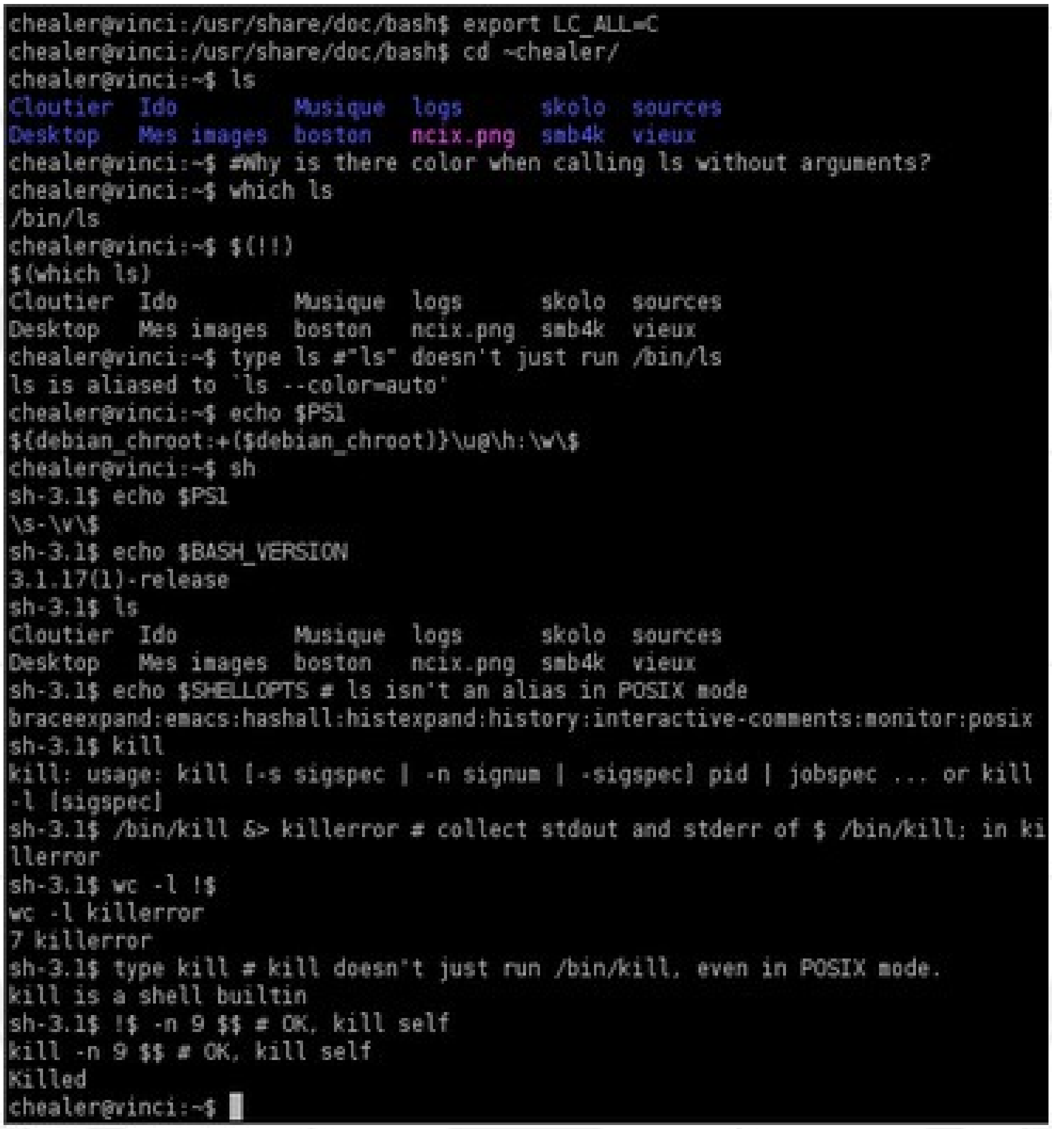
The Media Streaming Journal

October 2019



Covering Audio and Video Internet Broadcasting

Brought To You By

RADIOSOLUTION

www.radiosolution.info

publicdomainvectors.org/en/free-clipart/Vintage-microphone-vector-graphics/6111.html

The Media Streaming Journal Staff

Derek Bullard
Publication Director
info@radiosolution.info

David Childers
Editor In Chief
editor@radiosolution.info

Advertising
advertising@radiosolution.info

www.radiosolution.info

Welcome to The Media Streaming Journal

Greetings,

In the dark and ancient days before Microsoft Windows, computer users would create DOS batch scripts. Batch scripts would allow users to perform complicated functions or enables a user to create a logical sequence of actions within a textual run software application.

Shell Scripting is the Unix/ Linux version of creating DOS batch scripts. It is a beneficial capability that allows individuals to customize software applications or operating systems that can enhance application performance or system function. Unlike creating programs using a specific computer language, shell scripting enables the ability to control a system function or outcome.

It should be the goal of every broadcaster to know, understand, and be skilled in all aspects of Internet streaming, not just purchasing some streaming software and clicking a few buttons.

Please feel free to contact either the Publication Director (Derek Bullard) or myself if you have any questions or comments regarding The Media Streaming Journal.

Namaste

David Childers

www.linkedin.com/pub/david-childers/4/736/72a

The Grand Master of Digital Disaster
(Editor In Chief)

David Childers

The Grand Master of Digital Disaster

Current Member: International Association Of Internet Broadcasters

Former Member: Society of Motion Picture and Television Engineers

Published Author

Introduction To Internet Broadcasting
Amazon Publishing

Numerous Creative Commons Computer, Technical and Internet Broadcasting Guides
http://www.ScenicRadio.com/Library/BroadGuide/index.html

Newspaper Interviews

New York Times

Internet TV: Don't Touch That Mouse!
Tim Gnatek
July 1, 2004

Lagniappe - "Something Extra for Mobile"

Mobile Gets Hoaxed
Rob Holbert
Mar 16, 2016

Cited By

Five Essays on Copyright In the Digital Era
Ville Oksanen
2009

Turre Publishing
Helsinki Finland

Open Source Developer

Developed software architecture to continuously source multimedia content to Youtube Live servers.
Scenic Television – The sights and sounds of nature on the Internet.
http://www.ScenicRadio.com

Projects

Researched and developed documentation for Peercast P2P multimedia streaming project.
http://en.wikipedia.org/wiki/PeerCast

Researched and developed technical documentation for NSV / Winamp Television.
http://web.archive.org/web/20080601000000*/http://www.scvi.net

MidSummer Eve Webfest

A virtual International festival focusing on Digital art and Free Software that was coordinated by OrganicaDTM Design Studio.

Presentation and discussion regarding Internet multimedia content distribution.
http://web.archive.org/web/20061104230522/http://www.organicadtm.com/index.php?module=articles&func=display&catid=37&aid=61

LinkedIn Contact Information

http://www.linkedin.com/pub/david-childers/4/736/72a

The Media Streaming Journal

What is in this edition of the Media Streaming Journal

Advanced Bash-Scripting Guide
An in-depth exploration of the art of shell scripting

Bash Reference Manual
Reference Documentation for Bash

Join our technical discussion on Facebook

http://www.facebook.com/groups/internetradiosupport/

Magazine cover:

https://commons.wikimedia.org/wiki/File:Bash_demo.png

RADIOSOLUTION

www.radiosolution.info

Our Mission

Let our friendly, knowledgeable staff assist you to build your project, such as an online radio station using our high end reliable video and audio streaming technologies. We want to become your partner for all your hosting needs, as well as your one stop shop for radio products such as custom DJ drops and radio ID's.

Start An Internet Radio Station

Whatever you need to start Internet radio station, we will deliver! We provide high quality Internet Radio services to make your music radio project a success. We can provide Wowza, Icecast, SHOUTcast hosting and internet radio services to hobbyists, deejays, amateurs and established professionals. No radio station client is too big or too small for Radiosolution.

Choose between complete hassle-free service packages or new features to add to start internet radio station. Benefit from customized services and the latest in internet radio technology. You will receive professional, personalized and better Internet Radio Station services than you have received up till now. If you already have an Icecast or SHOUTcast hosting provider, we can still help you transfer your radio server over to us with no hassle and at no charge.

Internet Radio Station Services

Launch your internet, digital, satellite or AM/FM radio station anywhere in the world with all of the right tools. A broadcasting specialist is on standby to help you get started with an SHOUTcast or Icecast hosting package. We have servers ready for reliable streaming in North America and Europe. Our hosting packages have all the features you need to make your radio station project a success.

If you stream live or with an Auto DJ, we can provide you with the latest in web-based Cloud technology. You will love the simple to use control panel. Discover how easy it is to manage live deejays, upload fresh music and create custom scheduled programming. You will be able to track your listeners by getting real time statistics.

Starting your own Internet radio has never been easier. Get in touch with us anytime to start your Internet radio station.

Radiosolution is a SHOUTcast hosting provider located in Quebec Canada. We also offer Icecast, Wowza and Web Hosting services. Contact us to discuss the best option available as you start internet radio station. Radiosolution can provide personalized service in English, Dutch, and French. Starting an internet radio station can be intimidating, many people want to start one, but have no idea where to start. Radiosolution will be there for you every step of the way. Everyday people are searching the internet for free SHOUTcast servers. With Radiosolution SHOUTcast hosting we will allow you to try our services for FREE. By trying our services, you can be confident that you have chosen the best radio server hosting provider. You have nothing to loose because we offer a 30 day satisfaction guarantee. What are you waiting for? Contact us now! Radiosolution offers everything you need to start internet radio station. You will not need to go anywhere else. We can create your website, market your station and help you submit your station to online directories. We also feature the voice of Derek Bullard aka Dibblebee He can create affordable commercials, DJ intros, sweepers, jingles, ids and so much more.

Relax With The Sights And Sounds Of Nature

Scenic Television

Your Window To The World

Scenic Television is an Internet television station that broadcasts the sights and sounds of nature 24 hours a day. Savor exotic tropical beaches, or relax in a remote rain forest. Meditate at a bubbling stream, or relish the view of soft rolling waves at a lake. We have beautiful nature video from locations all around the world.

Scenic Television originates from the Gulf coast of South Alabama and broadcasts to a global audience. The television broadcast is accessible on any device with an Internet connection. Such electronic devices include desktop computers, laptops, tablets, smartphones, game platforms, and Internet-connected televisions.

http://www.scenictelevision.com

all-free-download.com/free-vector/download/magnifying_glass_clip_art_23181.html

We Are Your Information Resource

Are you looking for specialized data?

Are you swamped with information overload?

Do you need help finding the right information?

We Can Help You
Find The Information
That You Need

Our experienced data research analysts can wade through the vast information wasteland and find the information that you need.

We can save you both time and money.

We can streamline data requirement planning.

We can provide business critical information acquisition.

Contact us today

info@radiosolution.info

Advanced Bash-Scripting Guide
An in-depth exploration of the art of shell scripting
Mendel Cooper

A working knowledge of shell scripting is essential to anyone wishing to become reasonably proficient at system administration, even if they do not anticipate ever having to actually write a script. The craft of scripting is not hard to master, since scripts can be built in bite-sized sections and there is only a fairly small set of shell-specific operators and options to learn.

Bash Reference Manual
Reference Documentation for Bash
Edition 5.0, for Bash Version 5.0.
Chet Ramey, Case Western Reserve University
Brian Fox, Free Software Foundation

What follows is a tutorial on shell scripting. It relies heavily on examples to illustrate various features of the shell. The example scripts work -- they've been tested, insofar as possible -- and some of them are even useful in real life.

Advanced Bash-Scripting Guide

An in-depth exploration of the art of shell scripting

Mendel Cooper

<thegrendel.abs@gmail.com>

10

10 Mar 2014

Revision History

Revision 6.5	05 Apr 2012	Revised by: mc
'TUNGSTENBERRY' release		
Revision 6.6	27 Nov 2012	Revised by: mc
'YTTERBIUMBERRY' release		
Revision 10	10 Mar 2014	Revised by: mc
'PUBLICDOMAIN' release		

This tutorial assumes no previous knowledge of scripting or programming, yet progresses rapidly toward an intermediate/advanced level of instruction . . . *all the while sneaking in little nuggets of UNIX® wisdom and lore*. It serves as a textbook, a manual for self-study, and as a reference and source of knowledge on shell scripting techniques. The exercises and heavily-commented examples invite active reader participation, under the premise that **`the only way to really learn scripting is to write scripts`**.

This book is suitable for classroom use as a general introduction to programming concepts.

Dedication

For Anita, the source of all the magic

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Chapter 1. Shell Programming!

No programming language is perfect. There is not even a single best language; there are only languages well suited or perhaps poorly suited for particular purposes.

--Herbert Mayer

A working knowledge of shell scripting is essential to anyone wishing to become reasonably proficient at system administration, even if they do not anticipate ever having to actually write a script. Consider that as a Linux machine boots up, it executes the shell scripts in `/etc/rc.d` to restore the system configuration and set up services. A detailed understanding of these startup scripts is important for analyzing the behavior of a system, and possibly modifying it.

The craft of scripting is not hard to master, since scripts can be built in bite-sized sections and there is only a fairly small set of shell-specific operators and options [1] to learn. The syntax is simple -- even austere -- similar to that of invoking and chaining together utilities at the command line, and there are only a few "rules" governing their use. Most short scripts work right the first time, and debugging even the longer ones is straightforward.

```
In the early days of personal computing, the BASIC language enabled
anyone reasonably computer proficient to write programs on an early
generation of microcomputers. Decades later, the Bash scripting
language enables anyone with a rudimentary knowledge of Linux or
UNIX to do the same on modern machines.

We now have miniaturized single-board computers with amazing
capabilities, such as the Raspberry Pi.
Bash scripting provides a way to explore the capabilities of these
fascinating devices.
```

A shell script is a quick-and-dirty method of prototyping a complex application. Getting even a limited subset of the functionality to work in a script is often a useful first stage in project development. In this way, the structure of the application can be tested and tinkered with, and the major pitfalls found before proceeding to the final coding in *C*, *C++*, *Java*, Perl, or *Python*.

Shell scripting hearkens back to the classic UNIX philosophy of breaking complex projects into simpler subtasks, of chaining together components and utilities. Many consider this a better, or at least more esthetically pleasing approach to problem solving than using one of the new generation of high-powered all-in-one languages, such as *Perl*, which attempt to be all things to all people, but at the cost of forcing you to alter your thinking processes to fit the tool.

According to Herbert Mayer, "a useful language needs arrays, pointers, and a generic mechanism for building data structures." By these criteria, shell scripting falls somewhat short of being "useful." Or, perhaps not. . . .

When not to use shell scripts

- Resource-intensive tasks, especially where speed is a factor (sorting, hashing, recursion [2] ...)
- Procedures involving heavy-duty math operations, especially floating point arithmetic, arbitrary precision calculations, or complex numbers (use *C++* or *FORTRAN* instead)
- Cross-platform portability required (use *C* or *Java* instead)
- Complex applications, where structured programming is a necessity (type-checking of variables, function prototypes, etc.)
- Mission-critical applications upon which you are betting the future of the company
- Situations where *security* is important, where you need to guarantee the integrity of your system and protect against intrusion, cracking, and vandalism
- Project consists of subcomponents with interlocking dependencies
- Extensive file operations required (*Bash* is limited to serial file access, and that only in a particularly clumsy and inefficient line-by-line fashion.)
- Need native support for multi-dimensional arrays
- Need data structures, such as linked lists or trees
- Need to generate / manipulate graphics or GUIs
- Need direct access to system hardware or external peripherals
- Need port or socket I/O
- Need to use libraries or interface with legacy code
- Proprietary, closed-source applications (Shell scripts put the source code right out in the open for all the world to see.)

If any of the above applies, consider a more powerful scripting language -- perhaps *Perl*, *Tcl*, *Python*, *Ruby* -- or possibly a compiled language such as *C*, *C++*, or *Java*. Even then, prototyping the application as a shell script might still be a useful development step.

We will be using Bash, an acronym [3] for "Bourne-Again shell" and a pun on Stephen Bourne's now classic *Bourne* shell. Bash has become a *de facto* standard for shell scripting on most flavors of UNIX. Most of the principles this book covers apply equally well to scripting with other shells, such as the *Korn Shell*, from which Bash derives some of its features, [4] and the *C Shell* and its variants. (Note that *C Shell* programming is not recommended due to certain inherent problems, as pointed out in an October, 1993 Usenet post by Tom Christiansen.)

What follows is a tutorial on shell scripting. It relies heavily on examples to illustrate various features of the shell. The example scripts work -- they've been tested, insofar as possible -- and some of them are even useful in real life. The reader can play with the actual working code of the examples in the source archive (`scriptname.sh` or `scriptname.bash`), [5] give them *execute* permission (**`chmod u+rx scriptname`**), then run them to see what happens. Should the source archive not be available, then cut-and-paste from the HTML or pdf rendered versions. Be aware that some of the scripts presented here introduce features before they are explained, and this may require the reader to temporarily skip ahead for enlightenment.

Unless otherwise noted, the author of this book wrote the example scripts that follow.

His countenance was bold and bashed not.

--Edmund Spenser

Chapter 2. Starting Off With a Sha-Bang

Shell programming is a 1950s juke box . . .

--Larry Wall

In the simplest case, a script is nothing more than a list of system commands stored in a file. At the very least, this saves the effort of retyping that particular sequence of commands each time it is invoked.

Example 2-1. *cleanup*: A script to clean up log files in /var/log

```
# Cleanup
# Run as root, of course.

cd /var/log
cat /dev/null > messages
cat /dev/null > wtmp
echo "Log files cleaned up."
```

There is nothing unusual here, only a set of commands that could just as easily have been invoked one by one from the command-line on the console or in a terminal window. The advantages of placing the commands in a script go far beyond not having to retype them time and again. The script becomes a *program* -- a *tool* -- and it can easily be modified or customized for a particular application.

Example 2-2. *cleanup*: An improved clean-up script

```
#!/bin/bash
# Proper header for a Bash script.

# Cleanup, version 2

# Run as root, of course.
# Insert code here to print error message and exit if not root.

LOG_DIR=/var/log
# Variables are better than hard-coded values.
cd $LOG_DIR

cat /dev/null > messages
cat /dev/null > wtmp


echo "Logs cleaned up."

exit #  The right and proper method of "exiting" from a script.
     #  A bare "exit" (no parameter) returns the exit status
     #+ of the preceding command.
```

Now *that's* beginning to look like a real script. But we can go even farther . . .

Example 2-3. *cleanup*: An enhanced and generalized version of above scripts.

```
#!/bin/bash
# Cleanup, version 3
```

```
#  Warning:
#  -------
#  This script uses quite a number of features that will be explained
#+ later on.
#  By the time you've finished the first half of the book,
#+ there should be nothing mysterious about it.



LOG_DIR=/var/log
ROOT_UID=0     # Only users with $UID 0 have root privileges.
LINES=50       # Default number of lines saved.
E_XCD=86       # Can't change directory?
E_NOTROOT=87   # Non-root exit error.


# Run as root, of course.
if [ "$UID" -ne "$ROOT_UID" ]
then
  echo "Must be root to run this script."
  exit $E_NOTROOT
fi

if [ -n "$1" ]
# Test whether command-line argument is present (non-empty).
then
  lines=$1
else
  lines=$LINES # Default, if not specified on command-line.
fi


#  Stephane Chazelas suggests the following,
#+ as a better way of checking command-line arguments,
#+ but this is still a bit advanced for this stage of the tutorial.
#
#    E_WRONGARGS=85  # Non-numerical argument (bad argument format).
#
#    case "$1" in
#    ""      ) lines=50;;
#    *[!0-9]*) echo "Usage: `basename $0` lines-to-cleanup";
#     exit $E_WRONGARGS;;
#    *       ) lines=$1;;
#    esac
#
#* Skip ahead to "Loops" chapter to decipher all this.


cd $LOG_DIR

if [ `pwd` != "$LOG_DIR" ]  # or   if [ "$PWD" != "$LOG_DIR" ]
                            # Not in /var/log?
then
  echo "Can't change to $LOG_DIR."
  exit $E_XCD
fi  # Doublecheck if in right directory before messing with log file.

# Far more efficient is:
#
# cd /var/log || {
#   echo "Cannot change to necessary directory." >&2
#   exit $E_XCD;
```

```
# }



tail -n $lines messages > mesg.temp # Save last section of message log file.
mv mesg.temp messages               # Rename it as system log file.


#  cat /dev/null > messages
#* No longer needed, as the above method is safer.

cat /dev/null > wtmp  #  ': > wtmp' and '> wtmp'  have the same effect.
echo "Log files cleaned up."
#  Note that there are other log files in /var/log not affected
#+ by this script.

exit 0
#  A zero return value from the script upon exit indicates success
#+ to the shell.
```

Since you may not wish to wipe out the entire system log, this version of the script keeps the last section of the message log intact. You will constantly discover ways of fine-tuning previously written scripts for increased effectiveness.

* * *

The *sha-bang* (#!) [6] at the head of a script tells your system that this file is a set of commands to be fed to the command interpreter indicated. The #! is actually a two-byte [7] *magic number*, a special marker that designates a file type, or in this case an executable shell script (type **`man magic`** for more details on this fascinating topic). Immediately following the *sha-bang* is a *path name*. This is the path to the program that interprets the commands in the script, whether it be a shell, a programming language, or a utility. This command interpreter then executes the commands in the script, starting at the top (the line following the *sha-bang* line), and ignoring comments. [8]

```
#!/bin/sh
#!/bin/bash
#!/usr/bin/perl
#!/usr/bin/tcl
#!/bin/sed -f
#!/bin/awk -f
```

Each of the above script header lines calls a different command interpreter, be it `/bin/sh`, the default shell (**bash** in a Linux system) or otherwise. [9] Using **`#!/bin/sh`**, the default Bourne shell in most commercial variants of UNIX, makes the script portable to non-Linux machines, though you sacrifice Bash-specific features. The script will, however, conform to the POSIX [10] **sh** standard.

Note that the path given at the "sha-bang" must be correct, otherwise an error message -- usually "Command not found." -- will be the only result of running the script. [11]

#! can be omitted if the script consists only of a set of generic system commands, using no internal shell directives. The second example, above, requires the initial #!, since the variable assignment line, **`lines=50`**, uses a shell-specific construct. [12] Note again that **`#!/bin/sh`** invokes the default shell interpreter, which defaults to `/bin/bash` on a Linux machine.

This tutorial encourages a modular approach to constructing a script. Make note of and collect "boilerplate" code snippets that might be useful in future scripts. Eventually you will build quite an extensive library of nifty routines. As an example, the following script prolog tests whether the script has been invoked with the correct number of parameters.

```
E_WRONG_ARGS=85
script_parameters="-a -h -m -z"
#                  -a = all, -h = help, etc.

if [ $# -ne $Number_of_expected_args ]
then
  echo "Usage: `basename $0` $script_parameters"
  # `basename $0` is the script's filename.
  exit $E_WRONG_ARGS
fi
```

Many times, you will write a script that carries out one particular task. The first script in this chapter is an example. Later, it might occur to you to generalize the script to do other, similar tasks. Replacing the literal ("hard-wired") constants by variables is a step in that direction, as is replacing repetitive code blocks by functions.

2.1. Invoking the script

Having written the script, you can invoke it by **sh scriptname**, [13] or alternatively **bash scriptname**. (Not recommended is using **sh <scriptname**, since this effectively disables reading from `stdin` within the script.) Much more convenient is to make the script itself directly executable with a chmod.

Either:

chmod 555 scriptname (gives everyone read/execute permission) [14]

or

chmod +rx scriptname (gives everyone read/execute permission)

chmod u+rx scriptname (gives only the script owner read/execute permission)

Having made the script executable, you may now test it by **./scriptname**. [15] If it begins with a "sha-bang" line, invoking the script calls the correct command interpreter to run it.

As a final step, after testing and debugging, you would likely want to move it to `/usr/local/bin` (as *root*, of course), to make the script available to yourself and all other users as a systemwide executable. The script could then be invoked by simply typing **scriptname [ENTER]** from the command-line.

2.2. Preliminary Exercises

1. System administrators often write scripts to automate common tasks. Give several instances where such scripts would be useful.
2. Write a script that upon invocation shows the time and date, lists all logged-in users, and gives the system uptime. The script then saves this information to a logfile.

Part 2. Basics

Table of Contents

Chapter 3. Special Characters

What makes a character *special*? If it has a meaning beyond its *literal meaning*, a meta-meaning, then we refer to it as a *special character*. Along with commands and keywords, *special characters* are building blocks of Bash scripts.

Special Characters Found In Scripts and Elsewhere

#

Comments. Lines beginning with a # (with the exception of #!) are comments and will *not* be executed.

```
# This line is a comment.
```

Comments may also occur following the end of a command.

```
echo "A comment will follow." # Comment here.
#                            ^ Note whitespace before #
```

Comments may also follow whitespace at the beginning of a line.

```
	# A tab precedes this comment.
```

Comments may even be embedded within a pipe.

```
initial=( `cat "$startfile" | sed -e '/#/d' | tr -d '\n' |\
# Delete lines containing '#' comment character.
           sed -e 's/\./\. /g' -e 's/_/_ /g'` )
# Excerpted from life.sh script
```

A command may not follow a comment on the same line. There is no method of terminating the comment, in order for "live code" to begin on the same line. Use a new line for the next command.

Of course, a quoted or an escaped # in an echo statement does *not* begin a comment. Likewise, a # appears in certain parameter-substitution constructs and in numerical constant expressions.

```
echo "The # here does not begin a comment."
echo 'The # here does not begin a comment.'
echo The \# here does not begin a comment.
echo The # here begins a comment.

echo ${PATH#*:}       # Parameter substitution, not a comment.
echo $(( 2#101011 ))  # Base conversion, not a comment.

# Thanks, S.C.
```

The standard quoting and escape characters (" ' \) escape the #.

Certain pattern matching operations also use the #.

;

Command separator [semicolon]. Permits putting two or more commands on the same line.

```
echo hello; echo there
```

```
if [ -x "$filename" ]; then    #  Note the space after the semicolon.
#+                   ^^
  echo "File $filename exists."; cp $filename $filename.bak
else   #                       ^^
  echo "File $filename not found."; touch $filename
fi; echo "File test complete."
```

Note that the ";" sometimes needs to be *escaped*.

;;

Terminator in a case option [double semicolon].

```
case "$variable" in
  abc)  echo "\$variable = abc" ;;
  xyz)  echo "\$variable = xyz" ;;
esac
```

;;&, ;&

Terminators in a *case* option (version 4+ of Bash).

.

"dot" command [period]. Equivalent to source (see Example 15-22). This is a bash builtin.

.

"dot", as a component of a filename. When working with filenames, a leading dot is the prefix of a "hidden" file, a file that an ls will not normally show.

```
bash$ touch .hidden-file
bash$ ls -l
total 10
 -rw-r--r--    1 bozo      4034 Jul 18 22:04 data1.addressbook
 -rw-r--r--    1 bozo      4602 May 25 13:58 data1.addressbook.bak
 -rw-r--r--    1 bozo       877 Dec 17  2000 employment.addressbook


bash$ ls -al
total 14
 drwxrwxr-x    2 bozo  bozo      1024 Aug 29 20:54 ./
 drwx------   52 bozo  bozo      3072 Aug 29 20:51 ../
 -rw-r--r--    1 bozo  bozo      4034 Jul 18 22:04 data1.addressbook
 -rw-r--r--    1 bozo  bozo      4602 May 25 13:58 data1.addressbook.bak
 -rw-r--r--    1 bozo  bozo       877 Dec 17  2000 employment.addressbook
 -rw-rw-r--    1 bozo  bozo         0 Aug 29 20:54 .hidden-file
```

When considering directory names, *a single dot* represents the current working directory, and *two dots* denote the parent directory.

```
bash$ pwd
/home/bozo/projects

bash$ cd .
bash$ pwd
/home/bozo/projects

bash$ cd ..
bash$ pwd
/home/bozo/
```

The *dot* often appears as the destination (directory) of a file movement command, in this context meaning *current directory*.

```
bash$ cp /home/bozo/current_work/junk/* .
```

Copy all the "junk" files to $PWD.

.

"dot" character match. When matching characters, as part of a regular expression, a "dot" matches a single character.

"

partial quoting [double quote]. *"STRING"* preserves (from interpretation) most of the special characters within *STRING*. See Chapter 5.

'

full quoting [single quote]. *'STRING'* preserves all special characters within *STRING*. This is a stronger form of quoting than *"STRING"*. See Chapter 5.

,

comma operator. The *comma operator* [16] links together a series of arithmetic operations. All are evaluated, but only the last one is returned.

```
let "t2 = ((a = 9, 15 / 3))"
# Set "a = 9" and "t2 = 15 / 3"
```

The *comma* operator can also concatenate strings.

```
for file in /{,usr/}bin/*calc
#             ^    Find all executable files ending in "calc"
#+                 in /bin and /usr/bin directories.
do
        if [ -x "$file" ]
        then
          echo $file
        fi
done

# /bin/ipcalc
# /usr/bin/kcalc
# /usr/bin/oidcalc
# /usr/bin/oocalc


# Thank you, Rory Winston, for pointing this out.
```

,, ,

Lowercase conversion in *parameter substitution* (added in version 4 of Bash).

\

escape [backslash]. A quoting mechanism for single characters.

`\X` *escapes* the character *X*. This has the effect of "quoting" *X*, equivalent to *'X'*. The \ may be used to quote " and ', so they are expressed literally.

See Chapter 5 for an in-depth explanation of escaped characters.

/

Filename path separator [forward slash]. Separates the components of a filename (as in `/home/bozo/projects/Makefile`).

This is also the division arithmetic operator.

`

command substitution. The **`command`** construct makes available the output of **command** for assignment to a variable. This is also known as backquotes or backticks.

:

null command [colon]. This is the shell equivalent of a "NOP" (`no op`, a do-nothing operation). It may be considered a synonym for the shell builtin true. The ":" command is itself a *Bash* builtin, and its exit status is *true* (0).

```
:
echo $?   # 0
```

Endless loop:

```
while :
do
   operation-1
   operation-2
   ...
   operation-n
done

# Same as:
#    while true
#    do
#      ...
#    done
```

Placeholder in if/then test:

```
if condition
then :   # Do nothing and branch ahead
else     # Or else ...
   take-some-action
fi
```

Provide a placeholder where a binary operation is expected, see Example 8-2 and default parameters.

```
: ${username=`whoami`}
# ${username=`whoami`}   Gives an error without the leading :
#                        unless "username" is a command or builtin...

: ${1?"Usage: $0 ARGUMENT"}     # From "usage-message.sh example script.
```

Provide a placeholder where a command is expected in a here document. See Example 19-10.

Evaluate string of variables using parameter substitution (as in Example 10-7).

```
: ${HOSTNAME?} ${USER?} ${MAIL?}
#  Prints error message
#+ if one or more of essential environmental variables not set.
```

Variable expansion / substring replacement.

In combination with the > redirection operator, truncates a file to zero length, without changing its permissions. If the file did not previously exist, creates it.

```
: > data.xxx   # File "data.xxx" now empty.

# Same effect as   cat /dev/null >data.xxx
# However, this does not fork a new process, since ":" is a builtin.
```

See also Example 16-15.

In combination with the >> redirection operator, has no effect on a pre-existing target file (**`: >> target_file`**). If the file did not previously exist, creates it.

This applies to regular files, not pipes, symlinks, and certain special files.

May be used to begin a comment line, although this is not recommended. Using # for a comment turns off error checking for the remainder of that line, so almost anything may appear in a comment. However, this is not the case with :.

```
: This is a comment that generates an error, ( if [ $x -eq 3] ).
```

The ":" serves as a field separator, in `/etc/passwd`, and in the $PATH variable.

```
bash$ echo $PATH
/usr/local/bin:/bin:/usr/bin:/usr/X11R6/bin:/sbin:/usr/sbin:/usr/games
```

A *colon* is acceptable as a function name.

```
:()
{
  echo "The name of this function is "$FUNCNAME" "
  # Why use a colon as a function name?
  # It's a way of obfuscating your code.
}

:

# The name of this function is :
```

This is not portable behavior, and therefore not a recommended practice. In fact, more recent releases of Bash do not permit this usage. An underscore _ works, though.

A *colon* can serve as a placeholder in an otherwise empty function.

```
not_empty ()
{
  :
} # Contains a : (null command), and so is not empty.
```

!

reverse (or negate) the sense of a test or exit status [bang]. The ! operator inverts the exit status of the command to which it is applied (see Example 6-2). It also inverts the meaning of a test operator. This can, for example, change the sense of *equal* (=) to *not-equal* (!=). The ! operator is a Bash keyword.

In a different context, the ! also appears in indirect variable references.

In yet another context, from the *command line*, the ! invokes the Bash *history mechanism* (see Appendix L). Note that within a script, the history mechanism is disabled.

*

wild card [asterisk]. The * character serves as a "wild card" for filename expansion in globbing. By itself, it matches every filename in a given directory.

```
bash$ echo *
abs-book.sgml add-drive.sh agram.sh alias.sh
```

The * also represents any number (or zero) characters in a regular expression.

*

arithmetic operator. In the context of arithmetic operations, the * denotes multiplication.

** A double asterisk can represent the exponentiation operator or extended file-match *globbing*.

?

test operator. Within certain expressions, the ? indicates a test for a condition.

In a double-parentheses construct, the ? can serve as an element of a C-style *trinary* operator. [17]

`condition?result-if-true:result-if-false`

```
(( var0 = var1<98?9:21 ))
#                ^ ^

# if [ "$var1" -lt 98 ]
# then
#   var0=9
# else
#   var0=21
# fi
```

In a parameter substitution expression, the ? tests whether a variable has been set.

?

wild card. The ? character serves as a single-character "wild card" for filename expansion in globbing, as well as representing one character in an extended regular expression.

$

Variable substitution (contents of a variable).

```
var1=5
var2=23skidoo

echo $var1     # 5
echo $var2     # 23skidoo
```

A $ prefixing a variable name indicates the *value* the variable holds.

$

end-of-line. In a regular expression, a "$" addresses the end of a line of text.

${}

Parameter substitution.

$' ... '

Quoted string expansion. This construct expands single or multiple escaped octal or hex values into ASCII [18] or Unicode characters.

$*, $@

positional parameters.

$?

exit status variable. The $? variable holds the exit status of a command, a function, or of the script itself.

$$

process ID variable. The $$ variable holds the *process ID* [19] of the script in which it appears.

()

command group.

```
(a=hello; echo $a)
```

A listing of commands within `parentheses` starts a subshell.

Variables inside parentheses, within the subshell, are not visible to the rest of the script. The parent process, the script, cannot read variables created in the child process, the subshell.

```
a=123
( a=321; )

echo "a = $a"   # a = 123
# "a" within parentheses acts like a local variable.
```

array initialization.

```
Array=(element1 element2 element3)
```

{xxx,yyy,zzz,...}

Brace expansion.

```
echo \"{These,words,are,quoted}\"   # " prefix and suffix
# "These" "words" "are" "quoted"


cat {file1,file2,file3} > combined_file
# Concatenates the files file1, file2, and file3 into combined_file.

cp file22.{txt,backup}
# Copies "file22.txt" to "file22.backup"
```

A command may act upon a comma-separated list of file specs within `braces`. [20] Filename expansion (globbing) applies to the file specs between the braces.

No spaces allowed within the braces *unless* the spaces are quoted or escaped.

`echo {file1,file2}\ :{\ A," B",' C'}`

`file1 : A file1 : B file1 : C file2 : A file2 : B file2 : C`

{a..z}

Extended Brace expansion.

```
echo {a..z} # a b c d e f g h i j k l m n o p q r s t u v w x y z
# Echoes characters between a and z.

echo {0..3} # 0 1 2 3
# Echoes characters between 0 and 3.


base64_charset=( {A..Z} {a..z} {0..9} + / = )
# Initializing an array, using extended brace expansion.
# From vladz's "base64.sh" example script.
```

The *{a..z}* extended brace expansion construction is a feature introduced in version 3 of *Bash*.

{}

Block of code [curly brackets]. Also referred to as an *inline group*, this construct, in effect, creates an *anonymous function* (a function without a name). However, unlike in a "standard" function, the variables inside a code block remain visible to the remainder of the script.

```
bash$ { local a;
              a=123; }
bash: local: can only be used in a
function
```

```
a=123
{ a=321; }
echo "a = $a"   # a = 321   (value inside code block)

# Thanks, S.C.
```

The code block enclosed in braces may have I/O redirected to and from it.

Example 3-1. Code blocks and I/O redirection

```
#!/bin/bash
# Reading lines in /etc/fstab.

File=/etc/fstab

{
read line1
read line2
} < $File

echo "First line in $File is:"
echo "$line1"
echo
echo "Second line in $File is:"
echo "$line2"

exit 0

# Now, how do you parse the separate fields of each line?
# Hint: use awk, or . . .
# . . . Hans-Joerg Diers suggests using the "set" Bash builtin.
```

Example 3-2. Saving the output of a code block to a file

```
#!/bin/bash
# rpm-check.sh

#  Queries an rpm file for description, listing,
#+ and whether it can be installed.
#  Saves output to a file.
#
#  This script illustrates using a code block.

SUCCESS=0
E_NOARGS=65

if [ -z "$1" ]
then
```

```
  echo "Usage: `basename $0` rpm-file"
  exit $E_NOARGS
fi

{ # Begin code block.
  echo
  echo "Archive Description:"
  rpm -qpi $1       # Query description.
  echo
  echo "Archive Listing:"
  rpm -qpl $1       # Query listing.
  echo
  rpm -i --test $1  # Query whether rpm file can be installed.
  if [ "$?" -eq $SUCCESS ]
  then
    echo "$1 can be installed."
  else
    echo "$1 cannot be installed."
  fi
  echo              # End code block.
} > "$1.test"       # Redirects output of everything in block to file.

echo "Results of rpm test in file $1.test"

# See rpm man page for explanation of options.

exit 0
```

Unlike a command group within (parentheses), as above, a code block enclosed by {braces} will *not* normally launch a subshell. [21]

It is possible to iterate a code block using a non-standard *for-loop*.

{}

placeholder for text. Used after xargs -i (*replace strings* option). The {} double curly brackets are a placeholder for output text.

```
ls . | xargs -i -t cp ./{} $1
#            ^^         ^^

# From "ex42.sh" (copydir.sh) example.
```

{} \;

pathname. Mostly used in find constructs. This is *not* a shell builtin.

Definition: A *pathname* is a *filename* that includes the complete path. As an example, `/home/bozo/Notes/Thursday/schedule.txt`. This is sometimes referred to as the *absolute path*.

The ";" ends the `-exec` option of a **find** command sequence. It needs to be escaped to protect it from interpretation by the shell.

[ ]

test.

Test expression between **[ ]**. Note that **[** is part of the shell *builtin* test (and a synonym for it), *not* a link to the external command `/usr/bin/test`.

[[ ]]

test.

Test expression between [[ ]]. More flexible than the single-bracket [ ] test, this is a shell keyword.

See the discussion on the [[...]] construct.

[ ]

array element.

In the context of an array, brackets set off the numbering of each element of that array.

```
Array[1]=slot_1
echo ${Array[1]}
```

[ ]

range of characters.

As part of a regular expression, brackets delineate a range of characters to match.

$[...]

integer expansion.

Evaluate integer expression between $[ ].

```
a=3
b=7

echo $[$a+$b]   # 10
echo $[$a*$b]   # 21
```

Note that this usage is *deprecated*, and has been replaced by the ((...)) construct.

(( ))

integer expansion.

Expand and evaluate integer expression between (( )).

See the discussion on the ((...)) construct.

> &> >& >> < <>

redirection.

`scriptname >filename` redirects the output of `scriptname` to file `filename`. Overwrite `filename` if it already exists.

`command &>filename` redirects both the `stdout` and the `stderr` of `command` to `filename`.

This is useful for suppressing output when testing for a condition. For example, let us test whether a certain command exists.

```
bash$ type bogus_command &>/dev/null
```

```
bash$ echo $?
1
```

Or in a script:

```
command_test () { type "$1" &>/dev/null; }
#                                       ^

cmd=rmdir            # Legitimate command.
command_test $cmd; echo $?   # 0


cmd=bogus_command    # Illegitimate command
command_test $cmd; echo $?   # 1
```

`command >&2` redirects `stdout` of `command` to `stderr`.

`scriptname >>filename` appends the output of `scriptname` to file `filename`. If `filename` does not already exist, it is created.

`[i]<>filename` opens file `filename` for reading and writing, and assigns file descriptor i to it. If `filename` does not exist, it is created.

process substitution.

`(command)>`

`<(command)`

In a different context, the "<" and ">" characters act as string comparison operators.

In yet another context, the "<" and ">" characters act as integer comparison operators. See also Example 16-9.

<<

redirection used in a here document.

<<<

redirection used in a here string.

<, >

ASCII comparison.

```
veg1=carrots
veg2=tomatoes

if [[ "$veg1" < "$veg2" ]]
then
  echo "Although $veg1 precede $veg2 in the dictionary,"
  echo -n "this does not necessarily imply anything "
  echo "about my culinary preferences."
else
  echo "What kind of dictionary are you using, anyhow?"
fi
```

\<, \>

word boundary in a regular expression.

```
bash$ grep '\<the\>' textfile
```

|

pipe. Passes the output (`stdout`) of a previous command to the input (`stdin`) of the next one, or to the shell. This is a method of chaining commands together.

```
echo ls -l | sh
#  Passes the output of "echo ls -l" to the shell,
#+ with the same result as a simple "ls -l".


cat *.lst | sort | uniq
# Merges and sorts all ".lst" files, then deletes duplicate lines.
```

A pipe, as a classic method of interprocess communication, sends the `stdout` of one process to the `stdin` of another. In a typical case, a command, such as cat or echo, pipes a stream of data to a *filter*, a command that transforms its input for processing. [22]

```
cat $filename1 $filename2 | grep $search_word
```

For an interesting note on the complexity of using UNIX pipes, see the UNIX FAQ, Part 3.

The output of a command or commands may be piped to a script.

```
#!/bin/bash
# uppercase.sh : Changes input to uppercase.

tr 'a-z' 'A-Z'
#  Letter ranges must be quoted
#+ to prevent filename generation from single-letter filenames.

exit 0
```

Now, let us pipe the output of **ls -l** to this script.

```
bash$ ls -l | ./uppercase.sh
-RW-RW-R--    1 BOZO  BOZO       109 APR  7 19:49 1.TXT
 -RW-RW-R--    1 BOZO  BOZO       109 APR 14 16:48 2.TXT
 -RW-R--R--    1 BOZO  BOZO       725 APR 20 20:56 DATA-FILE
```

The `stdout` of each process in a pipe must be read as the `stdin` of the next. If this is not the case, the data stream will *block*, and the pipe will not behave as expected.

```
cat file1 file2 | ls -l | sort
# The output from "cat file1 file2" disappears.
```

A pipe runs as a child process, and therefore cannot alter script variables.

```
variable="initial_value"
echo "new_value" | read variable
echo "variable = $variable"     # variable = initial_value
```

If one of the commands in the pipe aborts, this prematurely terminates execution of the pipe. Called a *broken pipe*, this condition sends a *`SIGPIPE`* signal.

>|

force redirection (even if the noclobber option is set). This will forcibly overwrite an existing file.

||

OR logical operator. In a test construct, the || operator causes a return of 0 (success) if *either* of the linked test conditions is true.

&

Run job in background. A command followed by an & will run in the background.

```
bash$ sleep 10 &
[1] 850
[1]+  Done                    sleep 10
```

Within a script, commands and even loops may run in the background.

Example 3-3. Running a loop in the background

```
#!/bin/bash
# background-loop.sh

for i in 1 2 3 4 5 6 7 8 9 10            # First loop.
do
  echo -n "$i "
done & # Run this loop in background.
       # Will sometimes execute after second loop.

echo   # This 'echo' sometimes will not display.

for i in 11 12 13 14 15 16 17 18 19 20   # Second loop.
do
  echo -n "$i "
done

echo   # This 'echo' sometimes will not display.

# ======================================================

# The expected output from the script:
# 1 2 3 4 5 6 7 8 9 10
# 11 12 13 14 15 16 17 18 19 20

# Sometimes, though, you get:
# 11 12 13 14 15 16 17 18 19 20
# 1 2 3 4 5 6 7 8 9 10 bozo $
# (The second 'echo' doesn't execute. Why?)

# Occasionally also:
# 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
# (The first 'echo' doesn't execute. Why?)

# Very rarely something like:
# 11 12 13 1 2 3 4 5 6 7 8 9 10 14 15 16 17 18 19 20
# The foreground loop preempts the background one.

exit 0

#  Nasimuddin Ansari suggests adding    sleep 1
#+ after the   echo -n "$i"   in lines 6 and 14,
#+ for some real fun.
```

A command run in the background within a script may cause the script to hang, waiting for a keystroke. Fortunately, there is a remedy for this.

&&

AND logical operator. In a test construct, the && operator causes a return of 0 (success) only if *both* the linked test conditions are true.

-

option, prefix. Option flag for a command or filter. Prefix for an operator. Prefix for a default parameter in parameter substitution.

`COMMAND -[Option1][Option2][...]`

`ls -al`

`sort -dfu $filename`

```
if [ $file1 -ot $file2 ]
then #      ^
  echo "File $file1 is older than $file2."
fi

if [ "$a" -eq "$b" ]
then #    ^
  echo "$a is equal to $b."
fi

if [ "$c" -eq 24 -a "$d" -eq 47 ]
then #    ^              ^
  echo "$c equals 24 and $d equals 47."
fi


param2=${param1:-$DEFAULTVAL}
#                ^
```

--

The *double-dash* `--` prefixes *long* (verbatim) options to commands.

`sort --ignore-leading-blanks`

Used with a Bash builtin, it means the *end of options* to that particular command.

This provides a handy means of removing files whose *names begin with a dash*.

```
bash$ ls -l
-rw-r--r-- 1 bozo bozo 0 Nov 25 12:29 -badname


bash$ rm -- -badname

bash$ ls -l
total 0
```

The *double-dash* is also used in conjunction with set.

`set -- $variable` (as in Example 15-18)

-

redirection from/to `stdin` or `stdout` [dash].

```
bash$ cat -
abc
abc

...

Ctl-D
```

As expected, **`cat -`** echoes `stdin`, in this case keyboarded user input, to `stdout`. But, does I/O redirection using **-** have real-world applications?

```
(cd /source/directory && tar cf - . ) | (cd /dest/directory && tar xpvf -)
# Move entire file tree from one directory to another
# [courtesy Alan Cox <a.cox@swansea.ac.uk>, with a minor change]

# 1) cd /source/directory
#    Source directory, where the files to be moved are.
# 2) &&
#   "And-list": if the 'cd' operation successful,
#    then execute the next command.
# 3) tar cf - .
#    The 'c' option 'tar' archiving command creates a new archive,
#    the 'f' (file) option, followed by '-' designates the target file
#    as stdout, and do it in current directory tree ('.').
# 4) |
#    Piped to ...
# 5) ( ... )
#    a subshell
# 6) cd /dest/directory
#    Change to the destination directory.
# 7) &&
#   "And-list", as above
# 8) tar xpvf -
#    Unarchive ('x'), preserve ownership and file permissions ('p'),
#    and send verbose messages to stdout ('v'),
#    reading data from stdin ('f' followed by '-').
#
#    Note that 'x' is a command, and 'p', 'v', 'f' are options.
#
# Whew!



# More elegant than, but equivalent to:
#   cd source/directory
#   tar cf - . | (cd ../dest/directory; tar xpvf -)
#
#     Also having same effect:
# cp -a /source/directory/* /dest/directory
#     Or:
# cp -a /source/directory/* /source/directory/.[^.]* /dest/directory
#     If there are hidden files in /source/directory.
```

```
bunzip2 -c linux-2.6.16.tar.bz2 | tar xvf -
#  --uncompress tar file--      | --then pass it to "tar"--
#  If "tar" has not been patched to handle "bunzip2",
#+ this needs to be done in two discrete steps, using a pipe.
#  The purpose of the exercise is to unarchive "bzipped" kernel source.
```

Note that in this context the "-" is not itself a Bash operator, but rather an option recognized by certain UNIX utilities that write to `stdout`, such as **tar**, **cat**, etc.

```
bash$ echo "whatever" | cat -
whatever
```

Where a filename is expected, - redirects output to `stdout` (sometimes seen with **`tar cf`**), or accepts input from `stdin`, rather than from a file. This is a method of using a file-oriented utility as a filter in a pipe.

```
bash$ file
Usage: file [-bciknvzL] [-f namefile] [-m magicfiles] file...
```

By itself on the command-line, file fails with an error message.

Add a "-" for a more useful result. This causes the shell to await user input.

```
bash$ file -
abc
standard input:              ASCII text



bash$ file -
#!/bin/bash
standard input:              Bourne-Again shell script text executable
```

Now the command accepts input from `stdin` and analyzes it.

The "-" can be used to pipe `stdout` to other commands. This permits such stunts as prepending lines to a file.

Using diff to compare a file with a *section* of another:

`grep Linux file1 | diff file2 -`

Finally, a real-world example using - with tar.

Example 3-4. Backup of all files changed in last day

```
#!/bin/bash

#  Backs up all files in current directory modified within last 24 hours
#+ in a "tarball" (tarred and gzipped file).

BACKUPFILE=backup-$(date +%m-%d-%Y)
#                 Embeds date in backup filename.
#                 Thanks, Joshua Tschida, for the idea.
archive=${1:-$BACKUPFILE}
#  If no backup-archive filename specified on command-line,
#+ it will default to "backup-MM-DD-YYYY.tar.gz."

tar cvf - `find . -mtime -1 -type f -print` > $archive.tar
gzip $archive.tar
echo "Directory $PWD backed up in archive file \"$archive.tar.gz\"."
```

```
#  Stephane Chazelas points out that the above code will fail
#+ if there are too many files found
#+ or if any filenames contain blank characters.

# He suggests the following alternatives:
# -------------------------------------------------------------------
#   find . -mtime -1 -type f -print0 | xargs -0 tar rvf "$archive.tar"
#      using the GNU version of "find".


#   find . -mtime -1 -type f -exec tar rvf "$archive.tar" '{}' \;
#         portable to other UNIX flavors, but much slower.
# -------------------------------------------------------------------


exit 0
```

Filenames beginning with "-" may cause problems when coupled with the "-" redirection operator. A script should check for this and add an appropriate prefix to such filenames, for example `./-FILENAME`, `$PWD/-FILENAME`, or `$PATHNAME/-FILENAME`.

If the value of a variable begins with a -, this may likewise create problems.

```
var="-n"
echo $var
# Has the effect of "echo -n", and outputs nothing.
```

-

previous working directory. A **cd -** command changes to the previous working directory. This uses the $OLDPWD environmental variable.

Do not confuse the "-" used in this sense with the "-" redirection operator just discussed. The interpretation of the "-" depends on the context in which it appears.

-

Minus. Minus sign in an arithmetic operation.

=

Equals. Assignment operator

```
a=28
echo $a   # 28
```

In a different context, the "=" is a string comparison operator.

+

Plus. Addition arithmetic operator.

In a different context, the + is a Regular Expression operator.

+

Option. Option flag for a command or filter.

Certain commands and builtins use the + to enable certain options and the - to disable them. In parameter substitution, the + prefixes an alternate value that a variable expands to.

%

modulo. Modulo (remainder of a division) arithmetic operation.

```
let "z = 5 % 3"
echo $z  # 2
```

In a different context, the % is a pattern matching operator.

~

home directory [tilde]. This corresponds to the $HOME internal variable. `~bozo` is bozo's home directory, and **ls ~bozo** lists the contents of it. ~/ is the current user's home directory, and **ls ~/** lists the contents of it.

```
bash$ echo ~bozo
/home/bozo

bash$ echo ~
/home/bozo

bash$ echo ~/
/home/bozo/

bash$ echo ~:
/home/bozo:

bash$ echo ~nonexistent-user
~nonexistent-user
```

~+

current working directory. This corresponds to the $PWD internal variable.

~-

previous working directory. This corresponds to the $OLDPWD internal variable.

=~

regular expression match. This operator was introduced with version 3 of Bash.

^

beginning-of-line. In a regular expression, a "^" addresses the beginning of a line of text.

^, ^^

Uppercase conversion in *parameter substitution* (added in version 4 of Bash).

Control Characters

change the behavior of the terminal or text display. A control character is a **CONTROL + key** combination (pressed simultaneously). A control character may also be written in *octal* or *hexadecimal* notation, following an *escape*.

Control characters are not normally useful inside a script.

◊ **`Ctl-A`**

Moves cursor to beginning of line of text (on the command-line).

◊ **`Ctl-B`**

`Backspace` (nondestructive).

◊ **`Ctl-C`**

`Break`. Terminate a foreground job.

◊ **`Ctl-D`**

Log out from a shell (similar to exit).

EOF (end-of-file). This also terminates input from `stdin`.

When typing text on the console or in an *xterm* window, **Ctl-D** erases the character under the cursor. When there are no characters present, **Ctl-D** logs out of the session, as expected. In an *xterm* window, this has the effect of closing the window.

◊ **Ctl-E**

Moves cursor to end of line of text (on the command-line).

◊ **Ctl-F**

Moves cursor forward one character position (on the command-line).

◊ **Ctl-G**

BEL. On some old-time teletype terminals, this would actually ring a bell. In an *xterm* it might beep.

◊ **Ctl-H**

Rubout (destructive backspace). Erases characters the cursor backs over while backspacing.

```
#!/bin/bash
# Embedding Ctl-H in a string.

a="^H^H"                  # Two Ctl-H's -- backspaces
                          # ctl-V ctl-H, using vi/vim
echo "abcdef"             # abcdef
echo
echo -n "abcdef$a "       # abcd f
#  Space at end  ^              ^  Backspaces twice.
echo
echo -n "abcdef$a"        # abcdef
#  No space at end               ^ Doesn't backspace (why?).
                          # Results may not be quite as expected.
echo; echo

# Constantin Hagemeier suggests trying:
# a=$'\010\010'
# a=$'\b\b'
# a=$'\x08\x08'
# But, this does not change the results.

########################################

# Now, try this.

rubout="^H^H^H^H^H"       # 5 x Ctl-H.

echo -n "12345678"
sleep 2
echo -n "$rubout"
sleep 2
```

◊ **Ctl-I**

Horizontal tab.

◊ **Ctl-J**

Newline (line feed). In a script, may also be expressed in octal notation -- '\012' or in hexadecimal -- '\x0a'.

◊ **Ctl-K**

Vertical tab.

When typing text on the console or in an *xterm* window, **Ctl-K** erases from the character under the cursor to end of line. Within a script, **Ctl-K** may behave differently, as in Lee Lee Maschmeyer's example, below.

◊ **Ctl-L**

Formfeed (clear the terminal screen). In a terminal, this has the same effect as the clear command. When sent to a printer, a **Ctl-L** causes an advance to end of the paper sheet.

◊ **Ctl-M**

Carriage return.

```
#!/bin/bash
# Thank you, Lee Maschmeyer, for this example.

read -n 1 -s -p \
$'Control-M leaves cursor at beginning of this line. Press Enter. \x0d'
           # Of course, '0d' is the hex equivalent of Control-M.
echo >&2   #  The '-s' makes anything typed silent,
           #+ so it is necessary to go to new line explicitly.

read -n 1 -s -p $'Control-J leaves cursor on next line. \x0a'
           #  '0a' is the hex equivalent of Control-J, linefeed.
echo >&2

###

read -n 1 -s -p $'And Control-K\x0bgoes straight down.'
echo >&2   #  Control-K is vertical tab.

# A better example of the effect of a vertical tab is:

var=$'\x0aThis is the bottom line\x0bThis is the top line\x0a'
echo "$var"
#  This works the same way as the above example. However:
echo "$var" | col
#  This causes the right end of the line to be higher than the left end.
#  It also explains why we started and ended with a line feed --
#+ to avoid a garbled screen.

# As Lee Maschmeyer explains:
# --------------------------
#  In the [first vertical tab example] . . . the vertical tab
#+ makes the printing go straight down without a carriage return.
#  This is true only on devices, such as the Linux console,
#+ that can't go "backward."
#  The real purpose of VT is to go straight UP, not down.
```

```
#  It can be used to print superscripts on a printer.
#  The col utility can be used to emulate the proper behavior of VT.

exit 0
```

◊ **Ctl-N**

Erases a line of text recalled from *history buffer* [23] (on the command-line).

◊ **Ctl-O**

Issues a *newline* (on the command-line).

◊ **Ctl-P**

Recalls last command from *history buffer* (on the command-line).

◊ **Ctl-Q**

Resume (**XON**).

This resumes `stdin` in a terminal.

◊ **Ctl-R**

Backwards search for text in *history buffer* (on the command-line).

◊ **Ctl-S**

Suspend (**XOFF**).

This freezes `stdin` in a terminal. (Use Ctl-Q to restore input.)

◊ **Ctl-T**

Reverses the position of the character the cursor is on with the previous character (on the command-line).

◊ **Ctl-U**

Erase a line of input, from the cursor backward to beginning of line. In some settings, **Ctl-U** erases the entire line of input, *regardless of cursor position*.

◊ **Ctl-V**

When inputting text, **Ctl-V** permits inserting control characters. For example, the following two are equivalent:

```
echo -e '\x0a'
echo <Ctl-V><Ctl-J>
```

Ctl-V is primarily useful from within a text editor.

◊ **Ctl-W**

When typing text on the console or in an xterm window, **Ctl-W** erases from the character under the cursor backwards to the first instance of whitespace. In some settings, **Ctl-W** erases backwards to first non-alphanumeric character.

◊ **Ctl-X**

In certain word processing programs, *Cuts* highlighted text and copies to *clipboard*.

◊ **Ctl-Y**

Pastes back text previously erased (with **`Ctl-U`** or **`Ctl-W`**).

◊ **`Ctl-Z`**

Pauses a foreground job.

Substitute operation in certain word processing applications.

`EOF` (end-of-file) character in the MSDOS filesystem.

Whitespace

functions as a separator between commands and/or variables. Whitespace consists of either *spaces*, *tabs*, *blank lines*, or any combination thereof. [24] In some contexts, such as variable assignment, whitespace is not permitted, and results in a syntax error.

Blank lines have no effect on the action of a script, and are therefore useful for visually separating functional sections.

$IFS, the special variable separating *fields* of input to certain commands. It defaults to whitespace.

> **`Definition:`** A *field* is a discrete chunk of data expressed as a string of consecutive characters. Separating each field from adjacent fields is either *whitespace* or some other designated character (often determined by the $IFS). In some contexts, a field may be called a *record*.

To preserve *whitespace* within a string or in a variable, use quoting.

UNIX filters can target and operate on *whitespace* using the POSIX character class [:space:].

Chapter 4. Introduction to Variables and Parameters

Variables are how programming and scripting languages represent data. A variable is nothing more than a *label*, a name assigned to a location or set of locations in computer memory holding an item of data.

Variables appear in arithmetic operations and manipulation of quantities, and in string parsing.

4.1. Variable Substitution

The *name* of a variable is a placeholder for its *value*, the data it holds. Referencing (retrieving) its value is called *variable substitution*.

$

Let us carefully distinguish between the *name* of a variable and its *value*. If **`variable1`** is the name of a variable, then **`$variable1`** is a reference to its *value*, the data item it contains. [25]

```
bash$ variable1=23


bash$ echo variable1
variable1

bash$ echo $variable1
23
```

The only times a variable appears "naked" -- without the $ prefix -- is when declared or assigned, when *unset*, when exported, in an arithmetic expression within double parentheses ((...)), or in the special case of a variable representing a signal (see Example 32-5). Assignment may be with an = (as in *`var1=27`*), in a read statement, and at the head of a loop (*`for var2 in 1 2 3`*).

Enclosing a referenced value in *double quotes* (" ... ") does not interfere with variable substitution. This is called *partial quoting*, sometimes referred to as "weak quoting." Using single quotes (' ... ') causes the variable name to be used literally, and no substitution will take place. This is *full quoting*, sometimes referred to as 'strong quoting.' See Chapter 5 for a detailed discussion.

Note that **`$variable`** is actually a simplified form of **`${variable}`**. In contexts where the **`$variable`** syntax causes an error, the longer form may work (see Section 10.2, below).

Example 4-1. Variable assignment and substitution

```
#!/bin/bash
# ex9.sh

# Variables: assignment and substitution

a=375
hello=$a
#   ^ ^
```

```
#-------------------------------------------------------------------------
# No space permitted on either side of = sign when initializing variables.
# What happens if there is a space?

#  "VARIABLE =value"
#           ^
#% Script tries to run "VARIABLE" command with one argument, "=value".

#  "VARIABLE= value"
#            ^
#% Script tries to run "value" command with
#+ the environmental variable "VARIABLE" set to "".
#-------------------------------------------------------------------------


echo hello    # hello
# Not a variable reference, just the string "hello" ...

echo $hello   # 375
#    ^          This *is* a variable reference.
echo ${hello} # 375
#               Likewise a variable reference, as above.

# Quoting . . .
echo "$hello"    # 375
echo "${hello}"  # 375

echo

hello="A B  C   D"
echo $hello   # A B C D
echo "$hello" # A B  C   D
# As we see, echo $hello   and   echo "$hello"   give different results.
# =======================================
# Quoting a variable preserves whitespace.
# =======================================

echo

echo '$hello'  # $hello
#    ^      ^
#  Variable referencing disabled (escaped) by single quotes,
#+ which causes the "$" to be interpreted literally.

# Notice the effect of different types of quoting.


hello=    # Setting it to a null value.
echo "\$hello (null value) = $hello"      # $hello (null value) =
#  Note that setting a variable to a null value is not the same as
#+ unsetting it, although the end result is the same (see below).

# --------------------------------------------------------------

#  It is permissible to set multiple variables on the same line,
#+ if separated by white space.
#  Caution, this may reduce legibility, and may not be portable.

var1=21  var2=22  var3=$V3
echo
echo "var1=$var1   var2=$var2   var3=$var3"
```

```
# May cause problems with legacy versions of "sh" . . .

# -------------------------------------------------------------

echo; echo

numbers="one two three"
#           ^   ^
other_numbers="1 2 3"
#               ^ ^
#  If there is whitespace embedded within a variable,
#+ then quotes are necessary.
#  other_numbers=1 2 3                  # Gives an error message.
echo "numbers = $numbers"
echo "other_numbers = $other_numbers"   # other_numbers = 1 2 3
#  Escaping the whitespace also works.
mixed_bag=2\ ---\ Whatever
#           ^    ^ Space after escape (\).

echo "$mixed_bag"         # 2 --- Whatever

echo; echo

echo "uninitialized_variable = $uninitialized_variable"
# Uninitialized variable has null value (no value at all!).
uninitialized_variable=   #  Declaring, but not initializing it --
                          #+ same as setting it to a null value, as above.
echo "uninitialized_variable = $uninitialized_variable"
                          # It still has a null value.

uninitialized_variable=23       # Set it.
unset uninitialized_variable    # Unset it.
echo "uninitialized_variable = $uninitialized_variable"
                                # uninitialized_variable =
                                # It still has a null value.
echo

exit 0
```

An uninitialized variable has a "null" value -- no assigned value at all (*not* zero!).

```
if [ -z "$unassigned" ]
then
  echo "\$unassigned is NULL."
fi     # $unassigned is NULL.
```

Using a variable before assigning a value to it may cause problems. It is nevertheless possible to perform arithmetic operations on an uninitialized variable.

```
echo "$uninitialized"                                # (blank line)
let "uninitialized += 5"                             # Add 5 to it.
echo "$uninitialized"                                # 5

#  Conclusion:
#  An uninitialized variable has no value,
#+ however it evaluates as 0 in an arithmetic operation.
```

See also Example 15-23.

4.2. Variable Assignment

=

the assignment operator (*no space before and after*)

Do not confuse this with = and -eq, which test, rather than assign!

Note that = can be either an *assignment* or a *test* operator, depending on context.

Example 4-2. Plain Variable Assignment

```
#!/bin/bash
# Naked variables

echo

# When is a variable "naked", i.e., lacking the '$' in front?
# When it is being assigned, rather than referenced.

# Assignment
a=879
echo "The value of \"a\" is $a."

# Assignment using 'let'
let a=16+5
echo "The value of \"a\" is now $a."

echo

# In a 'for' loop (really, a type of disguised assignment):
echo -n "Values of \"a\" in the loop are: "
for a in 7 8 9 11
do
  echo -n "$a "
done

echo
echo

# In a 'read' statement (also a type of assignment):
echo -n "Enter \"a\" "
read a
echo "The value of \"a\" is now $a."

echo

exit 0
```

Example 4-3. Variable Assignment, plain and fancy

```
#!/bin/bash

a=23              # Simple case
echo $a
b=$a
```

```
echo $b

# Now, getting a little bit fancier (command substitution).

a=`echo Hello!`   # Assigns result of 'echo' command to 'a' ...
echo $a
#  Note that including an exclamation mark (!) within a
#+ command substitution construct will not work from the command-line,
#+ since this triggers the Bash "history mechanism."
#  Inside a script, however, the history functions are disabled by default.

a=`ls -l`         # Assigns result of 'ls -l' command to 'a'
echo $a           # Unquoted, however, it removes tabs and newlines.
echo
echo "$a"         # The quoted variable preserves whitespace.
                  # (See the chapter on "Quoting.")

exit 0
```

Variable assignment using the *$(...)* mechanism (a newer method than backquotes). This is likewise a form of command substitution.

```
# From /etc/rc.d/rc.local
R=$(cat /etc/redhat-release)
arch=$(uname -m)
```

4.3. Bash Variables Are Untyped

Unlike many other programming languages, Bash does not segregate its variables by "type." Essentially, *Bash variables are character strings*, but, depending on context, Bash permits arithmetic operations and comparisons on variables. The determining factor is whether the value of a variable contains only digits.

Example 4-4. Integer or string?

```
#!/bin/bash
# int-or-string.sh

a=2334                   # Integer.
let "a += 1"
echo "a = $a "           # a = 2335
echo                     # Integer, still.


b=${a/23/BB}             # Substitute "BB" for "23".
                         # This transforms $b into a string.
echo "b = $b"            # b = BB35
declare -i b             # Declaring it an integer doesn't help.
echo "b = $b"            # b = BB35

let "b += 1"             # BB35 + 1
echo "b = $b"            # b = 1
echo                     # Bash sets the "integer value" of a string to 0.

c=BB34
echo "c = $c"            # c = BB34
```

```
d=${c/BB/23}              # Substitute "23" for "BB".
                          # This makes $d an integer.
echo "d = $d"             # d = 2334
let "d += 1"              # 2334 + 1
echo "d = $d"             # d = 2335
echo


# What about null variables?
e=''                      # ... Or e="" ... Or e=
echo "e = $e"             # e =
let "e += 1"              # Arithmetic operations allowed on a null variable?
echo "e = $e"             # e = 1
echo                      # Null variable transformed into an integer.

# What about undeclared variables?
echo "f = $f"             # f =
let "f += 1"              # Arithmetic operations allowed?
echo "f = $f"             # f = 1
echo                      # Undeclared variable transformed into an integer.
#
# However ...
let "f /= $undecl_var"    # Divide by zero?
#   let: f /= : syntax error: operand expected (error token is " ")
# Syntax error! Variable $undecl_var is not set to zero here!
#
# But still ...
let "f /= 0"
#   let: f /= 0: division by 0 (error token is "0")
# Expected behavior.


#  Bash (usually) sets the "integer value" of null to zero
#+ when performing an arithmetic operation.
#  But, don't try this at home, folks!
#  It's undocumented and probably non-portable behavior.


# Conclusion: Variables in Bash are untyped,
#+ with all attendant consequences.

exit $?
```

Untyped variables are both a blessing and a curse. They permit more flexibility in scripting and make it easier to grind out lines of code (and give you enough rope to hang yourself!). However, they likewise permit subtle errors to creep in and encourage sloppy programming habits.

To lighten the burden of keeping track of variable types in a script, Bash *does* permit declaring variables.

4.4. Special Variable Types

`Local variables`
: Variables visible only within a code block or function (see also local variables in functions)

`Environmental variables`
: Variables that affect the behavior of the shell and user interface

> In a more general context, each process has an "environment", that is, a group of variables that the process may reference. In this sense, the shell behaves like any other

process.

Every time a shell starts, it creates shell variables that correspond to its own environmental variables. Updating or adding new environmental variables causes the shell to update its environment, and all the shell's *child processes* (the commands it executes) inherit this environment.

The space allotted to the environment is limited. Creating too many environmental variables or ones that use up excessive space may cause problems.

```
bash$ eval "`seq 10000 | sed -e 's/.*/export var&=ZZZZZZZZZZZZZZ/'`"

bash$ du
bash: /usr/bin/du: Argument list too long
```

Note: this "error" has been fixed, as of kernel version 2.6.23.

(Thank you, Stéphane Chazelas for the clarification, and for providing the above example.)

If a script sets environmental variables, they need to be "exported," that is, reported to the *environment* local to the script. This is the function of the export command.

A script can **export** variables only to child processes, that is, only to commands or processes which that particular script initiates. A script invoked from the command-line `cannot` export variables back to the command-line environment. *Child processes cannot export variables back to the parent processes that spawned them.*

`Definition:` A *child process* is a subprocess launched by another process, its parent.

`Positional parameters`

Arguments passed to the script from the command line [26] : `$0`, `$1`, `$2`, `$3 . . .`

`$0` is the name of the script itself, `$1` is the first argument, `$2` the second, `$3` the third, and so forth. [27] After `$9`, the arguments must be enclosed in brackets, for example, `${10}`, `${11}`, `${12}`.

The special variables $* and $@ denote *all* the positional parameters.

Example 4-5. Positional Parameters

```
#!/bin/bash

# Call this script with at least 10 parameters, for example
# ./scriptname 1 2 3 4 5 6 7 8 9 10
MINPARAMS=10

echo

echo "The name of this script is \"$0\"."
# Adds ./ for current directory
echo "The name of this script is \"`basename $0`\"."
```

```
# Strips out path name info (see 'basename')

echo

if [ -n "$1" ]              # Tested variable is quoted.
then
 echo "Parameter #1 is $1"  # Need quotes to escape #
fi

if [ -n "$2" ]
then
 echo "Parameter #2 is $2"
fi

if [ -n "$3" ]
then
 echo "Parameter #3 is $3"
fi

# ...


if [ -n "${10}" ]  # Parameters > $9 must be enclosed in {brackets}.
then
 echo "Parameter #10 is ${10}"
fi

echo "-----------------------------------"
echo "All the command-line parameters are: "$*""

if [ $# -lt "$MINPARAMS" ]
then
  echo
  echo "This script needs at least $MINPARAMS command-line arguments!"
fi

echo

exit 0
```

Bracket notation for positional parameters leads to a fairly simple way of referencing the *last* argument passed to a script on the command-line. This also requires indirect referencing.

```
args=$#           # Number of args passed.
lastarg=${!args}
# Note: This is an *indirect reference* to $args ...


# Or:       lastarg=${!#}             (Thanks, Chris Monson.)
# This is an *indirect reference* to the $# variable.
# Note that lastarg=${!$#} doesn't work.
```

Some scripts can perform different operations, depending on which name they are invoked with. For this to work, the script needs to check `$0`, the name it was invoked by. [28] There must also exist symbolic links to all the alternate names of the script. See Example 16-2.

If a script expects a command-line parameter but is invoked without one, this may cause a *null variable assignment*, generally an undesirable result. One way to prevent this is to append an extra character to both sides of the assignment statement using the expected positional parameter.

```
variable1_=$1_  # Rather than variable1=$1
# This will prevent an error, even if positional parameter is absent.

critical_argument01=$variable1_

# The extra character can be stripped off later, like so.
variable1=${variable1_/_/}
# Side effects only if $variable1_ begins with an underscore.
# This uses one of the parameter substitution templates discussed later.
# (Leaving out the replacement pattern results in a deletion.)

#  A more straightforward way of dealing with this is
#+ to simply test whether expected positional parameters have been passed.
if [ -z $1 ]
then
  exit $E_MISSING_POS_PARAM
fi


#  However, as Fabian Kreutz points out,
#+ the above method may have unexpected side-effects.
#  A better method is parameter substitution:
#         ${1:-$DefaultVal}
#  See the "Parameter Substition" section
#+ in the "Variables Revisited" chapter.
```

Example 4-6. *wh*, *whois* domain name lookup

```
#!/bin/bash
# ex18.sh

# Does a 'whois domain-name' lookup on any of 3 alternate servers:
#                     ripe.net, cw.net, radb.net

# Place this script -- renamed 'wh' -- in /usr/local/bin

# Requires symbolic links:
# ln -s /usr/local/bin/wh /usr/local/bin/wh-ripe
# ln -s /usr/local/bin/wh /usr/local/bin/wh-apnic
# ln -s /usr/local/bin/wh /usr/local/bin/wh-tucows

E_NOARGS=75


if [ -z "$1" ]
then
  echo "Usage: `basename $0` [domain-name]"
  exit $E_NOARGS
fi

# Check script name and call proper server.
case `basename $0` in    # Or:    case ${0##*/} in
    "wh"       ) whois $1@whois.tucows.com;;
    "wh-ripe"  ) whois $1@whois.ripe.net;;
    "wh-apnic" ) whois $1@whois.apnic.net;;
    "wh-cw"    ) whois $1@whois.cw.net;;
    *          ) echo "Usage: `basename $0` [domain-name]";;
```

```
esac

exit $?
```

The **shift** command reassigns the positional parameters, in effect shifting them to the left one notch.

`$1` <--- `$2`, `$2` <--- `$3`, `$3` <--- `$4`, etc.

The old `$1` disappears, but *`$0` (the script name) does not change*. If you use a large number of positional parameters to a script, **shift** lets you access those past `10`, although {bracket} notation also permits this.

Example 4-7. Using *shift*

```
#!/bin/bash
# shft.sh: Using 'shift' to step through all the positional parameters.

#  Name this script something like shft.sh,
#+ and invoke it with some parameters.
#+ For example:
#             sh shft.sh a b c def 83 barndoor

until [ -z "$1" ]  # Until all parameters used up . . .
do
  echo -n "$1 "
  shift
done

echo               # Extra linefeed.

# But, what happens to the "used-up" parameters?
echo "$2"
#  Nothing echoes!
#  When $2 shifts into $1 (and there is no $3 to shift into $2)
#+ then $2 remains empty.
#  So, it is not a parameter *copy*, but a *move*.

exit

#  See also the echo-params.sh script for a "shiftless"
#+ alternative method of stepping through the positional params.
```

The **shift** command can take a numerical parameter indicating how many positions to shift.

```
#!/bin/bash
# shift-past.sh

shift 3    # Shift 3 positions.
#  n=3; shift $n
#  Has the same effect.

echo "$1"

exit 0

# ======================== #
```

```
$ sh shift-past.sh 1 2 3 4 5
4

#  However, as Eleni Fragkiadaki, points out,
#+ attempting a 'shift' past the number of
#+ positional parameters ($#) returns an exit status of 1,
#+ and the positional parameters themselves do not change.
#  This means possibly getting stuck in an endless loop. . . .
#  For example:
#      until [ -z "$1" ]
#      do
#         echo -n "$1 "
#         shift 20    #  If less than 20 pos params,
#      done           #+ then loop never ends!
#
# When in doubt, add a sanity check. . . .
#           shift 20 || break
#                    ^^^^^^^^
```

The **shift** command works in a similar fashion on parameters passed to a function. See Example 36-18.

Chapter 5. Quoting

Quoting means just that, bracketing a string in quotes. This has the effect of protecting special characters in the string from reinterpretation or expansion by the shell or shell script. (A character is "special" if it has an interpretation other than its literal meaning. For example, the asterisk * represents a *wild card* character in globbing and Regular Expressions).

```
bash$ ls -l [Vv]*
-rw-rw-r--    1 bozo  bozo       324 Apr  2 15:05 VIEWDATA.BAT
 -rw-rw-r--    1 bozo  bozo       507 May  4 14:25 vartrace.sh
 -rw-rw-r--    1 bozo  bozo       539 Apr 14 17:11 viewdata.sh

bash$ ls -l '[Vv]*'
ls: [Vv]*: No such file or directory
```

In everyday speech or writing, when we "quote" a phrase, we set it apart and give it special meaning. In a Bash script, when we *quote* a string, we set it apart and protect its *literal* meaning.

Certain programs and utilities reinterpret or expand special characters in a quoted string. An important use of quoting is protecting a command-line parameter from the shell, but still letting the calling program expand it.

```
bash$ grep '[Ff]irst' *.txt
file1.txt:This is the first line of file1.txt.
 file2.txt:This is the First line of file2.txt.
```

Note that the unquoted **`grep [Ff]irst *.txt`** works under the Bash shell. [29]

Quoting can also suppress echo's "appetite" for newlines.

```
bash$ echo $(ls -l)
total 8 -rw-rw-r-- 1 bo bo 13 Aug 21 12:57 t.sh -rw-rw-r-- 1 bo bo 78 Aug 21 12:57 u.sh


bash$ echo "$(ls -l)"
total 8
 -rw-rw-r--  1 bo bo  13 Aug 21 12:57 t.sh
 -rw-rw-r--  1 bo bo  78 Aug 21 12:57 u.sh
```

5.1. Quoting Variables

When referencing a variable, it is generally advisable to enclose its name in double quotes. This prevents reinterpretation of all special characters within the quoted string -- except $, ` (backquote), and \ (escape). [30] Keeping $ as a special character within double quotes permits referencing a quoted variable (*`"$variable"`*), that is, replacing the variable with its value (see Example 4-1, above).

Use double quotes to prevent word splitting. [31] An argument enclosed in double quotes presents itself as a single word, even if it contains whitespace separators.

```
List="one two three"
```

```
for a in $List     # Splits the variable in parts at whitespace.
do
  echo "$a"
done
# one
# two
# three

echo "---"

for a in "$List"   # Preserves whitespace in a single variable.
do #     ^     ^
  echo "$a"
done
# one two three
```

A more elaborate example:

```
variable1="a variable containing five words"
COMMAND This is $variable1    # Executes COMMAND with 7 arguments:
# "This" "is" "a" "variable" "containing" "five" "words"

COMMAND "This is $variable1"  # Executes COMMAND with 1 argument:
# "This is a variable containing five words"


variable2=""    # Empty.

COMMAND $variable2 $variable2 $variable2
                # Executes COMMAND with no arguments.
COMMAND "$variable2" "$variable2" "$variable2"
                # Executes COMMAND with 3 empty arguments.
COMMAND "$variable2 $variable2 $variable2"
                # Executes COMMAND with 1 argument (2 spaces).

# Thanks, Stéphane Chazelas.
```

Enclosing the arguments to an **echo** statement in double quotes is necessary only when word splitting or preservation of whitespace is an issue.

Example 5-1. Echoing Weird Variables

```
#!/bin/bash
# weirdvars.sh: Echoing weird variables.

echo

var="'(]\\{}\$\""
echo $var        # '(]\{}$"
echo "$var"      # '(]\{}$"     Doesn't make a difference.

echo

IFS='\'
echo $var        # '(] {}$"     \ converted to space. Why?
echo "$var"      # '(]\{}$"

# Examples above supplied by Stephane Chazelas.
```

```
echo

var2="\\\\\""
echo $var2       #   "
echo "$var2"     # \\"
echo
# But ... var2="\\\\"" is illegal. Why?
var3='\\\\'
echo "$var3"     # \\\\
# Strong quoting works, though.


# ************************************************************ #
# As the first example above shows, nesting quotes is permitted.

echo "$(echo '"')"           # "
#    ^           ^


# At times this comes in useful.

var1="Two bits"
echo "\$var1 = "$var1""      # $var1 = Two bits
#    ^                ^

# Or, as Chris Hiestand points out ...

if [[ "$(du "$My_File1")" -gt "$(du "$My_File2")" ]]
#     ^     ^         ^ ^     ^     ^         ^ ^
then
  ...
fi
# ************************************************************ #
```

Single quotes (' ') operate similarly to double quotes, but do not permit referencing variables, since the special meaning of $ is turned off. Within single quotes, *every* special character except ' gets interpreted literally. Consider single quotes ("full quoting") to be a stricter method of quoting than double quotes ("partial quoting").

Since even the escape character (\) gets a literal interpretation within single quotes, trying to enclose a single quote within single quotes will not yield the expected result.

```
echo "Why can't I write 's between single quotes"

echo

# The roundabout method.
echo 'Why can'\''t I write '"'"'s between single quotes'
#    |-------|  |----------|   |-----------------------|
# Three single-quoted strings, with escaped and quoted single quotes between.

# This example courtesy of Stéphane Chazelas.
```

5.2. Escaping

Escaping is a method of quoting single characters. The escape (\) preceding a character tells the shell to interpret that character literally.

With certain commands and utilities, such as echo and sed, escaping a character may have the opposite effect - it can toggle on a special meaning for that character.

Special meanings of certain escaped characters

used with **echo** and **sed**

\n
: means newline

\r
: means return

\t
: means tab

\v
: means vertical tab

\b
: means backspace

\a
: means *alert* (beep or flash)

\0xx
: translates to the octal ASCII equivalent of `0nn`, where `nn` is a string of digits

The **`$' ... '`** quoted string-expansion construct is a mechanism that uses escaped octal or hex values to assign ASCII characters to variables, e.g., **quote=$'\042'**.

Example 5-2. Escaped Characters

```
#!/bin/bash
# escaped.sh: escaped characters

#############################################################
### First, let's show some basic escaped-character usage. ###
#############################################################

# Escaping a newline.
# ------------------

echo ""

echo "This will print
as two lines."
# This will print
# as two lines.

echo "This will print \
as one line."
# This will print as one line.

echo; echo

echo "============="


echo "\v\v\v\v"      # Prints \v\v\v\v literally.
# Use the -e option with 'echo' to print escaped characters.
```

```
echo "============="
echo "VERTICAL TABS"
echo -e "\v\v\v\v"   # Prints 4 vertical tabs.
echo "=============="

echo "QUOTATION MARK"
echo -e "\042"       # Prints " (quote, octal ASCII character 42).
echo "=============="



# The $'\X' construct makes the -e option unnecessary.

echo; echo "NEWLINE and (maybe) BEEP"
echo $'\n'           # Newline.
echo $'\a'           # Alert (beep).
                     # May only flash, not beep, depending on terminal.

# We have seen $'\nnn" string expansion, and now . . .

# =================================================================== #
# Version 2 of Bash introduced the $'\nnn' string expansion construct.
# =================================================================== #

echo "Introducing the \$\' ... \' string-expansion construct . . . "
echo ". . . featuring more quotation marks."

echo $'\t \042 \t'   # Quote (") framed by tabs.
# Note that  '\nnn' is an octal value.

# It also works with hexadecimal values, in an $'\xhhh' construct.
echo $'\t \x22 \t'  # Quote (") framed by tabs.
# Thank you, Greg Keraunen, for pointing this out.
# Earlier Bash versions allowed '\x022'.

echo


# Assigning ASCII characters to a variable.
# ----------------------------------------
quote=$'\042'        # " assigned to a variable.
echo "$quote Quoted string $quote and this lies outside the quotes."

echo

# Concatenating ASCII chars in a variable.
triple_underline=$'\137\137\137'  # 137 is octal ASCII code for '_'.
echo "$triple_underline UNDERLINE $triple_underline"

echo

ABC=$'\101\102\103\010'           # 101, 102, 103 are octal A, B, C.
echo $ABC

echo

escape=$'\033'                    # 033 is octal for escape.
echo "\"escape\" echoes as $escape"
#                                   no visible output.

echo
```

```
exit 0
```

A more elaborate example:

Example 5-3. Detecting key-presses

```
#!/bin/bash
# Author: Sigurd Solaas, 20 Apr 2011
# Used in ABS Guide with permission.
# Requires version 4.2+ of Bash.

key="no value yet"
while true; do
  clear
  echo "Bash Extra Keys Demo. Keys to try:"
  echo
  echo "* Insert, Delete, Home, End, Page_Up and Page_Down"
  echo "* The four arrow keys"
  echo "* Tab, enter, escape, and space key"
  echo "* The letter and number keys, etc."
  echo
  echo "    d = show date/time"
  echo "    q = quit"
  echo "================================"
  echo

 # Convert the separate home-key to home-key_num_7:
 if [ "$key" = $'\x1b\x4f\x48' ]; then
  key=$'\x1b\x5b\x31\x7e'
  #   Quoted string-expansion construct.
 fi

 # Convert the separate end-key to end-key_num_1.
 if [ "$key" = $'\x1b\x4f\x46' ]; then
  key=$'\x1b\x5b\x34\x7e'
 fi

 case "$key" in
  $'\x1b\x5b\x32\x7e')  # Insert
   echo Insert Key
  ;;
  $'\x1b\x5b\x33\x7e')  # Delete
   echo Delete Key
  ;;
  $'\x1b\x5b\x31\x7e')  # Home_key_num_7
   echo Home Key
  ;;
  $'\x1b\x5b\x34\x7e')  # End_key_num_1
   echo End Key
  ;;
  $'\x1b\x5b\x35\x7e')  # Page_Up
   echo Page_Up
  ;;
  $'\x1b\x5b\x36\x7e')  # Page_Down
   echo Page_Down
  ;;
  $'\x1b\x5b\x41')  # Up_arrow
   echo Up arrow
  ;;
  $'\x1b\x5b\x42')  # Down_arrow
   echo Down arrow
```

```
  ;;
  $'\x1b\x5b\x43')  # Right_arrow
   echo Right arrow
  ;;
  $'\x1b\x5b\x44')  # Left_arrow
   echo Left arrow
  ;;
  $'\x09')  # Tab
   echo Tab Key
  ;;
  $'\x0a')  # Enter
   echo Enter Key
  ;;
  $'\x1b')  # Escape
   echo Escape Key
  ;;
  $'\x20')  # Space
   echo Space Key
  ;;
  d)
   date
  ;;
  q)
  echo Time to quit...
  echo
  exit 0
  ;;
  *)
   echo You pressed: \'"$key"\'
  ;;
 esac

 echo
 echo "================================"

 unset K1 K2 K3
 read -s -N1 -p "Press a key: "
 K1="$REPLY"
 read -s -N2 -t 0.001
 K2="$REPLY"
 read -s -N1 -t 0.001
 K3="$REPLY"
 key="$K1$K2$K3"

done

exit $?
```

See also Example 37-1.

\"

gives the quote its literal meaning

```
echo "Hello"                     # Hello
echo "\"Hello\" ... he said."    # "Hello" ... he said.
```

\$

gives the dollar sign its literal meaning (variable name following \$ will not be referenced)

```
echo "\$variable01"           # $variable01
echo "The book cost \$7.98."  # The book cost $7.98.
```

\\

gives the backslash its literal meaning

```
echo "\\"  # Results in \

# Whereas . . .

echo "\"   # Invokes secondary prompt from the command-line.
           # In a script, gives an error message.

# However . . .

echo '\'   # Results in \
```

The behavior of \ depends on whether it is escaped, strong-quoted, weak-quoted, or appearing within command substitution or a here document.

```
                      #  Simple escaping and quoting
echo \z               #  z
echo \\z              # \z
echo '\z'             # \z
echo '\\z'            # \\z
echo "\z"             # \z
echo "\\z"            # \z

                      #  Command substitution
echo `echo \z`        #  z
echo `echo \\z`       #  z
echo `echo \\\z`      # \z
echo `echo \\\\z`     # \z
echo `echo \\\\\\z`   # \z
echo `echo \\\\\\\z`  # \\z
echo `echo "\z"`      # \z
echo `echo "\\z"`     # \z

                      # Here document
cat <<EOF
\z
EOF                   # \z

cat <<EOF
\\z
EOF                   # \z

# These examples supplied by Stéphane Chazelas.
```

Elements of a string assigned to a variable may be escaped, but the escape character alone may not be assigned to a variable.

```
variable=\
echo "$variable"
# Will not work - gives an error message:
# test.sh: : command not found
# A "naked" escape cannot safely be assigned to a variable.
#
#  What actually happens here is that the "\" escapes the newline and
#+ the effect is        variable=echo "$variable"
#+                      invalid variable assignment

variable=\
23skidoo
```

```
echo "$variable"        #  23skidoo
                        #  This works, since the second line
                        #+ is a valid variable assignment.

variable=\
#        \^    escape followed by space
echo "$variable"        # space

variable=\\
echo "$variable"        # \

variable=\\\
echo "$variable"
# Will not work - gives an error message:
# test.sh: \: command not found
#
#  First escape escapes second one, but the third one is left "naked",
#+ with same result as first instance, above.

variable=\\\\
echo "$variable"        # \\
                        # Second and fourth escapes escaped.
                        # This is o.k.
```

Escaping a space can prevent word splitting in a command's argument list.

```
file_list="/bin/cat /bin/gzip /bin/more /usr/bin/less /usr/bin/emacs-20.7"
# List of files as argument(s) to a command.

# Add two files to the list, and list all.
ls -l /usr/X11R6/bin/xsetroot /sbin/dump $file_list

echo "-------------------------------------------------------------------------"

# What happens if we escape a couple of spaces?
ls -l /usr/X11R6/bin/xsetroot\ /sbin/dump\ $file_list
# Error: the first three files concatenated into a single argument to 'ls -l'
#        because the two escaped spaces prevent argument (word) splitting.
```

The escape also provides a means of writing a multi-line command. Normally, each separate line constitutes a different command, but an escape at the end of a line *escapes the newline character*, and the command sequence continues on to the next line.

```
(cd /source/directory && tar cf - . ) | \
(cd /dest/directory && tar xpvf -)
# Repeating Alan Cox's directory tree copy command,
# but split into two lines for increased legibility.

# As an alternative:
tar cf - -C /source/directory . |
tar xpvf - -C /dest/directory
# See note below.
# (Thanks, Stéphane Chazelas.)
```

If a script line ends with a |, a pipe character, then a \, an escape, is not strictly necessary. It is, however, good programming practice to always escape the end of a line of code that continues to the following line.

```
echo "foo
```

```
bar"
#foo
#bar

echo

echo 'foo
bar'    # No difference yet.
#foo
#bar

echo

echo foo\
bar     # Newline escaped.
#foobar

echo

echo "foo\
bar"     # Same here, as \ still interpreted as escape within weak quotes.
#foobar

echo

echo 'foo\
bar'     # Escape character \ taken literally because of strong quoting.
#foo\
#bar

# Examples suggested by Stéphane Chazelas.
```

Chapter 6. Exit and Exit Status

... there are dark corners in the Bourne shell, and people use all of them.

--Chet Ramey

The **exit** command terminates a script, just as in a **C** program. It can also return a value, which is available to the script's parent process.

Every command returns an *exit status* (sometimes referred to as a *return status* or *exit code*). A successful command returns a 0, while an unsuccessful one returns a non-zero value that usually can be interpreted as an *error code*. Well-behaved UNIX commands, programs, and utilities return a 0 exit code upon successful completion, though there are some exceptions.

Likewise, functions within a script and the script itself return an exit status. The last command executed in the function or script determines the exit status. Within a script, an **`exit nnn`** command may be used to deliver an *`nnn`* exit status to the shell (*`nnn`* must be an integer in the 0 - 255 range).

When a script ends with an **exit** that has no parameter, the exit status of the script is the exit status of the last command executed in the script (previous to the **exit**).

```
#!/bin/bash

COMMAND_1

. . .

COMMAND_LAST

# Will exit with status of last command.

exit
```

The equivalent of a bare **exit** is **exit $?** or even just omitting the **exit**.

```
#!/bin/bash

COMMAND_1

. . .

COMMAND_LAST

# Will exit with status of last command.

exit $?
```

```
#!/bin/bash

COMMAND1

. . .

COMMAND_LAST
```

```
# Will exit with status of last command.
```

`$?` reads the exit status of the last command executed. After a function returns, `$?` gives the exit status of the last command executed in the function. This is Bash's way of giving functions a "return value." [32]

Following the execution of a pipe, a `$?` gives the exit status of the last command executed.

After a script terminates, a `$?` from the command-line gives the exit status of the script, that is, the last command executed in the script, which is, by convention, **0** on success or an integer in the range 1 - 255 on error.

Example 6-1. exit / exit status

```
#!/bin/bash

echo hello
echo $?    # Exit status 0 returned because command executed successfully.

lskdf      # Unrecognized command.
echo $?    # Non-zero exit status returned -- command failed to execute.

echo

exit 113   # Will return 113 to shell.
           # To verify this, type "echo $?" after script terminates.

#  By convention, an 'exit 0' indicates success,
#+ while a non-zero exit value means an error or anomalous condition.
#  See the "Exit Codes With Special Meanings" appendix.
```

$? is especially useful for testing the result of a command in a script (see Example 16-35 and Example 16-20).

The !, the *logical not* qualifier, reverses the outcome of a test or command, and this affects its exit status.

Example 6-2. Negating a condition using !

```
true    # The "true" builtin.
echo "exit status of \"true\" = $?"     # 0

! true
echo "exit status of \"! true\" = $?"   # 1
# Note that the "!" needs a space between it and the command.
#    !true   leads to a "command not found" error
#
# The '!' operator prefixing a command invokes the Bash history mechanism.

true
!true
# No error this time, but no negation either.
# It just repeats the previous command (true).


# =========================================================== #
# Preceding a _pipe_ with ! inverts the exit status returned.
ls | bogus_command     # bash: bogus_command: command not found
echo $?                # 127
```

```
! ls | bogus_command   # bash: bogus_command: command not found
echo $?                # 0
# Note that the ! does not change the execution of the pipe.
# Only the exit status changes.
# =========================================================== #

# Thanks, Stéphane Chazelas and Kristopher Newsome.
```

Certain exit status codes have reserved meanings and should not be user-specified in a script.

Chapter 7. Tests

Every reasonably complete programming language can test for a condition, then act according to the result of the test. Bash has the test command, various bracket and parenthesis operators, and the **if/then** construct.

7.1. Test Constructs

- An **if/then** construct tests whether the exit status of a list of commands is 0 (since 0 means "success" by UNIX convention), and if so, executes one or more commands.
- There exists a dedicated command called **[** (left bracket special character). It is a synonym for **test**, and a builtin for efficiency reasons. This command considers its arguments as comparison expressions or file tests and returns an exit status corresponding to the result of the comparison (0 for true, 1 for false).
- With version 2.02, Bash introduced the [[...]] *extended test command*, which performs comparisons in a manner more familiar to programmers from other languages. Note that **[[** is a keyword, not a command.

  Bash sees **`[[ $a -lt $b ]]`** as a single element, which returns an exit status.
-

  The ((...)) and let ... constructs return an exit status, *according to whether the arithmetic expressions they evaluate expand to a non-zero value*. These arithmetic-expansion constructs may therefore be used to perform arithmetic comparisons.

```
(( 0 && 1 ))                 # Logical AND
echo $?     # 1     ***
# And so ...
let "num = (( 0 && 1 ))"
echo $num   # 0
# But ...
let "num = (( 0 && 1 ))"
echo $?     # 1     ***


(( 200 || 11 ))              # Logical OR
echo $?     # 0     ***
# ...
let "num = (( 200 || 11 ))"
echo $num   # 1
let "num = (( 200 || 11 ))"
echo $?     # 0     ***


(( 200 | 11 ))               # Bitwise OR
echo $?                      # 0     ***
# ...
let "num = (( 200 | 11 ))"
echo $num                    # 203
let "num = (( 200 | 11 ))"
echo $?                      # 0     ***

# The "let" construct returns the same exit status
#+ as the double-parentheses arithmetic expansion.
```

Again, note that the *exit status* of an arithmetic expression is *not* an error value.

```
var=-2 && (( var+=2 ))
echo $?                     # 1

var=-2 && (( var+=2 )) && echo $var
                            # Will not echo $var!
```

- An **if** can test any command, not just conditions enclosed within brackets.

```
if cmp a b &> /dev/null  # Suppress output.
then echo "Files a and b are identical."
else echo "Files a and b differ."
fi

# The very useful "if-grep" construct:
# -----------------------------------
if grep -q Bash file
  then echo "File contains at least one occurrence of Bash."
fi

word=Linux
letter_sequence=inu
if echo "$word" | grep -q "$letter_sequence"
# The "-q" option to grep suppresses output.
then
  echo "$letter_sequence found in $word"
else
  echo "$letter_sequence not found in $word"
fi


if COMMAND_WHOSE_EXIT_STATUS_IS_0_UNLESS_ERROR_OCCURRED
  then echo "Command succeeded."
  else echo "Command failed."
fi
```

- *These last two examples courtesy of Stéphane Chazelas.*

Example 7-1. What is truth?

```
#!/bin/bash

#  Tip:
#  If you're unsure how a certain condition might evaluate,
#+ test it in an if-test.

echo

echo "Testing \"0\""
if [ 0 ]      # zero
then
  echo "0 is true."
else          # Or else ...
  echo "0 is false."
fi            # 0 is true.

echo
```

```
echo "Testing \"1\""
if [ 1 ]      # one
then
  echo "1 is true."
else
  echo "1 is false."
fi            # 1 is true.

echo

echo "Testing \"-1\""
if [ -1 ]     # minus one
then
  echo "-1 is true."
else
  echo "-1 is false."
fi            # -1 is true.

echo

echo "Testing \"NULL\""
if [ ]        # NULL (empty condition)
then
  echo "NULL is true."
else
  echo "NULL is false."
fi            # NULL is false.

echo

echo "Testing \"xyz\""
if [ xyz ]    # string
then
  echo "Random string is true."
else
  echo "Random string is false."
fi            # Random string is true.

echo

echo "Testing \"\$xyz\""
if [ $xyz ]   # Tests if $xyz is null, but...
              # it's only an uninitialized variable.
then
  echo "Uninitialized variable is true."
else
  echo "Uninitialized variable is false."
fi            # Uninitialized variable is false.

echo

echo "Testing \"-n \$xyz\""
if [ -n "$xyz" ]            # More pedantically correct.
then
  echo "Uninitialized variable is true."
else
  echo "Uninitialized variable is false."
fi            # Uninitialized variable is false.

echo
```

```
xyz=          # Initialized, but set to null value.

echo "Testing \"-n \$xyz\""
if [ -n "$xyz" ]
then
  echo "Null variable is true."
else
  echo "Null variable is false."
fi            # Null variable is false.


echo


# When is "false" true?

echo "Testing \"false\""
if [ "false" ]              #  It seems that "false" is just a string ...
then
  echo "\"false\" is true." #+ and it tests true.
else
  echo "\"false\" is false."
fi            # "false" is true.

echo

echo "Testing \"\$false\""  # Again, uninitialized variable.
if [ "$false" ]
then
  echo "\"\$false\" is true."
else
  echo "\"\$false\" is false."
fi            # "$false" is false.
              # Now, we get the expected result.

#  What would happen if we tested the uninitialized variable "$true"?

echo

exit 0
```

Exercise. Explain the behavior of Example 7-1, above.

```
if [ condition-true ]
then
   command 1
   command 2
   ...
else  # Or else ...
      # Adds default code block executing if original condition tests false.
   command 3
   command 4
   ...
fi
```

When *if* and *then* are on same line in a condition test, a semicolon must terminate the *if* statement. Both *if* and *then* are keywords. Keywords (or commands) begin statements, and before a new statement on the same line begins, the old one must terminate.

```
if [ -x "$filename" ]; then
```

Else if and elif

elif
: **`elif`** is a contraction for *else if*. The effect is to nest an inner if/then construct within an outer one.

```
if [ condition1 ]
then
   command1
   command2
   command3
elif [ condition2 ]
# Same as else if
then
   command4
   command5
else
   default-command
fi
```

The **`if test condition-true`** construct is the exact equivalent of **`if [ condition-true ]`**. As it happens, the left bracket, **[** , is a *token* [33] which invokes the **test** command. The closing right bracket, **]** , in an if/test should not therefore be strictly necessary, however newer versions of Bash require it.

The **test** command is a Bash builtin which tests file types and compares strings. Therefore, in a Bash script, **test** does *not* call the external `/usr/bin/test` binary, which is part of the *sh-utils* package. Likewise, **[** does not call `/usr/bin/[`, which is linked to `/usr/bin/test`.

```
bash$ type test
test is a shell builtin
bash$ type '['
[ is a shell builtin
bash$ type '[['
[[ is a shell keyword
bash$ type ']]'
]] is a shell keyword
bash$ type ']'
bash: type: ]: not found
```

If, for some reason, you wish to use `/usr/bin/test` in a Bash script, then specify it by full pathname.

Example 7-2. Equivalence of *test*, `/usr/bin/test`, [ ], and `/usr/bin/[`

```
#!/bin/bash

echo

if test -z "$1"
then
  echo "No command-line arguments."
else
  echo "First command-line argument is $1."
```

```
fi

echo

if /usr/bin/test -z "$1"      # Equivalent to "test" builtin.
#  ^^^^^^^^^^^^^              # Specifying full pathname.
then
  echo "No command-line arguments."
else
  echo "First command-line argument is $1."
fi

echo

if [ -z "$1" ]                # Functionally identical to above code blocks.
#   if [ -z "$1"                should work, but...
#+  Bash responds to a missing close-bracket with an error message.
then
  echo "No command-line arguments."
else
  echo "First command-line argument is $1."
fi

echo


if /usr/bin/[ -z "$1" ]       # Again, functionally identical to above.
# if /usr/bin/[ -z "$1"       # Works, but gives an error message.
#                             # Note:
#                               This has been fixed in Bash, version 3.x.
then
  echo "No command-line arguments."
else
  echo "First command-line argument is $1."
fi

echo

exit 0
```

The [[ ]] construct is the more versatile Bash version of [ ]. This is the *extended test command*, adopted from *ksh88*.

* * *

No filename expansion or word splitting takes place between [[and]], but there is parameter expansion and command substitution.

```
file=/etc/passwd

if [[ -e $file ]]
then
  echo "Password file exists."
fi
```

Using the **[[...]]** test construct, rather than **[...]** can prevent many logic errors in scripts. For example, the &&, ||, <, and > operators work within a [[ ]] test, despite giving an error within a [ ] construct.

Arithmetic evaluation of octal / hexadecimal constants takes place automatically within a [[...]] construct.

```
# [[ Octal and hexadecimal evaluation ]]
# Thank you, Moritz Gronbach, for pointing this out.


decimal=15
octal=017   # = 15 (decimal)
hex=0x0f    # = 15 (decimal)

if [ "$decimal" -eq "$octal" ]
then
  echo "$decimal equals $octal"
else
  echo "$decimal is not equal to $octal"       # 15 is not equal to 017
fi      # Doesn't evaluate within [ single brackets ]!


if [[ "$decimal" -eq "$octal" ]]
then
  echo "$decimal equals $octal"                # 15 equals 017
else
  echo "$decimal is not equal to $octal"
fi      # Evaluates within [[ double brackets ]]!

if [[ "$decimal" -eq "$hex" ]]
then
  echo "$decimal equals $hex"                  # 15 equals 0x0f
else
  echo "$decimal is not equal to $hex"
fi      # [[ $hexadecimal ]] also evaluates!
```

Following an **if**, neither the **test** command nor the test brackets ([ ] or [[ ]]) are strictly necessary.

```
dir=/home/bozo

if cd "$dir" 2>/dev/null; then   # "2>/dev/null" hides error message.
  echo "Now in $dir."
else
  echo "Can't change to $dir."
fi
```

The "if COMMAND" construct returns the exit status of COMMAND.

Similarly, a condition within test brackets may stand alone without an **if**, when used in combination with a list construct.

```
var1=20
var2=22
[ "$var1" -ne "$var2" ] && echo "$var1 is not equal to $var2"

home=/home/bozo
[ -d "$home" ] || echo "$home directory does not exist."
```

The (( )) construct expands and evaluates an arithmetic expression. If the expression evaluates as zero, it returns an exit status of 1, or "false". A non-zero expression returns an exit status of 0, or "true". This is in marked contrast to using the **test** and [ ] constructs previously discussed.

Example 7-3. Arithmetic Tests using (( ))

```
#!/bin/bash
# arith-tests.sh
# Arithmetic tests.

# The (( ... )) construct evaluates and tests numerical expressions.
# Exit status opposite from [ ... ] construct!

(( 0 ))
echo "Exit status of \"(( 0 ))\" is $?."         # 1

(( 1 ))
echo "Exit status of \"(( 1 ))\" is $?."         # 0

(( 5 > 4 ))                                      # true
echo "Exit status of \"(( 5 > 4 ))\" is $?."     # 0

(( 5 > 9 ))                                      # false
echo "Exit status of \"(( 5 > 9 ))\" is $?."     # 1

(( 5 == 5 ))                                     # true
echo "Exit status of \"(( 5 == 5 ))\" is $?."    # 0
# (( 5 = 5 ))  gives an error message.

(( 5 - 5 ))                                      # 0
echo "Exit status of \"(( 5 - 5 ))\" is $?."     # 1

(( 5 / 4 ))                                      # Division o.k.
echo "Exit status of \"(( 5 / 4 ))\" is $?."     # 0

(( 1 / 2 ))                                      # Division result < 1.
echo "Exit status of \"(( 1 / 2 ))\" is $?."     # Rounded off to 0.
                                                 # 1

(( 1 / 0 )) 2>/dev/null                          # Illegal division by 0.
#           ^^^^^^^^^^^
echo "Exit status of \"(( 1 / 0 ))\" is $?."     # 1

# What effect does the "2>/dev/null" have?
# What would happen if it were removed?
# Try removing it, then rerunning the script.

# ======================================= #

# (( ... )) also useful in an if-then test.

var1=5
var2=4

if (( var1 > var2 ))
then #^      ^      Note: Not $var1, $var2. Why?
  echo "$var1 is greater than $var2"
fi     # 5 is greater than 4

exit 0
```

7.2. File test operators

Returns true if...

-e
: file exists

-a
: file exists

  This is identical in effect to -e. It has been "deprecated," [34] and its use is discouraged.

-f
: file is a *regular* file (not a directory or device file)

-s
: file is not zero size

-d
: file is a directory

-b
: file is a block device

-c
: file is a character device

```
device0="/dev/sda2"    # /   (root directory)
if [ -b "$device0" ]
then
  echo "$device0 is a block device."
fi

# /dev/sda2 is a block device.



device1="/dev/ttyS1"   # PCMCIA modem card.
if [ -c "$device1" ]
then
  echo "$device1 is a character device."
fi

# /dev/ttyS1 is a character device.
```

-p
: file is a pipe

```
function show_input_type()
{
   [ -p /dev/fd/0 ] && echo PIPE || echo STDIN
}

show_input_type "Input"                           # STDIN
echo "Input" | show_input_type                    # PIPE

# This example courtesy of Carl Anderson.
```

-h
: file is a symbolic link

-L

file is a symbolic link

-S
: file is a socket

-t
: file (descriptor) is associated with a terminal device

  This test option may be used to check whether the `stdin` **`[ -t 0 ]`** or `stdout` **`[ -t 1 ]`** in a given script is a terminal.

-r
: file has read permission (*for the user running the test*)

-w
: file has write permission (for the user running the test)

-x
: file has execute permission (for the user running the test)

-g
: set-group-id (sgid) flag set on file or directory

  If a directory has the *`sgid`* flag set, then a file created within that directory belongs to the group that owns the directory, not necessarily to the group of the user who created the file. This may be useful for a directory shared by a workgroup.

-u
: set-user-id (suid) flag set on file

  A binary owned by *root* with *`set-user-id`* flag set runs with *root* privileges, even when an ordinary user invokes it. [35] This is useful for executables (such as **pppd** and **cdrecord**) that need to access system hardware. Lacking the *suid* flag, these binaries could not be invoked by a *non-root* user.

```
-rwsr-xr-t    1 root       178236 Oct  2  2000 /usr/sbin/pppd
```

  A file with the *`suid`* flag set shows an *s* in its permissions.

-k
: *`sticky bit`* set

  Commonly known as the *sticky bit,* the *save-text-mode* flag is a special type of file permission. If a file has this flag set, that file will be kept in cache memory, for quicker access. [36] If set on a directory, it restricts write permission. Setting the sticky bit adds a *t* to the permissions on the file or directory listing. This restricts altering or deleting specific files in that directory to the owner of those files.

```
drwxrwxrwt    7 root         1024 May 19 21:26 tmp/
```

  If a user does not own a directory that has the sticky bit set, but has write permission in that directory, she can only delete those files that she owns in it. This keeps users from inadvertently overwriting or deleting each other's files in a publicly accessible directory, such as `/tmp`. (The *owner* of the directory or *root* can, of course, delete or rename files there.)

-O
: you are owner of file

-G
: group-id of file same as yours

-N
: file modified since it was last read

f1 -nt f2
: file *f1* is newer than *f2*

f1 -ot f2
: file *f1* is older than *f2*

f1 -ef f2
: files *f1* and *f2* are hard links to the same file

!
: "not" -- reverses the sense of the tests above (returns true if condition absent).

Example 7-4. Testing for broken links

```
#!/bin/bash
# broken-link.sh
# Written by Lee bigelow <ligelowbee@yahoo.com>
# Used in ABS Guide with permission.

#  A pure shell script to find dead symlinks and output them quoted
#+ so they can be fed to xargs and dealt with :)
#+ eg. sh broken-link.sh /somedir /someotherdir|xargs rm
#
#  This, however, is a better method:
#
#  find "somedir" -type l -print0|\
#  xargs -r0 file|\
#  grep "broken symbolic"|
#  sed -e 's/^\|: *broken symbolic.*$/"/g'
#
#+ but that wouldn't be pure Bash, now would it.
#  Caution: beware the /proc file system and any circular links!
################################################################


#  If no args are passed to the script set directories-to-search
#+ to current directory.  Otherwise set the directories-to-search
#+ to the args passed.
######################

[ $# -eq 0 ] && directorys=`pwd` || directorys=$@


#  Setup the function linkchk to check the directory it is passed
#+ for files that are links and don't exist, then print them quoted.
#  If one of the elements in the directory is a subdirectory then
#+ send that subdirectory to the linkcheck function.
##########

linkchk () {
    for element in $1/*; do
      [ -h "$element" -a ! -e "$element" ] && echo \"$element\"
      [ -d "$element" ] && linkchk $element
    # Of course, '-h' tests for symbolic link, '-d' for directory.
    done
}

#  Send each arg that was passed to the script to the linkchk() function
#+ if it is a valid directoy.  If not, then print the error message
```

```
#+ and usage info.
##################
for directory in $directorys; do
    if [ -d $directory ]
        then linkchk $directory
        else
            echo "$directory is not a directory"
            echo "Usage: $0 dir1 dir2 ..."
    fi
done

exit $?
```

Example 31-1, Example 11-8, Example 11-3, Example 31-3, and Example A-1 also illustrate uses of the file test operators.

7.3. Other Comparison Operators

A *binary* comparison operator compares two variables or quantities. *Note that integer and string comparison use a different set of operators.*

integer comparison

-eq
: is equal to

  `if [ "$a" -eq "$b" ]`

-ne
: is not equal to

  `if [ "$a" -ne "$b" ]`

-gt
: is greater than

  `if [ "$a" -gt "$b" ]`

-ge
: is greater than or equal to

  `if [ "$a" -ge "$b" ]`

-lt
: is less than

  `if [ "$a" -lt "$b" ]`

-le
: is less than or equal to

  `if [ "$a" -le "$b" ]`

<
: is less than (within double parentheses)

  `(("$a" < "$b"))`

<=
: is less than or equal to (within double parentheses)

`(("$a" <= "$b"))`

>

is greater than (within double parentheses)

`(("$a" > "$b"))`

>=

is greater than or equal to (within double parentheses)

`(("$a" >= "$b"))`

string comparison

=

is equal to

`if [ "$a" = "$b" ]`

Note the whitespace framing the =.

`if [ "$a"="$b" ]` is *not* equivalent to the above.

==

is equal to

`if [ "$a" == "$b" ]`

This is a synonym for =.

The == comparison operator behaves differently within a double-brackets test than within single brackets.

```
[[ $a == z* ]]   # True if $a starts with an "z" (pattern matching).
[[ $a == "z*" ]] # True if $a is equal to z* (literal matching).

[ $a == z* ]     # File globbing and word splitting take place.
[ "$a" == "z*" ] # True if $a is equal to z* (literal matching).

# Thanks, Stéphane Chazelas
```

!=

is not equal to

`if [ "$a" != "$b" ]`

This operator uses pattern matching within a [[...]] construct.

<

is less than, in ASCII alphabetical order

`if [[ "$a" < "$b" ]]`

`if [ "$a" \< "$b" ]`

Note that the "<" needs to be escaped within a **[ ]** construct.

>

is greater than, in ASCII alphabetical order

if [["$a" > "$b"]]

if ["$a" \> "$b"]

Note that the ">" needs to be escaped within a **[ ]** construct.

See Example 27-11 for an application of this comparison operator.

-z

string is *null*, that is, has zero length

```
 String=''   # Zero-length ("null") string variable.

if [ -z "$String" ]
then
  echo "\$String is null."
else
  echo "\$String is NOT null."
fi     # $String is null.
```

-n

string is not *null.*

The **-n** test requires that the string be quoted within the test brackets. Using an unquoted string with *! -z*, or even just the unquoted string alone within test brackets (see Example 7-6) normally works, however, this is an unsafe practice. *Always* quote a tested string. [37]

Example 7-5. Arithmetic and string comparisons

```
#!/bin/bash

a=4
b=5

#  Here "a" and "b" can be treated either as integers or strings.
#  There is some blurring between the arithmetic and string comparisons,
#+ since Bash variables are not strongly typed.

#  Bash permits integer operations and comparisons on variables
#+ whose value consists of all-integer characters.
#  Caution advised, however.

echo

if [ "$a" -ne "$b" ]
then
  echo "$a is not equal to $b"
  echo "(arithmetic comparison)"
fi

echo
```

```
if [ "$a" != "$b" ]
then
  echo "$a is not equal to $b."
  echo "(string comparison)"
  #     "4"  != "5"
  # ASCII 52 != ASCII 53
fi

# In this particular instance, both "-ne" and "!=" work.

echo

exit 0
```

Example 7-6. Testing whether a string is *null*

```
#!/bin/bash
#  str-test.sh: Testing null strings and unquoted strings,
#+ but not strings and sealing wax, not to mention cabbages and kings . . .

# Using   if [ ... ]

# If a string has not been initialized, it has no defined value.
# This state is called "null" (not the same as zero!).

if [ -n $string1 ]    # string1 has not been declared or initialized.
then
  echo "String \"string1\" is not null."
else
  echo "String \"string1\" is null."
fi                    # Wrong result.
# Shows $string1 as not null, although it was not initialized.

echo

# Let's try it again.

if [ -n "$string1" ]  # This time, $string1 is quoted.
then
  echo "String \"string1\" is not null."
else
  echo "String \"string1\" is null."
fi                    # Quote strings within test brackets!

echo

if [ $string1 ]       # This time, $string1 stands naked.
then
  echo "String \"string1\" is not null."
else
  echo "String \"string1\" is null."
fi                    # This works fine.
# The [ ... ] test operator alone detects whether the string is null.
# However it is good practice to quote it (if [ "$string1" ]).
#
# As Stephane Chazelas points out,
#    if [ $string1 ]    has one argument, "]"
#    if [ "$string1" ]  has two arguments, the empty "$string1" and "]"


echo
```

```
string1=initialized

if [ $string1 ]       # Again, $string1 stands unquoted.
then
  echo "String \"string1\" is not null."
else
  echo "String \"string1\" is null."
fi                    # Again, gives correct result.
# Still, it is better to quote it ("$string1"), because . . .


string1="a = b"

if [ $string1 ]       # Again, $string1 stands unquoted.
then
  echo "String \"string1\" is not null."
else
  echo "String \"string1\" is null."
fi                    # Not quoting "$string1" now gives wrong result!

exit 0   # Thank you, also, Florian Wisser, for the "heads-up".
```

Example 7-7. *zmore*

```
#!/bin/bash
# zmore

# View gzipped files with 'more' filter.

E_NOARGS=85
E_NOTFOUND=86
E_NOTGZIP=87

if [ $# -eq 0 ] # same effect as:  if [ -z "$1" ]
# $1 can exist, but be empty:  zmore "" arg2 arg3
then
  echo "Usage: `basename $0` filename" >&2
  # Error message to stderr.
  exit $E_NOARGS
  # Returns 85 as exit status of script (error code).
fi

filename=$1

if [ ! -f "$filename" ]   # Quoting $filename allows for possible spaces.
then
  echo "File $filename not found!" >&2   # Error message to stderr.
  exit $E_NOTFOUND
fi

if [ ${filename##*.} != "gz" ]
# Using bracket in variable substitution.
then
  echo "File $1 is not a gzipped file!"
  exit $E_NOTGZIP
fi

zcat $1 | more
```

```
# Uses the 'more' filter.
# May substitute 'less' if desired.

exit $?   # Script returns exit status of pipe.
#  Actually "exit $?" is unnecessary, as the script will, in any case,
#+ return the exit status of the last command executed.
```

compound comparison

-a
: logical and

  exp1 -a exp2 returns true if *both* exp1 and exp2 are true.

-o
: logical or

  exp1 -o exp2 returns true if either exp1 *or* exp2 is true.

These are similar to the Bash comparison operators **&&** and **||**, used within double brackets.

```
[[ condition1 && condition2 ]]
```

The **-o** and **-a** operators work with the test command or occur within single test brackets.

```
if [ "$expr1" -a "$expr2" ]
then
  echo "Both expr1 and expr2 are true."
else
  echo "Either expr1 or expr2 is false."
fi
```

But, as *rihad* points out:

```
[ 1 -eq 1 ] && [ -n "`echo true 1>&2`" ]   # true
[ 1 -eq 2 ] && [ -n "`echo true 1>&2`" ]   # (no output)
# ^^^^^^^ False condition. So far, everything as expected.

# However ...
[ 1 -eq 2 -a -n "`echo true 1>&2`" ]       # true
# ^^^^^^^ False condition. So, why "true" output?

# Is it because both condition clauses within brackets evaluate?
[[ 1 -eq 2 && -n "`echo true 1>&2`" ]]     # (no output)
# No, that's not it.

# Apparently && and || "short-circuit" while -a and -o do not.
```

Refer to Example 8-3, Example 27-17, and Example A-29 to see compound comparison operators in action.

7.4. Nested *if/then* Condition Tests

Condition tests using the *if/then* construct may be nested. The net result is equivalent to using the *&&* compound comparison operator.

```
a=3
```

```
if [ "$a" -gt 0 ]
then
  if [ "$a" -lt 5 ]
  then
    echo "The value of \"a\" lies somewhere between 0 and 5."
  fi
fi

# Same result as:

if [ "$a" -gt 0 ] && [ "$a" -lt 5 ]
then
  echo "The value of \"a\" lies somewhere between 0 and 5."
fi
```

Example 37-4 and Example 17-11 demonstrate nested `if/then` condition tests.

7.5. Testing Your Knowledge of Tests

The systemwide `xinitrc` file can be used to launch the X server. This file contains quite a number of *if/then* tests. The following is excerpted from an "ancient" version of `xinitrc` (*Red Hat 7.1*, or thereabouts).

```
if [ -f $HOME/.Xclients ]; then
  exec $HOME/.Xclients
elif [ -f /etc/X11/xinit/Xclients ]; then
  exec /etc/X11/xinit/Xclients
else
     # failsafe settings.  Although we should never get here
     # (we provide fallbacks in Xclients as well) it can't hurt.
     xclock -geometry 100x100-5+5 &
     xterm -geometry 80x50-50+150 &
     if [ -f /usr/bin/netscape -a -f /usr/share/doc/HTML/index.html ]; then
             netscape /usr/share/doc/HTML/index.html &
     fi
fi
```

Explain the *test* constructs in the above snippet, then examine an updated version of the file, `/etc/X11/xinit/xinitrc`, and analyze the *if/then* test constructs there. You may need to refer ahead to the discussions of grep, sed, and regular expressions.

Chapter 8. Operations and Related Topics

8.1. Operators

assignment

`variable assignment`
: Initializing or changing the value of a variable

=
: All-purpose assignment operator, which works for both arithmetic and string assignments.

```
var=27
category=minerals  # No spaces allowed after the "=".
```

Do not confuse the "=" assignment operator with the = test operator.

```
#   =  as a test operator

if [ "$string1" = "$string2" ]
then
   command
fi

#  if [ "X$string1" = "X$string2" ] is safer,
#+ to prevent an error message should one of the variables be empty.
#  (The prepended "X" characters cancel out.)
```

arithmetic operators

+
: plus

-
: minus

*
: multiplication

/
: division

**
: exponentiation

```
# Bash, version 2.02, introduced the "**" exponentiation operator.

let "z=5**3"    # 5 * 5 * 5
echo "z = $z"   # z = 125
```

%
: modulo, or mod (returns the *remainder* of an integer division operation)

```
bash$ expr 5 % 3
2
```

5/3 = 1, with remainder 2

This operator finds use in, among other things, generating numbers within a specific range (see Example 9-11 and Example 9-15) and formatting program output (see Example 27-16 and Example A-6). It can even be used to generate prime numbers, (see Example A-15). Modulo turns up surprisingly often in numerical recipes.

Example 8-1. Greatest common divisor

```
#!/bin/bash
# gcd.sh: greatest common divisor
#         Uses Euclid's algorithm

#  The "greatest common divisor" (gcd) of two integers
#+ is the largest integer that will divide both, leaving no remainder.

#  Euclid's algorithm uses successive division.
#    In each pass,
#+      dividend <---  divisor
#+      divisor  <---  remainder
#+   until remainder = 0.
#    The gcd = dividend, on the final pass.
#
#  For an excellent discussion of Euclid's algorithm, see
#+ Jim Loy's site, http://www.jimloy.com/number/euclids.htm.


# ------------------------------------------------------
# Argument check
ARGS=2
E_BADARGS=85

if [ $# -ne "$ARGS" ]
then
  echo "Usage: `basename $0` first-number second-number"
  exit $E_BADARGS
fi
# ------------------------------------------------------


gcd ()
{

  dividend=$1             #  Arbitrary assignment.
  divisor=$2              #! It doesn't matter which of the two is larger.
                          #  Why not?

  remainder=1             #  If an uninitialized variable is used inside
                          #+ test brackets, an error message results.

  until [ "$remainder" -eq 0 ]
  do    #  ^^^^^^^^^^  Must be previously initialized!
    let "remainder = $dividend % $divisor"
    dividend=$divisor     # Now repeat with 2 smallest numbers.
    divisor=$remainder
  done                    # Euclid's algorithm

}                         # Last $dividend is the gcd.


gcd $1 $2
```

```
echo; echo "GCD of $1 and $2 = $dividend"; echo


# Exercises :
# ---------
# 1) Check command-line arguments to make sure they are integers,
#+   and exit the script with an appropriate error message if not.
# 2) Rewrite the gcd () function to use local variables.

exit 0
```

+=

plus-equal (increment variable by a constant) [38]

let "var += 5" results in *var* being incremented by 5.

-=

minus-equal (decrement variable by a constant)

*=

times-equal (multiply variable by a constant)

let "var *= 4" results in *var* being multiplied by 4.

/=

slash-equal (divide variable by a constant)

%=

mod-equal (*remainder* of dividing variable by a constant)

Arithmetic operators often occur in an expr or let expression.

Example 8-2. Using Arithmetic Operations

```
#!/bin/bash
# Counting to 11 in 10 different ways.

n=1; echo -n "$n "

let "n = $n + 1"   # let "n = n + 1"  also works.
echo -n "$n "


: $((n = $n + 1))
#  ":" necessary because otherwise Bash attempts
#+ to interpret "$((n = $n + 1))" as a command.
echo -n "$n "

(( n = n + 1 ))
#  A simpler alternative to the method above.
#  Thanks, David Lombard, for pointing this out.
echo -n "$n "

n=$(($n + 1))
echo -n "$n "

: $[ n = $n + 1 ]
#  ":" necessary because otherwise Bash attempts
#+ to interpret "$[ n = $n + 1 ]" as a command.
#  Works even if "n" was initialized as a string.
```

```
echo -n "$n "

n=$[ $n + 1 ]
#  Works even if "n" was initialized as a string.
#* Avoid this type of construct, since it is obsolete and nonportable.
#  Thanks, Stephane Chazelas.
echo -n "$n "

# Now for C-style increment operators.
# Thanks, Frank Wang, for pointing this out.

let "n++"          # let "++n"  also works.
echo -n "$n "

(( n++ ))          # (( ++n ))  also works.
echo -n "$n "

: $(( n++ ))       # : $(( ++n )) also works.
echo -n "$n "

: $[ n++ ]         # : $[ ++n ] also works
echo -n "$n "

echo

exit 0
```

Integer variables in older versions of Bash were signed *long* (32-bit) integers, in the range of -2147483648 to 2147483647. An operation that took a variable outside these limits gave an erroneous result.

```
echo $BASH_VERSION   # 1.14

a=2147483646
echo "a = $a"        # a = 2147483646
let "a+=1"           # Increment "a".
echo "a = $a"        # a = 2147483647
let "a+=1"           # increment "a" again, past the limit.
echo "a = $a"        # a = -2147483648
                     #      ERROR: out of range,
                     # +    and the leftmost bit, the sign bit,
                     # +    has been set, making the result negative.
```

As of version >= 2.05b, Bash supports 64-bit integers.

Bash does not understand floating point arithmetic. It treats numbers containing a decimal point as strings.

```
a=1.5

let "b = $a + 1.3"  # Error.
# t2.sh: let: b = 1.5 + 1.3: syntax error in expression
#                            (error token is ".5 + 1.3")

echo "b = $b"       # b=1
```

Use bc in scripts that that need floating point calculations or math library functions.

bitwise operators. The bitwise operators seldom make an appearance in shell scripts. Their chief use seems to be manipulating and testing values read from ports or sockets. "Bit flipping" is more relevant to compiled languages, such as C and C++, which provide direct access to system hardware. However, see *vladz's* ingenious use of bitwise operators in his *base64.sh* (Example A-54) script.

bitwise operators

<<
: bitwise left shift (multiplies by `2` for each shift position)

<<=
: *left-shift-equal*

  `let "var <<= 2"` results in *`var`* left-shifted `2` bits (multiplied by `4`)

>>
: bitwise right shift (divides by `2` for each shift position)

>>=
: *right-shift-equal* (inverse of <<=)

&
: bitwise AND

&=
: bitwise *AND-equal*

|
: bitwise OR

|=
: bitwise *OR-equal*

~
: bitwise NOT

^
: bitwise XOR

^=
: bitwise *XOR-equal*

logical (boolean) operators

!
: NOT

```
if [ ! -f $FILENAME ]
then
  ...
```

&&
: AND

```
if [ $condition1 ] && [ $condition2 ]
#  Same as:  if [ $condition1 -a $condition2 ]
#  Returns true if both condition1 and condition2 hold true...

if [[ $condition1 && $condition2 ]]    # Also works.
#  Note that && operator not permitted inside brackets
#+ of [ ... ] construct.
```

&& may also be used, depending on context, in an and list to concatenate commands.

||

OR

```
if [ $condition1 ] || [ $condition2 ]
# Same as:  if [ $condition1 -o $condition2 ]
# Returns true if either condition1 or condition2 holds true...

if [[ $condition1 || $condition2 ]]    # Also works.
#  Note that || operator not permitted inside brackets
#+ of a [ ... ] construct.
```

Bash tests the exit status of each statement linked with a logical operator.

Example 8-3. Compound Condition Tests Using && and ||

```
#!/bin/bash

a=24
b=47

if [ "$a" -eq 24 ] && [ "$b" -eq 47 ]
then
  echo "Test #1 succeeds."
else
  echo "Test #1 fails."
fi

# ERROR:   if [ "$a" -eq 24 && "$b" -eq 47 ]
#+         attempts to execute  ' [ "$a" -eq 24 '
#+         and fails to finding matching ']'.
#
#  Note:  if [[ $a -eq 24 && $b -eq 24 ]]  works.
#  The double-bracket if-test is more flexible
#+ than the single-bracket version.
#    (The "&&" has a different meaning in line 17 than in line 6.)
#    Thanks, Stephane Chazelas, for pointing this out.


if [ "$a" -eq 98 ] || [ "$b" -eq 47 ]
then
  echo "Test #2 succeeds."
else
  echo "Test #2 fails."
fi


#  The -a and -o options provide
#+ an alternative compound condition test.
#  Thanks to Patrick Callahan for pointing this out.


if [ "$a" -eq 24 -a "$b" -eq 47 ]
then
  echo "Test #3 succeeds."
else
  echo "Test #3 fails."
fi
```

```
if [ "$a" -eq 98 -o "$b" -eq 47 ]
then
  echo "Test #4 succeeds."
else
  echo "Test #4 fails."
fi


a=rhino
b=crocodile
if [ "$a" = rhino ] && [ "$b" = crocodile ]
then
  echo "Test #5 succeeds."
else
  echo "Test #5 fails."
fi

exit 0
```

The && and || operators also find use in an arithmetic context.

```
bash$ echo $(( 1 && 2 )) $((3 && 0)) $((4 || 0)) $((0 || 0))
1 0 1 0
```

miscellaneous operators

,
: Comma operator

The **comma operator** chains together two or more arithmetic operations. All the operations are evaluated (with possible *side effects*. [39]

```
let "t1 = ((5 + 3, 7 - 1, 15 - 4))"
echo "t1 = $t1"           ^^^^^^  # t1 = 11
# Here t1 is set to the result of the last operation. Why?

let "t2 = ((a = 9, 15 / 3))"      # Set "a" and calculate "t2".
echo "t2 = $t2    a = $a"         # t2 = 5    a = 9
```

The comma operator finds use mainly in for loops. See Example 11-13.

8.2. Numerical Constants

A shell script interprets a number as decimal (base 10), unless that number has a special prefix or notation. A number preceded by a *0* is *octal* (base 8). A number preceded by *0x* is *hexadecimal* (base 16). A number with an embedded *#* evaluates as *BASE#NUMBER* (with range and notational restrictions).

Example 8-4. Representation of numerical constants

```
#!/bin/bash
# numbers.sh: Representation of numbers in different bases.

# Decimal: the default
let "dec = 32"
echo "decimal number = $dec"             # 32
```

```
# Nothing out of the ordinary here.


# Octal: numbers preceded by '0' (zero)
let "oct = 032"
echo "octal number = $oct"              # 26
# Expresses result in decimal.
# --------- ------ -- -------


# Hexadecimal: numbers preceded by '0x' or '0X'
let "hex = 0x32"
echo "hexadecimal number = $hex"        # 50

echo $((0x9abc))                        # 39612
#     ^^      ^^   double-parentheses arithmetic expansion/evaluation
# Expresses result in decimal.



# Other bases: BASE#NUMBER
# BASE between 2 and 64.
# NUMBER must use symbols within the BASE range, see below.


let "bin = 2#111100111001101"
echo "binary number = $bin"             # 31181

let "b32 = 32#77"
echo "base-32 number = $b32"            # 231

let "b64 = 64#@_"
echo "base-64 number = $b64"            # 4031
# This notation only works for a limited range (2 - 64) of ASCII characters.
# 10 digits + 26 lowercase characters + 26 uppercase characters + @ + _


echo

echo $((36#zz)) $((2#10101010)) $((16#AF16)) $((53#1aA))
                                        # 1295 170 44822 3375


#  Important note:
#  --------------
#  Using a digit out of range of the specified base notation
#+ gives an error message.

let "bad_oct = 081"
# (Partial) error message output:
#  bad_oct = 081: value too great for base (error token is "081")
#              Octal numbers use only digits in the range 0 - 7.

exit $?   # Exit value = 1 (error)

# Thanks, Rich Bartell and Stephane Chazelas, for clarification.
```

8.3. The Double-Parentheses Construct

Similar to the let command, the **((...))** construct permits arithmetic expansion and evaluation. In its simplest form, **`a=$(( 5 + 3 ))`** would set **`a`** to **`5 + 3`**, or **`8`**. However, this double-parentheses construct is also a mechanism for allowing C-style manipulation of variables in Bash, for example, `(( var++ ))`.

Example 8-5. C-style manipulation of variables

```
#!/bin/bash
# c-vars.sh
# Manipulating a variable, C-style, using the (( ... )) construct.


echo

(( a = 23 ))  #  Setting a value, C-style,
              #+ with spaces on both sides of the "=".
echo "a (initial value) = $a"   # 23

(( a++ ))     #  Post-increment 'a', C-style.
echo "a (after a++) = $a"       # 24

(( a-- ))     #  Post-decrement 'a', C-style.
echo "a (after a--) = $a"       # 23


(( ++a ))     #  Pre-increment 'a', C-style.
echo "a (after ++a) = $a"       # 24

(( --a ))     #  Pre-decrement 'a', C-style.
echo "a (after --a) = $a"       # 23

echo

########################################################
#  Note that, as in C, pre- and post-decrement operators
#+ have different side-effects.

n=1; let --n && echo "True" || echo "False"  # False
n=1; let n-- && echo "True" || echo "False"  # True

#  Thanks, Jeroen Domburg.
########################################################

echo

(( t = a<45?7:11 ))   # C-style trinary operator.
#       ^  ^ ^
echo "If a < 45, then t = 7, else t = 11."  # a = 23
echo "t = $t "                              # t = 7

echo


# -----------------
# Easter Egg alert!
```

```
# -----------------
#  Chet Ramey seems to have snuck a bunch of undocumented C-style
#+ constructs into Bash (actually adapted from ksh, pretty much).
#  In the Bash docs, Ramey calls (( ... )) shell arithmetic,
#+ but it goes far beyond that.
#  Sorry, Chet, the secret is out.

# See also "for" and "while" loops using the (( ... )) construct.

# These work only with version 2.04 or later of Bash.

exit
```

See also Example 11-13 and Example 8-4.

8.4. Operator Precedence

In a script, operations execute in order of *precedence*: the higher precedence operations execute *before* the lower precedence ones. [40]

Table 8-1. Operator Precedence

Operator	Meaning	Comments
		HIGHEST PRECEDENCE
`var++ var--`	post-increment, post-decrement	C-style operators
`++var --var`	pre-increment, pre-decrement	
`! ~`	negation	logical / bitwise, inverts sense of following operator
`**`	exponentiation	arithmetic operation
`* / %`	multiplication, division, modulo	arithmetic operation
`+ -`	addition, subtraction	arithmetic operation
`<< >>`	left, right shift	bitwise
`-z -n`	*unary* comparison	string is/is-not null
`-e -f -t -x, etc.`	*unary* comparison	file-test
`< -lt > -gt <= -le >= -ge`	*compound* comparison	string and integer
`-nt -ot -ef`	*compound* comparison	file-test
`== -eq != -ne`	equality / inequality	test operators, string and integer
`&`	AND	bitwise
`^`	XOR	*exclusive* OR, bitwise
`|`	OR	bitwise

`&& -a`	AND	logical, *compound* comparison
`|| -o`	OR	logical, *compound* comparison
`?:`	trinary operator	C-style
`=`	assignment	(do not confuse with equality *test*)
`*= /= %= += -= <<= >>= &=`	combination assignment	times-equal, divide-equal, mod-equal, etc.
`,`	comma	links a sequence of operations
		LOWEST PRECEDENCE

In practice, all you really need to remember is the following:

* The "My Dear Aunt Sally" mantra (*multiply, divide, add, subtract*) for the familiar arithmetic operations.
* The *compound* logical operators, **&&**, **||**, **-a**, and **-o** have low precedence.
* The order of evaluation of equal-precedence operators is usually *left-to-right*.

Now, let's utilize our knowledge of operator precedence to analyze a couple of lines from the `/etc/init.d/functions file`, as found in the *Fedora Core* Linux distro.

```
while [ -n "$remaining" -a "$retry" -gt 0 ]; do

# This looks rather daunting at first glance.


# Separate the conditions:
while [ -n "$remaining" -a "$retry" -gt 0 ]; do
#       --condition 1-- ^^ --condition 2-

#  If variable "$remaining" is not zero length
#+      AND (-a)
#+ variable "$retry" is greater-than zero
#+ then
#+ the [ expresion-within-condition-brackets ] returns success (0)
#+ and the while-loop executes an iteration.
#  ==============================================================
#  Evaluate "condition 1" and "condition 2" ***before***
#+ ANDing them. Why? Because the AND (-a) has a lower precedence
#+ than the -n and -gt operators,
#+ and therefore gets evaluated *last*.

#################################################################

if [ -f /etc/sysconfig/i18n -a -z "${NOLOCALE:-}" ] ; then


# Again, separate the conditions:
if [ -f /etc/sysconfig/i18n -a -z "${NOLOCALE:-}" ] ; then
#    --condition 1--------- ^^ --condition 2-----

#  If file "/etc/sysconfig/i18n" exists
#+      AND (-a)
#+ variable $NOLOCALE is zero length
#+ then
```

```
#+ the [ test-expresion-within-condition-brackets ] returns success (0)
#+ and the commands following execute.
#
#  As before, the AND (-a) gets evaluated *last*
#+ because it has the lowest precedence of the operators within
#+ the test brackets.
#  ==============================================================
#  Note:
#  ${NOLOCALE:-} is a parameter expansion that seems redundant.
#  But, if $NOLOCALE has not been declared, it gets set to *null*,
#+ in effect declaring it.
#  This makes a difference in some contexts.
```

To avoid confusion or error in a complex sequence of test operators, break up the sequence into bracketed sections.

```
if [ "$v1" -gt "$v2"  -o  "$v1" -lt "$v2"  -a  -e "$filename" ]
# Unclear what's going on here...

if [[ "$v1" -gt "$v2" ]] || [[ "$v1" -lt "$v2" ]] && [[ -e "$filename" ]]
# Much better -- the condition tests are grouped in logical sections.
```

Part 3. Beyond the Basics

Table of Contents

Chapter 9. Another Look at Variables

Used properly, variables can add power and flexibility to scripts. This requires learning their subtleties and nuances.

9.1. Internal Variables

Builtin variables:
: variables affecting bash script behavior

`$BASH`
: The path to the *Bash* binary itself

```
bash$ echo $BASH
/bin/bash
```

`$BASH_ENV`
: An environmental variable pointing to a Bash startup file to be read when a script is invoked

`$BASH_SUBSHELL`
: A variable indicating the subshell level. This is a new addition to Bash, version 3.

See Example 21-1 for usage.

`$BASHPID`
: *Process ID* of the current instance of Bash. This is not the same as the $$ variable, but it often gives the same result.

```
bash4$ echo $$
11015


bash4$ echo $BASHPID
11015


bash4$ ps ax | grep bash4
11015 pts/2    R      0:00 bash4
```

But ...

```
#!/bin/bash4

echo "\$\$ outside of subshell = $$"                              # 9602
echo "\$BASH_SUBSHELL  outside of subshell = $BASH_SUBSHELL"      # 0
echo "\$BASHPID outside of subshell = $BASHPID"                   # 9602

echo

( echo "\$\$ inside of subshell = $$"                             # 9602
  echo "\$BASH_SUBSHELL inside of subshell = $BASH_SUBSHELL"      # 1
  echo "\$BASHPID inside of subshell = $BASHPID" )                # 9603
  # Note that $$ returns PID of parent process.
```

`$BASH_VERSINFO[n]`
: A 6-element array containing version information about the installed release of Bash. This is similar to `$BASH_VERSION`, below, but a bit more detailed.

```
# Bash version info:

for n in 0 1 2 3 4 5
do
  echo "BASH_VERSINFO[$n] = ${BASH_VERSINFO[$n]}"
done

# BASH_VERSINFO[0] = 3                      # Major version no.
# BASH_VERSINFO[1] = 00                     # Minor version no.
# BASH_VERSINFO[2] = 14                     # Patch level.
# BASH_VERSINFO[3] = 1                      # Build version.
# BASH_VERSINFO[4] = release                # Release status.
# BASH_VERSINFO[5] = i386-redhat-linux-gnu  # Architecture
                                            # (same as $MACHTYPE).
```

$BASH_VERSION
: The version of Bash installed on the system

```
bash$ echo $BASH_VERSION
3.2.25(1)-release
```

```
tcsh% echo $BASH_VERSION
BASH_VERSION: Undefined variable.
```

Checking $BASH_VERSION is a good method of determining which shell is running. $SHELL does not necessarily give the correct answer.

$CDPATH
: A colon-separated list of search paths available to the cd command, similar in function to the $PATH variable for binaries. The `$CDPATH` variable may be set in the local `~/.bashrc` file.

```
bash$ cd bash-doc
bash: cd: bash-doc: No such file or directory


bash$ CDPATH=/usr/share/doc
bash$ cd bash-doc
/usr/share/doc/bash-doc


bash$ echo $PWD
/usr/share/doc/bash-doc
```

$DIRSTACK
: The top value in the directory stack [41] (affected by pushd and popd)

This builtin variable corresponds to the dirs command, however **dirs** shows the entire contents of the directory stack.

$EDITOR
: The default editor invoked by a script, usually **vi** or **emacs**.

$EUID
: "effective" user ID number

Identification number of whatever identity the current user has assumed, perhaps by means of su.

The `$EUID` is not necessarily the same as the $UID.

`$FUNCNAME`

Name of the current function

```
xyz23 ()
{
  echo "$FUNCNAME now executing."  # xyz23 now executing.
}

xyz23

echo "FUNCNAME = $FUNCNAME"        # FUNCNAME =
                                   # Null value outside a function.
```

See also Example A-50.

`$GLOBIGNORE`

A list of filename patterns to be excluded from matching in globbing.

`$GROUPS`

Groups current user belongs to

This is a listing (array) of the group id numbers for current user, as recorded in `/etc/passwd` and `/etc/group`.

```
root# echo $GROUPS
0


root# echo ${GROUPS[1]}
1


root# echo ${GROUPS[5]}
6
```

`$HOME`

Home directory of the user, usually `/home/username` (see Example 10-7)

`$HOSTNAME`

The hostname command assigns the system host name at bootup in an init script. However, the `gethostname()` function sets the Bash internal variable `$HOSTNAME`. See also Example 10-7.

`$HOSTTYPE`

host type

Like $MACHTYPE, identifies the system hardware.

```
bash$ echo $HOSTTYPE
i686
```

`$IFS`

internal field separator

This variable determines how Bash recognizes fields, or word boundaries, when it interprets character strings.

$IFS defaults to whitespace (space, tab, and newline), but may be changed, for example, to parse a comma-separated data file. Note that $* uses the first character held in `$IFS`. See Example 5-1.

```
bash$ echo "$IFS"

(With $IFS set to default, a blank line displays.)



bash$ echo "$IFS" | cat -vte
 ^I$
 $
(Show whitespace: here a single space, ^I [horizontal tab],
  and newline, and display "$" at end-of-line.)



bash$ bash -c 'set w x y z; IFS=":-;"; echo "$*"'
w:x:y:z
(Read commands from string and assign any arguments to pos params.)
```

Set `$IFS` to eliminate whitespace in pathnames.

```
IFS="$(printf '\n\t')"   # Per David Wheeler.
```

`$IFS` does not handle whitespace the same as it does other characters.

Example 9-1. $IFS and whitespace

```
#!/bin/bash
# ifs.sh


var1="a+b+c"
var2="d-e-f"
var3="g,h,i"

IFS=+
# The plus sign will be interpreted as a separator.
echo $var1     # a b c
echo $var2     # d-e-f
echo $var3     # g,h,i

echo

IFS="-"
# The plus sign reverts to default interpretation.
# The minus sign will be interpreted as a separator.
echo $var1     # a+b+c
echo $var2     # d e f
echo $var3     # g,h,i

echo

IFS=","
# The comma will be interpreted as a separator.
# The minus sign reverts to default interpretation.
echo $var1     # a+b+c
echo $var2     # d-e-f
echo $var3     # g h i

echo
```

```
IFS=" "
# The space character will be interpreted as a separator.
# The comma reverts to default interpretation.
echo $var1     # a+b+c
echo $var2     # d-e-f
echo $var3     # g,h,i

# ======================================================== #

# However ...
# $IFS treats whitespace differently than other characters.

output_args_one_per_line()
{
  for arg
  do
    echo "[$arg]"
  done #  ^    ^   Embed within brackets, for your viewing pleasure.
}

echo; echo "IFS=\" \""
echo "-------"

IFS=" "
var=" a  b c   "
#    ^ ^^   ^^^
output_args_one_per_line $var  # output_args_one_per_line `echo " a  b c   "`
# [a]
# [b]
# [c]


echo; echo "IFS=:"
echo "-----"

IFS=:
var=":a::b:c:::"               # Same pattern as above,
#    ^ ^^   ^^^                #+ but substituting ":" for " "  ...
output_args_one_per_line $var
# []
# [a]
# []
# [b]
# [c]
# []
# []

# Note "empty" brackets.
# The same thing happens with the "FS" field separator in awk.


echo

exit
```

(Many thanks, Stéphane Chazelas, for clarification and above examples.)

See also Example 16-41, Example 11-8, and Example 19-14 for instructive examples of using `$IFS`.

`$IGNOREEOF`

Ignore EOF: how many end-of-files (control-D) the shell will ignore before logging out.

`$LC_COLLATE`

Often set in the .bashrc or /etc/profile files, this variable controls collation order in filename expansion and pattern matching. If mishandled, LC_COLLATE can cause unexpected results in filename globbing.

As of version 2.05 of Bash, filename globbing no longer distinguishes between lowercase and uppercase letters in a character range between brackets. For example, **ls [A-M]*** would match both File1.txt and file1.txt. To revert to the customary behavior of bracket matching, set LC_COLLATE to C by an **export LC_COLLATE=C** in /etc/profile and/or ~/.bashrc.

$LC_CTYPE
: This internal variable controls character interpretation in globbing and pattern matching.

$LINENO
: This variable is the line number of the shell script in which this variable appears. It has significance only within the script in which it appears, and is chiefly useful for debugging purposes.

```
# *** BEGIN DEBUG BLOCK ***
last_cmd_arg=$_  # Save it.

echo "At line number $LINENO, variable \"v1\" = $v1"
echo "Last command argument processed = $last_cmd_arg"
# *** END DEBUG BLOCK ***
```

$MACHTYPE
: machine type

Identifies the system hardware.

```
bash$ echo $MACHTYPE
i686
```

$OLDPWD
: Old working directory ("OLD-Print-Working-Directory", previous directory you were in).

$OSTYPE
: operating system type

```
bash$ echo $OSTYPE
linux
```

$PATH
: Path to binaries, usually /usr/bin/, /usr/X11R6/bin/, /usr/local/bin, etc.

When given a command, the shell automatically does a hash table search on the directories listed in the *path* for the executable. The path is stored in the environmental variable, $PATH, a list of directories, separated by colons. Normally, the system stores the $PATH definition in /etc/profile and/or ~/.bashrc (see Appendix H).

```
bash$ echo $PATH
/bin:/usr/bin:/usr/local/bin:/usr/X11R6/bin:/sbin:/usr/sbin
```

PATH=${PATH}:/opt/bin appends the /opt/bin directory to the current path. In a script, it may be expedient to temporarily add a directory to the path in this way. When the script exits, this restores the original $PATH (a child process, such as a script, may not change the environment of the parent process, the shell).

The current "working directory", `./`, is usually omitted from the `$PATH` as a security measure.

`$PIPESTATUS`

Array variable holding exit status(es) of last executed *foreground* pipe.

```
bash$ echo $PIPESTATUS
0

bash$ ls -al | bogus_command
bash: bogus_command: command not found
bash$ echo ${PIPESTATUS[1]}
127

bash$ ls -al | bogus_command
bash: bogus_command: command not found
bash$ echo $?
127
```

The members of the `$PIPESTATUS` array hold the exit status of each respective command executed in a pipe. `$PIPESTATUS[0]` holds the exit status of the first command in the pipe, `$PIPESTATUS[1]` the exit status of the second command, and so on.

The `$PIPESTATUS` variable may contain an erroneous 0 value in a login shell (in releases prior to 3.0 of Bash).

```
tcsh% bash

bash$ who | grep nobody | sort
bash$ echo ${PIPESTATUS[*]}
0
```

The above lines contained in a script would produce the expected `0 1 0` output.

Thank you, Wayne Pollock for pointing this out and supplying the above example.

The `$PIPESTATUS` variable gives unexpected results in some contexts.

```
bash$ echo $BASH_VERSION
3.00.14(1)-release

bash$ $ ls | bogus_command | wc
bash: bogus_command: command not found
 0       0       0

bash$ echo ${PIPESTATUS[@]}
141 127 0
```

Chet Ramey attributes the above output to the behavior of ls. If *ls* writes to a *pipe* whose output is not read, then *SIGPIPE* kills it, and its exit status is 141. Otherwise its exit status is 0, as expected. This likewise is the case for tr.

`$PIPESTATUS` is a "volatile" variable. It needs to be captured immediately after the pipe in question, before any other command intervenes.

```
bash$ $ ls | bogus_command | wc
bash: bogus_command: command not found
 0       0       0

bash$ echo ${PIPESTATUS[@]}
0 127 0

bash$ echo ${PIPESTATUS[@]}
0
```

The pipefail option may be useful in cases where `$PIPESTATUS` does not give the desired information.

`$PPID`

The `$PPID` of a process is the process ID (`pid`) of its parent process. [42]

Compare this with the pidof command.

`$PROMPT_COMMAND`

A variable holding a command to be executed just before the primary prompt, `$PS1` is to be displayed.

`$PS1`

This is the main prompt, seen at the command-line.

`$PS2`

The secondary prompt, seen when additional input is expected. It displays as ">".

`$PS3`

The tertiary prompt, displayed in a select loop (see Example 11-30).

`$PS4`

The quartenary prompt, shown at the beginning of each line of output when invoking a script with the -x *[verbose trace]* option. It displays as "+".

As a debugging aid, it may be useful to embed diagnostic information in `$PS4`.

```
P4='$(read time junk < /proc/$$/schedstat; echo "@@@ $time @@@ " )'
# Per suggestion by Erik Brandsberg.
set -x
# Various commands follow ...
```

`$PWD`

Working directory (directory you are in at the time)

This is the analog to the pwd builtin command.

```
#!/bin/bash

E_WRONG_DIRECTORY=85

clear # Clear the screen.

TargetDirectory=/home/bozo/projects/GreatAmericanNovel

cd $TargetDirectory
echo "Deleting stale files in $TargetDirectory."

if [ "$PWD" != "$TargetDirectory" ]
then    # Keep from wiping out wrong directory by accident.
```

```
  echo "Wrong directory!"
  echo "In $PWD, rather than $TargetDirectory!"
  echo "Bailing out!"
  exit $E_WRONG_DIRECTORY
fi

rm -rf *
rm .[A-Za-z0-9]*    # Delete dotfiles.
# rm -f .[^.]* ..?*   to remove filenames beginning with multiple dots.
# (shopt -s dotglob; rm -f *)   will also work.
# Thanks, S.C. for pointing this out.

#  A filename (`basename`) may contain all characters in the 0 - 255 range,
#+ except "/".
#  Deleting files beginning with weird characters, such as -
#+ is left as an exercise. (Hint: rm ./-weirdname or rm -- -weirdname)
result=$?   # Result of delete operations. If successful = 0.

echo
ls -al              # Any files left?
echo "Done."
echo "Old files deleted in $TargetDirectory."
echo

# Various other operations here, as necessary.

exit $result
```

$REPLY

The default value when a variable is not supplied to read. Also applicable to select menus, but only supplies the item number of the variable chosen, not the value of the variable itself.

```
#!/bin/bash
# reply.sh

# REPLY is the default value for a 'read' command.

echo
echo -n "What is your favorite vegetable? "
read

echo "Your favorite vegetable is $REPLY."
#  REPLY holds the value of last "read" if and only if
#+ no variable supplied.

echo
echo -n "What is your favorite fruit? "
read fruit
echo "Your favorite fruit is $fruit."
echo "but..."
echo "Value of \$REPLY is still $REPLY."
#  $REPLY is still set to its previous value because
#+ the variable $fruit absorbed the new "read" value.

echo

exit 0
```

$SECONDS

The number of seconds the script has been running.

```
#!/bin/bash
```

```
TIME_LIMIT=10
INTERVAL=1

echo
echo "Hit Control-C to exit before $TIME_LIMIT seconds."
echo

while [ "$SECONDS" -le "$TIME_LIMIT" ]
do   #   $SECONDS is an internal shell variable.
  if [ "$SECONDS" -eq 1 ]
  then
    units=second
  else
    units=seconds
  fi

  echo "This script has been running $SECONDS $units."
  #  On a slow or overburdened machine, the script may skip a count
  #+ every once in a while.
  sleep $INTERVAL
done

echo -e "\a"  # Beep!

exit 0
```

`$SHELLOPTS`

The list of enabled shell options, a readonly variable.

```
bash$ echo $SHELLOPTS
braceexpand:hashall:histexpand:monitor:history:interactive-comments:emacs
```

`$SHLVL`

Shell level, how deeply Bash is nested. [43] If, at the command-line, $SHLVL is 1, then in a script it will increment to 2.

This variable is *not* affected by subshells. Use $BASH_SUBSHELL when you need an indication of subshell nesting.

`$TMOUT`

If the *$TMOUT* environmental variable is set to a non-zero value `time`, then the shell prompt will time out after `$time` seconds. This will cause a logout.

As of version 2.05b of Bash, it is now possible to use *$TMOUT* in a script in combination with read.

```
# Works in scripts for Bash, versions 2.05b and later.

TMOUT=3    # Prompt times out at three seconds.

echo "What is your favorite song?"
echo "Quickly now, you only have $TMOUT seconds to answer!"
read song

if [ -z "$song" ]
then
  song="(no answer)"
  # Default response.
fi
```

```
echo "Your favorite song is $song."
```

There are other, more complex, ways of implementing timed input in a script. One alternative is to set up a timing loop to signal the script when it times out. This also requires a signal handling routine to trap (see Example 32-5) the interrupt generated by the timing loop (whew!).

Example 9-2. Timed Input

```
#!/bin/bash
# timed-input.sh

# TMOUT=3    Also works, as of newer versions of Bash.

TIMER_INTERRUPT=14
TIMELIMIT=3  # Three seconds in this instance.
             # May be set to different value.

PrintAnswer()
{
  if [ "$answer" = TIMEOUT ]
  then
    echo $answer
  else       # Don't want to mix up the two instances.
    echo "Your favorite veggie is $answer"
    kill $!  #  Kills no-longer-needed TimerOn function
             #+ running in background.
             #  $! is PID of last job running in background.
  fi

}


TimerOn()
{
  sleep $TIMELIMIT && kill -s 14 $$ &
  # Waits 3 seconds, then sends sigalarm to script.
}


Int14Vector()
{
  answer="TIMEOUT"
  PrintAnswer
  exit $TIMER_INTERRUPT
}

trap Int14Vector $TIMER_INTERRUPT
# Timer interrupt (14) subverted for our purposes.

echo "What is your favorite vegetable "
TimerOn
read answer
PrintAnswer


#  Admittedly, this is a kludgy implementation of timed input.
#  However, the "-t" option to "read" simplifies this task.
#  See the "t-out.sh" script.
#  However, what about timing not just single user input,
```

```
#+ but an entire script?

#  If you need something really elegant ...
#+ consider writing the application in C or C++,
#+ using appropriate library functions, such as 'alarm' and 'setitimer.'

exit 0
```

An alternative is using stty.

Example 9-3. Once more, timed input

```
#!/bin/bash
# timeout.sh

#  Written by Stephane Chazelas,
#+ and modified by the document author.

INTERVAL=5                # timeout interval

timedout_read() {
  timeout=$1
  varname=$2
  old_tty_settings=`stty -g`
  stty -icanon min 0 time ${timeout}0
  eval read $varname      # or just  read $varname
  stty "$old_tty_settings"
  # See man page for "stty."
}

echo; echo -n "What's your name? Quick! "
timedout_read $INTERVAL your_name

#  This may not work on every terminal type.
#  The maximum timeout depends on the terminal.
#+ (it is often 25.5 seconds).

echo

if [ ! -z "$your_name" ]  # If name input before timeout ...
then
  echo "Your name is $your_name."
else
  echo "Timed out."
fi

echo

# The behavior of this script differs somewhat from "timed-input.sh."
# At each keystroke, the counter resets.

exit 0
```

Perhaps the simplest method is using the `-t` option to read.

Example 9-4. Timed *read*

```
#!/bin/bash
```

```
# t-out.sh [time-out]
# Inspired by a suggestion from "syngin seven" (thanks).


TIMELIMIT=4         # 4 seconds

read -t $TIMELIMIT variable <&1
#                           ^^^
#  In this instance, "<&1" is needed for Bash 1.x and 2.x,
#  but unnecessary for Bash 3+.

echo

if [ -z "$variable" ]  # Is null?
then
  echo "Timed out, variable still unset."
else
  echo "variable = $variable"
fi

exit 0
```

`$UID`

User ID number

Current user's user identification number, as recorded in /etc/passwd

This is the current user's real id, even if she has temporarily assumed another identity through su. $UID is a readonly variable, not subject to change from the command line or within a script, and is the counterpart to the id builtin.

Example 9-5. Am I root?

```
#!/bin/bash
# am-i-root.sh:   Am I root or not?

ROOT_UID=0   # Root has $UID 0.

if [ "$UID" -eq "$ROOT_UID" ]  # Will the real "root" please stand up?
then
  echo "You are root."
else
  echo "You are just an ordinary user (but mom loves you just the same)."
fi

exit 0


# ============================================================= #
# Code below will not execute, because the script already exited.

# An alternate method of getting to the root of matters:

ROOTUSER_NAME=root

username=`id -nu`              # Or...   username=`whoami`
if [ "$username" = "$ROOTUSER_NAME" ]
then
  echo "Rooty, toot, toot. You are root."
```

```
else
  echo "You are just a regular fella."
fi
```

See also Example 2-3.

The variables `$ENV`, `$LOGNAME`, `$MAIL`, `$TERM`, `$USER`, and `$USERNAME` are *not* Bash builtins. These are, however, often set as environmental variables in one of the Bash or *login* startup files. `$SHELL`, the name of the user's login shell, may be set from `/etc/passwd` or in an "init" script, and it is likewise not a Bash builtin.

```
tcsh% echo $LOGNAME
bozo
tcsh% echo $SHELL
/bin/tcsh
tcsh% echo $TERM
rxvt

bash$ echo $LOGNAME
bozo
bash$ echo $SHELL
/bin/tcsh
bash$ echo $TERM
rxvt
```

Positional Parameters

`$0`, `$1`, `$2`, etc.
: Positional parameters, passed from command line to script, passed to a function, or set to a variable (see Example 4-5 and Example 15-16)

`$#`
: Number of command-line arguments [44] or positional parameters (see Example 36-2)

`$*`
: All of the positional parameters, seen as a single word

"`$*`" must be quoted.

`$@`
: Same as $*, but each parameter is a quoted string, that is, the parameters are passed on intact, without interpretation or expansion. This means, among other things, that each parameter in the argument list is seen as a separate word.

Of course, "`$@`" should be quoted.

Example 9-6. *arglist*: Listing arguments with $* and $@

```
#!/bin/bash
# arglist.sh
# Invoke this script with several arguments, such as "one two three" ...

E_BADARGS=85

if [ ! -n "$1" ]
then
  echo "Usage: `basename $0` argument1 argument2 etc."
```

```
  exit $E_BADARGS
fi

echo

index=1          # Initialize count.

echo "Listing args with \"\$*\":"
for arg in "$*"  # Doesn't work properly if "$*" isn't quoted.
do
  echo "Arg #$index = $arg"
  let "index+=1"
done             # $* sees all arguments as single word.
echo "Entire arg list seen as single word."

echo

index=1          # Reset count.
                 # What happens if you forget to do this?

echo "Listing args with \"\$@\":"
for arg in "$@"
do
  echo "Arg #$index = $arg"
  let "index+=1"
done             # $@ sees arguments as separate words.
echo "Arg list seen as separate words."

echo

index=1          # Reset count.

echo "Listing args with \$* (unquoted):"
for arg in $*
do
  echo "Arg #$index = $arg"
  let "index+=1"
done             # Unquoted $* sees arguments as separate words.
echo "Arg list seen as separate words."

exit 0
```

Following a **shift**, the `$@` holds the remaining command-line parameters, lacking the previous `$1`, which was lost.

```
#!/bin/bash
# Invoke with ./scriptname 1 2 3 4 5

echo "$@"    # 1 2 3 4 5
shift
echo "$@"    # 2 3 4 5
shift
echo "$@"    # 3 4 5

# Each "shift" loses parameter $1.
# "$@" then contains the remaining parameters.
```

The `$@` special parameter finds use as a tool for filtering input into shell scripts. The **cat "$@"** construction accepts input to a script either from `stdin` or from files given as parameters to the script. See Example 16-24 and Example 16-25.

The `$*` and `$@` parameters sometimes display inconsistent and puzzling behavior, depending on the setting of $IFS.

Example 9-7. Inconsistent `$*` and `$@` behavior

```
#!/bin/bash

#  Erratic behavior of the "$*" and "$@" internal Bash variables,
#+ depending on whether or not they are quoted.
#  Demonstrates inconsistent handling of word splitting and linefeeds.


set -- "First one" "second" "third:one" "" "Fifth: :one"
# Setting the script arguments, $1, $2, $3, etc.

echo

echo 'IFS unchanged, using "$*"'
c=0
for i in "$*"               # quoted
do echo "$((c+=1)): [$i]"   # This line remains the same in every instance.
                            # Echo args.
done
echo ---

echo 'IFS unchanged, using $*'
c=0
for i in $*                 # unquoted
do echo "$((c+=1)): [$i]"
done
echo ---

echo 'IFS unchanged, using "$@"'
c=0
for i in "$@"
do echo "$((c+=1)): [$i]"
done
echo ---

echo 'IFS unchanged, using $@'
c=0
for i in $@
do echo "$((c+=1)): [$i]"
done
echo ---

IFS=:
echo 'IFS=":", using "$*"'
c=0
for i in "$*"
do echo "$((c+=1)): [$i]"
done
echo ---

echo 'IFS=":", using $*'
c=0
for i in $*
do echo "$((c+=1)): [$i]"
done
echo ---
```

```
var=$*
echo 'IFS=":", using "$var" (var=$*)'
c=0
for i in "$var"
do echo "$((c+=1)): [$i]"
done
echo ---

echo 'IFS=":", using $var (var=$*)'
c=0
for i in $var
do echo "$((c+=1)): [$i]"
done
echo ---

var="$*"
echo 'IFS=":", using $var (var="$*")'
c=0
for i in $var
do echo "$((c+=1)): [$i]"
done
echo ---

echo 'IFS=":", using "$var" (var="$*")'
c=0
for i in "$var"
do echo "$((c+=1)): [$i]"
done
echo ---

echo 'IFS=":", using "$@"'
c=0
for i in "$@"
do echo "$((c+=1)): [$i]"
done
echo ---

echo 'IFS=":", using $@'
c=0
for i in $@
do echo "$((c+=1)): [$i]"
done
echo ---

var=$@
echo 'IFS=":", using $var (var=$@)'
c=0
for i in $var
do echo "$((c+=1)): [$i]"
done
echo ---

echo 'IFS=":", using "$var" (var=$@)'
c=0
for i in "$var"
do echo "$((c+=1)): [$i]"
done
echo ---

var="$@"
echo 'IFS=":", using "$var" (var="$@")'
c=0
```

```
for i in "$var"
do echo "$((c+=1)): [$i]"
done
echo ---

echo 'IFS=":", using $var (var="$@")'
c=0
for i in $var
do echo "$((c+=1)): [$i]"
done

echo

# Try this script with ksh or zsh -y.

exit 0

#  This example script written by Stephane Chazelas,
#+ and slightly modified by the document author.
```

The **$@** and **$*** parameters differ only when between double quotes.

Example 9-8. `$*` and `$@` when `$IFS` is empty

```
#!/bin/bash

#  If $IFS set, but empty,
#+ then "$*" and "$@" do not echo positional params as expected.

mecho ()       # Echo positional parameters.
{
echo "$1,$2,$3";
}


IFS=""         # Set, but empty.
set a b c      # Positional parameters.

mecho "$*"     # abc,,
#                   ^^
mecho $*       # a,b,c

mecho $@       # a,b,c
mecho "$@"     # a,b,c

#  The behavior of $* and $@ when $IFS is empty depends
#+ on which Bash or sh version being run.
#  It is therefore inadvisable to depend on this "feature" in a script.


# Thanks, Stephane Chazelas.

exit
```

Other Special Parameters

`$-`

Flags passed to script (using set). See Example 15-16.

This was originally a *ksh* construct adopted into Bash, and unfortunately it does not seem to work reliably in Bash scripts. One possible use for it is to have a script self-test whether it is interactive.

`$!`

PID (process ID) of last job run in background

```
LOG=$0.log

COMMAND1="sleep 100"

echo "Logging PIDs background commands for script: $0" >> "$LOG"
# So they can be monitored, and killed as necessary.
echo >> "$LOG"

# Logging commands.

echo -n "PID of \"$COMMAND1\":  " >> "$LOG"
${COMMAND1} &
echo $! >> "$LOG"
# PID of "sleep 100":  1506

# Thank you, Jacques Lederer, for suggesting this.
```

Using `$!` for job control:

```
possibly_hanging_job & { sleep ${TIMEOUT}; eval 'kill -9 $!' &> /dev/null; }
# Forces completion of an ill-behaved program.
# Useful, for example, in init scripts.

# Thank you, Sylvain Fourmanoit, for this creative use of the "!" variable.
```

Or, alternately:

```
# This example by Matthew Sage.
# Used with permission.

TIMEOUT=30   # Timeout value in seconds
count=0

possibly_hanging_job & {
        while ((count < TIMEOUT )); do
                eval '[ ! -d "/proc/$!" ] && ((count = TIMEOUT))'
                # /proc is where information about running processes is found.
                # "-d" tests whether it exists (whether directory exists).
                # So, we're waiting for the job in question to show up.
                ((count++))
                sleep 1
        done
        eval '[ -d "/proc/$!" ] && kill -15 $!'
        # If the hanging job is running, kill it.
}

#  -------------------------------------------------------------- #

#  However, this may not not work as specified if another process
#+ begins to run after the "hanging_job" . . .
#  In such a case, the wrong job may be killed.
#  Ariel Meragelman suggests the following fix.

TIMEOUT=30
count=0
```

```
# Timeout value in seconds
possibly_hanging_job & {

while ((count < TIMEOUT )); do
  eval '[ ! -d "/proc/$lastjob" ] && ((count = TIMEOUT))'
  lastjob=$!
  ((count++))
  sleep 1
done
eval '[ -d "/proc/$lastjob" ] && kill -15 $lastjob'

}

exit
```

`$_`
: Special variable set to final argument of previous command executed.

Example 9-9. Underscore variable

```
#!/bin/bash

echo $_              #  /bin/bash
                     #  Just called /bin/bash to run the script.
                     #  Note that this will vary according to
                     #+ how the script is invoked.

du >/dev/null        #  So no output from command.
echo $_              #  du

ls -al >/dev/null    #  So no output from command.
echo $_              #  -al  (last argument)

:
echo $_              #  :
```

`$?`
: Exit status of a command, function, or the script itself (see Example 24-7)

`$$`
: Process ID (*PID*) of the script itself. [45] The `$$` variable often finds use in scripts to construct "unique" temp file names (see Example 32-6, Example 16-31, and Example 15-27). This is usually simpler than invoking mktemp.

9.2. Typing variables: declare or typeset

The *declare* or *typeset* builtins, which are exact synonyms, permit modifying the properties of variables. This is a very weak form of the *typing* [46] available in certain programming languages. The *declare* command is specific to version 2 or later of Bash. The *typeset* command also works in ksh scripts.

declare/typeset options

-r *readonly*
: (**declare -r var1** works the same as **readonly var1**)

This is the rough equivalent of the **C** *const* type qualifier. An attempt to change the value of a *readonly* variable fails with an error message.

```
declare -r var1=1
echo "var1 = $var1"   # var1 = 1

(( var1++ ))          # x.sh: line 4: var1: readonly variable
```

-i *integer*

```
declare -i number
# The script will treat subsequent occurrences of "number" as an integer.

number=3
echo "Number = $number"     # Number = 3

number=three
echo "Number = $number"     # Number = 0
# Tries to evaluate the string "three" as an integer.
```

Certain arithmetic operations are permitted for declared integer variables without the need for expr or let.

```
n=6/3
echo "n = $n"       # n = 6/3

declare -i n
n=6/3
echo "n = $n"       # n = 2
```

-a *array*

```
declare -a indices
```

The variable *indices* will be treated as an array.

-f *function(s)*

```
declare -f
```

A **declare -f** line with no arguments in a script causes a listing of all the functions previously defined in that script.

```
declare -f function_name
```

A **declare -f function_name** in a script lists just the function named.

-x export

```
declare -x var3
```

This declares a variable as available for exporting outside the environment of the script itself.

-x var=$value

```
declare -x var3=373
```

The **declare** command permits assigning a value to a variable in the same statement as setting its properties.

Example 9-10. Using *declare* to type variables

```
#!/bin/bash
```

```
func1 ()
{
  echo This is a function.
}

declare -f        # Lists the function above.

echo

declare -i var1   # var1 is an integer.
var1=2367
echo "var1 declared as $var1"
var1=var1+1       # Integer declaration eliminates the need for 'let'.
echo "var1 incremented by 1 is $var1."
# Attempt to change variable declared as integer.
echo "Attempting to change var1 to floating point value, 2367.1."
var1=2367.1       # Results in error message, with no change to variable.
echo "var1 is still $var1"

echo

declare -r var2=13.36         # 'declare' permits setting a variable property
                              #+ and simultaneously assigning it a value.
echo "var2 declared as $var2" # Attempt to change readonly variable.
var2=13.37                    # Generates error message, and exit from script.

echo "var2 is still $var2"    # This line will not execute.

exit 0                        # Script will not exit here.
```

Using the *declare* builtin restricts the scope of a variable.

```
foo ()
{
FOO="bar"
}

bar ()
{
foo
echo $FOO
}

bar   # Prints bar.
```

However . . .

```
foo (){
declare FOO="bar"
}

bar ()
{
foo
echo $FOO
}

bar  # Prints nothing.


# Thank you, Michael Iatrou, for pointing this out.
```

9.2.1. Another use for *declare*

The *declare* command can be helpful in identifying variables, environmental or otherwise. This can be especially useful with arrays.

```
bash$ declare | grep HOME
HOME=/home/bozo


bash$ zzy=68
bash$ declare | grep zzy
zzy=68


bash$ Colors=([0]="purple" [1]="reddish-orange" [2]="light green")
bash$ echo ${Colors[@]}
purple reddish-orange light green
bash$ declare | grep Colors
Colors=([0]="purple" [1]="reddish-orange" [2]="light green")
```

9.3. $RANDOM: generate random integer

Anyone who attempts to generate random numbers by deterministic means is, of course, living in a state of sin.

--John von Neumann

`$RANDOM` is an internal Bash function (not a constant) that returns a *pseudorandom* [47] integer in the range 0 - 32767. It should *not* be used to generate an encryption key.

Example 9-11. Generating random numbers

```
#!/bin/bash

# $RANDOM returns a different random integer at each invocation.
# Nominal range: 0 - 32767 (signed 16-bit integer).

MAXCOUNT=10
count=1

echo
echo "$MAXCOUNT random numbers:"
echo "-----------------"
while [ "$count" -le $MAXCOUNT ]      # Generate 10 ($MAXCOUNT) random integers.
do
  number=$RANDOM
  echo $number
  let "count += 1"  # Increment count.
done
echo "-----------------"

# If you need a random int within a certain range, use the 'modulo' operator.
```

```
# This returns the remainder of a division operation.

RANGE=500

echo

number=$RANDOM
let "number %= $RANGE"
#           ^^
echo "Random number less than $RANGE  ---  $number"

echo



#  If you need a random integer greater than a lower bound,
#+ then set up a test to discard all numbers below that.

FLOOR=200

number=0   #initialize
while [ "$number" -le $FLOOR ]
do
  number=$RANDOM
done
echo "Random number greater than $FLOOR ---  $number"
echo

   # Let's examine a simple alternative to the above loop, namely
   #       let "number = $RANDOM + $FLOOR"
   # That would eliminate the while-loop and run faster.
   # But, there might be a problem with that. What is it?



# Combine above two techniques to retrieve random number between two limits.
number=0   #initialize
while [ "$number" -le $FLOOR ]
do
  number=$RANDOM
  let "number %= $RANGE"  # Scales $number down within $RANGE.
done
echo "Random number between $FLOOR and $RANGE ---  $number"
echo



# Generate binary choice, that is, "true" or "false" value.
BINARY=2
T=1
number=$RANDOM

let "number %= $BINARY"
#  Note that    let "number >>= 14"    gives a better random distribution
#+ (right shifts out everything except last binary digit).
if [ "$number" -eq $T ]
then
  echo "TRUE"
else
  echo "FALSE"
fi
```

```
echo


# Generate a toss of the dice.
SPOTS=6   # Modulo 6 gives range 0 - 5.
          # Incrementing by 1 gives desired range of 1 - 6.
          # Thanks, Paulo Marcel Coelho Aragao, for the simplification.
die1=0
die2=0
# Would it be better to just set SPOTS=7 and not add 1? Why or why not?

# Tosses each die separately, and so gives correct odds.

    let "die1 = $RANDOM % $SPOTS +1" # Roll first one.
    let "die2 = $RANDOM % $SPOTS +1" # Roll second one.
    #  Which arithmetic operation, above, has greater precedence --
    #+ modulo (%) or addition (+)?


let "throw = $die1 + $die2"
echo "Throw of the dice = $throw"
echo


exit 0
```

Example 9-12. Picking a random card from a deck

```
#!/bin/bash
# pick-card.sh

# This is an example of choosing random elements of an array.


# Pick a card, any card.

Suites="Clubs
Diamonds
Hearts
Spades"

Denominations="2
3
4
5
6
7
8
9
10
Jack
Queen
King
Ace"

# Note variables spread over multiple lines.


suite=($Suites)                # Read into array variable.
denomination=($Denominations)

```

```
num_suites=${#suite[*]}        # Count how many elements.
num_denominations=${#denomination[*]}

echo -n "${denomination[$((RANDOM%num_denominations))]} of "
echo ${suite[$((RANDOM%num_suites))]}


# $bozo sh pick-cards.sh
# Jack of Clubs


# Thank you, "jipe," for pointing out this use of $RANDOM.
exit 0
```

Example 9-13. Brownian Motion Simulation

```
#!/bin/bash
# brownian.sh
# Author: Mendel Cooper
# Reldate: 10/26/07
# License: GPL3

#  ----------------------------------------------------------------
#  This script models Brownian motion:
#+ the random wanderings of tiny particles in a fluid,
#+ as they are buffeted by random currents and collisions.
#+ This is colloquially known as the "Drunkard's Walk."

#  It can also be considered as a stripped-down simulation of a
#+ Galton Board, a slanted board with a pattern of pegs,
#+ down which rolls a succession of marbles, one at a time.
#+ At the bottom is a row of slots or catch basins in which
#+ the marbles come to rest at the end of their journey.
#  Think of it as a kind of bare-bones Pachinko game.
#  As you see by running the script,
#+ most of the marbles cluster around the center slot.
#+ This is consistent with the expected binomial distribution.
#  As a Galton Board simulation, the script
#+ disregards such parameters as
#+ board tilt-angle, rolling friction of the marbles,
#+ angles of impact, and elasticity of the pegs.
#  To what extent does this affect the accuracy of the simulation?
#  ----------------------------------------------------------------

PASSES=500            #  Number of particle interactions / marbles.
ROWS=10               #  Number of "collisions" (or horiz. peg rows).
RANGE=3               #  0 - 2 output range from $RANDOM.
POS=0                 #  Left/right position.
RANDOM=$$             #  Seeds the random number generator from PID
                      #+ of script.

declare -a Slots      # Array holding cumulative results of passes.
NUMSLOTS=21           # Number of slots at bottom of board.


Initialize_Slots () { # Zero out all elements of the array.
for i in $( seq $NUMSLOTS )
do
  Slots[$i]=0
done
```

```
echo                  # Blank line at beginning of run.
  }


Show_Slots () {
echo; echo
echo -n " "
for i in $( seq $NUMSLOTS )   # Pretty-print array elements.
do
  printf "%3d" ${Slots[$i]}   # Allot three spaces per result.
done

echo # Row of slots:
echo " |__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|"
echo "                                ||"
echo #  Note that if the count within any particular slot exceeds 99,
     #+ it messes up the display.
     #  Running only(!) 500 passes usually avoids this.
  }


Move () {              # Move one unit right / left, or stay put.
  Move=$RANDOM         # How random is $RANDOM? Well, let's see ...
  let "Move %= RANGE"  # Normalize into range of 0 - 2.
  case "$Move" in
    0 ) ;;                   # Do nothing, i.e., stay in place.
    1 ) ((POS--));;          # Left.
    2 ) ((POS++));;          # Right.
    * ) echo -n "Error ";;   # Anomaly! (Should never occur.)
  esac
  }


Play () {                    # Single pass (inner loop).
i=0
while [ "$i" -lt "$ROWS" ]   # One event per row.
do
  Move
  ((i++));
done

SHIFT=11                     # Why 11, and not 10?
let "POS += $SHIFT"          # Shift "zero position" to center.
(( Slots[$POS]++ ))          # DEBUG: echo $POS

# echo -n "$POS "

  }


Run () {                     # Outer loop.
p=0
while [ "$p" -lt "$PASSES" ]
do
  Play
  (( p++ ))
  POS=0                      # Reset to zero. Why?
done
  }
```

```
# --------------
# main ()
Initialize_Slots
Run
Show_Slots
# --------------

exit $?

#  Exercises:
#  ---------
#  1) Show the results in a vertical bar graph, or as an alternative,
#+    a scattergram.
#  2) Alter the script to use /dev/urandom instead of $RANDOM.
#     Will this make the results more random?
#  3) Provide some sort of "animation" or graphic output
#     for each marble played.
```

Jipe points out a set of techniques for generating random numbers within a range.

```
#  Generate random number between 6 and 30.
   rnumber=$((RANDOM%25+6))

#  Generate random number in the same 6 - 30 range,
#+ but the number must be evenly divisible by 3.
   rnumber=$(((RANDOM%30/3+1)*3))

#  Note that this will not work all the time.
#  It fails if $RANDOM%30 returns 0.

#  Frank Wang suggests the following alternative:
   rnumber=$(( RANDOM%27/3*3+6 ))
```

Bill Gradwohl came up with an improved formula that works for positive numbers.

```
rnumber=$(((RANDOM%(max-min+divisibleBy))/divisibleBy*divisibleBy+min))
```

Here Bill presents a versatile function that returns a random number between two specified values.

Example 9-14. Random between values

```
#!/bin/bash
# random-between.sh
# Random number between two specified values.
# Script by Bill Gradwohl, with minor modifications by the document author.
# Corrections in lines 187 and 189 by Anthony Le Clezio.
# Used with permission.


randomBetween() {
   #  Generates a positive or negative random number
   #+ between $min and $max
   #+ and divisible by $divisibleBy.
   #  Gives a "reasonably random" distribution of return values.
   #
   #  Bill Gradwohl - Oct 1, 2003

   syntax() {
   # Function embedded within function.
      echo
      echo    "Syntax: randomBetween [min] [max] [multiple]"
```

```
        echo
        echo -n "Expects up to 3 passed parameters, "
        echo    "but all are completely optional."
        echo    "min is the minimum value"
        echo    "max is the maximum value"
        echo -n "multiple specifies that the answer must be "
        echo     "a multiple of this value."
        echo    "    i.e. answer must be evenly divisible by this number."
        echo
        echo    "If any value is missing, defaults area supplied as: 0 32767 1"
        echo -n "Successful completion returns 0, "
        echo     "unsuccessful completion returns"
        echo    "function syntax and 1."
        echo -n "The answer is returned in the global variable "
        echo    "randomBetweenAnswer"
        echo -n "Negative values for any passed parameter are "
        echo    "handled correctly."
    }

    local min=${1:-0}
    local max=${2:-32767}
    local divisibleBy=${3:-1}
    # Default values assigned, in case parameters not passed to function.

    local x
    local spread

    # Let's make sure the divisibleBy value is positive.
    [ ${divisibleBy} -lt 0 ] && divisibleBy=$((0-divisibleBy))

    # Sanity check.
    if [ $# -gt 3 -o ${divisibleBy} -eq 0 -o  ${min} -eq ${max} ]; then
       syntax
       return 1
    fi

    # See if the min and max are reversed.
    if [ ${min} -gt ${max} ]; then
       # Swap them.
       x=${min}
       min=${max}
       max=${x}
    fi

    #  If min is itself not evenly divisible by $divisibleBy,
    #+ then fix the min to be within range.
    if [ $((min/divisibleBy*divisibleBy)) -ne ${min} ]; then
       if [ ${min} -lt 0 ]; then
          min=$((min/divisibleBy*divisibleBy))
       else
          min=$((((min/divisibleBy)+1)*divisibleBy))
       fi
    fi

    #  If max is itself not evenly divisible by $divisibleBy,
    #+ then fix the max to be within range.
    if [ $((max/divisibleBy*divisibleBy)) -ne ${max} ]; then
       if [ ${max} -lt 0 ]; then
          max=$((((max/divisibleBy)-1)*divisibleBy))
       else
          max=$((max/divisibleBy*divisibleBy))
       fi
```

```
   fi

   #  ----------------------------------------------------------------------
   #  Now, to do the real work.

   #  Note that to get a proper distribution for the end points,
   #+ the range of random values has to be allowed to go between
   #+ 0 and abs(max-min)+divisibleBy, not just abs(max-min)+1.

   #  The slight increase will produce the proper distribution for the
   #+ end points.

   #  Changing the formula to use abs(max-min)+1 will still produce
   #+ correct answers, but the randomness of those answers is faulty in
   #+ that the number of times the end points ($min and $max) are returned
   #+ is considerably lower than when the correct formula is used.
   #  ----------------------------------------------------------------------

   spread=$((max-min))
   #  Omair Eshkenazi points out that this test is unnecessary,
   #+ since max and min have already been switched around.
   [ ${spread} -lt 0 ] && spread=$((0-spread))
   let spread+=divisibleBy
   randomBetweenAnswer=$(((RANDOM%spread)/divisibleBy*divisibleBy+min))

   return 0

   #  However, Paulo Marcel Coelho Aragao points out that
   #+ when $max and $min are not divisible by $divisibleBy,
   #+ the formula fails.
   #
   #  He suggests instead the following formula:
   #    rnumber = $(((RANDOM%(max-min+1)+min)/divisibleBy*divisibleBy))

}

# Let's test the function.
min=-14
max=20
divisibleBy=3


#  Generate an array of expected answers and check to make sure we get
#+ at least one of each answer if we loop long enough.

declare -a answer
minimum=${min}
maximum=${max}
   if [ $((minimum/divisibleBy*divisibleBy)) -ne ${minimum} ]; then
      if [ ${minimum} -lt 0 ]; then
         minimum=$((minimum/divisibleBy*divisibleBy))
      else
         minimum=$((((minimum/divisibleBy)+1)*divisibleBy))
      fi
   fi


   #  If max is itself not evenly divisible by $divisibleBy,
   #+ then fix the max to be within range.

   if [ $((maximum/divisibleBy*divisibleBy)) -ne ${maximum} ]; then
      if [ ${maximum} -lt 0 ]; then
```

```
         maximum=$((((maximum/divisibleBy)-1)*divisibleBy))
      else
         maximum=$((maximum/divisibleBy*divisibleBy))
      fi
   fi


#  We need to generate only positive array subscripts,
#+ so we need a displacement that that will guarantee
#+ positive results.

disp=$((0-minimum))
for ((i=${minimum}; i<=${maximum}; i+=divisibleBy)); do
   answer[i+disp]=0
done


# Now loop a large number of times to see what we get.
loopIt=1000   #  The script author suggests 100000,
              #+ but that takes a good long while.

for ((i=0; i<${loopIt}; ++i)); do

   #  Note that we are specifying min and max in reversed order here to
   #+ make the function correct for this case.

   randomBetween ${max} ${min} ${divisibleBy}

   # Report an error if an answer is unexpected.
   [ ${randomBetweenAnswer} -lt ${min} -o ${randomBetweenAnswer} -gt ${max} ] \
   && echo MIN or MAX error - ${randomBetweenAnswer}!
   [ $((randomBetweenAnswer%${divisibleBy})) -ne 0 ] \
   && echo DIVISIBLE BY error - ${randomBetweenAnswer}!

   # Store the answer away statistically.
   answer[randomBetweenAnswer+disp]=$((answer[randomBetweenAnswer+disp]+1))
done



# Let's check the results

for ((i=${minimum}; i<=${maximum}; i+=divisibleBy)); do
   [ ${answer[i+disp]} -eq 0 ] \
   && echo "We never got an answer of $i." \
   || echo "${i} occurred ${answer[i+disp]} times."
done


exit 0
```

Just how random is `$RANDOM`? The best way to test this is to write a script that tracks the distribution of "random" numbers generated by `$RANDOM`. Let's roll a `$RANDOM` die a few times . . .

Example 9-15. Rolling a single die with RANDOM

```
#!/bin/bash
# How random is RANDOM?

RANDOM=$$       # Reseed the random number generator using script process ID.
```

```
PIPS=6          # A die has 6 pips.
MAXTHROWS=600   # Increase this if you have nothing better to do with your time.
throw=0         # Number of times the dice have been cast.

ones=0          #  Must initialize counts to zero,
twos=0          #+ since an uninitialized variable is null, NOT zero.
threes=0
fours=0
fives=0
sixes=0

print_result ()
{
echo
echo "ones =   $ones"
echo "twos =   $twos"
echo "threes = $threes"
echo "fours =  $fours"
echo "fives =  $fives"
echo "sixes =  $sixes"
echo
}

update_count()
{
case "$1" in
  0) ((ones++));;   # Since a die has no "zero", this corresponds to 1.
  1) ((twos++));;   # And this to 2.
  2) ((threes++));; # And so forth.
  3) ((fours++));;
  4) ((fives++));;
  5) ((sixes++));;
esac
}

echo


while [ "$throw" -lt "$MAXTHROWS" ]
do
  let "die1 = RANDOM % $PIPS"
  update_count $die1
  let "throw += 1"
done

print_result

exit $?

#  The scores should distribute evenly, assuming RANDOM is random.
#  With $MAXTHROWS at 600, all should cluster around 100,
#+ plus-or-minus 20 or so.
#
#  Keep in mind that RANDOM is a ***pseudorandom*** generator,
#+ and not a spectacularly good one at that.

#  Randomness is a deep and complex subject.
#  Sufficiently long "random" sequences may exhibit
#+ chaotic and other "non-random" behavior.

# Exercise (easy):
```

```
# ---------------
# Rewrite this script to flip a coin 1000 times.
# Choices are "HEADS" and "TAILS."
```

As we have seen in the last example, it is best to *reseed* the `RANDOM` generator each time it is invoked. Using the same seed for `RANDOM` repeats the same series of numbers. [48] (This mirrors the behavior of the `random()` function in *C*.)

Example 9-16. Reseeding RANDOM

```
#!/bin/bash
# seeding-random.sh: Seeding the RANDOM variable.
# v 1.1, reldate 09 Feb 2013

MAXCOUNT=25       # How many numbers to generate.
SEED=

random_numbers ()
{
local count=0
local number

while [ "$count" -lt "$MAXCOUNT" ]
do
  number=$RANDOM
  echo -n "$number "
  let "count++"
done
}

echo; echo

SEED=1
RANDOM=$SEED      # Setting RANDOM seeds the random number generator.
echo "Random seed = $SEED"
random_numbers


RANDOM=$SEED      # Same seed for RANDOM . . .
echo; echo "Again, with same random seed ..."
echo "Random seed = $SEED"
random_numbers    # . . . reproduces the exact same number series.
                  #
                  # When is it useful to duplicate a "random" series?

echo; echo

SEED=2
RANDOM=$SEED      # Trying again, but with a different seed . . .
echo "Random seed = $SEED"
random_numbers    # . . . gives a different number series.

echo; echo

# RANDOM=$$  seeds RANDOM from process id of script.
# It is also possible to seed RANDOM from 'time' or 'date' commands.

# Getting fancy...
SEED=$(head -1 /dev/urandom | od -N 1 | awk '{ print $2 }'| sed s/^0*//)
#  Pseudo-random output fetched
```

```
#+ from /dev/urandom (system pseudo-random device-file),
#+ then converted to line of printable (octal) numbers by "od",
#+ then "awk" retrieves just one number for SEED,
#+ finally "sed" removes any leading zeros.
RANDOM=$SEED
echo "Random seed = $SEED"
random_numbers

echo; echo

exit 0
```

The `/dev/urandom` pseudo-device file provides a method of generating much more "random" pseudorandom numbers than the `$RANDOM` variable. **`dd if=/dev/urandom of=targetfile bs=1 count=XX`** creates a file of well-scattered pseudorandom numbers. However, assigning these numbers to a variable in a script requires a workaround, such as filtering through od (as in above example, Example 16-14, and Example A-36), or even piping to md5sum (see Example 36-16).

There are also other ways to generate pseudorandom numbers in a script. **Awk** provides a convenient means of doing this.

Example 9-17. Pseudorandom numbers, using awk

```
#!/bin/bash
#  random2.sh: Returns a pseudorandom number in the range 0 - 1,
#+ to 6 decimal places. For example: 0.822725
#  Uses the awk rand() function.

AWKSCRIPT=' { srand(); print rand() } '
#            Command(s)/parameters passed to awk
# Note that srand() reseeds awk's random number generator.


echo -n "Random number between 0 and 1 = "

echo | awk "$AWKSCRIPT"
# What happens if you leave out the 'echo'?

exit 0


# Exercises:
# ---------

# 1) Using a loop construct, print out 10 different random numbers.
#      (Hint: you must reseed the srand() function with a different seed
#+     in each pass through the loop. What happens if you omit this?)

# 2) Using an integer multiplier as a scaling factor, generate random numbers
#+   in the range of 10 to 100.

# 3) Same as exercise #2, above, but generate random integers this time.
```

The date command also lends itself to generating pseudorandom integer sequences.

Chapter 10. Manipulating Variables

10.1. Manipulating Strings

Bash supports a surprising number of string manipulation operations. Unfortunately, these tools lack a unified focus. Some are a subset of parameter substitution, and others fall under the functionality of the UNIX expr command. This results in inconsistent command syntax and overlap of functionality, not to mention confusion.

String Length

${#string}
expr length $string
: These are the equivalent of *strlen()* in *C*.

expr "$string" : '.*'

```
stringZ=abcABC123ABCabc

echo ${#stringZ}                 # 15
echo `expr length $stringZ`      # 15
echo `expr "$stringZ" : '.*'`    # 15
```

Example 10-1. Inserting a blank line between paragraphs in a text file

```
#!/bin/bash
# paragraph-space.sh
# Ver. 2.1, Reldate 29Jul12 [fixup]

# Inserts a blank line between paragraphs of a single-spaced text file.
# Usage: $0 <FILENAME

MINLEN=60        # Change this value? It's a judgment call.
#  Assume lines shorter than $MINLEN characters ending in a period
#+ terminate a paragraph. See exercises below.

while read line  # For as many lines as the input file has ...
do
  echo "$line"   # Output the line itself.

  len=${#line}
  if [[ "$len" -lt "$MINLEN" && "$line" =~ [*{\.}]$ ]]
# if [[ "$len" -lt "$MINLEN" && "$line" =~ \[*\.\] ]]
# An update to Bash broke the previous version of this script. Ouch!
# Thank you, Halim Srama, for pointing this out and suggesting a fix.
    then echo    #  Add a blank line immediately
  fi             #+ after a short line terminated by a period.
done

exit

# Exercises:
# ---------
#  1) The script usually inserts a blank line at the end
```

```
#+    of the target file. Fix this.
#  2) Line 17 only considers periods as sentence terminators.
#     Modify this to include other common end-of-sentence characters,
#+    such as ?, !, and ".
```

Length of Matching Substring at Beginning of String

expr match "$string" '$substring'

$substring is a regular expression.

expr "$string" : '$substring'

$substring is a regular expression.

```
stringZ=abcABC123ABCabc
#       |------|
#       12345678

echo `expr match "$stringZ" 'abc[A-Z]*.2'`   # 8
echo `expr "$stringZ" : 'abc[A-Z]*.2'`       # 8
```

Index

expr index $string $substring

Numerical position in $string of first character in $substring that matches.

```
stringZ=abcABC123ABCabc
#       123456 ...
echo `expr index "$stringZ" C12`             # 6
                                             # C position.

echo `expr index "$stringZ" 1c`              # 3
# 'c' (in #3 position) matches before '1'.
```

This is the near equivalent of *strchr()* in *C*.

Substring Extraction

${string:position}

Extracts substring from *$string* at *$position*.

If the `$string` parameter is "*" or "@", then this extracts the positional parameters, [49] starting at `$position`.

${string:position:length}

Extracts *$length* characters of substring from *$string* at *$position*.

```
stringZ=abcABC123ABCabc
#       0123456789.....
#       0-based indexing.

echo ${stringZ:0}                            # abcABC123ABCabc
echo ${stringZ:1}                            # bcABC123ABCabc
echo ${stringZ:7}                            # 23ABCabc

echo ${stringZ:7:3}                          # 23A
                                             # Three characters of substring.
```

```
# Is it possible to index from the right end of the string?

echo ${stringZ:-4}                           # abcABC123ABCabc
# Defaults to full string, as in ${parameter:-default}.
# However . . .

echo ${stringZ:(-4)}                         # Cabc
echo ${stringZ: -4}                          # Cabc
# Now, it works.
# Parentheses or added space "escape" the position parameter.

# Thank you, Dan Jacobson, for pointing this out.
```

The *position* and *length* arguments can be "parameterized," that is, represented as a variable, rather than as a numerical constant.

Example 10-2. Generating an 8-character "random" string

```
#!/bin/bash
# rand-string.sh
# Generating an 8-character "random" string.

if [ -n "$1" ]  #  If command-line argument present,
then            #+ then set start-string to it.
  str0="$1"
else            #  Else use PID of script as start-string.
  str0="$$"
fi

POS=2  # Starting from position 2 in the string.
LEN=8  # Extract eight characters.

str1=$( echo "$str0" | md5sum | md5sum )
#  Doubly scramble     ^^^^^^   ^^^^^^
#+ by piping and repiping to md5sum.

randstring="${str1:$POS:$LEN}"
# Can parameterize ^^^^ ^^^^

echo "$randstring"

exit $?

# bozo$ ./rand-string.sh my-password
# 1bdd88c4

#  No, this is is not recommended
#+ as a method of generating hack-proof passwords.
```

If the `$string` parameter is "*" or "@", then this extracts a maximum of `$length` positional parameters, starting at `$position`.

```
echo ${*:2}          # Echoes second and following positional parameters.
echo ${@:2}          # Same as above.

echo ${*:2:3}        # Echoes three positional parameters, starting at second.
```

expr substr $string $position $length

Extracts *$length* characters from *$string* starting at *$position*.

```
stringZ=abcABC123ABCabc
#       123456789......
#       1-based indexing.

echo `expr substr $stringZ 1 2`              # ab
echo `expr substr $stringZ 4 3`              # ABC
```

expr match "$string" '\($substring\)'
: Extracts *$substring* at beginning of *$string*, where *$substring* is a regular expression.

expr "$string" : '\($substring\)'
: Extracts *$substring* at beginning of *$string*, where *$substring* is a regular expression.

```
stringZ=abcABC123ABCabc
#       =======

echo `expr match "$stringZ" '\(.[b-c]*[A-Z]..[0-9]\)'`   # abcABC1
echo `expr "$stringZ" : '\(.[b-c]*[A-Z]..[0-9]\)'`       # abcABC1
echo `expr "$stringZ" : '\(.......\)'`                   # abcABC1
# All of the above forms give an identical result.
```

expr match "$string" '.*\($substring\)'
: Extracts *$substring* at *end* of *$string*, where *$substring* is a regular expression.

expr "$string" : '.*\($substring\)'
: Extracts *$substring* at *end* of *$string*, where *$substring* is a regular expression.

```
stringZ=abcABC123ABCabc
#                ======

echo `expr match "$stringZ" '.*\([A-C][A-C][A-C][a-c]*\)'`    # ABCabc
echo `expr "$stringZ" : '.*\(......\)'`                       # ABCabc
```

Substring Removal

${string#substring}
: Deletes shortest match of *$substring* from *front* of *$string*.

${string##substring}
: Deletes longest match of *$substring* from *front* of *$string*.

```
stringZ=abcABC123ABCabc
#       |----|          shortest
#       |----------|    longest

echo ${stringZ#a*C}      # 123ABCabc
# Strip out shortest match between 'a' and 'C'.

echo ${stringZ##a*C}     # abc
# Strip out longest match between 'a' and 'C'.



# You can parameterize the substrings.

X='a*C'

echo ${stringZ#$X}      # 123ABCabc
echo ${stringZ##$X}     # abc
                        # As above.
```

${string%substring}

Deletes shortest match of *$substring* from *back* of *$string*.

For example:

```
# Rename all filenames in $PWD with "TXT" suffix to a "txt" suffix.
# For example, "file1.TXT" becomes "file1.txt" . . .

SUFF=TXT
suff=txt

for i in $(ls *.$SUFF)
do
  mv -f $i ${i%.$SUFF}.$suff
  #  Leave unchanged everything *except* the shortest pattern match
  #+ starting from the right-hand-side of the variable $i . . .
done ### This could be condensed into a "one-liner" if desired.

# Thank you, Rory Winston.
```

${string%%substring}

Deletes longest match of *$substring* from *back* of *$string*.

```
stringZ=abcABC123ABCabc
#                    ||     shortest
#        |------------|     longest

echo ${stringZ%b*c}      # abcABC123ABCa
# Strip out shortest match between 'b' and 'c', from back of $stringZ.

echo ${stringZ%%b*c}     # a
# Strip out longest match between 'b' and 'c', from back of $stringZ.
```

This operator is useful for generating filenames.

Example 10-3. Converting graphic file formats, with filename change

```
#!/bin/bash
#  cvt.sh:
#  Converts all the MacPaint image files in a directory to "pbm" format.

#  Uses the "macptopbm" binary from the "netpbm" package,
#+ which is maintained by Brian Henderson (bryanh@giraffe-data.com).
#  Netpbm is a standard part of most Linux distros.

OPERATION=macptopbm
SUFFIX=pbm          # New filename suffix.

if [ -n "$1" ]
then
  directory=$1      # If directory name given as a script argument...
else
  directory=$PWD    # Otherwise use current working directory.
fi

#  Assumes all files in the target directory are MacPaint image files,
#+ with a ".mac" filename suffix.

for file in $directory/*    # Filename globbing.
do
```

```
  filename=${file%.*c}      #  Strip ".mac" suffix off filename
                            #+ ('.*c' matches everything
                            #+ between '.' and 'c', inclusive).
  $OPERATION $file > "$filename.$SUFFIX"
                            # Redirect conversion to new filename.
  rm -f $file               # Delete original files after converting.
  echo "$filename.$SUFFIX"  # Log what is happening to stdout.
done

exit 0

# Exercise:
# --------
#  As it stands, this script converts *all* the files in the current
#+ working directory.
#  Modify it to work *only* on files with a ".mac" suffix.



# *** And here's another way to do it. *** #

#!/bin/bash
# Batch convert into different graphic formats.
# Assumes imagemagick installed (standard in most Linux distros).

INFMT=png   # Can be tif, jpg, gif, etc.
OUTFMT=pdf  # Can be tif, jpg, gif, pdf, etc.

for pic in *"$INFMT"
do
  p2=$(ls "$pic" | sed -e s/\.$INFMT//)
  # echo $p2
    convert "$pic" $p2.$OUTFMT
    done

exit $?
```

Example 10-4. Converting streaming audio files to *ogg*

```
#!/bin/bash
# ra2ogg.sh: Convert streaming audio files (*.ra) to ogg.

# Uses the "mplayer" media player program:
#      http://www.mplayerhq.hu/homepage
# Uses the "ogg" library and "oggenc":
#      http://www.xiph.org/
#
# This script may need appropriate codecs installed, such as sipr.so ...
# Possibly also the compat-libstdc++ package.


OFILEPREF=${1%%ra}      # Strip off the "ra" suffix.
OFILESUFF=wav           # Suffix for wav file.
OUTFILE="$OFILEPREF""$OFILESUFF"
E_NOARGS=85

if [ -z "$1" ]          # Must specify a filename to convert.
then
  echo "Usage: `basename $0` [filename]"
  exit $E_NOARGS
fi
```

```
#########################################################################
mplayer "$1" -ao pcm:file=$OUTFILE
oggenc "$OUTFILE"  # Correct file extension automatically added by oggenc.
#########################################################################

rm "$OUTFILE"      # Delete intermediate *.wav file.
                   # If you want to keep it, comment out above line.

exit $?

#  Note:
#  ----
#  On a Website, simply clicking on a *.ram streaming audio file
#+ usually only downloads the URL of the actual *.ra audio file.
#  You can then use "wget" or something similar
#+ to download the *.ra file itself.


#  Exercises:
#  ---------
#  As is, this script converts only *.ra filenames.
#  Add flexibility by permitting use of *.ram and other filenames.
#
#  If you're really ambitious, expand the script
#+ to do automatic downloads and conversions of streaming audio files.
#  Given a URL, batch download streaming audio files (using "wget")
#+ and convert them on the fly.
```

A simple emulation of getopt using substring-extraction constructs.

Example 10-5. Emulating *getopt*

```
#!/bin/bash
# getopt-simple.sh
# Author: Chris Morgan
# Used in the ABS Guide with permission.


getopt_simple()
{
    echo "getopt_simple()"
    echo "Parameters are '$*'"
    until [ -z "$1" ]
    do
      echo "Processing parameter of: '$1'"
      if [ ${1:0:1} = '/' ]
      then
          tmp=${1:1}               # Strip off leading '/' . . .
          parameter=${tmp%%=*}     # Extract name.
          value=${tmp##*=}         # Extract value.
          echo "Parameter: '$parameter', value: '$value'"
          eval $parameter=$value
      fi
      shift
    done
}
```

```
# Pass all options to getopt_simple().
getopt_simple $*

echo "test is '$test'"
echo "test2 is '$test2'"

exit 0  # See also, UseGetOpt.sh, a modified version of this script.

---

sh getopt_example.sh /test=value1 /test2=value2

Parameters are '/test=value1 /test2=value2'
Processing parameter of: '/test=value1'
Parameter: 'test', value: 'value1'
Processing parameter of: '/test2=value2'
Parameter: 'test2', value: 'value2'
test is 'value1'
test2 is 'value2'
```

Substring Replacement

${string/substring/replacement}
: Replace first *match* of `$substring` with `$replacement`. [50]

${string//substring/replacement}
: Replace all matches of `$substring` with `$replacement`.

```
stringZ=abcABC123ABCabc

echo ${stringZ/abc/xyz}       # xyzABC123ABCabc
                              # Replaces first match of 'abc' with 'xyz'.

echo ${stringZ//abc/xyz}      # xyzABC123ABCxyz
                              # Replaces all matches of 'abc' with # 'xyz'.

echo  ---------------
echo "$stringZ"               # abcABC123ABCabc
echo  ---------------
                              # The string itself is not altered!

# Can the match and replacement strings be parameterized?
match=abc
repl=000
echo ${stringZ/$match/$repl}  # 000ABC123ABCabc
#              ^      ^         ^^^
echo ${stringZ//$match/$repl} # 000ABC123ABC000
# Yes!          ^      ^        ^^^         ^^^

echo

# What happens if no $replacement string is supplied?
echo ${stringZ/abc}           # ABC123ABCabc
echo ${stringZ//abc}          # ABC123ABC
# A simple deletion takes place.
```

${string/#substring/replacement}
: If `$substring` matches *front* end of `$string`, substitute `$replacement` for `$substring`.

${string/%substring/replacement}
: If `$substring` matches *back* end of `$string`, substitute `$replacement` for `$substring`.

```
stringZ=abcABC123ABCabc

echo ${stringZ/#abc/XYZ}          # XYZABC123ABCabc
                                  # Replaces front-end match of 'abc' with 'XYZ'.

echo ${stringZ/%abc/XYZ}          # abcABC123ABCXYZ
                                  # Replaces back-end match of 'abc' with 'XYZ'.
```

10.1.1. Manipulating strings using awk

A Bash script may invoke the string manipulation facilities of awk as an alternative to using its built-in operations.

Example 10-6. Alternate ways of extracting and locating substrings

```
#!/bin/bash
# substring-extraction.sh

String=23skidoo1
#      012345678    Bash
#      123456789    awk
# Note different string indexing system:
# Bash numbers first character of string as 0.
# Awk  numbers first character of string as 1.

echo ${String:2:4} # position 3 (0-1-2), 4 characters long
                                         # skid

# The awk equivalent of ${string:pos:length} is substr(string,pos,length).
echo | awk '
{ print substr("'"${String}"'",3,4)      # skid
}
'
#  Piping an empty "echo" to awk gives it dummy input,
#+ and thus makes it unnecessary to supply a filename.

echo "----"

# And likewise:

echo | awk '
{ print index("'"${String}"'", "skid")      # 3
}                                           # (skid starts at position 3)
'   # The awk equivalent of "expr index" ...

exit 0
```

10.1.2. Further Reference

For more on string manipulation in scripts, refer to Section 10.2 and the relevant section of the expr command listing.

Script examples:

1. Example 16-9
2. Example 10-9
3. Example 10-10
4. Example 10-11
5. Example 10-13
6. Example A-36
7. Example A-41

10.2. Parameter Substitution

Manipulating and/or expanding variables

`${parameter}`

Same as *`$parameter`*, i.e., value of the variable *`parameter`*. In certain contexts, only the less ambiguous *`${parameter}`* form works.

May be used for concatenating variables with strings.

```
your_id=${USER}-on-${HOSTNAME}
echo "$your_id"
#
echo "Old \$PATH = $PATH"
PATH=${PATH}:/opt/bin  # Add /opt/bin to $PATH for duration of script.
echo "New \$PATH = $PATH"
```

`${parameter-default}`, **`${parameter:-default}`**

If parameter not set, use default.

```
var1=1
var2=2
# var3 is unset.

echo ${var1-$var2}   # 1
echo ${var3-$var2}   # 2
#           ^          Note the $ prefix.



echo ${username-`whoami`}
# Echoes the result of `whoami`, if variable $username is still unset.
```

`${parameter-default}` and *`${parameter:-default}`* are almost equivalent. The extra : makes a difference only when *`parameter`* has been declared, but is null.

```
#!/bin/bash
# param-sub.sh

#  Whether a variable has been declared
#+ affects triggering of the default option
#+ even if the variable is null.

username0=
echo "username0 has been declared, but is set to null."
```

```
echo "username0 = ${username0-`whoami`}"
# Will not echo.

echo

echo username1 has not been declared.
echo "username1 = ${username1-`whoami`}"
# Will echo.

username2=
echo "username2 has been declared, but is set to null."
echo "username2 = ${username2:-`whoami`}"
#                             ^
# Will echo because of :- rather than just - in condition test.
# Compare to first instance, above.


#

# Once again:

variable=
# variable has been declared, but is set to null.

echo "${variable-0}"    # (no output)
echo "${variable:-1}"   # 1
#               ^

unset variable

echo "${variable-2}"    # 2
echo "${variable:-3}"   # 3

exit 0
```

The *default parameter* construct finds use in providing "missing" command-line arguments in scripts.

```
DEFAULT_FILENAME=generic.data
filename=${1:-$DEFAULT_FILENAME}
#  If not otherwise specified, the following command block operates
#+ on the file "generic.data".
#  Begin-Command-Block
#  ...
#  ...
#  ...
#  End-Command-Block



#  From "hanoi2.bash" example:
DISKS=${1:-E_NOPARAM}   # Must specify how many disks.
#  Set $DISKS to $1 command-line-parameter,
#+ or to $E_NOPARAM if that is unset.
```

See also Example 3-4, Example 31-2, and Example A-6.

Compare this method with using an *and list* to supply a default command-line argument.

`${parameter=default}`, **`${parameter:=default}`**

If parameter not set, set it to *default*.

Both forms nearly equivalent. The : makes a difference only when `$parameter` has been declared and is null, [51] as above.

```
echo ${var=abc}   # abc
echo ${var=xyz}   # abc
# $var had already been set to abc, so it did not change.
```

`${parameter+alt_value}`, **`${parameter:+alt_value}`**

If parameter set, use **`alt_value`**, else use null string.

Both forms nearly equivalent. The : makes a difference only when *`parameter`* has been declared and is null, see below.

```
echo "###### \${parameter+alt_value} ########"
echo

a=${param1+xyz}
echo "a = $a"      # a =

param2=
a=${param2+xyz}
echo "a = $a"      # a = xyz

param3=123
a=${param3+xyz}
echo "a = $a"      # a = xyz

echo
echo "###### \${parameter:+alt_value} ########"
echo

a=${param4:+xyz}
echo "a = $a"      # a =

param5=
a=${param5:+xyz}
echo "a = $a"      # a =
# Different result from   a=${param5+xyz}

param6=123
a=${param6:+xyz}
echo "a = $a"      # a = xyz
```

`${parameter?err_msg}`, **`${parameter:?err_msg}`**

If parameter set, use it, else print *err_msg* and *abort the script* with an exit status of 1.

Both forms nearly equivalent. The : makes a difference only when *`parameter`* has been declared and is null, as above.

Example 10-7. Using parameter substitution and error messages

```
#!/bin/bash

#  Check some of the system's environmental variables.
#  This is good preventative maintenance.
#  If, for example, $USER, the name of the person at the console, is not set,
#+ the machine will not recognize you.
```

```
: ${HOSTNAME?} ${USER?} ${HOME?} ${MAIL?}
  echo
  echo "Name of the machine is $HOSTNAME."
  echo "You are $USER."
  echo "Your home directory is $HOME."
  echo "Your mail INBOX is located in $MAIL."
  echo
  echo "If you are reading this message,"
  echo "critical environmental variables have been set."
  echo
  echo

# ------------------------------------------------------

#  The ${variablename?} construction can also check
#+ for variables set within the script.

ThisVariable=Value-of-ThisVariable
#  Note, by the way, that string variables may be set
#+ to characters disallowed in their names.
: ${ThisVariable?}
echo "Value of ThisVariable is $ThisVariable".

echo; echo


: ${ZZXy23AB?"ZZXy23AB has not been set."}
#  Since ZZXy23AB has not been set,
#+ then the script terminates with an error message.

# You can specify the error message.
# : ${variablename?"ERROR MESSAGE"}


# Same result with:   dummy_variable=${ZZXy23AB?}
#                     dummy_variable=${ZZXy23AB?"ZXy23AB has not been set."}
#
#                     echo ${ZZXy23AB?} >/dev/null

#  Compare these methods of checking whether a variable has been set
#+ with "set -u" . . .



echo "You will not see this message, because script already terminated."

HERE=0
exit $HERE   # Will NOT exit here.

# In fact, this script will return an exit status (echo $?) of 1.
```

Example 10-8. Parameter substitution and "usage" messages

```
#!/bin/bash
# usage-message.sh

: ${1?"Usage: $0 ARGUMENT"}
#  Script exits here if command-line parameter absent,
#+ with following error message.
#    usage-message.sh: 1: Usage: usage-message.sh ARGUMENT
```

```
echo "These two lines echo only if command-line parameter given."
echo "command-line parameter = \"$1\""

exit 0  # Will exit here only if command-line parameter present.

# Check the exit status, both with and without command-line parameter.
# If command-line parameter present, then "$?" is 0.
# If not, then "$?" is 1.
```

Parameter substitution and/or expansion. The following expressions are the complement to the **match** `in` **expr** string operations (see Example 16-9). These particular ones are used mostly in parsing file path names.

Variable length / Substring removal

`${#var}`

`String length` (number of characters in `$var`). For an array, **${#array}** is the length of the first element in the array.

Exceptions:

- **${#*}** and **${#@}** give the *number of positional parameters*.
- For an array, **${#array[*]}** and **${#array[@]}** give the number of elements in the array.

Example 10-9. Length of a variable

```
#!/bin/bash
# length.sh

E_NO_ARGS=65

if [ $# -eq 0 ]  # Must have command-line args to demo script.
then
  echo "Please invoke this script with one or more command-line arguments."
  exit $E_NO_ARGS
fi

var01=abcdEFGH28ij
echo "var01 = ${var01}"
echo "Length of var01 = ${#var01}"
# Now, let's try embedding a space.
var02="abcd EFGH28ij"
echo "var02 = ${var02}"
echo "Length of var02 = ${#var02}"

echo "Number of command-line arguments passed to script = ${#@}"
echo "Number of command-line arguments passed to script = ${#*}"

exit 0
```

`${var#Pattern}`, **`${var##Pattern}`**

${var#Pattern} Remove from `$var` the *shortest* part of `$Pattern` that matches the *`front end`* of `$var`.

${var##Pattern} Remove from `$var` the *longest* part of `$Pattern` that matches the *`front end`* of `$var`.

A usage illustration from Example A-7:

```
# Function from "days-between.sh" example.
# Strips leading zero(s) from argument passed.

strip_leading_zero () #  Strip possible leading zero(s)
{                     #+ from argument passed.
  return=${1#0}       #  The "1" refers to "$1" -- passed arg.
}                     #  The "0" is what to remove from "$1" -- strips zeros.
```

Manfred Schwarb's more elaborate variation of the above:

```
strip_leading_zero2 () # Strip possible leading zero(s), since otherwise
{                      # Bash will interpret such numbers as octal values.
  shopt -s extglob     # Turn on extended globbing.
  local val=${1##+(0)} # Use local variable, longest matching series of 0's.
  shopt -u extglob     # Turn off extended globbing.
  _strip_leading_zero2=${val:-0}
                       # If input was 0, return 0 instead of "".
}
```

Another usage illustration:

```
echo `basename $PWD`        # Basename of current working directory.
echo "${PWD##*/}"           # Basename of current working directory.
echo
echo `basename $0`          # Name of script.
echo $0                     # Name of script.
echo "${0##*/}"             # Name of script.
echo
filename=test.data
echo "${filename##*.}"      # data
                            # Extension of filename.
```

`${var%Pattern}`, **`${var%%Pattern}`**

${var%Pattern} Remove from `$var` the *shortest* part of `$Pattern` that matches the *`back end`* of `$var`.

${var%%Pattern} Remove from `$var` the *longest* part of `$Pattern` that matches the *`back end`* of `$var`.

Version 2 of Bash added additional options.

Example 10-10. Pattern matching in parameter substitution

```
#!/bin/bash
# patt-matching.sh

# Pattern matching  using the # ## % %% parameter substitution operators.

var1=abcd12345abc6789
pattern1=a*c  # * (wild card) matches everything between a - c.

```

```
echo
echo "var1 = $var1"           # abcd12345abc6789
echo "var1 = ${var1}"         # abcd12345abc6789
                              # (alternate form)
echo "Number of characters in ${var1} = ${#var1}"
echo

echo "pattern1 = $pattern1"   # a*c  (everything between 'a' and 'c')
echo "--------------"
echo '${var1#$pattern1}  =' "${var1#$pattern1}"    #         d12345abc6789
# Shortest possible match, strips out first 3 characters  abcd12345abc6789
#                                     ^^^^^               |-|
echo '${var1##$pattern1} =' "${var1##$pattern1}"   #                  6789
# Longest possible match, strips out first 12 characters  abcd12345abc6789
#                                    ^^^^^                |----------|

echo; echo; echo

pattern2=b*9            # everything between 'b' and '9'
echo "var1 = $var1"     # Still  abcd12345abc6789
echo
echo "pattern2 = $pattern2"
echo "--------------"
echo '${var1%pattern2}  =' "${var1%$pattern2}"     #     abcd12345a
# Shortest possible match, strips out last 6 characters  abcd12345abc6789
#                                     ^^^^                        |----|
echo '${var1%%pattern2} =' "${var1%%$pattern2}"    #     a
# Longest possible match, strips out last 12 characters  abcd12345abc6789
#                                    ^^^^                 |-------------|

# Remember, # and ## work from the left end (beginning) of string,
#           % and %% work from the right end.

echo

exit 0
```

Example 10-11. Renaming file extensions:

```
#!/bin/bash
# rfe.sh: Renaming file extensions.
#
#         rfe old_extension new_extension
#
# Example:
# To rename all *.gif files in working directory to *.jpg,
#          rfe gif jpg


E_BADARGS=65

case $# in
  0|1)             # The vertical bar means "or" in this context.
  echo "Usage: `basename $0` old_file_suffix new_file_suffix"
  exit $E_BADARGS  # If 0 or 1 arg, then bail out.
  ;;
esac


for filename in *.$1
# Traverse list of files ending with 1st argument.
```

```
do
  mv $filename ${filename%$1}$2
  #  Strip off part of filename matching 1st argument,
  #+ then append 2nd argument.
done

exit 0
```

Variable expansion / Substring replacement

These constructs have been adopted from *ksh*.

`${var:pos}`

Variable `var` expanded, starting from offset `pos`.

`${var:pos:len}`

Expansion to a max of `len` characters of variable `var`, from offset `pos`. See Example A-13 for an example of the creative use of this operator.

`${var/Pattern/Replacement}`

First match of `Pattern`, within `var` replaced with `Replacement`.

If `Replacement` is omitted, then the first match of `Pattern` is replaced by *nothing*, that is, deleted.

`${var//Pattern/Replacement}`

Global replacement. All matches of `Pattern`, within `var` replaced with `Replacement`.

As above, if `Replacement` is omitted, then all occurrences of `Pattern` are replaced by *nothing*, that is, deleted.

Example 10-12. Using pattern matching to parse arbitrary strings

```
#!/bin/bash

var1=abcd-1234-defg
echo "var1 = $var1"

t=${var1#*-*}
echo "var1 (with everything, up to and including first - stripped out) = $t"
#  t=${var1#*-}  works just the same,
#+ since # matches the shortest string,
#+ and * matches everything preceding, including an empty string.
# (Thanks, Stephane Chazelas, for pointing this out.)

t=${var1##*-*}
echo "If var1 contains a \"-\", returns empty string...   var1 = $t"


t=${var1%*-*}
echo "var1 (with everything from the last - on stripped out) = $t"

echo

# -------------------------------------------
path_name=/home/bozo/ideas/thoughts.for.today
# -------------------------------------------
echo "path_name = $path_name"
t=${path_name##/*/}
echo "path_name, stripped of prefixes = $t"
# Same effect as   t=`basename $path_name` in this particular case.
```

```
#  t=${path_name%/}; t=${t##*/}   is a more general solution,
#+ but still fails sometimes.
#  If $path_name ends with a newline, then `basename $path_name` will not work,
#+ but the above expression will.
# (Thanks, S.C.)

t=${path_name%/*.*}
# Same effect as   t=`dirname $path_name`
echo "path_name, stripped of suffixes = $t"
# These will fail in some cases, such as "../", "/foo////", # "foo/", "/".
#  Removing suffixes, especially when the basename has no suffix,
#+ but the dirname does, also complicates matters.
# (Thanks, S.C.)

echo

t=${path_name:11}
echo "$path_name, with first 11 chars stripped off = $t"
t=${path_name:11:5}
echo "$path_name, with first 11 chars stripped off, length 5 = $t"

echo

t=${path_name/bozo/clown}
echo "$path_name with \"bozo\" replaced  by \"clown\" = $t"
t=${path_name/today/}
echo "$path_name with \"today\" deleted = $t"
t=${path_name//o/O}
echo "$path_name with all o's capitalized = $t"
t=${path_name//o/}
echo "$path_name with all o's deleted = $t"

exit 0
```

`${var/#Pattern/Replacement}`
: If *prefix* of `var` matches `Pattern`, then substitute `Replacement` for `Pattern`.

`${var/%Pattern/Replacement}`
: If *suffix* of `var` matches `Pattern`, then substitute `Replacement` for `Pattern`.

Example 10-13. Matching patterns at prefix or suffix of string

```
#!/bin/bash
# var-match.sh:
# Demo of pattern replacement at prefix / suffix of string.

v0=abc1234zip1234abc    # Original variable.
echo "v0 = $v0"         # abc1234zip1234abc
echo

# Match at prefix (beginning) of string.
v1=${v0/#abc/ABCDEF}    # abc1234zip1234abc
                        # |-|
echo "v1 = $v1"         # ABCDEF1234zip1234abc
                        # |----|

# Match at suffix (end) of string.
v2=${v0/%abc/ABCDEF}    # abc1234zip123abc
                        #              |-|
echo "v2 = $v2"         # abc1234zip1234ABCDEF
                        #               |----|
```

```
echo

#  --------------------------------------------------------
#  Must match at beginning / end of string,
#+ otherwise no replacement results.
#  --------------------------------------------------------
v3=${v0/#123/000}       # Matches, but not at beginning.
echo "v3 = $v3"         # abc1234zip1234abc
                        # NO REPLACEMENT.
v4=${v0/%123/000}       # Matches, but not at end.
echo "v4 = $v4"         # abc1234zip1234abc
                        # NO REPLACEMENT.

exit 0
```

${!varprefix*}**, **${!varprefix@}

Matches *names* of all previously declared variables beginning with *varprefix*.

```
# This is a variation on indirect reference, but with a * or @.
# Bash, version 2.04, adds this feature.

xyz23=whatever
xyz24=

a=${!xyz*}         #  Expands to *names* of declared variables
# ^ ^   ^           + beginning with "xyz".
echo "a = $a"      #  a = xyz23 xyz24
a=${!xyz@}         #  Same as above.
echo "a = $a"      #  a = xyz23 xyz24

echo "---"

abc23=something_else
b=${!abc*}
echo "b = $b"      #  b = abc23
c=${!b}            #  Now, the more familiar type of indirect reference.
echo $c            #  something_else
```

Chapter 11. Loops and Branches

What needs this iteration, woman?

--Shakespeare, `Othello`

Operations on code blocks are the key to structured and organized shell scripts. Looping and branching constructs provide the tools for accomplishing this.

11.1. Loops

A *loop* is a block of code that *iterates* [52] a list of commands as long as the *loop control condition* is true.

for loops

for *`arg`* in *`[list]`*

This is the basic looping construct. It differs significantly from its *C* counterpart.

for *`arg`* in [*`list`*]
do
 `command(s)`...
done

During each pass through the loop, *`arg`* takes on the value of each successive variable in the *`list`*.

```
for arg in "$var1" "$var2" "$var3" ... "$varN"
# In pass 1 of the loop, arg = $var1
# In pass 2 of the loop, arg = $var2
# In pass 3 of the loop, arg = $var3
# ...
# In pass N of the loop, arg = $varN

# Arguments in [list] quoted to prevent possible word splitting.
```

The argument *`list`* may contain wild cards.

If *do* is on same line as *for*, there needs to be a semicolon after list.

for *`arg`* in [*`list`*] ; do

Example 11-1. Simple *for* loops

```
#!/bin/bash
# Listing the planets.

for planet in Mercury Venus Earth Mars Jupiter Saturn Uranus Neptune Pluto
do
  echo $planet  # Each planet on a separate line.
```

```
done

echo; echo

for planet in "Mercury Venus Earth Mars Jupiter Saturn Uranus Neptune Pluto"
    # All planets on same line.
    # Entire 'list' enclosed in quotes creates a single variable.
    # Why? Whitespace incorporated into the variable.
do
  echo $planet
done

echo; echo "Whoops! Pluto is no longer a planet!"

exit 0
```

Each **`[list]`** element may contain multiple parameters. This is useful when processing parameters in groups. In such cases, use the set command (see Example 15-16) to force parsing of each **`[list]`** element and assignment of each component to the positional parameters.

Example 11-2. *for* loop with two parameters in each [list] element

```
#!/bin/bash
# Planets revisited.

# Associate the name of each planet with its distance from the sun.

for planet in "Mercury 36" "Venus 67" "Earth 93"  "Mars 142" "Jupiter 483"
do
  set -- $planet  #  Parses variable "planet"
                  #+ and sets positional parameters.
  #  The "--" prevents nasty surprises if $planet is null or
  #+ begins with a dash.

  #  May need to save original positional parameters,
  #+ since they get overwritten.
  #  One way of doing this is to use an array,
  #         original_params=("$@")

  echo "$1              $2,000,000 miles from the sun"
  #-------two  tabs---concatenate zeroes onto parameter $2
done

# (Thanks, S.C., for additional clarification.)

exit 0
```

A variable may supply the **`[list]`** in a *for loop*.

Example 11-3. *Fileinfo:* operating on a file list contained in a variable

```
#!/bin/bash
# fileinfo.sh

FILES="/usr/sbin/accept
/usr/sbin/pwck
```

```
/usr/sbin/chroot
/usr/bin/fakefile
/sbin/badblocks
/sbin/ypbind"     # List of files you are curious about.
                  # Threw in a dummy file, /usr/bin/fakefile.

echo

for file in $FILES
do

  if [ ! -e "$file" ]       # Check if file exists.
  then
    echo "$file does not exist."; echo
    continue                # On to next.
   fi

  ls -l $file | awk '{ print $8 "         file size: " $5 }'  # Print 2 fields.
  whatis `basename $file`   # File info.
  # Note that the whatis database needs to have been set up for this to work.
  # To do this, as root run /usr/bin/makewhatis.
  echo
done

exit 0
```

The **`[list]`** in a *for loop* may be parameterized.

Example 11-4. Operating on a parameterized file list

```
#!/bin/bash

filename="*txt"

for file in $filename
do
 echo "Contents of $file"
 echo "---"
 cat "$file"
 echo
done
```

If the **`[list]`** in a *for loop* contains wild cards (* and ?) used in filename expansion, then globbing takes place.

Example 11-5. Operating on files with a *for* loop

```
#!/bin/bash
# list-glob.sh: Generating [list] in a for-loop, using "globbing" ...
# Globbing = filename expansion.

echo

for file in *
#           ^  Bash performs filename expansion
#+             on expressions that globbing recognizes.
do
```

```
  ls -l "$file"  # Lists all files in $PWD (current directory).
  #  Recall that the wild card character "*" matches every filename,
  #+ however, in "globbing," it doesn't match dot-files.

  #  If the pattern matches no file, it is expanded to itself.
  #  To prevent this, set the nullglob option
  #+   (shopt -s nullglob).
  #  Thanks, S.C.
done

echo; echo

for file in [jx]*
do
  rm -f $file    # Removes only files beginning with "j" or "x" in $PWD.
  echo "Removed file \"$file\"".
done

echo

exit 0
```

Omitting the **`in [list]`** part of a *for loop* causes the loop to operate on $@ -- the positional parameters. A particularly clever illustration of this is Example A-15. See also Example 15-17.

Example 11-6. Missing `in [list]` in a *for* loop

```
#!/bin/bash

#  Invoke this script both with and without arguments,
#+ and see what happens.

for a
do
 echo -n "$a "
done

#  The 'in list' missing, therefore the loop operates on '$@'
#+ (command-line argument list, including whitespace).

echo

exit 0
```

It is possible to use command substitution to generate the **`[list]`** in a *for loop*. See also Example 16-54, Example 11-11 and Example 16-48.

Example 11-7. Generating the `[list]` in a *for* loop with command substitution

```
#!/bin/bash
#  for-loopcmd.sh: for-loop with [list]
#+ generated by command substitution.

NUMBERS="9 7 3 8 37.53"

for number in `echo $NUMBERS`  # for number in 9 7 3 8 37.53
```

```
do
  echo -n "$number "
done

echo
exit 0
```

Here is a somewhat more complex example of using command substitution to create the **`[list]`**.

Example 11-8. A *grep* replacement for binary files

```
#!/bin/bash
# bin-grep.sh: Locates matching strings in a binary file.

# A "grep" replacement for binary files.
# Similar effect to "grep -a"

E_BADARGS=65
E_NOFILE=66

if [ $# -ne 2 ]
then
  echo "Usage: `basename $0` search_string filename"
  exit $E_BADARGS
fi

if [ ! -f "$2" ]
then
  echo "File \"$2\" does not exist."
  exit $E_NOFILE
fi


IFS=$'\012'       # Per suggestion of Anton Filippov.
                  # was:  IFS="\n"
for word in $( strings "$2" | grep "$1" )
# The "strings" command lists strings in binary files.
# Output then piped to "grep", which tests for desired string.
do
  echo $word
done

# As S.C. points out, lines 23 - 30 could be replaced with the simpler
#    strings "$2" | grep "$1" | tr -s "$IFS" '[\n*]'


#  Try something like  "./bin-grep.sh mem /bin/ls"
#+ to exercise this script.

exit 0
```

More of the same.

Example 11-9. Listing all users on the system

```
#!/bin/bash
# userlist.sh

PASSWORD_FILE=/etc/passwd
```

```
n=1           # User number

for name in $(awk 'BEGIN{FS=":"}{print $1}' < "$PASSWORD_FILE" )
# Field separator = :    ^^^^^^
# Print first field              ^^^^^^^^
# Get input from password file  /etc/passwd  ^^^^^^^^^^^^^^^^^
do
  echo "USER #$n = $name"
  let "n += 1"
done


# USER #1 = root
# USER #2 = bin
# USER #3 = daemon
# ...
# USER #33 = bozo

exit $?

#  Discussion:
#  ----------
#  How is it that an ordinary user, or a script run by same,
#+ can read /etc/passwd? (Hint: Check the /etc/passwd file permissions.)
#  Is this a security hole? Why or why not?
```

Yet another example of the **[list]** resulting from command substitution.

Example 11-10. Checking all the binaries in a directory for authorship

```
#!/bin/bash
# findstring.sh:
# Find a particular string in the binaries in a specified directory.

directory=/usr/bin/
fstring="Free Software Foundation"  # See which files come from the FSF.

for file in $( find $directory -type f -name '*' | sort )
do
  strings -f $file | grep "$fstring" | sed -e "s%$directory%%"
  #  In the "sed" expression,
  #+ it is necessary to substitute for the normal "/" delimiter
  #+ because "/" happens to be one of the characters filtered out.
  #  Failure to do so gives an error message. (Try it.)
done

exit $?

#  Exercise (easy):
#  ---------------
#  Convert this script to take command-line parameters
#+ for $directory and $fstring.
```

A final example of **[list]** / command substitution, but this time the "command" is a function.

```
generate_list ()
{
  echo "one two three"
}

for word in $(generate_list)  # Let "word" grab output of function.
```

```
do
  echo "$word"
done

# one
# two
# three
```

The output of a *for loop* may be piped to a command or commands.

Example 11-11. Listing the *symbolic links* in a directory

```
#!/bin/bash
# symlinks.sh: Lists symbolic links in a directory.


directory=${1-`pwd`}
#  Defaults to current working directory,
#+ if not otherwise specified.
#  Equivalent to code block below.
# ----------------------------------------------------------
# ARGS=1                 # Expect one command-line argument.
#
# if [ $# -ne "$ARGS" ]  # If not 1 arg...
# then
#   directory=`pwd`      # current working directory
# else
#   directory=$1
# fi
# ----------------------------------------------------------

echo "symbolic links in directory \"$directory\""

for file in "$( find $directory -type l )"   # -type l = symbolic links
do
  echo "$file"
done | sort                                  # Otherwise file list is unsorted.
#  Strictly speaking, a loop isn't really necessary here,
#+ since the output of the "find" command is expanded into a single word.
#  However, it's easy to understand and illustrative this way.

#  As Dominik 'Aeneas' Schnitzer points out,
#+ failing to quote  $( find $directory -type l )
#+ will choke on filenames with embedded whitespace.
#  containing whitespace.

exit 0


# --------------------------------------------------------
# Jean Helou proposes the following alternative:

echo "symbolic links in directory \"$directory\""
# Backup of the current IFS. One can never be too cautious.
OLDIFS=$IFS
IFS=:

for file in $(find $directory -type l -printf "%p$IFS")
do     #                              ^^^^^^^^^^^^^^^^
```

```
      echo "$file"
      done|sort

# And, James "Mike" Conley suggests modifying Helou's code thusly:

OLDIFS=$IFS
IFS='' # Null IFS means no word breaks
for file in $( find $directory -type l )
do
  echo $file
  done | sort

#  This works in the "pathological" case of a directory name having
#+ an embedded colon.
#  "This also fixes the pathological case of the directory name having
#+  a colon (or space in earlier example) as well."
```

The `stdout` of a loop may be redirected to a file, as this slight modification to the previous example shows.

Example 11-12. Symbolic links in a directory, saved to a file

```
#!/bin/bash
# symlinks.sh: Lists symbolic links in a directory.

OUTFILE=symlinks.list                         # save-file

directory=${1-`pwd`}
#  Defaults to current working directory,
#+ if not otherwise specified.


echo "symbolic links in directory \"$directory\"" > "$OUTFILE"
echo "---------------------------" >> "$OUTFILE"

for file in "$( find $directory -type l )"    # -type l = symbolic links
do
  echo "$file"
done | sort >> "$OUTFILE"                     # stdout of loop
#           ^^^^^^^^^^^^^                       redirected to save file.

# echo "Output file = $OUTFILE"

exit $?
```

There is an alternative syntax to a *for loop* that will look very familiar to C programmers. This requires double parentheses.

Example 11-13. A C-style *for* loop

```
#!/bin/bash
# Multiple ways to count up to 10.

echo

# Standard syntax.
for a in 1 2 3 4 5 6 7 8 9 10
do
```

```
  echo -n "$a "
done

echo; echo

# +==========================================+

# Using "seq" ...
for a in `seq 10`
do
  echo -n "$a "
done

echo; echo

# +==========================================+

# Using brace expansion ...
# Bash, version 3+.
for a in {1..10}
do
  echo -n "$a "
done

echo; echo

# +==========================================+

# Now, let's do the same, using C-like syntax.

LIMIT=10

for ((a=1; a <= LIMIT ; a++))  # Double parentheses, and naked "LIMIT"
do
  echo -n "$a "
done                           # A construct borrowed from ksh93.

echo; echo

# +=========================================================================+

# Let's use the C "comma operator" to increment two variables simultaneously.

for ((a=1, b=1; a <= LIMIT ; a++, b++))
do  # The comma concatenates operations.
  echo -n "$a-$b "
done

echo; echo

exit 0
```

See also Example 27-16, Example 27-17, and Example A-6.

Now, a *for loop* used in a "real-life" context.

Example 11-14. Using *efax* in batch mode

```
#!/bin/bash
# Faxing (must have 'efax' package installed).

EXPECTED_ARGS=2
E_BADARGS=85
MODEM_PORT="/dev/ttyS2"   # May be different on your machine.
#                ^^^^^      PCMCIA modem card default port.

if [ $# -ne $EXPECTED_ARGS ]
# Check for proper number of command-line args.
then
   echo "Usage: `basename $0` phone# text-file"
   exit $E_BADARGS
fi


if [ ! -f "$2" ]
then
  echo "File $2 is not a text file."
  #     File is not a regular file, or does not exist.
  exit $E_BADARGS
fi


fax make $2              #  Create fax-formatted files from text files.

for file in $(ls $2.0*)  #  Concatenate the converted files.
                         #  Uses wild card (filename "globbing")
                         #+ in variable list.
do
  fil="$fil $file"
done

efax -d "$MODEM_PORT"  -t "T$1" $fil   # Finally, do the work.
# Trying adding  -o1  if above line fails.


#  As S.C. points out, the for-loop can be eliminated with
#     efax -d /dev/ttyS2 -o1 -t "T$1" $2.0*
#+ but it's not quite as instructive [grin].

exit $?   # Also, efax sends diagnostic messages to stdout.
```

The keywords **do** and **done** delineate the *for-loop* command block. However, these may, in certain contexts, be omitted by framing the command block within curly brackets

```
for((n=1; n<=10; n++))
# No do!
{
  echo -n "* $n *"
}
# No done!


# Outputs:
# * 1 ** 2 ** 3 ** 4 ** 5 ** 6 ** 7 ** 8 ** 9 ** 10 *
# And, echo $? returns 0, so Bash does not register an error.
```

```
echo


#  But, note that in a classic for-loop:    for n in [list] ...
#+ a terminal semicolon is required.

for n in 1 2 3
{  echo -n "$n "; }
#               ^


# Thank you, YongYe, for pointing this out.
```

while

This construct tests for a condition at the top of a loop, and keeps looping as long as that condition is true (returns a 0 exit status). In contrast to a for loop, a *while loop* finds use in situations where the number of loop repetitions is not known beforehand.

while [*condition*]
do
 command(s)...
done

The bracket construct in a *while loop* is nothing more than our old friend, the test brackets used in an *if/then* test. In fact, a *while loop* can legally use the more versatile double-brackets construct (while [[condition]]).

As is the case with *for loops*, placing the *do* on the same line as the condition test requires a semicolon.

while [*condition*] ; do

Note that the *test brackets* are *not* mandatory in a *while* loop. See, for example, the getopts construct.

Example 11-15. Simple *while* loop

```
#!/bin/bash

var0=0
LIMIT=10

while [ "$var0" -lt "$LIMIT" ]
#      ^                    ^
# Spaces, because these are "test-brackets" . . .
do
  echo -n "$var0 "        # -n suppresses newline.
  #             ^           Space, to separate printed out numbers.

  var0=`expr $var0 + 1`   # var0=$(($var0+1))  also works.
                          # var0=$((var0 + 1)) also works.
                          # let "var0 += 1"    also works.
done                      # Various other methods also work.

echo
```

```
exit 0
```

Example 11-16. Another *while* loop

```
#!/bin/bash

echo
                               # Equivalent to:
while [ "$var1" != "end" ]     # while test "$var1" != "end"
do
  echo "Input variable #1 (end to exit) "
  read var1                    # Not 'read $var1' (why?).
  echo "variable #1 = $var1"   # Need quotes because of "#" . . .
  # If input is 'end', echoes it here.
  # Does not test for termination condition until top of loop.
  echo
done

exit 0
```

A *while loop* may have multiple conditions. Only the final condition determines when the loop terminates. This necessitates a slightly different loop syntax, however.

Example 11-17. *while* loop with multiple conditions

```
#!/bin/bash

var1=unset
previous=$var1

while echo "previous-variable = $previous"
      echo
      previous=$var1
      [ "$var1" != end ] # Keeps track of what $var1 was previously.
      # Four conditions on *while*, but only the final one controls loop.
      # The *last* exit status is the one that counts.
do
echo "Input variable #1 (end to exit) "
  read var1
  echo "variable #1 = $var1"
done

# Try to figure out how this all works.
# It's a wee bit tricky.

exit 0
```

As with a *for loop*, a *while loop* may employ C-style syntax by using the double-parentheses construct (see also Example 8-5).

Example 11-18. C-style syntax in a *while* loop

```
#!/bin/bash
# wh-loopc.sh: Count to 10 in a "while" loop.
```

```
LIMIT=10                  # 10 iterations.
a=1

while [ "$a" -le $LIMIT ]
do
  echo -n "$a "
  let "a+=1"
done                      # No surprises, so far.

echo; echo

# +=================================================================+

# Now, we'll repeat with C-like syntax.

((a = 1))      # a=1
# Double parentheses permit space when setting a variable, as in C.

while (( a <= LIMIT ))    #  Double parentheses,
do                        #+ and no "$" preceding variables.
  echo -n "$a "
  ((a += 1))              # let "a+=1"
  # Yes, indeed.
  # Double parentheses permit incrementing a variable with C-like syntax.
done

echo

# C and Java programmers can feel right at home in Bash.

exit 0
```

Inside its test brackets, a *while loop* can call a function.

```
t=0

condition ()
{
  ((t++))

  if [ $t -lt 5 ]
  then
    return 0  # true
  else
    return 1  # false
  fi
}

while condition
#     ^^^^^^^^^
#     Function call -- four loop iterations.
do
  echo "Still going: t = $t"
done

# Still going: t = 1
# Still going: t = 2
# Still going: t = 3
# Still going: t = 4
```

Similar to the if-test construct, a *while* loop can omit the test brackets.

```
while condition
do
   command(s) ...
done
```

By coupling the power of the read command with a *while loop*, we get the handy while read construct, useful for reading and parsing files.

```
cat $filename |   # Supply input from a file.
while read line   # As long as there is another line to read ...
do
  ...
done

# =========== Snippet from "sd.sh" example script ========== #

  while read value   # Read one data point at a time.
  do
    rt=$(echo "scale=$SC; $rt + $value" | bc)
    (( ct++ ))
  done

  am=$(echo "scale=$SC; $rt / $ct" | bc)

  echo $am; return $ct   # This function "returns" TWO values!
  #  Caution: This little trick will not work if $ct > 255!
  #  To handle a larger number of data points,
  #+ simply comment out the "return $ct" above.
} <"$datafile"   # Feed in data file.
```

A *while loop* may have its `stdin` redirected to a file by a < at its end.

A *while loop* may have its `stdin` supplied by a pipe.

until

This construct tests for a condition at the top of a loop, and keeps looping as long as that condition is *false* (opposite of *while loop*).

until [*condition-is-true*]
do
 command(s)...
done

Note that an *until loop* tests for the terminating condition at the *top* of the loop, differing from a similar construct in some programming languages.

As is the case with *for loops*, placing the *do* on the same line as the condition test requires a semicolon.

until [*condition-is-true*] ; do

Example 11-19. *until* loop

```
#!/bin/bash

END_CONDITION=end

until [ "$var1" = "$END_CONDITION" ]
# Tests condition here, at top of loop.
do
  echo "Input variable #1 "
  echo "($END_CONDITION to exit)"
  read var1
  echo "variable #1 = $var1"
  echo
done

#                     ---                        #

#  As with "for" and "while" loops,
#+ an "until" loop permits C-like test constructs.

LIMIT=10
var=0

until (( var > LIMIT ))
do  # ^^ ^     ^     ^^   No brackets, no $ prefixing variables.
  echo -n "$var "
  (( var++ ))
done    # 0 1 2 3 4 5 6 7 8 9 10


exit 0
```

How to choose between a *for* loop or a *while* loop or *until* loop? In **C**, you would typically use a *for* loop when the number of loop iterations is known beforehand. With *Bash*, however, the situation is fuzzier. The Bash *for* loop is more loosely structured and more flexible than its equivalent in other languages. Therefore, feel free to use whatever type of loop gets the job done in the simplest way.

11.2. Nested Loops

A *nested loop* is a loop within a loop, an inner loop within the body of an outer one. How this works is that the first pass of the outer loop triggers the inner loop, which executes to completion. Then the second pass of the outer loop triggers the inner loop again. This repeats until the outer loop finishes. Of course, a *break* within either the inner or outer loop would interrupt this process.

Example 11-20. Nested Loop

```
#!/bin/bash
# nested-loop.sh: Nested "for" loops.

outer=1             # Set outer loop counter.

# Beginning of outer loop.
for a in 1 2 3 4 5
do
  echo "Pass $outer in outer loop."
  echo "---------------------"
```

```
  inner=1           # Reset inner loop counter.

  # ===============================================
  # Beginning of inner loop.
  for b in 1 2 3 4 5
  do
    echo "Pass $inner in inner loop."
    let "inner+=1"  # Increment inner loop counter.
  done
  # End of inner loop.
  # ===============================================

  let "outer+=1"    # Increment outer loop counter.
  echo              # Space between output blocks in pass of outer loop.
done
# End of outer loop.

exit 0
```

See Example 27-11 for an illustration of nested while loops, and Example 27-13 to see a while loop nested inside an until loop.

11.3. Loop Control

Tournez cent tours, tournez mille tours,

Tournez souvent et tournez toujours . . .

--Verlaine, "Chevaux de bois"

Commands affecting loop behavior

break, **continue**

The **break** and **continue** loop control commands [53] correspond exactly to their counterparts in other programming languages. The **break** command terminates the loop (*breaks* out of it), while **continue** causes a jump to the next iteration of the loop, skipping all the remaining commands in that particular loop cycle.

Example 11-21. Effects of *break* and continue in a loop

```
#!/bin/bash

LIMIT=19  # Upper limit

echo
echo "Printing Numbers 1 through 20 (but not 3 and 11)."

a=0

while [ $a -le "$LIMIT" ]
do
 a=$(($a+1))

 if [ "$a" -eq 3 ] || [ "$a" -eq 11 ]  # Excludes 3 and 11.
 then
   continue      # Skip rest of this particular loop iteration.
```

```
 fi

 echo -n "$a "   # This will not execute for 3 and 11.
done

# Exercise:
# Why does the loop print up to 20?

echo; echo

echo Printing Numbers 1 through 20, but something happens after 2.

##################################################################

# Same loop, but substituting 'break' for 'continue'.

a=0

while [ "$a" -le "$LIMIT" ]
do
 a=$(($a+1))

 if [ "$a" -gt 2 ]
 then
   break  # Skip entire rest of loop.
 fi

 echo -n "$a "
done

echo; echo; echo

exit 0
```

The **break** command may optionally take a parameter. A plain **break** terminates only the innermost loop in which it is embedded, but a **break N** breaks out of *N* levels of loop.

Example 11-22. Breaking out of multiple loop levels

```
#!/bin/bash
# break-levels.sh: Breaking out of loops.

# "break N" breaks out of N level loops.

for outerloop in 1 2 3 4 5
do
  echo -n "Group $outerloop:   "

  # --------------------------------------------------------
  for innerloop in 1 2 3 4 5
  do
    echo -n "$innerloop "

    if [ "$innerloop" -eq 3 ]
    then
      break  # Try   break 2   to see what happens.
             # ("Breaks" out of both inner and outer loops.)
    fi
  done
```

```
  # ------------------------------------------------------

  echo
done

echo

exit 0
```

The **continue** command, similar to **break**, optionally takes a parameter. A plain **continue** cuts short the current iteration within its loop and begins the next. A **continue N** terminates all remaining iterations at its loop level and continues with the next iteration at the loop, `N` levels above.

Example 11-23. Continuing at a higher loop level

```
#!/bin/bash
# The "continue N" command, continuing at the Nth level loop.

for outer in I II III IV V           # outer loop
do
  echo; echo -n "Group $outer: "

  # --------------------------------------------------------------------
  for inner in 1 2 3 4 5 6 7 8 9 10  # inner loop
  do

    if [[ "$inner" -eq 7 && "$outer" = "III" ]]
    then
      continue 2  # Continue at loop on 2nd level, that is "outer loop".
                  # Replace above line with a simple "continue"
                  # to see normal loop behavior.
    fi

    echo -n "$inner "  # 7 8 9 10 will not echo on "Group III."
  done
  # --------------------------------------------------------------------

done

echo; echo

# Exercise:
# Come up with a meaningful use for "continue N" in a script.

exit 0
```

Example 11-24. Using *continue N* in an actual task

```
# Albert Reiner gives an example of how to use "continue N":
# ---------------------------------------------------------

#  Suppose I have a large number of jobs that need to be run, with
#+ any data that is to be treated in files of a given name pattern
#+ in a directory. There are several machines that access
#+ this directory, and I want to distribute the work over these
#+ different boxen.
#  Then I usually nohup something like the following on every box:

while true
```

```
do
  for n in .iso.*
  do
    [ "$n" = ".iso.opts" ] && continue
    beta=${n#.iso.}
    [ -r .Iso.$beta ] && continue
    [ -r .lock.$beta ] && sleep 10 && continue
    lockfile -r0 .lock.$beta || continue
    echo -n "$beta: " `date`
    run-isotherm $beta
    date
    ls -alF .Iso.$beta
    [ -r .Iso.$beta ] && rm -f .lock.$beta
    continue 2
  done
  break
done

exit 0

#  The details, in particular the sleep N, are particular to my
#+ application, but the general pattern is:

while true
do
  for job in {pattern}
  do
    {job already done or running} && continue
    {mark job as running, do job, mark job as done}
    continue 2
  done
  break        # Or something like `sleep 600' to avoid termination.
done

#  This way the script will stop only when there are no more jobs to do
#+ (including jobs that were added during runtime). Through the use
#+ of appropriate lockfiles it can be run on several machines
#+ concurrently without duplication of calculations [which run a couple
#+ of hours in my case, so I really want to avoid this]. Also, as search
#+ always starts again from the beginning, one can encode priorities in
#+ the file names. Of course, one could also do this without `continue 2',
#+ but then one would have to actually check whether or not some job
#+ was done (so that we should immediately look for the next job) or not
#+ (in which case we terminate or sleep for a long time before checking
#+ for a new job).
```

The **continue N** construct is difficult to understand and tricky to use in any meaningful context. It is probably best avoided.

11.4. Testing and Branching

The **case** and **select** constructs are technically not loops, since they do not iterate the execution of a code block. Like loops, however, they direct program flow according to conditions at the top or bottom of the block.

Controlling program flow in a code block

case (in) / esac

The **case** construct is the shell scripting analog to `switch` in **C/C++**. It permits branching to one of a number of code blocks, depending on condition tests. It serves as a kind of shorthand for multiple if/then/else statements and is an appropriate tool for creating menus.

case "$*variable*" in

"$*condition1*")
command...
;;

"$*condition2*")
command...
;;

esac

◊ Quoting the variables is not mandatory, since word splitting does not take place.
◊ Each test line ends with a right paren). [54]
◊ Each condition block ends with a *double* semicolon ;;.
◊ If a condition tests *true*, then the associated commands execute and the **case** block terminates.
◊ The entire **case** block ends with an **esac** (*case* spelled backwards).

Example 11-25. Using *case*

```
#!/bin/bash
# Testing ranges of characters.

echo; echo "Hit a key, then hit return."
read Keypress

case "$Keypress" in
  [[:lower:]]   ) echo "Lowercase letter";;
  [[:upper:]]   ) echo "Uppercase letter";;
  [0-9]         ) echo "Digit";;
  *             ) echo "Punctuation, whitespace, or other";;
esac      #  Allows ranges of characters in [square brackets],
          #+ or POSIX ranges in [[double square brackets.

#  In the first version of this example,
#+ the tests for lowercase and uppercase characters were
#+ [a-z] and [A-Z].
#  This no longer works in certain locales and/or Linux distros.
#  POSIX is more portable.
#  Thanks to Frank Wang for pointing this out.

#  Exercise:
#  --------
#  As the script stands, it accepts a single keystroke, then terminates.
#  Change the script so it accepts repeated input,
#+ reports on each keystroke, and terminates only when "X" is hit.
#  Hint: enclose everything in a "while" loop.
```

```
exit 0
```

Example 11-26. Creating menus using *case*

```
#!/bin/bash

# Crude address database

clear # Clear the screen.

echo "          Contact List"
echo "          ------- ----"
echo "Choose one of the following persons:"
echo
echo "[E]vans, Roland"
echo "[J]ones, Mildred"
echo "[S]mith, Julie"
echo "[Z]ane, Morris"
echo

read person

case "$person" in
# Note variable is quoted.

  "E" | "e" )
  # Accept upper or lowercase input.
  echo
  echo "Roland Evans"
  echo "4321 Flash Dr."
  echo "Hardscrabble, CO 80753"
  echo "(303) 734-9874"
  echo "(303) 734-9892 fax"
  echo "revans@zzy.net"
  echo "Business partner & old friend"
  ;;
# Note double semicolon to terminate each option.

  "J" | "j" )
  echo
  echo "Mildred Jones"
  echo "249 E. 7th St., Apt. 19"
  echo "New York, NY 10009"
  echo "(212) 533-2814"
  echo "(212) 533-9972 fax"
  echo "milliej@loisaida.com"
  echo "Ex-girlfriend"
  echo "Birthday: Feb. 11"
  ;;

# Add info for Smith & Zane later.

          * )
   # Default option.
   # Empty input (hitting RETURN) fits here, too.
   echo
   echo "Not yet in database."
  ;;

esac
```

```
echo

#  Exercise:
#  --------
#  Change the script so it accepts multiple inputs,
#+ instead of terminating after displaying just one address.

exit 0
```

An exceptionally clever use of **case** involves testing for command-line parameters.

```
#! /bin/bash

case "$1" in
  "") echo "Usage: ${0##*/} <filename>"; exit $E_PARAM;;
                      # No command-line parameters,
                      # or first parameter empty.
# Note that ${0##*/} is ${var##pattern} param substitution.
                      # Net result is $0.

  -*) FILENAME=./$1;;   #  If filename passed as argument ($1)
                      #+ starts with a dash,
                      #+ replace it with ./$1
                      #+ so further commands don't interpret it
                      #+ as an option.

  * ) FILENAME=$1;;     # Otherwise, $1.
esac
```

Here is a more straightforward example of command-line parameter handling:

```
#! /bin/bash


while [ $# -gt 0 ]; do    # Until you run out of parameters . . .
  case "$1" in
    -d|--debug)
              # "-d" or "--debug" parameter?
              DEBUG=1
              ;;
    -c|--conf)
              CONFFILE="$2"
              shift
              if [ ! -f $CONFFILE ]; then
                echo "Error: Supplied file doesn't exist!"
                exit $E_CONFFILE     # File not found error.
              fi
              ;;
  esac
  shift       # Check next set of parameters.
done

#  From Stefano Falsetto's "Log2Rot" script,
#+ part of his "rottlog" package.
#  Used with permission.
```

Example 11-27. Using *command substitution* to generate the *case* variable

```
#!/bin/bash
# case-cmd.sh: Using command substitution to generate a "case" variable.
```

```
case $( arch ) in   # $( arch ) returns machine architecture.
                    # Equivalent to 'uname -m' ...
  i386 ) echo "80386-based machine";;
  i486 ) echo "80486-based machine";;
  i586 ) echo "Pentium-based machine";;
  i686 ) echo "Pentium2+-based machine";;
  *    ) echo "Other type of machine";;
esac

exit 0
```

A **case** construct can filter strings for globbing patterns.

Example 11-28. Simple string matching

```
#!/bin/bash
# match-string.sh: Simple string matching
#                  using a 'case' construct.

match_string ()
{ # Exact string match.
  MATCH=0
  E_NOMATCH=90
  PARAMS=2     # Function requires 2 arguments.
  E_BAD_PARAMS=91

  [ $# -eq $PARAMS ] || return $E_BAD_PARAMS

  case "$1" in
  "$2") return $MATCH;;
  *   ) return $E_NOMATCH;;
  esac

}


a=one
b=two
c=three
d=two


match_string $a     # wrong number of parameters
echo $?             # 91

match_string $a $b  # no match
echo $?             # 90

match_string $b $d  # match
echo $?             # 0


exit 0
```

Example 11-29. Checking for alphabetic input

```
#!/bin/bash
```

```
# isalpha.sh: Using a "case" structure to filter a string.

SUCCESS=0
FAILURE=1   #  Was FAILURE=-1,
            #+ but Bash no longer allows negative return value.

isalpha ()  # Tests whether *first character* of input string is alphabetic.
{
if [ -z "$1" ]                # No argument passed?
then
  return $FAILURE
fi

case "$1" in
  [a-zA-Z]*) return $SUCCESS;;  # Begins with a letter?
  *        ) return $FAILURE;;
esac
}             # Compare this with "isalpha ()" function in C.


isalpha2 ()   # Tests whether *entire string* is alphabetic.
{
  [ $# -eq 1 ] || return $FAILURE

  case $1 in
  *[!a-zA-Z]*|"") return $FAILURE;;
               *) return $SUCCESS;;
  esac
}

isdigit ()    # Tests whether *entire string* is numerical.
{             # In other words, tests for integer variable.
  [ $# -eq 1 ] || return $FAILURE

  case $1 in
    *[!0-9]*|"") return $FAILURE;;
              *) return $SUCCESS;;
  esac
}



check_var ()  # Front-end to isalpha ().
{
if isalpha "$@"
then
  echo "\"$*\" begins with an alpha character."
  if isalpha2 "$@"
  then        # No point in testing if first char is non-alpha.
    echo "\"$*\" contains only alpha characters."
  else
    echo "\"$*\" contains at least one non-alpha character."
  fi
else
  echo "\"$*\" begins with a non-alpha character."
              # Also "non-alpha" if no argument passed.
fi

echo

}
```

```
digit_check ()  # Front-end to isdigit ().
{
if isdigit "$@"
then
  echo "\"$*\" contains only digits [0 - 9]."
else
  echo "\"$*\" has at least one non-digit character."
fi

echo

}

a=23skidoo
b=H3llo
c=-What?
d=What?
e=$(echo $b)   # Command substitution.
f=AbcDef
g=27234
h=27a34
i=27.34

check_var $a
check_var $b
check_var $c
check_var $d
check_var $e
check_var $f
check_var     # No argument passed, so what happens?
#
digit_check $g
digit_check $h
digit_check $i


exit 0        # Script improved by S.C.

# Exercise:
# --------
#  Write an 'isfloat ()' function that tests for floating point numbers.
#  Hint: The function duplicates 'isdigit ()',
#+ but adds a test for a mandatory decimal point.
```

select

The **select** construct, adopted from the Korn Shell, is yet another tool for building menus.

select *`variable`* [in *`list`*]
do
 `command`...
 break
done

This prompts the user to enter one of the choices presented in the variable list. Note that **select** uses the `$PS3` prompt (`#? `) by default, but this may be changed.

Example 11-30. Creating menus using *select*

```
#!/bin/bash

PS3='Choose your favorite vegetable: ' # Sets the prompt string.
                                       # Otherwise it defaults to #? .

echo

select vegetable in "beans" "carrots" "potatoes" "onions" "rutabagas"
do
  echo
  echo "Your favorite veggie is $vegetable."
  echo "Yuck!"
  echo
  break  # What happens if there is no 'break' here?
done

exit

# Exercise:
# --------
#  Fix this script to accept user input not specified in
#+ the "select" statement.
#  For example, if the user inputs "peas,"
#+ the script would respond "Sorry. That is not on the menu."
```

If **`in list`** is omitted, then **select** uses the list of command line arguments (`$@`) passed to the script or the function containing the **select** construct.

Compare this to the behavior of a

for `variable` [in `list`]

construct with the **`in list`** omitted.

Example 11-31. Creating menus using *select* in a function

```
#!/bin/bash

PS3='Choose your favorite vegetable: '

echo

choice_of()
{
select vegetable
# [in list] omitted, so 'select' uses arguments passed to function.
do
  echo
  echo "Your favorite veggie is $vegetable."
  echo "Yuck!"
  echo
  break
done
}

choice_of beans rice carrots radishes rutabaga spinach
#         $1    $2   $3      $4       $5       $6
#         passed to choice_of() function
```

```
exit 0
```

See also Example 37-3.

Chapter 12. Command Substitution

Command substitution reassigns the output of a command [55] or even multiple commands; it literally plugs the command output into another context. [56]

The classic form of command substitution uses *backquotes* (`...`). Commands within backquotes (backticks) generate command-line text.

```
script_name=`basename $0`
echo "The name of this script is $script_name."
```

The output of commands can be used as arguments to another command, to set a variable, and even for generating the argument list in a for loop.

```
rm `cat filename`   # "filename" contains a list of files to delete.
#
# S. C. points out that "arg list too long" error might result.
# Better is              xargs rm -- < filename
# ( -- covers those cases where "filename" begins with a "-" )

textfile_listing=`ls *.txt`
# Variable contains names of all *.txt files in current working directory.
echo $textfile_listing

textfile_listing2=$(ls *.txt)   # The alternative form of command substitution.
echo $textfile_listing2
# Same result.

# A possible problem with putting a list of files into a single string
# is that a newline may creep in.
#
# A safer way to assign a list of files to a parameter is with an array.
#      shopt -s nullglob    # If no match, filename expands to nothing.
#      textfile_listing=( *.txt )
#
# Thanks, S.C.
```

Command substitution invokes a subshell.

Command substitution may result in word splitting.

```
COMMAND `echo a b`     # 2 args: a and b

COMMAND "`echo a b`"   # 1 arg: "a b"

COMMAND `echo`         # no arg

COMMAND "`echo`"       # one empty arg


# Thanks, S.C.
```

Even when there is no word splitting, command substitution can remove trailing newlines.

```
# cd "`pwd`"  # This should always work.
# However...
```

```
mkdir 'dir with trailing newline
'

cd 'dir with trailing newline
'

cd "`pwd`"  # Error message:
# bash: cd: /tmp/file with trailing newline: No such file or directory

cd "$PWD"   # Works fine.





old_tty_setting=$(stty -g)   # Save old terminal setting.
echo "Hit a key "
stty -icanon -echo           # Disable "canonical" mode for terminal.
                             # Also, disable *local* echo.
key=$(dd bs=1 count=1 2> /dev/null)   # Using 'dd' to get a keypress.
stty "$old_tty_setting"      # Restore old setting.
echo "You hit ${#key} key."  # ${#variable} = number of characters in $variable
#
# Hit any key except RETURN, and the output is "You hit 1 key."
# Hit RETURN, and it's "You hit 0 key."
# The newline gets eaten in the command substitution.

#Code snippet by Stéphane Chazelas.
```

Using **echo** to output an *unquoted* variable set with command substitution removes trailing newlines characters from the output of the reassigned command(s). This can cause unpleasant surprises.

```
dir_listing=`ls -l`
echo $dir_listing     # unquoted

# Expecting a nicely ordered directory listing.

# However, what you get is:
# total 3 -rw-rw-r-- 1 bozo bozo 30 May 13 17:15 1.txt -rw-rw-r-- 1 bozo
# bozo 51 May 15 20:57 t2.sh -rwxr-xr-x 1 bozo bozo 217 Mar 5 21:13 wi.sh

# The newlines disappeared.


echo "$dir_listing"   # quoted
# -rw-rw-r--    1 bozo       30 May 13 17:15 1.txt
# -rw-rw-r--    1 bozo       51 May 15 20:57 t2.sh
# -rwxr-xr-x    1 bozo      217 Mar  5 21:13 wi.sh
```

Command substitution even permits setting a variable to the contents of a file, using either redirection or the cat command.

```
variable1=`<file1`      #  Set "variable1" to contents of "file1".
variable2=`cat file2`   #  Set "variable2" to contents of "file2".
                        #  This, however, forks a new process,
                        #+ so the line of code executes slower than the above version.

#  Note that the variables may contain embedded whitespace,
#+ or even (horrors), control characters.
```

```
#  It is not necessary to explicitly assign a variable.
echo "` <$0`"           # Echoes the script itself to stdout.
```

```
#  Excerpts from system file, /etc/rc.d/rc.sysinit
#+ (on a Red Hat Linux installation)


if [ -f /fsckoptions ]; then
        fsckoptions=`cat /fsckoptions`
...
fi
#
#
if [ -e "/proc/ide/${disk[$device]}/media" ] ; then
             hdmedia=`cat /proc/ide/${disk[$device]}/media`
...
fi
#
#
if [ ! -n "`uname -r | grep -- "-"`" ]; then
       ktag="`cat /proc/version`"
...
fi
#
#
if [ $usb = "1" ]; then
    sleep 5
    mouseoutput=`cat /proc/bus/usb/devices 2>/dev/null|grep -E "^I.*Cls=03.*Prot=02"`
    kbdoutput=`cat /proc/bus/usb/devices 2>/dev/null|grep -E "^I.*Cls=03.*Prot=01"`
...
fi
```

Do not set a variable to the contents of a *long* text file unless you have a very good reason for doing so. Do not set a variable to the contents of a *binary* file, even as a joke.

Example 12-1. Stupid script tricks

```
#!/bin/bash
# stupid-script-tricks.sh: Don't try this at home, folks.
# From "Stupid Script Tricks," Volume I.

exit 99  ### Comment out this line if you dare.

dangerous_variable=`cat /boot/vmlinuz`   # The compressed Linux kernel itself.

echo "string-length of \$dangerous_variable = ${#dangerous_variable}"
# string-length of $dangerous_variable = 794151
# (Newer kernels are bigger.)
# Does not give same count as 'wc -c /boot/vmlinuz'.

# echo "$dangerous_variable"
# Don't try this! It would hang the script.


#  The document author is aware of no useful applications for
#+ setting a variable to the contents of a binary file.
```

```
exit 0
```

Notice that a *buffer overrun* does not occur. This is one instance where an interpreted language, such as Bash, provides more protection from programmer mistakes than a compiled language.

Command substitution permits setting a variable to the output of a loop. The key to this is grabbing the output of an echo command within the loop.

Example 12-2. Generating a variable from a loop

```
#!/bin/bash
# csubloop.sh: Setting a variable to the output of a loop.

variable1=`for i in 1 2 3 4 5
do
  echo -n "$i"                 #  The 'echo' command is critical
done`                          #+ to command substitution here.

echo "variable1 = $variable1"  # variable1 = 12345


i=0
variable2=`while [ "$i" -lt 10 ]
do
  echo -n "$i"                 # Again, the necessary 'echo'.
  let "i += 1"                 # Increment.
done`

echo "variable2 = $variable2"  # variable2 = 0123456789

#  Demonstrates that it's possible to embed a loop
#+ inside a variable declaration.

exit 0
```

Command substitution makes it possible to extend the toolset available to Bash. It is simply a matter of writing a program or script that outputs to `stdout` (like a well-behaved UNIX tool should) and assigning that output to a variable.

```
#include <stdio.h>

/*  "Hello, world." C program  */

int main()
{
  printf( "Hello, world.\n" );
  return (0);
}
```

```
bash$ gcc -o hello hello.c
```

```
#!/bin/bash
# hello.sh

greeting=`./hello`
```

```
echo $greeting
```

```
bash$ sh hello.sh
Hello, world.
```

The **$(...)** form has superseded backticks for command substitution.

```
output=$(sed -n /"$1"/p $file)   # From "grp.sh"        example.

# Setting a variable to the contents of a text file.
File_contents1=$(cat $file1)
File_contents2=$(<$file2)        # Bash permits this also.
```

The **$(...)** form of command substitution treats a double backslash in a different way than **`...`**.

```
bash$ echo `echo \\`


bash$ echo $(echo \\)
\
```

The **$(...)** form of command substitution permits nesting. [57]

```
word_count=$( wc -w $(echo * | awk '{print $8}') )
```

Or, for something a bit more elaborate . . .

Example 12-3. Finding anagrams

```
#!/bin/bash
# agram2.sh
# Example of nested command substitution.

#  Uses "anagram" utility
#+ that is part of the author's "yawl" word list package.
#  http://ibiblio.org/pub/Linux/libs/yawl-0.3.2.tar.gz
#  http://bash.deta.in/yawl-0.3.2.tar.gz

E_NOARGS=86
E_BADARG=87
MINLEN=7

if [ -z "$1" ]
then
  echo "Usage $0 LETTERSET"
  exit $E_NOARGS         # Script needs a command-line argument.
elif [ ${#1} -lt $MINLEN ]
then
  echo "Argument must have at least $MINLEN letters."
  exit $E_BADARG
fi



FILTER='.......'         # Must have at least 7 letters.
#       1234567
Anagrams=( $(echo $(anagram $1 | grep $FILTER) ) )
```

```
#          $(     $(  nested command sub.    ) )
#        (              array assignment         )

echo
echo "${#Anagrams[*]}  7+ letter anagrams found"
echo
echo ${Anagrams[0]}      # First anagram.
echo ${Anagrams[1]}      # Second anagram.
                         # Etc.

# echo "${Anagrams[*]}"  # To list all the anagrams in a single line . . .

#  Look ahead to the Arrays chapter for enlightenment on
#+ what's going on here.

# See also the agram.sh script for an exercise in anagram finding.

exit $?
```

Examples of command substitution in shell scripts:

1. Example 11-8
2. Example 11-27
3. Example 9-16
4. Example 16-3
5. Example 16-22
6. Example 16-17
7. Example 16-54
8. Example 11-14
9. Example 11-11
10. Example 16-32
11. Example 20-8
12. Example A-16
13. Example 29-3
14. Example 16-47
15. Example 16-48
16. Example 16-49

Chapter 13. Arithmetic Expansion

Arithmetic expansion provides a powerful tool for performing (integer) arithmetic operations in scripts. Translating a string into a numerical expression is relatively straightforward using *backticks*, *double parentheses*, or *let*.

Variations

Arithmetic expansion with backticks (often used in conjunction with expr)

```
z=`expr $z + 3`          # The 'expr' command performs the expansion.
```

Arithmetic expansion with double parentheses, and using let

The use of *backticks* (*backquotes*) in arithmetic expansion has been superseded by *double parentheses* -- **((...))** and **$((...))** -- and also by the very convenient let construction.

```
z=$(($z+3))
z=$((z+3))                                  #  Also correct.
                                            #  Within double parentheses,
                                            #+ parameter dereferencing
                                            #+ is optional.

# $((EXPRESSION)) is arithmetic expansion.  #  Not to be confused with
                                            #+ command substitution.



# You may also use operations within double parentheses without assignment.

  n=0
  echo "n = $n"                             # n = 0

  (( n += 1 ))                              # Increment.
# (( $n += 1 )) is incorrect!
  echo "n = $n"                             # n = 1


let z=z+3
let "z += 3"  #  Quotes permit the use of spaces in variable assignment.
              #  The 'let' operator actually performs arithmetic evaluation,
              #+ rather than expansion.
```

Examples of arithmetic expansion in scripts:

1. Example 16-9
2. Example 11-15
3. Example 27-1
4. Example 27-11
5. Example A-16

Chapter 14. Recess Time

This bizarre little intermission gives the reader a chance to relax and maybe laugh a bit.

```
Fellow Linux user, greetings! You are reading something which
will bring you luck and good fortune. Just e-mail a copy of
this document to 10 of your friends. Before making the copies,
send a 100-line Bash script to the first person on the list
at the bottom of this letter. Then delete their name and add
yours to the bottom of the list.

Don't break the chain! Make the copies within 48 hours.
Wilfred P. of Brooklyn failed to send out his ten copies and
woke the next morning to find his job description changed
to "COBOL programmer." Howard L. of Newport News sent
out his ten copies and within a month had enough hardware
to build a 100-node Beowulf cluster dedicated to playing
Tuxracer. Amelia V. of Chicago laughed at this letter
and broke the chain. Shortly thereafter, a fire broke out
in her terminal and she now spends her days writing
documentation for MS Windows.

Don't break the chain!  Send out your ten copies today!
```

Courtesy 'NIX "fortune cookies", with some alterations and many apologies

Part 4. Commands

Mastering the commands on your Linux machine is an indispensable prelude to writing effective shell scripts.

This section covers the following commands:

- . (See also source)
- ac
- adduser
- agetty
- agrep
- ar
- arch
- at
- autoload
- awk (See also Using **awk** for math operations)
- badblocks
- banner
- basename
- batch
- bc
- bg
- bind
- bison
- builtin
- bzgrep
- bzip2
- cal
- caller
- cat
- cd
- chattr
- chfn
- chgrp
- chkconfig
- chmod
- chown
- chroot
- cksum
- clear
- clock
- cmp
- col
- colrm
- column
- comm
- command
- compgen

- complete
- compress
- coproc
- cp
- cpio
- cron
- crypt
- csplit
- cu
- cut
- date
- dc
- dd
- debugfs
- declare
- depmod
- df
- dialog
- diff
- diff3
- diffstat
- dig
- dirname
- dirs
- disown
- dmesg
- doexec
- dos2unix
- du
- dump
- dumpe2fs
- e2fsck
- echo
- egrep
- enable
- enscript
- env
- eqn
- eval
- exec
- exit (Related topic: exit status)
- expand
- export
- expr
- factor
- false
- fdformat
- fdisk
- fg
- fgrep
- file

- find
- finger
- flex
- flock
- fmt
- fold
- free
- fsck
- ftp
- fuser
- getfacl
- getopt
- getopts
- gettext
- getty
- gnome-mount
- grep
- groff
- groupmod
- groups (Related topic: the $GROUPS variable)
- gs
- gzip
- halt
- hash
- hdparm
- head
- help
- hexdump
- host
- hostid
- hostname (Related topic: the $HOSTNAME variable)
- hwclock
- iconv
- id (Related topic: the $UID variable)
- ifconfig
- info
- infocmp
- init
- insmod
- install
- ip
- ipcalc
- iptables
- iwconfig
- jobs
- join
- jot
- kill
- killall
- last
- lastcomm

- more
- mount
- msgfmt
- mv
- nc
- netconfig
- netstat
- newgrp
- nice
- nl
- nm
- nmap
- nohup
- nslookup
- objdump
- od
- openssl
- passwd
- paste
- patch (Related topic: diff)
- pathchk
- pax
- pgrep
- pidof
- ping
- pkill
- popd
- pr
- printenv
- printf
- procinfo
- ps
- pstree
- ptx
- pushd
- pwd (Related topic: the $PWD variable)
- quota
- rcp
- rdev
- rdist
- read
- readelf
- readlink
- readonly
- reboot
- recode
- renice
- reset
- resize
- restore
- rev

- uucp
- uudecode
- uuencode
- uux
- vacation
- vdir
- vmstat
- vrfy
- w
- wait
- wall
- watch
- wc
- wget
- whatis
- whereis
- which
- who
- whoami
- whois
- write
- xargs
- xrandr
- xz
- yacc
- yes
- zcat
- zdiff
- zdump
- zegrep
- zfgrep
- zgrep
- zip

Table of Contents

Chapter 15. Internal Commands and Builtins

A *builtin* is a **command** contained within the Bash tool set, literally *built in*. This is either for performance reasons -- builtins execute faster than external commands, which usually require *forking off* [58] a separate process -- or because a particular builtin needs direct access to the shell internals.

When a command or the shell itself initiates (or *spawns*) a new subprocess to carry out a task, this is called *forking*. This new process is the *child*, and the process that *forked* it off is the *parent*. While the *child process* is doing its work, the *parent process* is still executing.

Note that while a *parent process* gets the *process ID* of the *child process*, and can thus pass arguments to it, *the reverse is not true*. This can create problems that are subtle and hard to track down.

Example 15-1. A script that spawns multiple instances of itself

```
#!/bin/bash
# spawn.sh


PIDS=$(pidof sh $0)  # Process IDs of the various instances of this script.
P_array=( $PIDS )    # Put them in an array (why?).
echo $PIDS           # Show process IDs of parent and child processes.
let "instances = ${#P_array[*]} - 1"  # Count elements, less 1.
                                      # Why subtract 1?
echo "$instances instance(s) of this script running."
echo "[Hit Ctl-C to exit.]"; echo


sleep 1              # Wait.
sh $0                # Play it again, Sam.

exit 0               # Not necessary; script will never get to here.
                     # Why not?

#  After exiting with a Ctl-C,
#+ do all the spawned instances of the script die?
#  If so, why?

# Note:
# ----
# Be careful not to run this script too long.
# It will eventually eat up too many system resources.

#  Is having a script spawn multiple instances of itself
#+ an advisable scripting technique.
#  Why or why not?
```

Generally, a Bash *builtin* does not fork a subprocess when it executes within a script. An external system command or filter in a script usually *will* fork a subprocess.

A builtin may be a synonym to a system command of the same name, but Bash reimplements it internally. For example, the Bash **echo** command is not the same as `/bin/echo`, although their behavior is almost identical.

```
#!/bin/bash

echo "This line uses the \"echo\" builtin."
/bin/echo "This line uses the /bin/echo system command."
```

A *keyword* is a *reserved* word, token or operator. Keywords have a special meaning to the shell, and indeed are the building blocks of the shell's syntax. As examples, *for*, *while*, *do*, and *!* are keywords. Similar to a builtin, a keyword is hard-coded into Bash, but unlike a *builtin*, a keyword is not in itself a command, but *a subunit of a command construct*. [59]

I/O

echo

prints (to `stdout`) an expression or variable (see Example 4-1).

```
echo Hello
echo $a
```

An **echo** requires the `-e` option to print escaped characters. See Example 5-2.

Normally, each **echo** command prints a terminal newline, but the `-n` option suppresses this.

An **echo** can be used to feed a sequence of commands down a pipe.

```
if echo "$VAR" | grep -q txt   # if [[ $VAR = *txt* ]]
then
  echo "$VAR contains the substring sequence \"txt\""
fi
```

An **echo**, in combination with command substitution can set a variable.

`a=`echo "HELLO" | tr A-Z a-z``

See also Example 16-22, Example 16-3, Example 16-47, and Example 16-48.

Be aware that **echo `command`** deletes any linefeeds that the output of *`command`* generates.

The $IFS (internal field separator) variable normally contains \n (linefeed) as one of its set of whitespace characters. Bash therefore splits the output of *`command`* at linefeeds into arguments to **echo**. Then **echo** outputs these arguments, separated by spaces.

```
bash$ ls -l /usr/share/apps/kjezz/sounds
-rw-r--r--    1 root     root         1407 Nov  7  2000 reflect.au
 -rw-r--r--    1 root     root          362 Nov  7  2000 seconds.au




bash$ echo `ls -l /usr/share/apps/kjezz/sounds`
total 40 -rw-r--r-- 1 root root 716 Nov 7 2000 reflect.au -rw-r--r-- 1 root root ...
```

So, how can we embed a linefeed within an echoed character string?

```
# Embedding a linefeed?
echo "Why doesn't this string \n split on two lines?"
# Doesn't split.
```

```
# Let's try something else.

echo

echo $"A line of text containing
a linefeed."
# Prints as two distinct lines (embedded linefeed).
# But, is the "$" variable prefix really necessary?

echo

echo "This string splits
on two lines."
# No, the "$" is not needed.

echo
echo "---------------"
echo

echo -n $"Another line of text containing
a linefeed."
# Prints as two distinct lines (embedded linefeed).
# Even the -n option fails to suppress the linefeed here.

echo
echo
echo "---------------"
echo
echo

# However, the following doesn't work as expected.
# Why not? Hint: Assignment to a variable.
string1=$"Yet another line of text containing
a linefeed (maybe)."

echo $string1
# Yet another line of text containing a linefeed (maybe).
#                                    ^
# Linefeed becomes a space.

# Thanks, Steve Parker, for pointing this out.
```

This command is a shell builtin, and not the same as `/bin/echo`, although its behavior is similar.

```
bash$ type -a echo
echo is a shell builtin
 echo is /bin/echo
```

printf

The **printf**, formatted print, command is an enhanced **echo**. It is a limited variant of the *C* language `printf()` library function, and its syntax is somewhat different.

printf *`format-string`*... *`parameter`*...

This is the Bash *builtin* version of the `/bin/printf` or `/usr/bin/printf` command. See the **printf** manpage (of the system command) for in-depth coverage.

Older versions of Bash may not support **printf**.

Example 15-2. *printf* in action

```
#!/bin/bash
# printf demo

declare -r PI=3.14159265358979     # Read-only variable, i.e., a constant.
declare -r DecimalConstant=31373

Message1="Greetings,"
Message2="Earthling."

echo

printf "Pi to 2 decimal places = %1.2f" $PI
echo
printf "Pi to 9 decimal places = %1.9f" $PI  # It even rounds off correctly.

printf "\n"                                  # Prints a line feed,
                                             # Equivalent to 'echo' . . .

printf "Constant = \t%d\n" $DecimalConstant  # Inserts tab (\t).

printf "%s %s \n" $Message1 $Message2

echo

# ==========================================#
# Simulation of C function, sprintf().
# Loading a variable with a formatted string.

echo

Pi12=$(printf "%1.12f" $PI)
echo "Pi to 12 decimal places = $Pi12"      # Roundoff error!

Msg=`printf "%s %s \n" $Message1 $Message2`
echo $Msg; echo $Msg

#  As it happens, the 'sprintf' function can now be accessed
#+ as a loadable module to Bash,
#+ but this is not portable.

exit 0
```

Formatting error messages is a useful application of **printf**

```
E_BADDIR=85

var=nonexistent_directory

error()
{
  printf "$@" >&2
  # Formats positional params passed, and sends them to stderr.
  echo
  exit $E_BADDIR
}

cd $var || error $"Can't cd to %s." "$var"
```

```
# Thanks, S.C.
```

See also Example 36-17.

read

"Reads" the value of a variable from `stdin`, that is, interactively fetches input from the keyboard. The `-a` option lets **read** get array variables (see Example 27-6).

Example 15-3. Variable assignment, using *read*

```
#!/bin/bash
# "Reading" variables.

echo -n "Enter the value of variable 'var1': "
# The -n option to echo suppresses newline.

read var1
# Note no '$' in front of var1, since it is being set.

echo "var1 = $var1"


echo

# A single 'read' statement can set multiple variables.
echo -n "Enter the values of variables 'var2' and 'var3' "
echo =n "(separated by a space or tab): "
read var2 var3
echo "var2 = $var2      var3 = $var3"
#  If you input only one value,
#+ the other variable(s) will remain unset (null).

exit 0
```

A **read** without an associated variable assigns its input to the dedicated variable $REPLY.

Example 15-4. What happens when *read* has no variable

```
#!/bin/bash
# read-novar.sh

echo

# -------------------------- #
echo -n "Enter a value: "
read var
echo "\"var\" = "$var""
# Everything as expected here.
# -------------------------- #

echo

# ------------------------------------------------------------------ #
echo -n "Enter another value: "
read           #  No variable supplied for 'read', therefore...
               #+ Input to 'read' assigned to default variable, $REPLY.
var="$REPLY"
echo "\"var\" = "$var""
# This is equivalent to the first code block.
```

```
# ------------------------------------------------------------------ #

echo
echo "========================="
echo


#  This example is similar to the "reply.sh" script.
#  However, this one shows that $REPLY is available
#+ even after a 'read' to a variable in the conventional way.


# ================================================================= #

#  In some instances, you might wish to discard the first value read.
#  In such cases, simply ignore the $REPLY variable.

{ # Code block.
read            # Line 1, to be discarded.
read line2      # Line 2, saved in variable.
  } <$0
echo "Line 2 of this script is:"
echo "$line2"   #   # read-novar.sh
echo            #   #!/bin/bash  line discarded.

# See also the soundcard-on.sh script.

exit 0
```

Normally, inputting a **** suppresses a newline during input to a **read**. The `-r` option causes an inputted **** to be interpreted literally.

Example 15-5. Multi-line input to *read*

```
#!/bin/bash

echo

echo "Enter a string terminated by a \\, then press <ENTER>."
echo "Then, enter a second string (no \\ this time), and again press <ENTER>."

read var1     # The "\" suppresses the newline, when reading $var1.
              #     first line \
              #     second line

echo "var1 = $var1"
#     var1 = first line second line

#  For each line terminated by a "\"
#+ you get a prompt on the next line to continue feeding characters into var1.

echo; echo

echo "Enter another string terminated by a \\ , then press <ENTER>."
read -r var2  # The -r option causes the "\" to be read literally.
              #     first line \

echo "var2 = $var2"
#     var2 = first line \
```

```
# Data entry terminates with the first <ENTER>.

echo

exit 0
```

The **read** command has some interesting options that permit echoing a prompt and even reading keystrokes without hitting **ENTER**.

```
# Read a keypress without hitting ENTER.

read -s -n1 -p "Hit a key " keypress
echo; echo "Keypress was "\"$keypress\""."

# -s option means do not echo input.
# -n N option means accept only N characters of input.
# -p option means echo the following prompt before reading input.

# Using these options is tricky, since they need to be in the correct order.
```

The `-n` option to **read** also allows detection of the **arrow keys** and certain of the other unusual keys.

Example 15-6. Detecting the arrow keys

```
#!/bin/bash
# arrow-detect.sh: Detects the arrow keys, and a few more.
# Thank you, Sandro Magi, for showing me how.

# --------------------------------------------
# Character codes generated by the keypresses.
arrowup='\[A'
arrowdown='\[B'
arrowrt='\[C'
arrowleft='\[D'
insert='\[2'
delete='\[3'
# --------------------------------------------

SUCCESS=0
OTHER=65

echo -n "Press a key...  "
# May need to also press ENTER if a key not listed above pressed.
read -n3 key                      # Read 3 characters.

echo -n "$key" | grep "$arrowup"  #Check if character code detected.
if [ "$?" -eq $SUCCESS ]
then
  echo "Up-arrow key pressed."
  exit $SUCCESS
fi

echo -n "$key" | grep "$arrowdown"
if [ "$?" -eq $SUCCESS ]
then
  echo "Down-arrow key pressed."
  exit $SUCCESS
fi
```

```
echo -n "$key" | grep "$arrowrt"
if [ "$?" -eq $SUCCESS ]
then
  echo "Right-arrow key pressed."
  exit $SUCCESS
fi

echo -n "$key" | grep "$arrowleft"
if [ "$?" -eq $SUCCESS ]
then
  echo "Left-arrow key pressed."
  exit $SUCCESS
fi

echo -n "$key" | grep "$insert"
if [ "$?" -eq $SUCCESS ]
then
  echo "\"Insert\" key pressed."
  exit $SUCCESS
fi

echo -n "$key" | grep "$delete"
if [ "$?" -eq $SUCCESS ]
then
  echo "\"Delete\" key pressed."
  exit $SUCCESS
fi


echo " Some other key pressed."

exit $OTHER

# ========================================= #

#  Mark Alexander came up with a simplified
#+ version of the above script (Thank you!).
#  It eliminates the need for grep.

#!/bin/bash

  uparrow=$'\x1b[A'
  downarrow=$'\x1b[B'
  leftarrow=$'\x1b[D'
  rightarrow=$'\x1b[C'

  read -s -n3 -p "Hit an arrow key: " x

  case "$x" in
  $uparrow)
     echo "You pressed up-arrow"
     ;;
  $downarrow)
     echo "You pressed down-arrow"
     ;;
  $leftarrow)
     echo "You pressed left-arrow"
     ;;
  $rightarrow)
     echo "You pressed right-arrow"
     ;;
```

```
  esac

exit $?

# ========================================= #

# Antonio Macchi has a simpler alternative.

#!/bin/bash

while true
do
  read -sn1 a
  test "$a" == `echo -en "\e"` || continue
  read -sn1 a
  test "$a" == "[" || continue
  read -sn1 a
  case "$a" in
    A)  echo "up";;
    B)  echo "down";;
    C)  echo "right";;
    D)  echo "left";;
  esac
done

# ========================================= #

#  Exercise:
#  --------
#  1) Add detection of the "Home," "End," "PgUp," and "PgDn" keys.
```

The `-n` option to **read** will not detect the **ENTER** (newline) key.

The `-t` option to **read** permits timed input (see Example 9-4 and Example A-41).

The `-u` option takes the file descriptor of the target file.

The **read** command may also "read" its variable value from a file redirected to `stdin`. If the file contains more than one line, only the first line is assigned to the variable. If **read** has more than one parameter, then each of these variables gets assigned a successive whitespace-delineated string. Caution!

Example 15-7. Using *read* with file redirection

```
#!/bin/bash

read var1 <data-file
echo "var1 = $var1"
# var1 set to the entire first line of the input file "data-file"

read var2 var3 <data-file
echo "var2 = $var2   var3 = $var3"
# Note non-intuitive behavior of "read" here.
# 1) Rewinds back to the beginning of input file.
# 2) Each variable is now set to a corresponding string,
#    separated by whitespace, rather than to an entire line of text.
```

```
# 3) The final variable gets the remainder of the line.
# 4) If there are more variables to be set than whitespace-terminated strings
#    on the first line of the file, then the excess variables remain empty.

echo "------------------------------------------------"

# How to resolve the above problem with a loop:
while read line
do
  echo "$line"
done <data-file
# Thanks, Heiner Steven for pointing this out.

echo "------------------------------------------------"

# Use $IFS (Internal Field Separator variable) to split a line of input to
# "read", if you do not want the default to be whitespace.

echo "List of all users:"
OIFS=$IFS; IFS=:       # /etc/passwd uses ":" for field separator.
while read name passwd uid gid fullname ignore
do
  echo "$name ($fullname)"
done </etc/passwd   # I/O redirection.
IFS=$OIFS              # Restore original $IFS.
# This code snippet also by Heiner Steven.



#  Setting the $IFS variable within the loop itself
#+ eliminates the need for storing the original $IFS
#+ in a temporary variable.
#  Thanks, Dim Segebart, for pointing this out.
echo "------------------------------------------------"
echo "List of all users:"

while IFS=: read name passwd uid gid fullname ignore
do
  echo "$name ($fullname)"
done </etc/passwd   # I/O redirection.

echo
echo "\$IFS still $IFS"

exit 0
```

Piping output to a *read*, using echo to set variables will fail.

Yet, piping the output of cat *seems* to work.

```
cat file1 file2 |
while read line
do
echo $line
done
```

However, as Bjön Eriksson shows:

Example 15-8. Problems reading from a pipe

```
#!/bin/sh
# readpipe.sh
# This example contributed by Bjon Eriksson.

### shopt -s lastpipe

last="(null)"
cat $0 |
while read line
do
    echo "{$line}"
    last=$line
done

echo
echo "++++++++++++++++++++++"
printf "\nAll done, last: $last\n" #  The output of this line
                                   #+ changes if you uncomment line 5.
                                   #  (Bash, version -ge 4.2 required.)

exit 0  # End of code.
        # (Partial) output of script follows.
        # The 'echo' supplies extra brackets.

#############################################

./readpipe.sh

{#!/bin/sh}
{last="(null)"}
{cat $0 |}
{while read line}
{do}
{echo "{$line}"}
{last=$line}
{done}
{printf "nAll done, last: $lastn"}


All done, last: (null)

The variable (last) is set within the loop/subshell
but its value does not persist outside the loop.
```

The *gendiff* script, usually found in `/usr/bin` on many Linux distros, pipes the output of find to a *while read* construct.

```
find $1 \( -name "*$2" -o -name ".*$2" \) -print |
while read f; do
. . .
```

It is possible to *paste* text into the input field of a *read* (but *not* multiple lines!). See Example A-38.

Filesystem

cd

The familiar **cd** change directory command finds use in scripts where execution of a command requires being in a specified directory.

```
(cd /source/directory && tar cf - . ) | (cd /dest/directory && tar xpvf -)
```

[from the previously cited example by Alan Cox]

The `-P` (physical) option to **cd** causes it to ignore symbolic links.

cd - changes to $OLDPWD, the previous working directory.

The **cd** command does not function as expected when presented with two forward slashes.

```
bash$ cd //
bash$ pwd
//
```

The output should, of course, be `/`. This is a problem both from the command-line and in a script.

pwd

Print Working Directory. This gives the user's (or script's) current directory (see Example 15-9). The effect is identical to reading the value of the builtin variable $PWD.

pushd, **popd**, **dirs**

This command set is a mechanism for bookmarking working directories, a means of moving back and forth through directories in an orderly manner. A pushdown stack is used to keep track of directory names. Options allow various manipulations of the directory stack.

`pushd dir-name` pushes the path *`dir-name`* onto the directory stack (to the *top* of the stack) and simultaneously changes the current working directory to *`dir-name`*

popd removes (pops) the top directory path name off the directory stack and simultaneously changes the current working directory to the directory now at the *top* of the stack.

dirs lists the contents of the directory stack (compare this with the $DIRSTACK variable). A successful **pushd** or **popd** will automatically invoke **dirs**.

Scripts that require various changes to the current working directory without hard-coding the directory name changes can make good use of these commands. Note that the implicit `$DIRSTACK` array variable, accessible from within a script, holds the contents of the directory stack.

Example 15-9. Changing the current working directory

```
#!/bin/bash

dir1=/usr/local
dir2=/var/spool

pushd $dir1
# Will do an automatic 'dirs' (list directory stack to stdout).
echo "Now in directory `pwd`." # Uses back-quoted 'pwd'.
```

```
# Now, do some stuff in directory 'dir1'.
pushd $dir2
echo "Now in directory `pwd`."

# Now, do some stuff in directory 'dir2'.
echo "The top entry in the DIRSTACK array is $DIRSTACK."
popd
echo "Now back in directory `pwd`."

# Now, do some more stuff in directory 'dir1'.
popd
echo "Now back in original working directory `pwd`."

exit 0

# What happens if you don't 'popd' -- then exit the script?
# Which directory do you end up in? Why?
```

Variables

let

The **let** command carries out *arithmetic* operations on variables. [60] In many cases, it functions as a less complex version of expr.

Example 15-10. Letting *let* do arithmetic.

```
#!/bin/bash

echo

let a=11            # Same as 'a=11'
let a=a+5           # Equivalent to  let "a = a + 5"
                    # (Double quotes and spaces make it more readable.)
echo "11 + 5 = $a"  # 16

let "a <<= 3"       # Equivalent to  let "a = a << 3"
echo "\"\$a\" (=16) left-shifted 3 places = $a"
                    # 128

let "a /= 4"        # Equivalent to  let "a = a / 4"
echo "128 / 4 = $a" # 32

let "a -= 5"        # Equivalent to  let "a = a - 5"
echo "32 - 5 = $a"  # 27

let "a *=  10"      # Equivalent to  let "a = a * 10"
echo "27 * 10 = $a" # 270

let "a %= 8"        # Equivalent to  let "a = a % 8"
echo "270 modulo 8 = $a  (270 / 8 = 33, remainder $a)"
                    # 6


# Does "let" permit C-style operators?
# Yes, just as the (( ... )) double-parentheses construct does.

let a++             # C-style (post) increment.
echo "6++ = $a"     # 6++ = 7
let a--             # C-style decrement.
```

```
echo "7-- = $a"     # 7-- = 6
# Of course, ++a, etc., also allowed . . .
echo


# Trinary operator.

# Note that $a is 6, see above.
let "t = a<7?7:11"   # True
echo $t  # 7

let a++
let "t = a<7?7:11"   # False
echo $t  #     11

exit
```

The *let* command can, in certain contexts, return a surprising exit status.

```
# Evgeniy Ivanov points out:

var=0
echo $?     # 0
            # As expected.

let var++
echo $?     # 1
            # The command was successful, so why isn't $?=0 ???
            # Anomaly!

let var++
echo $?     # 0
            # As expected.


# Likewise . . .

let var=0
echo $?     # 1
            # The command was successful, so why isn't $?=0 ???

#  However, as Jeff Gorak points out,
#+ this is part of the design spec for 'let' . . .
# "If the last ARG evaluates to 0, let returns 1;
#  let returns 0 otherwise." ['help let']
```

eval

`eval arg1 [arg2] ... [argN]`

Combines the arguments in an expression or list of expressions and `evaluates` them. Any variables within the expression are expanded. The net result is to **convert a string into a command**.

The **eval** command can be used for code generation from the command-line or within a script.

```
bash$ command_string="ps ax"
bash$ process="ps ax"
bash$ eval "$command_string" | grep "$process"
26973 pts/3    R+     0:00 grep --color ps ax
```

```
26974 pts/3    R+     0:00 ps ax
```

Each invocation of *eval* forces a re-*evaluation* of its arguments.

```
a='$b'
b='$c'
c=d

echo $a             # $b
                    # First level.
eval echo $a        # $c
                    # Second level.
eval eval echo $a   # d
                    # Third level.

# Thank you, E. Choroba.
```

Example 15-11. Showing the effect of *eval*

```
#!/bin/bash
# Exercising "eval" ...

y=`eval ls -l`  #  Similar to y=`ls -l`
echo $y         #+ but linefeeds removed because "echoed" variable is unquoted.
echo
echo "$y"       #  Linefeeds preserved when variable is quoted.

echo; echo

y=`eval df`     #  Similar to y=`df`
echo $y         #+ but linefeeds removed.

#  When LF's not preserved, it may make it easier to parse output,
#+ using utilities such as "awk".

echo
echo "==========================================================="
echo

eval "`seq 3 | sed -e 's/.*/echo var&=ABCDEFGHIJ/'`"
# var1=ABCDEFGHIJ
# var2=ABCDEFGHIJ
# var3=ABCDEFGHIJ

echo
echo "==========================================================="
echo


# Now, showing how to do something useful with "eval" . . .
# (Thank you, E. Choroba!)

version=3.4     #  Can we split the version into major and minor
                #+ part in one command?
echo "version = $version"
eval major=${version/./;minor=}     #  Replaces '.' in version by ';minor='
                                    #  The substitution yields '3; minor=4'
                                    #+ so eval does minor=4, major=3
```

```
echo Major: $major, minor: $minor   #  Major: 3, minor: 4
```

Example 15-12. Using *eval* to select among variables

```
#!/bin/bash
# arr-choice.sh

#  Passing arguments to a function to select
#+ one particular variable out of a group.

arr0=( 10 11 12 13 14 15 )
arr1=( 20 21 22 23 24 25 )
arr2=( 30 31 32 33 34 35 )
#       0  1  2  3  4  5      Element number (zero-indexed)


choose_array ()
{
  eval array_member=\${arr${array_number}[element_number]}
  #                 ^       ^^^^^^^^^^^^
  #  Using eval to construct the name of a variable,
  #+ in this particular case, an array name.

  echo "Element $element_number of array $array_number is $array_member"
} #  Function can be rewritten to take parameters.

array_number=0    # First array.
element_number=3
choose_array      # 13

array_number=2    # Third array.
element_number=4
choose_array      # 34

array_number=3    # Null array (arr3 not allocated).
element_number=4
choose_array      # (null)

# Thank you, Antonio Macchi, for pointing this out.
```

Example 15-13. *Echoing* the *command-line parameters*

```
#!/bin/bash
# echo-params.sh

# Call this script with a few command-line parameters.
# For example:
#     sh echo-params.sh first second third fourth fifth

params=$#              # Number of command-line parameters.
param=1                # Start at first command-line param.

while [ "$param" -le "$params" ]
do
  echo -n "Command-line parameter "
  echo -n \$$param     #  Gives only the *name* of variable.
#          ^^^         #  $1, $2, $3, etc.
                       #  Why?
                       #  \$ escapes the first "$"
```

```
                        #+ so it echoes literally,
                        #+ and $param dereferences "$param" . . .
                        #+ . . . as expected.
  echo -n " = "
  eval echo \$$param    #  Gives the *value* of variable.
# ^^^^      ^^^         #  The "eval" forces the *evaluation*
                        #+ of \$$
                        #+ as an indirect variable reference.

(( param ++ ))          # On to the next.
done

exit $?

# =================================================

$ sh echo-params.sh first second third fourth fifth
Command-line parameter $1 = first
Command-line parameter $2 = second
Command-line parameter $3 = third
Command-line parameter $4 = fourth
Command-line parameter $5 = fifth
```

Example 15-14. Forcing a log-off

```
#!/bin/bash
# Killing ppp to force a log-off.
# For dialup connection, of course.

# Script should be run as root user.

SERPORT=ttyS3
#  Depending on the hardware and even the kernel version,
#+ the modem port on your machine may be different --
#+ /dev/ttyS1 or /dev/ttyS2.


killppp="eval kill -9 `ps ax | awk '/ppp/ { print $1 }'`"
#                      -------- process ID of ppp -------

$killppp                     # This variable is now a command.


# The following operations must be done as root user.

chmod 666 /dev/$SERPORT      # Restore r+w permissions, or else what?
#  Since doing a SIGKILL on ppp changed the permissions on the serial port,
#+ we restore permissions to previous state.

rm /var/lock/LCK..$SERPORT   # Remove the serial port lock file. Why?

exit $?

# Exercises:
# ---------
# 1) Have script check whether root user is invoking it.
# 2) Do a check on whether the process to be killed
#+   is actually running before attempting to kill it.
# 3) Write an alternate version of this script based on 'fuser':
#+      if [ fuser -s /dev/modem ]; then . . .
```

Example 15-15. A version of *rot13*

```
#!/bin/bash
# A version of "rot13" using 'eval'.
# Compare to "rot13.sh" example.

setvar_rot_13()              # "rot13" scrambling
{
  local varname=$1 varvalue=$2
  eval $varname='$(echo "$varvalue" | tr a-z n-za-m)'
}


setvar_rot_13 var "foobar"   # Run "foobar" through rot13.
echo $var                    # sbbone

setvar_rot_13 var "$var"     # Run "sbbone" through rot13.
                             # Back to original variable.
echo $var                    # foobar

# This example by Stephane Chazelas.
# Modified by document author.

exit 0
```

Here is another example of using *eval* to *evaluate* a complex expression, this one from an earlier version of YongYe's Tetris game script.

```
eval ${1}+=\"${x} ${y} \"
```

Example A-53 uses *eval* to convert array elements into a command list.

The *eval* command occurs in the older version of indirect referencing.

```
eval var=\$$var
```

The *eval* command can be used to parameterize *brace expansion*.

The **eval** command can be risky, and normally should be avoided when there exists a reasonable alternative. An **`eval $COMMANDS`** executes the contents of *`COMMANDS`*, which may contain such unpleasant surprises as **rm -rf ***. Running an **eval** on unfamiliar code written by persons unknown is living dangerously.

set

The **set** command changes the value of internal script variables/options. One use for this is to toggle option flags which help determine the behavior of the script. Another application for it is to reset the positional parameters that a script sees as the result of a command (**`set `command``**). The script can then parse the fields of the command output.

Example 15-16. Using *set* with positional parameters

```
#!/bin/bash
# ex34.sh
# Script "set-test"

# Invoke this script with three command-line parameters,
# for example, "sh ex34.sh one two three".
```

```
echo
echo "Positional parameters before  set \`uname -a\` :"
echo "Command-line argument #1 = $1"
echo "Command-line argument #2 = $2"
echo "Command-line argument #3 = $3"


set `uname -a` # Sets the positional parameters to the output
               # of the command `uname -a`

echo
echo +++++
echo $_        # +++++
# Flags set in script.
echo $-        # hB
#                Anomalous behavior?
echo

echo "Positional parameters after  set \`uname -a\` :"
# $1, $2, $3, etc. reinitialized to result of `uname -a`
echo "Field #1 of 'uname -a' = $1"
echo "Field #2 of 'uname -a' = $2"
echo "Field #3 of 'uname -a' = $3"
echo \#\#\#
echo $_        # ###
echo

exit 0
```

More fun with positional parameters.

Example 15-17. Reversing the positional parameters

```
#!/bin/bash
# revposparams.sh: Reverse positional parameters.
# Script by Dan Jacobson, with stylistic revisions by document author.


set a\ b c d\ e;
#     ^      ^     Spaces escaped
#       ^ ^        Spaces not escaped
OIFS=$IFS; IFS=:;
#               ^   Saving old IFS and setting new one.

echo

until [ $# -eq 0 ]
do          #      Step through positional parameters.
  echo "### k0 = "$k""     # Before
  k=$1:$k;  #      Append each pos param to loop variable.
#     ^
  echo "### k = "$k""      # After
  echo
  shift;
done

set $k  #  Set new positional parameters.
echo -
echo $# #  Count of positional parameters.
echo -
```

```
echo

for i   #  Omitting the "in list" sets the variable -- i --
        #+ to the positional parameters.
do
  echo $i  # Display new positional parameters.
done

IFS=$OIFS  # Restore IFS.

#  Question:
#  Is it necessary to set an new IFS, internal field separator,
#+ in order for this script to work properly?
#  What happens if you don't? Try it.
#  And, why use the new IFS -- a colon -- in line 17,
#+ to append to the loop variable?
#  What is the purpose of this?

exit 0

$ ./revposparams.sh

### k0 =
### k = a b

### k0 = a b
### k = c a b

### k0 = c a b
### k = d e c a b

-
3
-

d e
c
a b
```

Invoking **set** without any options or arguments simply lists all the environmental and other variables that have been initialized.

```
bash$ set
AUTHORCOPY=/home/bozo/posts
 BASH=/bin/bash
 BASH_VERSION=$'2.05.8(1)-release'
 ...
 XAUTHORITY=/home/bozo/.Xauthority
 _=/etc/bashrc
 variable22=abc
 variable23=xzy
```

Using **set** with the -- option explicitly assigns the contents of a variable to the positional parameters. If no variable follows the -- it *unsets* the positional parameters.

Example 15-18. Reassigning the positional parameters

```
#!/bin/bash
```

```
variable="one two three four five"

set -- $variable
# Sets positional parameters to the contents of "$variable".

first_param=$1
second_param=$2
shift; shift        # Shift past first two positional params.
# shift 2             also works.
remaining_params="$*"

echo
echo "first parameter = $first_param"             # one
echo "second parameter = $second_param"           # two
echo "remaining parameters = $remaining_params"   # three four five

echo; echo

# Again.
set -- $variable
first_param=$1
second_param=$2
echo "first parameter = $first_param"             # one
echo "second parameter = $second_param"           # two

# ======================================================

set --
# Unsets positional parameters if no variable specified.

first_param=$1
second_param=$2
echo "first parameter = $first_param"             # (null value)
echo "second parameter = $second_param"           # (null value)

exit 0
```

See also Example 11-2 and Example 16-56.

unset

The **unset** command deletes a shell variable, effectively setting it to *null*. Note that this command does not affect positional parameters.

```
bash$ unset PATH

bash$ echo $PATH

bash$
```

Example 15-19. "Unsetting" a variable

```
#!/bin/bash
# unset.sh: Unsetting a variable.

variable=hello                       #  Initialized.
echo "variable = $variable"

unset variable                       #  Unset.
                                     #  In this particular context,
                                     #+ same effect as:   variable=
echo "(unset) variable = $variable"  #  $variable is null.
```

```
if [ -z "$variable" ]                #  Try a string-length test.
then
  echo "\$variable has zero length."
fi

exit 0
```

In most contexts, an *undeclared* variable and one that has been *unset* are equivalent. However, the ${parameter:-default} parameter substitution construct can distinguish between the two.

export

The **export** [61] command makes available variables to all child processes of the running script or shell. One important use of the **export** command is in startup files, to initialize and make accessible environmental variables to subsequent user processes.

Unfortunately, there is no way to export variables back to the parent process, to the process that called or invoked the script or shell.

Example 15-20. Using *export* to pass a variable to an embedded *awk* script

```
#!/bin/bash

#  Yet another version of the "column totaler" script (col-totaler.sh)
#+ that adds up a specified column (of numbers) in the target file.
#  This uses the environment to pass a script variable to 'awk' . . .
#+ and places the awk script in a variable.


ARGS=2
E_WRONGARGS=85

if [ $# -ne "$ARGS" ] # Check for proper number of command-line args.
then
   echo "Usage: `basename $0` filename column-number"
   exit $E_WRONGARGS
fi

filename=$1
column_number=$2

#===== Same as original script, up to this point =====#

export column_number
# Export column number to environment, so it's available for retrieval.


# -----------------------------------------------
awkscript='{ total += $ENVIRON["column_number"] }
END { print total }'
# Yes, a variable can hold an awk script.
# -----------------------------------------------

# Now, run the awk script.
awk "$awkscript" "$filename"

# Thanks, Stephane Chazelas.
```

```
exit 0
```

It is possible to initialize and export variables in the same operation, as in **export var1=xxx**.

However, as Greg Keraunen points out, in certain situations this may have a different effect than setting a variable, then exporting it.

```
bash$ export var=(a b); echo ${var[0]}
(a b)



bash$ var=(a b); export var; echo ${var[0]}
a
```

A variable to be exported may require special treatment. See Example M-2.

declare, **typeset**
: The declare and typeset commands specify and/or restrict properties of variables.

readonly
: Same as declare -r, sets a variable as read-only, or, in effect, as a constant. Attempts to change the variable fail with an error message. This is the shell analog of the *C* language **const** type qualifier.

getopts
: This powerful tool parses command-line arguments passed to the script. This is the Bash analog of the getopt external command and the *getopt* library function familiar to *C* programmers. It permits passing and concatenating multiple options [62] and associated arguments to a script (for example `scriptname -abc -e /usr/local`).

The **getopts** construct uses two implicit variables. `$OPTIND` is the argument pointer (*OPTion INDex*) and `$OPTARG` (*OPTion ARGument*) the (optional) argument attached to an option. A colon following the option name in the declaration tags that option as having an associated argument.

A **getopts** construct usually comes packaged in a while loop, which processes the options and arguments one at a time, then increments the implicit `$OPTIND` variable to point to the next.

1. The arguments passed from the command-line to the script must be preceded by a dash (-). It is the prefixed - that lets **getopts** recognize command-line arguments as *options*. In fact, **getopts** will not process arguments without the prefixed -, and will terminate option processing at the first argument encountered lacking them.
2. The **getopts** template differs slightly from the standard while loop, in that it lacks condition brackets.
3. The **getopts** construct is a highly functional replacement for the traditional getopt external command.

```
while getopts ":abcde:fg" Option
# Initial declaration.
# a, b, c, d, e, f, and g are the options (flags) expected.
```

```
# The : after option 'e' shows it will have an argument passed with it.
do
  case $Option in
    a ) # Do something with variable 'a'.
    b ) # Do something with variable 'b'.
    ...
    e)  # Do something with 'e', and also with $OPTARG,
        # which is the associated argument passed with option 'e'.
    ...
    g ) # Do something with variable 'g'.
  esac
done
shift $(($OPTIND - 1))
# Move argument pointer to next.

# All this is not nearly as complicated as it looks <grin>.
```

Example 15-21. Using *getopts* to read the options/arguments passed to a script

```
#!/bin/bash
# ex33.sh: Exercising getopts and OPTIND
#          Script modified 10/09/03 at the suggestion of Bill Gradwohl.


# Here we observe how 'getopts' processes command-line arguments to script.
# The arguments are parsed as "options" (flags) and associated arguments.

# Try invoking this script with:
#   'scriptname -mn'
#   'scriptname -oq qOption' (qOption can be some arbitrary string.)
#   'scriptname -qXXX -r'
#
#   'scriptname -qr'
#+      - Unexpected result, takes "r" as the argument to option "q"
#   'scriptname -q -r'
#+      - Unexpected result, same as above
#   'scriptname -mnop -mnop'  - Unexpected result
#   (OPTIND is unreliable at stating where an option came from.)
#
#  If an option expects an argument ("flag:"), then it will grab
#+ whatever is next on the command-line.

NO_ARGS=0
E_OPTERROR=85

if [ $# -eq "$NO_ARGS" ]    # Script invoked with no command-line args?
then
  echo "Usage: `basename $0` options (-mnopqrs)"
  exit $E_OPTERROR          # Exit and explain usage.
                            # Usage: scriptname -options
                            # Note: dash (-) necessary
fi


while getopts ":mnopq:rs" Option
do
  case $Option in
    m     ) echo "Scenario #1: option -m-   [OPTIND=${OPTIND}]";;
    n | o ) echo "Scenario #2: option -$Option-   [OPTIND=${OPTIND}]";;
    p     ) echo "Scenario #3: option -p-   [OPTIND=${OPTIND}]";;
    q     ) echo "Scenario #4: option -q-\
```

```
                  with argument \"$OPTARG\"   [OPTIND=${OPTIND}]";;
    #  Note that option 'q' must have an associated argument,
    #+ otherwise it falls through to the default.
    r | s ) echo "Scenario #5: option -$Option-";;
    *     ) echo "Unimplemented option chosen.";;   # Default.
  esac
done

shift $(($OPTIND - 1))
#  Decrements the argument pointer so it points to next argument.
#  $1 now references the first non-option item supplied on the command-line
#+ if one exists.

exit $?

#   As Bill Gradwohl states,
#  "The getopts mechanism allows one to specify:  scriptname -mnop -mnop
#+  but there is no reliable way to differentiate what came
#+ from where by using OPTIND."
#  There are, however, workarounds.
```

Script Behavior

source, . (dot command)
: This command, when invoked from the command-line, executes a script. Within a script, a **`source file-name`** loads the file `file-name`. *Sourcing* a file (dot-command) *imports* code into the script, appending to the script (same effect as the **`#include`** directive in a *C* program). The net result is the same as if the "sourced" lines of code were physically present in the body of the script. This is useful in situations when multiple scripts use a common data file or function library.

Example 15-22. "Including" a data file

```
#!/bin/bash
#  Note that this example must be invoked with bash, i.e., bash ex38.sh
#+ not  sh ex38.sh !

. data-file    # Load a data file.
# Same effect as "source data-file", but more portable.

#  The file "data-file" must be present in current working directory,
#+ since it is referred to by its basename.

# Now, let's reference some data from that file.

echo "variable1 (from data-file) = $variable1"
echo "variable3 (from data-file) = $variable3"

let "sum = $variable2 + $variable4"
echo "Sum of variable2 + variable4 (from data-file) = $sum"
echo "message1 (from data-file) is \"$message1\""
#                                  Escaped quotes
echo "message2 (from data-file) is \"$message2\""

print_message This is the message-print function in the data-file.


exit $?
```

File `data-file` for Example 15-22, above. Must be present in same directory.

```
# This is a data file loaded by a script.
# Files of this type may contain variables, functions, etc.
# It loads with a 'source' or '.' command from a shell script.

# Let's initialize some variables.

variable1=23
variable2=474
variable3=5
variable4=97

message1="Greetings from *** line $LINENO *** of the data file!"
message2="Enough for now. Goodbye."

print_message ()
{   # Echoes any message passed to it.

  if [ -z "$1" ]
  then
    return 1 # Error, if argument missing.
  fi

  echo

  until [ -z "$1" ]
  do             # Step through arguments passed to function.
    echo -n "$1" # Echo args one at a time, suppressing line feeds.
    echo -n " "  # Insert spaces between words.
    shift        # Next one.
  done

  echo

  return 0
}
```

If the *sourced* file is itself an executable script, then it will run, then return control to the script that called it. A *sourced* executable script may use a return for this purpose.

Arguments may be (optionally) passed to the *sourced* file as positional parameters.

```
source $filename $arg1 arg2
```

It is even possible for a script to *source* itself, though this does not seem to have any practical applications.

Example 15-23. A (useless) script that sources itself

```
#!/bin/bash
# self-source.sh: a script sourcing itself "recursively."
# From "Stupid Script Tricks," Volume II.

MAXPASSCNT=100    # Maximum number of execution passes.

echo -n  "$pass_count  "
#  At first execution pass, this just echoes two blank spaces,
```

```
#+ since $pass_count still uninitialized.

let "pass_count += 1"
#  Assumes the uninitialized variable $pass_count
#+ can be incremented the first time around.
#  This works with Bash and pdksh, but
#+ it relies on non-portable (and possibly dangerous) behavior.
#  Better would be to initialize $pass_count to 0 before incrementing.

while [ "$pass_count" -le $MAXPASSCNT ]
do
  . $0   # Script "sources" itself, rather than calling itself.
         # ./$0 (which would be true recursion) doesn't work here. Why?
done

#  What occurs here is not actually recursion,
#+ since the script effectively "expands" itself, i.e.,
#+ generates a new section of code
#+ with each pass through the 'while' loop',
#  with each 'source' in line 20.
#
#  Of course, the script interprets each newly 'sourced' "#!" line
#+ as a comment, and not as the start of a new script.

echo

exit 0   # The net effect is counting from 1 to 100.
         # Very impressive.

# Exercise:
# --------
# Write a script that uses this trick to actually do something useful.
```

exit

Unconditionally terminates a script. [63] The **exit** command may optionally take an integer argument, which is returned to the shell as the exit status of the script. It is good practice to end all but the simplest scripts with an `exit 0`, indicating a successful run.

If a script terminates with an **exit** lacking an argument, the exit status of the script is the exit status of the last command executed in the script, not counting the **exit**. This is equivalent to an **exit $?**.

An **exit** command may also be used to terminate a subshell.

exec

This shell builtin replaces the current process with a specified command. Normally, when the shell encounters a command, it forks off a child process to actually execute the command. Using the **exec** builtin, the shell does not fork, and the command *exec*'ed replaces the shell. When used in a script, therefore, it forces an exit from the script when the **exec**'ed command terminates. [64]

Example 15-24. Effects of *exec*

```
#!/bin/bash

exec echo "Exiting \"$0\" at line $LINENO."   # Exit from script here.
# $LINENO is an internal Bash variable set to the line number it's on.

# ----------------------------------
```

```
# The following lines never execute.

echo "This echo fails to echo."

exit 99                        #  This script will not exit here.
                               #  Check exit value after script terminates
                               #+ with an 'echo $?'.
                               #  It will *not* be 99.
```

Example 15-25. A script that *exec's* itself

```
#!/bin/bash
# self-exec.sh

# Note: Set permissions on this script to 555 or 755,
#       then call it with ./self-exec.sh or sh ./self-exec.sh.

echo

echo "This line appears ONCE in the script, yet it keeps echoing."
echo "The PID of this instance of the script is still $$."
#     Demonstrates that a subshell is not forked off.

echo "==================== Hit Ctl-C to exit ===================="

sleep 1

exec $0   #  Spawns another instance of this same script
          #+ that replaces the previous one.

echo "This line will never echo!"  # Why not?

exit 99                            # Will not exit here!
                                   # Exit code will not be 99!
```

An **exec** also serves to reassign file descriptors. For example, **`exec <zzz-file`** replaces `stdin` with the file `zzz-file`.

The `-exec` option to find is *`not`* the same as the **exec** shell builtin.

shopt

This command permits changing *shell options* on the fly (see Example 25-1 and Example 25-2). It often appears in the Bash startup files, but also has its uses in scripts. Needs version 2 or later of Bash.

```
shopt -s cdspell
# Allows minor misspelling of directory names with 'cd'
# Option -s sets, -u unsets.

cd /hpme  # Oops! Mistyped '/home'.
pwd       # /home
          # The shell corrected the misspelling.
```

caller

Putting a **caller** command inside a function echoes to `stdout` information about the *caller* of that function.

```
#!/bin/bash

function1 ()
{
```

```
  # Inside function1 ().
  caller 0   # Tell me about it.
}

function1    # Line 9 of script.

# 9 main test.sh
# ^                 Line number that the function was called from.
#   ^^^^            Invoked from "main" part of script.
#        ^^^^^^^    Name of calling script.

caller 0     # Has no effect because it's not inside a function.
```

A **caller** command can also return *caller* information from a script sourced within another script. Analogous to a function, this is a "subroutine call."

You may find this command useful in debugging.

Commands

true

A command that returns a successful (zero) exit status, but does nothing else.

```
bash$ true
bash$ echo $?
0
```

```
# Endless loop
while true   # alias for ":"
do
   operation-1
   operation-2
   ...
   operation-n
   # Need a way to break out of loop or script will hang.
done
```

false

A command that returns an unsuccessful exit status, but does nothing else.

```
bash$ false
bash$ echo $?
1
```

```
# Testing "false"
if false
then
  echo "false evaluates \"true\""
else
  echo "false evaluates \"false\""
fi
# false evaluates "false"


# Looping while "false" (null loop)
while false
do
   # The following code will not execute.
```

```
   operation-1
   operation-2
   ...
   operation-n
   # Nothing happens!
done
```

type [cmd]

Similar to the which external command, **type cmd** identifies "cmd." Unlike **which**, **type** is a Bash builtin. The useful `-a` option to **type** identifies *keywords* and *builtins*, and also locates system commands with identical names.

```
bash$ type '['
[ is a shell builtin
bash$ type -a '['
[ is a shell builtin
 [ is /usr/bin/[


bash$ type type
type is a shell builtin
```

The **type** command can be useful for testing whether a certain command exists.

hash [cmds]

Records the *path* name of specified commands -- in the shell *hash table* [65] -- so the shell or script will not need to search the $PATH on subsequent calls to those commands. When **hash** is called with no arguments, it simply lists the commands that have been hashed. The `-r` option resets the hash table.

bind

The **bind** builtin displays or modifies *readline* [66] key bindings.

help

Gets a short usage summary of a shell builtin. This is the counterpart to whatis, but for builtins. The display of *help* information got a much-needed update in the version 4 release of Bash.

```
bash$ help exit
exit: exit [n]
    Exit the shell with a status of N.  If N is omitted, the exit status
    is that of the last command executed.
```

15.1. Job Control Commands

Certain of the following job control commands take a *job identifier* as an argument. See the table at end of the chapter.

jobs

Lists the jobs running in the background, giving the *job number*. Not as useful as ps.

It is all too easy to confuse *jobs* and *processes*. Certain builtins, such as **kill**, **disown**, and **wait** accept either a job number or a process number as an argument. The fg, bg and **jobs** commands accept only a job number.

```
bash$ sleep 100 &
[1] 1384
```

```
bash $ jobs
[1]+  Running                 sleep 100 &
```

"1" is the job number (jobs are maintained by the current shell). "1384" is the PID or *process ID number* (processes are maintained by the system). To kill this job/process, either a **kill %1** or a **kill 1384** works.

Thanks, S.C.

disown
: Remove job(s) from the shell's table of active jobs.

fg, **bg**
: The **fg** command switches a job running in the background into the foreground. The **bg** command restarts a suspended job, and runs it in the background. If no job number is specified, then the **fg** or **bg** command acts upon the currently running job.

wait
: Suspend script execution until all jobs running in background have terminated, or until the job number or process ID specified as an option terminates. Returns the exit status of waited-for command.

You may use the **wait** command to prevent a script from exiting before a background job finishes executing (this would create a dreaded orphan process).

Example 15-26. Waiting for a process to finish before proceeding

```
#!/bin/bash

ROOT_UID=0   # Only users with $UID 0 have root privileges.
E_NOTROOT=65
E_NOPARAMS=66

if [ "$UID" -ne "$ROOT_UID" ]
then
  echo "Must be root to run this script."
  # "Run along kid, it's past your bedtime."
  exit $E_NOTROOT
fi

if [ -z "$1" ]
then
  echo "Usage: `basename $0` find-string"
  exit $E_NOPARAMS
fi


echo "Updating 'locate' database..."
echo "This may take a while."
updatedb /usr &     # Must be run as root.

wait
# Don't run the rest of the script until 'updatedb' finished.
# You want the the database updated before looking up the file name.

locate $1

#  Without the 'wait' command, in the worse case scenario,
#+ the script would exit while 'updatedb' was still running,
#+ leaving it as an orphan process.
```

```
exit 0
```

Optionally, **wait** can take a *job identifier* as an argument, for example, *wait%1* or *wait $PPID*. [67] See the job id table.

Within a script, running a command in the background with an ampersand (&) may cause the script to har
ENTER is hit. This seems to occur with commands that write to `stdout`. It can be a major annoyance.

```
#!/bin/bash
# test.sh

ls -l &
echo "Done."
```

```
bash$ ./test.sh
Done.
 [bozo@localhost test-scripts]$ total 1
 -rwxr-xr-x    1 bozo     bozo           34 Oct 11 15:09 test.sh
 _
```

```
As Walter Brameld IV explains it:

As far as I can tell, such scripts don't actually hang. It ju
seems that they do because the background command writes text
the console after the prompt. The user gets the impression th
the prompt was never displayed. Here's the sequence of events

1. Script launches background command.
2. Script exits.
3. Shell displays the prompt.
4. Background command continues running and writing text to t
   console.
5. Background command finishes.
6. User doesn't see a prompt at the bottom of the output, thi
   is hanging.
```

Placing a **wait** after the background command seems to remedy this.

```
#!/bin/bash
# test.sh

ls -l &
echo "Done."
wait
```

```
bash$ ./test.sh
Done.
 [bozo@localhost test-scripts]$ total 1
 -rwxr-xr-x    1 bozo     bozo           34 Oct 11 15:09 test.sh
```

Redirecting the output of the command to a file or even to `/dev/null` also takes care of this problem.

suspend

This has a similar effect to **Control-Z**, but it suspends the shell (the shell's parent process should resume it at an appropriate time).

logout

Exit a login shell, optionally specifying an exit status.

times

Gives statistics on the system time elapsed when executing commands, in the following form:

```
0m0.020s 0m0.020s
```

This capability is of relatively limited value, since it is not common to profile and benchmark shell scripts.

kill

Forcibly terminate a process by sending it an appropriate *terminate* signal (see Example 17-6).

Example 15-27. A script that kills itself

```
#!/bin/bash
# self-destruct.sh

kill $$  # Script kills its own process here.
         # Recall that "$$" is the script's PID.

echo "This line will not echo."
# Instead, the shell sends a "Terminated" message to stdout.

exit 0   # Normal exit? No!

#  After this script terminates prematurely,
#+ what exit status does it return?
#
# sh self-destruct.sh
# echo $?
# 143
#
# 143 = 128 + 15
#              TERM signal
```

`kill -l` lists all the signals (as does the file `/usr/include/asm/signal.h`). A **`kill -9`** is a *sure kill*, which will usually terminate a process that stubbornly refuses to die with a plain **kill**. Sometimes, a **`kill -15`** works. A *zombie* process, that is, a child process that has terminated, but that the parent process has not (yet) killed, cannot be killed by a logged-on user -- you can't kill something that is already dead -- but **init** will generally clean it up sooner or later.

killall

The **killall** command kills a running process by *name*, rather than by process ID. If there are multiple instances of a particular command running, then doing a *killall* on that command will terminate them *all*.

This refers to the **killall** command in `/usr/bin`, *not* the killall script in `/etc/rc.d/init.d`.

command

The **command** directive disables aliases and functions for the command immediately following it.

```
bash$ command ls
```

This is one of three shell directives that effect script command processing. The others are builtin and enable.

builtin

Invoking **builtin BUILTIN_COMMAND** runs the command `BUILTIN_COMMAND` as a shell builtin, temporarily disabling both functions and external system commands with the same name.

enable

This either enables or disables a shell builtin command. As an example, `enable -n kill` disables the shell builtin kill, so that when Bash subsequently encounters *kill*, it invokes the external command `/bin/kill`.

The `-a` option to *enable* lists all the shell builtins, indicating whether or not they are enabled. The `-f filename` option lets *enable* load a builtin as a shared library (DLL) module from a properly compiled object file. [68].

autoload

This is a port to Bash of the *ksh* autoloader. With **autoload** in place, a function with an *autoload* declaration will load from an external file at its first invocation. [69] This saves system resources.

Note that *autoload* is not a part of the core Bash installation. It needs to be loaded in with `enable -f` (see above).

Table 15-1. Job identifiers

Notation	Meaning
`%N`	Job number [N]
`%S`	Invocation (command-line) of job begins with string *S*
`%?S`	Invocation (command-line) of job contains within it string *S*
`%%`	"current" job (last job stopped in foreground or started in background)
`%+`	"current" job (last job stopped in foreground or started in background)
`%-`	Last job
`$!`	Last background process

Chapter 16. External Filters, Programs and Commands

Standard UNIX commands make shell scripts more versatile. The power of scripts comes from coupling system commands and shell directives with simple programming constructs.

16.1. Basic Commands

The first commands a novice learns

ls

The basic file "list" command. It is all too easy to underestimate the power of this humble command. For example, using the `-R`, recursive option, **ls** provides a tree-like listing of a directory structure. Other useful options are `-S`, sort listing by file size, `-t`, sort by file modification time, `-v`, sort by (numerical) version numbers embedded in the filenames, [70] `-b`, show escape characters, and `-i`, show file inodes (see Example 16-4).

```
bash$ ls -l
-rw-rw-r-- 1 bozo bozo 0 Sep 14 18:44 chapter10.txt
 -rw-rw-r-- 1 bozo bozo 0 Sep 14 18:44 chapter11.txt
 -rw-rw-r-- 1 bozo bozo 0 Sep 14 18:44 chapter12.txt
 -rw-rw-r-- 1 bozo bozo 0 Sep 14 18:44 chapter1.txt
 -rw-rw-r-- 1 bozo bozo 0 Sep 14 18:44 chapter2.txt
 -rw-rw-r-- 1 bozo bozo 0 Sep 14 18:44 chapter3.txt
 -rw-rw-r-- 1 bozo bozo 0 Sep 14 18:49 Chapter_headings.txt
 -rw-rw-r-- 1 bozo bozo 0 Sep 14 18:49 Preface.txt


bash$ ls -lv
 total 0
 -rw-rw-r-- 1 bozo bozo 0 Sep 14 18:49 Chapter_headings.txt
 -rw-rw-r-- 1 bozo bozo 0 Sep 14 18:49 Preface.txt
 -rw-rw-r-- 1 bozo bozo 0 Sep 14 18:44 chapter1.txt
 -rw-rw-r-- 1 bozo bozo 0 Sep 14 18:44 chapter2.txt
 -rw-rw-r-- 1 bozo bozo 0 Sep 14 18:44 chapter3.txt
 -rw-rw-r-- 1 bozo bozo 0 Sep 14 18:44 chapter10.txt
 -rw-rw-r-- 1 bozo bozo 0 Sep 14 18:44 chapter11.txt
 -rw-rw-r-- 1 bozo bozo 0 Sep 14 18:44 chapter12.txt
```

The *ls* command returns a non-zero exit status when attempting to list a non-existent file.

```
bash$ ls abc
ls: abc: No such file or directory


bash$ echo $?
2
```

Example 16-1. Using *ls* to create a table of contents for burning a CDR disk

```
#!/bin/bash
# ex40.sh (burn-cd.sh)
# Script to automate burning a CDR.


SPEED=10         # May use higher speed if your hardware supports it.
IMAGEFILE=cdimage.iso
CONTENTSFILE=contents
# DEVICE=/dev/cdrom     For older versions of cdrecord
DEVICE="1,0,0"
DEFAULTDIR=/opt  # This is the directory containing the data to be burned.
                 # Make sure it exists.
                 # Exercise: Add a test for this.

# Uses Joerg Schilling's "cdrecord" package:
# http://www.fokus.fhg.de/usr/schilling/cdrecord.html

#  If this script invoked as an ordinary user, may need to suid cdrecord
#+ chmod u+s /usr/bin/cdrecord, as root.
#  Of course, this creates a security hole, though a relatively minor one.

if [ -z "$1" ]
then
  IMAGE_DIRECTORY=$DEFAULTDIR
  # Default directory, if not specified on command-line.
else
    IMAGE_DIRECTORY=$1
fi

# Create a "table of contents" file.
ls -lRF $IMAGE_DIRECTORY > $IMAGE_DIRECTORY/$CONTENTSFILE
# The "l" option gives a "long" file listing.
# The "R" option makes the listing recursive.
# The "F" option marks the file types (directories get a trailing /).
echo "Creating table of contents."

# Create an image file preparatory to burning it onto the CDR.
mkisofs -r -o $IMAGEFILE $IMAGE_DIRECTORY
echo "Creating ISO9660 file system image ($IMAGEFILE)."

# Burn the CDR.
echo "Burning the disk."
echo "Please be patient, this will take a while."
wodim -v -isosize dev=$DEVICE $IMAGEFILE
#  In newer Linux distros, the "wodim" utility assumes the
#+ functionality of "cdrecord."
exitcode=$?
echo "Exit code = $exitcode"

exit $exitcode
```

cat, **tac**

cat, an acronym for *concatenate*, lists a file to `stdout`. When combined with redirection (> or >>), it is commonly used to concatenate files.

```
# Uses of 'cat'
cat filename                          # Lists the file.

cat file.1 file.2 file.3 > file.123   # Combines three files into one.
```

The `-n` option to **cat** inserts consecutive numbers before all lines of the target file(s). The `-b` option numbers only the non-blank lines. The `-v` option echoes nonprintable characters, using ^ notation.

The `-s` option squeezes multiple consecutive blank lines into a single blank line.

See also Example 16-28 and Example 16-24.

In a pipe, it may be more efficient to redirect the `stdin` to a file, rather than to **cat** the file.

```
cat filename | tr a-z A-Z

tr a-z A-Z < filename   #  Same effect, but starts one less process,
                        #+ and also dispenses with the pipe.
```

tac, is the inverse of *cat*, listing a file backwards from its end.

rev

reverses each line of a file, and outputs to `stdout`. This does not have the same effect as **tac**, as it preserves the order of the lines, but flips each one around (mirror image).

```
bash$ cat file1.txt
This is line 1.
 This is line 2.


bash$ tac file1.txt
This is line 2.
 This is line 1.


bash$ rev file1.txt
.1 enil si sihT
 .2 enil si sihT
```

cp

This is the file copy command. **`cp file1 file2`** copies `file1` to `file2`, overwriting `file2` if it already exists (see Example 16-6).

Particularly useful are the `-a` archive flag (for copying an entire directory tree), the `-u` update flag (which prevents overwriting identically-named newer files), and the `-r` and `-R` recursive flags.

```
cp -u source_dir/* dest_dir
#  "Synchronize" dest_dir to source_dir
#+  by copying over all newer and not previously existing files.
```

mv

This is the file *move* command. It is equivalent to a combination of **cp** and **rm**. It may be used to move multiple files to a directory, or even to rename a directory. For some examples of using **mv** in a script, see Example 10-11 and Example A-2.

When used in a non-interactive script, **mv** takes the `-f` (*force*) option to bypass user input.

When a directory is moved to a preexisting directory, it becomes a subdirectory of the destination directory.

```
bash$ mv source_directory target_directory
```

```
bash$ ls -lF target_directory
total 1
 drwxrwxr-x    2 bozo  bozo      1024 May 28 19:20 source_directory/
```

rm

Delete (remove) a file or files. The `-f` option forces removal of even readonly files, and is useful for bypassing user input in a script.

The *rm* command will, by itself, fail to remove filenames beginning with a dash. Why? Because *rm* sees a dash-prefixed filename as an *option*.

```
bash$ rm -badname
rm: invalid option -- b
 Try `rm --help' for more information.
```

One clever workaround is to precede the filename with a " -- " (the *end-of-options* flag).

```
bash$ rm -- -badname
```

Another method to is to preface the filename to be removed with a `dot-slash` .

```
bash$ rm ./-badname
```

When used with the recursive flag `-r`, this command removes files all the way down the directory tree from the current directory. A careless **rm -rf *** can wipe out a big chunk of a directory structure.

rmdir

Remove directory. The directory must be empty of all files -- including "invisible" *dotfiles* [71] -- for this command to succeed.

mkdir

Make directory, creates a new directory. For example, **`mkdir -p project/programs/December`** creates the named directory. The *-p* option automatically creates any necessary parent directories.

chmod

Changes the attributes of an existing file or directory (see Example 15-14).

```
chmod +x filename
# Makes "filename" executable for all users.

chmod u+s filename
# Sets "suid" bit on "filename" permissions.
# An ordinary user may execute "filename" with same privileges as the file's owner.
# (This does not apply to shell scripts.)
```

```
chmod 644 filename
#  Makes "filename" readable/writable to owner, readable to others
#+ (octal mode).

chmod 444 filename
#  Makes "filename" read-only for all.
#  Modifying the file (for example, with a text editor)
#+ not allowed for a user who does not own the file (except for root),
#+ and even the file owner must force a file-save
#+ if she modifies the file.
```

```
#  Same restrictions apply for deleting the file.
```

```
chmod 1777 directory-name
#  Gives everyone read, write, and execute permission in directory,
#+ however also sets the "sticky bit".
#  This means that only the owner of the directory,
#+ owner of the file, and, of course, root
#+ can delete any particular file in that directory.

chmod 111 directory-name
#  Gives everyone execute-only permission in a directory.
#  This means that you can execute and READ the files in that directory
#+ (execute permission necessarily includes read permission
#+ because you can't execute a file without being able to read it).
#  But you can't list the files or search for them with the "find" command.
#  These restrictions do not apply to root.

chmod 000 directory-name
#  No permissions at all for that directory.
#  Can't read, write, or execute files in it.
#  Can't even list files in it or "cd" to it.
#  But, you can rename (mv) the directory
#+ or delete it (rmdir) if it is empty.
#  You can even symlink to files in the directory,
#+ but you can't read, write, or execute the symlinks.
#  These restrictions do not apply to root.
```

chattr

Change file **attr**ibutes. This is analogous to **chmod** above, but with different options and a different invocation syntax, and it works only on *ext2/ext3* filesystems.

One particularly interesting **chattr** option is `i`. A **chattr +i `filename`** marks the file as immutable. The file cannot be modified, linked to, or deleted, *not even by root*. This file attribute can be set or removed only by *root*. In a similar fashion, the `a` option marks the file as append only.

```
root# chattr +i file1.txt


root# rm file1.txt

rm: remove write-protected regular file `file1.txt'? y
 rm: cannot remove `file1.txt': Operation not permitted
```

If a file has the `s` (secure) attribute set, then when it is deleted its block is overwritten with binary zeroes. [72]

If a file has the `u` (undelete) attribute set, then when it is deleted, its contents can still be retrieved (undeleted).

If a file has the `c` (compress) attribute set, then it will automatically be compressed on writes to disk, and uncompressed on reads.

The file attributes set with **chattr** do not show in a file listing (**ls -l**).

ln

Creates links to pre-existings files. A "link" is a reference to a file, an alternate name for it. The **ln** command permits referencing the linked file by more than one name and is a superior alternative to aliasing (see Example 4-6).

The **ln** creates only a reference, a pointer to the file only a few bytes in size.

The **ln** command is most often used with the `-s`, symbolic or "soft" link flag. Advantages of using the `-s` flag are that it permits linking across file systems or to directories.

The syntax of the command is a bit tricky. For example: **`ln -s oldfile newfile`** links the previously existing `oldfile` to the newly created link, `newfile`.

If a file named `newfile` has previously existed, an error message will result.

Which type of link to use?

As John Macdonald explains it:

Both of these [types of links] provide a certain measure of dual reference -- if you edit the contents of the file using any name, your changes will affect both the original name and either a hard or soft new name. The differences between them occurs when you work at a higher level. The advantage of a hard link is that the new name is totally independent of the old name -- if you remove or rename the old name, that does not affect the hard link, which continues to point to the data while it would leave a soft link hanging pointing to the old name which is no longer there. The advantage of a soft link is that it can refer to a different file system (since it is just a reference to a file name, not to actual data). And, unlike a hard link, a symbolic link can refer to a directory.

Links give the ability to invoke a script (or any other type of executable) with multiple names, and having that script behave according to how it was invoked.

Example 16-2. Hello or Good-bye

```
#!/bin/bash
# hello.sh: Saying "hello" or "goodbye"
#+          depending on how script is invoked.

# Make a link in current working directory ($PWD) to this script:
#    ln -s hello.sh goodbye
# Now, try invoking this script both ways:
# ./hello.sh
# ./goodbye


HELLO_CALL=65
GOODBYE_CALL=66

if [ $0 = "./goodbye" ]
then
  echo "Good-bye!"
  # Some other goodbye-type commands, as appropriate.
  exit $GOODBYE_CALL
fi

echo "Hello!"
# Some other hello-type commands, as appropriate.
exit $HELLO_CALL
```

man, **info**

These commands access the manual and information pages on system commands and installed utilities. When available, the *info* pages usually contain more detailed descriptions than do the *man* pages.

There have been various attempts at "automating" the writing of *man pages*. For a script that makes a tentative first step in that direction, see Example A-39.

16.2. Complex Commands

Commands for more advanced users

find

-exec *COMMAND* \;

Carries out *COMMAND* on each file that **find** matches. The command sequence terminates with ; (the ";" is escaped to make certain the shell passes it to **find** literally, without interpreting it as a special character).

```
bash$ find ~/ -name '*.txt'
/home/bozo/.kde/share/apps/karm/karmdata.txt
 /home/bozo/misc/irmeyc.txt
 /home/bozo/test-scripts/1.txt
```

If *COMMAND* contains {}, then **find** substitutes the full path name of the selected file for "{}".

```
find ~/ -name 'core*' -exec rm {} \;
# Removes all core dump files from user's home directory.
```

```
find /home/bozo/projects -mtime -1
#                               ^   Note minus sign!
#  Lists all files in /home/bozo/projects directory tree
#+ that were modified within the last day (current_day - 1).
#
find /home/bozo/projects -mtime 1
#  Same as above, but modified *exactly* one day ago.
#
#  mtime = last modification time of the target file
#  ctime = last status change time (via 'chmod' or otherwise)
#  atime = last access time

DIR=/home/bozo/junk_files
find "$DIR" -type f -atime +5 -exec rm {} \;
#                          ^           ^^
#  Curly brackets are placeholder for the path name output by "find."
#
#  Deletes all files in "/home/bozo/junk_files"
#+ that have not been accessed in *at least* 5 days (plus sign ... +5).
#
#  "-type filetype", where
#  f = regular file
#  d = directory
#  l = symbolic link, etc.
```

```
#
#  (The 'find' manpage and info page have complete option listings.)
```

```
find /etc -exec grep '[0-9][0-9]*[.][0-9][0-9]*[.][0-9][0-9]*[.][0-9][0-9]*' {} \;

# Finds all IP addresses (xxx.xxx.xxx.xxx) in /etc directory files.
# There a few extraneous hits. Can they be filtered out?

# Possibly by:

find /etc -type f -exec cat '{}' \; | tr -c '.[:digit:]' '\n' \
| grep '^[^.][^.]*\.[^.][^.]*\.[^.][^.]*\.[^.][^.]*$'
#
#  [:digit:] is one of the character classes
#+ introduced with the POSIX 1003.2 standard.

# Thanks, Stéphane Chazelas.
```

The `-exec` option to **find** should not be confused with the exec shell builtin.

Example 16-3. *Badname*, eliminate file names in current directory containing bad characters and whitespace.

```
#!/bin/bash
# badname.sh
# Delete filenames in current directory containing bad characters.

for filename in *
do
  badname=`echo "$filename" | sed -n /[\+\{\;\"\\\=\?~\(\)\<\>\&\*\|\$]/p`
# badname=`echo "$filename" | sed -n '/[+{;"\=?~()<>&*|$]/p'`  also works.
# Deletes files containing these nasties:     + { ; " \ = ? ~ ( ) < > & * | $
#
  rm $badname 2>/dev/null
#             ^^^^^^^^^^^ Error messages deep-sixed.
done

# Now, take care of files containing all manner of whitespace.
find . -name "* *" -exec rm -f {} \;
# The path name of the file that _find_ finds replaces the "{}".
# The '\' ensures that the ';' is interpreted literally, as end of command.

exit 0

#---------------------------------------------------------------------
# Commands below this line will not execute because of _exit_ command.

# An alternative to the above script:
find . -name '*[+{;"\\=?~()<>&*|$ ]*' -maxdepth 0 \
-exec rm -f '{}' \;
#  The "-maxdepth 0" option ensures that _find_ will not search
#+ subdirectories below $PWD.

# (Thanks, S.C.)
```

Example 16-4. Deleting a file by its *inode* number

```
#!/bin/bash
```

```
# idelete.sh: Deleting a file by its inode number.

#  This is useful when a filename starts with an illegal character,
#+ such as ? or -.

ARGCOUNT=1                      # Filename arg must be passed to script.
E_WRONGARGS=70
E_FILE_NOT_EXIST=71
E_CHANGED_MIND=72

if [ $# -ne "$ARGCOUNT" ]
then
  echo "Usage: `basename $0` filename"
  exit $E_WRONGARGS
fi

if [ ! -e "$1" ]
then
  echo "File \""$1"\" does not exist."
  exit $E_FILE_NOT_EXIST
fi

inum=`ls -i | grep "$1" | awk '{print $1}'`
# inum = inode (index node) number of file
# -----------------------------------------------------------------------
# Every file has an inode, a record that holds its physical address info.
# -----------------------------------------------------------------------

echo; echo -n "Are you absolutely sure you want to delete \"$1\" (y/n)? "
# The '-v' option to 'rm' also asks this.
read answer
case "$answer" in
[nN]) echo "Changed your mind, huh?"
      exit $E_CHANGED_MIND
      ;;
*)    echo "Deleting file \"$1\".";;
esac

find . -inum $inum -exec rm {} \;
#                           ^^
#        Curly brackets are placeholder
#+       for text output by "find."
echo "File "\"$1"\" deleted!"

exit 0
```

The **find** command also works without the `-exec` option.

```
#!/bin/bash
#  Find suid root files.
#  A strange suid file might indicate a security hole,
#+ or even a system intrusion.

directory="/usr/sbin"
# Might also try /sbin, /bin, /usr/bin, /usr/local/bin, etc.
permissions="+4000"  # suid root (dangerous!)


for file in $( find "$directory" -perm "$permissions" )
do
  ls -ltF --author "$file"
done
```

See Example 16-30, Example 3-4, and Example 11-10 for scripts using **find**. Its manpage provides more detail on this complex and powerful command.

xargs

A filter for feeding arguments to a command, and also a tool for assembling the commands themselves. It breaks a data stream into small enough chunks for filters and commands to process. Consider it as a powerful replacement for backquotes. In situations where command substitution fails with a too many arguments error, substituting **xargs** often works. [73] Normally, **xargs** reads from `stdin` or from a pipe, but it can also be given the output of a file.

The default command for **xargs** is echo. This means that input piped to **xargs** may have linefeeds and other whitespace characters stripped out.

```
bash$ ls -l
total 0
 -rw-rw-r--    1 bozo  bozo         0 Jan 29 23:58 file1
 -rw-rw-r--    1 bozo  bozo         0 Jan 29 23:58 file2



bash$ ls -l | xargs
total 0 -rw-rw-r-- 1 bozo bozo 0 Jan 29 23:58 file1 -rw-rw-r-- 1 bozo bozo 0 Jan...



bash$ find ~/mail -type f | xargs grep "Linux"
./misc:User-Agent: slrn/0.9.8.1 (Linux)
 ./sent-mail-jul-2005: hosted by the Linux Documentation Project.
 ./sent-mail-jul-2005: (Linux Documentation Project Site, rtf version)
 ./sent-mail-jul-2005: Subject: Criticism of Bozo's Windows/Linux article
 ./sent-mail-jul-2005: while mentioning that the Linux ext2/ext3 filesystem
 . . .
```

`ls | xargs -p -l gzip` gzips every file in current directory, one at a time, prompting before each operation.

Note that *xargs* processes the arguments passed to it sequentially, *one at a time*.

```
bash$ find /usr/bin | xargs file
/usr/bin:          directory
 /usr/bin/foomatic-ppd-options:          perl script text executable
 . . .
```

An interesting *xargs* option is `-n NN`, which limits to *NN* the number of arguments passed.

`ls | xargs -n 8 echo` lists the files in the current directory in `8` columns.

Another useful option is `-0`, in combination with **`find -print0`** or **`grep -lZ`**. This allows handling arguments containing whitespace or quotes.

`find / -type f -print0 | xargs -0 grep -liwZ GUI | xargs -0 rm -f`

grep -rliwZ GUI / | xargs -0 rm -f

Either of the above will remove any file containing "GUI". *(Thanks, S.C.)*

Or:

```
cat /proc/"$pid"/"$OPTION" | xargs -0 echo
#  Formats output:         ^^^^^^^^^^^^^^^
#  From Han Holl's fixup of "get-commandline.sh"
#+ script in "/dev and /proc" chapter.
```

The -P option to *xargs* permits running processes in parallel. This speeds up execution in a machine with a multicore CPU.

```
#!/bin/bash

ls *gif | xargs -t -n1 -P2 gif2png
# Converts all the gif images in current directory to png.

# Options:
# =======
# -t    Print command to stderr.
# -n1   At most 1 argument per command line.
# -P2   Run up to 2 processes simultaneously.

# Thank you, Roberto Polli, for the inspiration.
```

Example 16-5. Logfile: Using *xargs* to monitor system log

```
#!/bin/bash

# Generates a log file in current directory
# from the tail end of /var/log/messages.

# Note: /var/log/messages must be world readable
# if this script invoked by an ordinary user.
#         #root chmod 644 /var/log/messages

LINES=5

( date; uname -a ) >>logfile
# Time and machine name
echo ---------------------------------------------------------- >>logfile
tail -n $LINES /var/log/messages | xargs | fmt -s >>logfile
echo >>logfile
echo >>logfile

exit 0

#  Note:
#  ----
#  As Frank Wang points out,
#+ unmatched quotes (either single or double quotes) in the source file
#+ may give xargs indigestion.
#
#  He suggests the following substitution for line 15:
#  tail -n $LINES /var/log/messages | tr -d "\"'" | xargs | fmt -s >>logfile
```

```
#  Exercise:
#  --------
#  Modify this script to track changes in /var/log/messages at intervals
#+ of 20 minutes.
#  Hint: Use the "watch" command.
```

As in **find**, a curly bracket pair serves as a placeholder for replacement text.

Example 16-6. Copying files in current directory to another

```
#!/bin/bash
# copydir.sh

#  Copy (verbose) all files in current directory ($PWD)
#+ to directory specified on command-line.

E_NOARGS=85

if [ -z "$1" ]   # Exit if no argument given.
then
  echo "Usage: `basename $0` directory-to-copy-to"
  exit $E_NOARGS
fi

ls . | xargs -i -t cp ./{} $1
#            ^^ ^^      ^^
#  -t is "verbose" (output command-line to stderr) option.
#  -i is "replace strings" option.
#  {} is a placeholder for output text.
#  This is similar to the use of a curly-bracket pair in "find."
#
#  List the files in current directory (ls .),
#+ pass the output of "ls" as arguments to "xargs" (-i -t options),
#+ then copy (cp) these arguments ({}) to new directory ($1).
#
#  The net result is the exact equivalent of
#+   cp * $1
#+ unless any of the filenames has embedded "whitespace" characters.

exit 0
```

Example 16-7. Killing processes by name

```
#!/bin/bash
# kill-byname.sh: Killing processes by name.
# Compare this script with kill-process.sh.

#  For instance,
#+ try "./kill-byname.sh xterm" --
#+ and watch all the xterms on your desktop disappear.

#  Warning:
#  -------
#  This is a fairly dangerous script.
#  Running it carelessly (especially as root)
```

```
#+ can cause data loss and other undesirable effects.

E_BADARGS=66

if test -z "$1"  # No command-line arg supplied?
then
  echo "Usage: `basename $0` Process(es)_to_kill"
  exit $E_BADARGS
fi


PROCESS_NAME="$1"
ps ax | grep "$PROCESS_NAME" | awk '{print $1}' | xargs -i kill {} 2&>/dev/null
#                                                       ^^      ^^

# ---------------------------------------------------------------
# Notes:
# -i is the "replace strings" option to xargs.
# The curly brackets are the placeholder for the replacement.
# 2&>/dev/null suppresses unwanted error messages.
#
# Can  grep "$PROCESS_NAME" be replaced by pidof "$PROCESS_NAME"?
# ---------------------------------------------------------------

exit $?

#  The "killall" command has the same effect as this script,
#+ but using it is not quite as educational.
```

Example 16-8. Word frequency analysis using *xargs*

```
#!/bin/bash
# wf2.sh: Crude word frequency analysis on a text file.

# Uses 'xargs' to decompose lines of text into single words.
# Compare this example to the "wf.sh" script later on.


# Check for input file on command-line.
ARGS=1
E_BADARGS=85
E_NOFILE=86

if [ $# -ne "$ARGS" ]
# Correct number of arguments passed to script?
then
  echo "Usage: `basename $0` filename"
  exit $E_BADARGS
fi

if [ ! -f "$1" ]       # Does file exist?
then
  echo "File \"$1\" does not exist."
  exit $E_NOFILE
fi



#####################################################
cat "$1" | xargs -n1 | \
#  List the file, one word per line.
```

```
tr A-Z a-z | \
#  Shift characters to lowercase.
sed -e 's/\.//g'  -e 's/\,//g' -e 's/ /\
/g' | \
#  Filter out periods and commas, and
#+ change space between words to linefeed,
sort | uniq -c | sort -nr
#  Finally remove duplicates, prefix occurrence count
#+ and sort numerically.
#####################################################

#  This does the same job as the "wf.sh" example,
#+ but a bit more ponderously, and it runs more slowly (why?).

exit $?
```

expr

All-purpose expression evaluator: Concatenates and evaluates the arguments according to the operation given (arguments must be separated by spaces). Operations may be arithmetic, comparison, string, or logical.

`expr 3 + 5`
: returns 8

`expr 5 % 3`
: returns 2

`expr 1 / 0`
: returns the error message, expr: division by zero

  Illegal arithmetic operations not allowed.

`expr 5 \* 3`
: returns 15

  The multiplication operator must be escaped when used in an arithmetic expression with **expr**.

``y=`expr $y + 1` ``
: Increment a variable, with the same effect as **`let y=y+1`** and **`y=$(($y+1))`**. This is an example of arithmetic expansion.

``z=`expr substr $string $position $length` ``
: Extract substring of $length characters, starting at $position.

Example 16-9. Using *expr*

```
#!/bin/bash

# Demonstrating some of the uses of 'expr'
# =======================================

echo

# Arithmetic Operators
# ---------- ---------

echo "Arithmetic Operators"
echo
a=`expr 5 + 3`
echo "5 + 3 = $a"
```

```
a=`expr $a + 1`
echo
echo "a + 1 = $a"
echo "(incrementing a variable)"

a=`expr 5 % 3`
# modulo
echo
echo "5 mod 3 = $a"

echo
echo

# Logical Operators
# ------- ---------

#  Returns 1 if true, 0 if false,
#+ opposite of normal Bash convention.

echo "Logical Operators"
echo

x=24
y=25
b=`expr $x = $y`         # Test equality.
echo "b = $b"            # 0  ( $x -ne $y )
echo

a=3
b=`expr $a \> 10`
echo 'b=`expr $a \> 10`, therefore...'
echo "If a > 10, b = 0 (false)"
echo "b = $b"            # 0  ( 3 ! -gt 10 )
echo

b=`expr $a \< 10`
echo "If a < 10, b = 1 (true)"
echo "b = $b"            # 1  ( 3 -lt 10 )
echo
# Note escaping of operators.

b=`expr $a \<= 3`
echo "If a <= 3, b = 1 (true)"
echo "b = $b"            # 1  ( 3 -le 3 )
# There is also a "\>=" operator (greater than or equal to).


echo
echo



# String Operators
# ------ ---------

echo "String Operators"
echo

a=1234zipper43231
echo "The string being operated upon is \"$a\"."

# length: length of string
```

```
b=`expr length $a`
echo "Length of \"$a\" is $b."

# index: position of first character in substring
#        that matches a character in string
b=`expr index $a 23`
echo "Numerical position of first \"2\" in \"$a\" is \"$b\"."

# substr: extract substring, starting position & length specified
b=`expr substr $a 2 6`
echo "Substring of \"$a\", starting at position 2,\
and 6 chars long is \"$b\"."


#  The default behavior of the 'match' operations is to
#+ search for the specified match at the BEGINNING of the string.
#
#       Using Regular Expressions ...
b=`expr match "$a" '[0-9]*'`               #  Numerical count.
echo Number of digits at the beginning of \"$a\" is $b.
b=`expr match "$a" '\([0-9]*\)'`           #  Note that escaped parentheses
#                   ==      ==             #+ trigger substring match.
echo "The digits at the beginning of \"$a\" are \"$b\"."

echo

exit 0
```

The : (null) operator can substitute for **match**. For example, **`` b=`expr $a : [0-9]*` ``** is the exact equivalent of **`` b=`expr match $a [0-9]*` ``** in the above listing.

```
#!/bin/bash

echo
echo "String operations using \"expr \$string : \" construct"
echo "==================================================="
echo

a=1234zipper5FLIPPER43231

echo "The string being operated upon is \"`expr "$a" : '\(.*\)'`\"."
#     Escaped parentheses grouping operator.            ==  ==

#       ***************************
#+          Escaped parentheses
#+           match a substring
#       ***************************


#  If no escaped parentheses ...
#+ then 'expr' converts the string operand to an integer.

echo "Length of \"$a\" is `expr "$a" : '.*'`."   # Length of string

echo "Number of digits at the beginning of \"$a\" is `expr "$a" : '[0-9]*'`."

# ------------------------------------------------------------------------- #

echo

echo "The digits at the beginning of \"$a\" are `expr "$a" : '\([0-9]*\)'`."
```

```
#                                                              ==       ==
echo "The first 7 characters of \"$a\" are `expr "$a" : '\(.......\)'`."
#          =====                                          ==        ==
# Again, escaped parentheses force a substring match.
#
echo "The last 7 characters of \"$a\" are `expr "$a" : '.*\(.......\)'`."
#          ====                  end of string operator  ^^
#  (In fact, means skip over one or more of any characters until specified
#+  substring found.)

echo

exit 0
```

The above script illustrates how **expr** uses the *escaped parentheses* -- \(... \) -- grouping operator in tandem with regular expression parsing to match a substring. Here is a another example, this time from "real life."

```
# Strip the whitespace from the beginning and end.
LRFDATE=`expr "$LRFDATE" : '[[:space:]]*\(.*\)[[:space:]]*$'`

#  From Peter Knowles' "booklistgen.sh" script
#+ for converting files to Sony Librie/PRS-50X format.
#  (http://booklistgensh.peterknowles.com)
```

Perl, sed, and awk have far superior string parsing facilities. A short **sed** or **awk** "subroutine" within a script (see Section 36.2) is an attractive alternative to **expr**.

See Section 10.1 for more on using **expr** in string operations.

16.3. Time / Date Commands

Time/date and timing

date

Simply invoked, **date** prints the date and time to `stdout`. Where this command gets interesting is in its formatting and parsing options.

Example 16-10. Using *date*

```
#!/bin/bash
# Exercising the 'date' command

echo "The number of days since the year's beginning is `date +%j`."
# Needs a leading '+' to invoke formatting.
# %j gives day of year.

echo "The number of seconds elapsed since 01/01/1970 is `date +%s`."
#  %s yields number of seconds since "UNIX epoch" began,
#+ but how is this useful?

prefix=temp
suffix=$(date +%s)  # The "+%s" option to 'date' is GNU-specific.
filename=$prefix.$suffix
echo "Temporary filename = $filename"
#  It's great for creating "unique and random" temp filenames,
#+ even better than using $$.
```

```
# Read the 'date' man page for more formatting options.

exit 0
```

The `-u` option gives the UTC (Universal Coordinated Time).

```
bash$ date
Fri Mar 29 21:07:39 MST 2002



bash$ date -u
Sat Mar 30 04:07:42 UTC 2002
```

This option facilitates calculating the time between different dates.

Example 16-11. *Date* calculations

```
#!/bin/bash
# date-calc.sh
# Author: Nathan Coulter
# Used in ABS Guide with permission (thanks!).

MPHR=60    # Minutes per hour.
HPD=24     # Hours per day.

diff () {
        printf '%s' $(( $(date -u -d"$TARGET" +%s) -
                        $(date -u -d"$CURRENT" +%s)))
#                       %d = day of month.
}


CURRENT=$(date -u -d '2007-09-01 17:30:24' '+%F %T.%N %Z')
TARGET=$(date -u -d'2007-12-25 12:30:00' '+%F %T.%N %Z')
# %F = full date, %T = %H:%M:%S, %N = nanoseconds, %Z = time zone.

printf '\nIn 2007, %s ' \
       "$(date -d"$CURRENT +
        $(( $(diff) /$MPHR /$MPHR /$HPD / 2 )) days" '+%d %B')"
#       %B = name of month                ^ halfway
printf 'was halfway between %s ' "$(date -d"$CURRENT" '+%d %B')"
printf 'and %s\n' "$(date -d"$TARGET" '+%d %B')"

printf '\nOn %s at %s, there were\n' \
        $(date -u -d"$CURRENT" +%F) $(date -u -d"$CURRENT" +%T)
DAYS=$(( $(diff) / $MPHR / $MPHR / $HPD ))
CURRENT=$(date -d"$CURRENT +$DAYS days" '+%F %T.%N %Z')
HOURS=$(( $(diff) / $MPHR / $MPHR ))
CURRENT=$(date -d"$CURRENT +$HOURS hours" '+%F %T.%N %Z')
MINUTES=$(( $(diff) / $MPHR ))
CURRENT=$(date -d"$CURRENT +$MINUTES minutes" '+%F %T.%N %Z')
printf '%s days, %s hours, ' "$DAYS" "$HOURS"
printf '%s minutes, and %s seconds ' "$MINUTES" "$(diff)"
printf 'until Christmas Dinner!\n\n'

#  Exercise:
#  --------
#  Rewrite the diff () function to accept passed parameters,
```

```
#+ rather than using global variables.
```

The *date* command has quite a number of *output* options. For example `%N` gives the nanosecond portion of the current time. One interesting use for this is to generate random integers.

```
date +%N | sed -e 's/000$//' -e 's/^0//'
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
#  Strip off leading and trailing zeroes, if present.
#  Length of generated integer depends on
#+ how many zeroes stripped off.

# 115281032
# 63408725
# 394504284
```

There are many more options (try **man date**).

```
date +%j
# Echoes day of the year (days elapsed since January 1).

date +%k%M
# Echoes hour and minute in 24-hour format, as a single digit string.



# The 'TZ' parameter permits overriding the default time zone.
date                 # Mon Mar 28 21:42:16 MST 2005
TZ=EST date          # Mon Mar 28 23:42:16 EST 2005
# Thanks, Frank Kannemann and Pete Sjoberg, for the tip.


SixDaysAgo=$(date --date='6 days ago')
OneMonthAgo=$(date --date='1 month ago')  # Four weeks back (not a month!)
OneYearAgo=$(date --date='1 year ago')
```

See also Example 3-4 and Example A-43.

zdump

Time zone dump: echoes the time in a specified time zone.

```
bash$ zdump EST
EST  Tue Sep 18 22:09:22 2001 EST
```

time

Outputs verbose timing statistics for executing a command.

`time ls -l /` gives something like this:

```
real    0m0.067s
 user    0m0.004s
 sys     0m0.005s
```

See also the very similar times command in the previous section.

As of version 2.0 of Bash, **time** became a shell reserved word, with slightly altered behavior in a pipeline.

touch

Utility for updating access/modification times of a file to current system time or other specified time, but also useful for creating a new file. The command **`touch zzz`** will create a new file of zero length, named `zzz`, assuming that `zzz` did not previously exist. Time-stamping empty files in this way is useful for storing date information, for example in keeping track of modification times on a project.

The **touch** command is equivalent to **`: >> newfile`** or **`>> newfile`** (for ordinary files).

Before doing a cp -u (*copy/update*), use **touch** to update the time stamp of files you don't wish overwritten.

As an example, if the directory `/home/bozo/tax_audit` contains the files `spreadsheet-051606.data`, `spreadsheet-051706.data`, and `spreadsheet-051806.data`, then doing a **touch spreadsheet*.data** will protect these files from being overwritten by files with the same names during a **cp -u /home/bozo/financial_info/spreadsheet*data /home/bozo/tax_audit**.

at

The **at** job control command executes a given set of commands at a specified time. Superficially, it resembles cron, however, **at** is chiefly useful for one-time execution of a command set.

`at 2pm January 15` prompts for a set of commands to execute at that time. These commands should be shell-script compatible, since, for all practical purposes, the user is typing in an executable shell script a line at a time. Input terminates with a Ctl-D.

Using either the `-f` option or input redirection (<), **at** reads a command list from a file. This file is an executable shell script, though it should, of course, be non-interactive. Particularly clever is including the run-parts command in the file to execute a different set of scripts.

```
bash$ at 2:30 am Friday < at-jobs.list
job 2 at 2000-10-27 02:30
```

batch

The **batch** job control command is similar to **at**, but it runs a command list when the system load drops below `.8`. Like **at**, it can read commands from a file with the `-f` option.

> The concept of *batch processing* dates back to the era of mainframe computers. It means running a set of commands without user intervention.

cal

Prints a neatly formatted monthly calendar to `stdout`. Will do current year or a large range of past and future years.

sleep

This is the shell equivalent of a *wait loop*. It pauses for a specified number of seconds, doing nothing. It can be useful for timing or in processes running in the background, checking for a specific event every so often (polling), as in Example 32-6.

```
sleep 3     # Pauses 3 seconds.
```

The **sleep** command defaults to seconds, but minute, hours, or days may also be specified.

```
sleep 3 h   # Pauses 3 hours!
```

The watch command may be a better choice than **sleep** for running commands at timed intervals.

usleep

Microsleep (the *u* may be read as the Greek *mu*, or *micro-* prefix). This is the same as **sleep**, above, but "sleeps" in microsecond intervals. It can be used for fine-grained timing, or for polling an ongoing process at very frequent intervals.

```
usleep 30     # Pauses 30 microseconds.
```

This command is part of the Red Hat *initscripts / rc-scripts* package.

The **usleep** command does not provide particularly accurate timing, and is therefore unsuitable for critical timing loops.

hwclock, **clock**

The **hwclock** command accesses or adjusts the machine's hardware clock. Some options require *root* privileges. The `/etc/rc.d/rc.sysinit` startup file uses **hwclock** to set the system time from the hardware clock at bootup.

The **clock** command is a synonym for **hwclock**.

16.4. Text Processing Commands

Commands affecting text and text files

sort

File sort utility, often used as a filter in a pipe. This command sorts a *text stream* or file forwards or backwards, or according to various keys or character positions. Using the `-m` option, it merges presorted input files. The *info page* lists its many capabilities and options. See Example 11-10, Example 11-11, and Example A-8.

tsort

Topological sort, reading in pairs of whitespace-separated strings and sorting according to input patterns. The original purpose of **tsort** was to sort a list of dependencies for an obsolete version of the *ld* linker in an "ancient" version of UNIX.

The results of a *tsort* will usually differ markedly from those of the standard **sort** command, above.

uniq

This filter removes duplicate lines from a sorted file. It is often seen in a pipe coupled with sort.

```
cat list-1 list-2 list-3 | sort | uniq > final.list
# Concatenates the list files,
# sorts them,
# removes duplicate lines,
# and finally writes the result to an output file.
```

The useful `-c` option prefixes each line of the input file with its number of occurrences.

```
bash$ cat testfile
This line occurs only once.
 This line occurs twice.
 This line occurs twice.
 This line occurs three times.
```

```
 This line occurs three times.
 This line occurs three times.


bash$ uniq -c testfile
      1 This line occurs only once.
       2 This line occurs twice.
       3 This line occurs three times.


bash$ sort testfile | uniq -c | sort -nr
      3 This line occurs three times.
       2 This line occurs twice.
       1 This line occurs only once.
```

The **`sort INPUTFILE | uniq -c | sort -nr`** command string produces a *frequency of occurrence* listing on the `INPUTFILE` file (the `-nr` options to **sort** cause a reverse numerical sort). This template finds use in analysis of log files and dictionary lists, and wherever the lexical structure of a document needs to be examined.

Example 16-12. Word Frequency Analysis

```
#!/bin/bash
# wf.sh: Crude word frequency analysis on a text file.
# This is a more efficient version of the "wf2.sh" script.


# Check for input file on command-line.
ARGS=1
E_BADARGS=85
E_NOFILE=86

if [ $# -ne "$ARGS" ]  # Correct number of arguments passed to script?
then
  echo "Usage: `basename $0` filename"
  exit $E_BADARGS
fi

if [ ! -f "$1" ]       # Check if file exists.
then
  echo "File \"$1\" does not exist."
  exit $E_NOFILE
fi



########################################################
# main ()
sed -e 's/\.//g'  -e 's/\,//g' -e 's/ /\
/g' "$1" | tr 'A-Z' 'a-z' | sort | uniq -c | sort -nr
#                           =========================
#                            Frequency of occurrence

#  Filter out periods and commas, and
#+ change space between words to linefeed,
#+ then shift characters to lowercase, and
#+ finally prefix occurrence count and sort numerically.

#  Arun Giridhar suggests modifying the above to:
```

```
#  . . . | sort | uniq -c | sort +1 [-f] | sort +0 -nr
#  This adds a secondary sort key, so instances of
#+ equal occurrence are sorted alphabetically.
#  As he explains it:
#  "This is effectively a radix sort, first on the
#+ least significant column
#+ (word or string, optionally case-insensitive)
#+ and last on the most significant column (frequency)."
#
#  As Frank Wang explains, the above is equivalent to
#+       . . . | sort | uniq -c | sort +0 -nr
#+ and the following also works:
#+       . . . | sort | uniq -c | sort -k1nr -k
########################################################

exit 0

# Exercises:
# ---------
# 1) Add 'sed' commands to filter out other punctuation,
#+   such as semicolons.
# 2) Modify the script to also filter out multiple spaces and
#+   other whitespace.
```

```
bash$ cat testfile
This line occurs only once.
 This line occurs twice.
 This line occurs twice.
 This line occurs three times.
 This line occurs three times.
 This line occurs three times.


bash$ ./wf.sh testfile
      6 this
       6 occurs
       6 line
       3 times
       3 three
       2 twice
       1 only
       1 once
```

expand, **unexpand**

The **expand** filter converts tabs to spaces. It is often used in a pipe.

The **unexpand** filter converts spaces to tabs. This reverses the effect of **expand**.

cut

A tool for extracting fields from files. It is similar to the `print $N` command set in awk, but more limited. It may be simpler to use *cut* in a script than *awk*. Particularly important are the `-d` (delimiter) and `-f` (field specifier) options.

Using **cut** to obtain a listing of the mounted filesystems:

```
cut -d ' ' -f1,2 /etc/mtab
```

Using **cut** to list the OS and kernel version:

```
uname -a | cut -d" " -f1,3,11,12
```

Using **cut** to extract message headers from an e-mail folder:

```
bash$ grep '^Subject:' read-messages | cut -c10-80
Re: Linux suitable for mission-critical apps?
 MAKE MILLIONS WORKING AT HOME!!!
 Spam complaint
 Re: Spam complaint
```

Using **cut** to parse a file:

```
# List all the users in /etc/passwd.

FILENAME=/etc/passwd

for user in $(cut -d: -f1 $FILENAME)
do
  echo $user
done

# Thanks, Oleg Philon for suggesting this.
```

`cut -d ' ' -f2,3 filename` is equivalent to **`awk -F'[ ]' '{ print $2, $3 }' filename`**

It is even possible to specify a linefeed as a delimiter. The trick is to actually embed a linefeed (**RETURN**) in the command sequence.

```
bash$ cut -d'
 ' -f3,7,19 testfile
This is line 3 of testfile.
 This is line 7 of testfile.
 This is line 19 of testfile.
```

Thank you, Jaka Kranjc, for pointing this out.

See also Example 16-48.

paste

Tool for merging together different files into a single, multi-column file. In combination with cut, useful for creating system log files.

```
bash$ cat items
alphabet blocks
 building blocks
 cables

bash$ cat prices
$1.00/dozen
 $2.50 ea.
 $3.75

bash$ paste items prices
alphabet blocks $1.00/dozen
 building blocks $2.50 ea.
 cables  $3.75
```

join

Consider this a special-purpose cousin of **paste**. This powerful utility allows merging two files in a meaningful fashion, which essentially creates a simple version of a relational database.

The **join** command operates on exactly two files, but pastes together only those lines with a common tagged field (usually a numerical label), and writes the result to `stdout`. The files to be joined should be sorted according to the tagged field for the matchups to work properly.

```
File: 1.data

100 Shoes
200 Laces
300 Socks
```

```
File: 2.data

100 $40.00
200 $1.00
300 $2.00
```

```
bash$ join 1.data 2.data
File: 1.data 2.data

 100 Shoes $40.00
 200 Laces $1.00
 300 Socks $2.00
```

The tagged field appears only once in the output.

head

lists the beginning of a file to `stdout`. The default is `10` lines, but a different number can be specified. The command has a number of interesting options.

Example 16-13. Which files are scripts?

```
#!/bin/bash
# script-detector.sh: Detects scripts within a directory.

TESTCHARS=2    # Test first 2 characters.
SHABANG='#!'   # Scripts begin with a "sha-bang."

for file in *  # Traverse all the files in current directory.
do
  if [[ `head -c$TESTCHARS "$file"` = "$SHABANG" ]]
  #      head -c2                      #!
  #  The '-c' option to "head" outputs a specified
  #+ number of characters, rather than lines (the default).
  then
    echo "File \"$file\" is a script."
  else
    echo "File \"$file\" is *not* a script."
  fi
done

exit 0

#  Exercises:
#  ---------
#  1) Modify this script to take as an optional argument
#+    the directory to scan for scripts
#+    (rather than just the current working directory).
```

```
#
#  2) As it stands, this script gives "false positives" for
#+    Perl, awk, and other scripting language scripts.
#     Correct this.
```

Example 16-14. Generating 10-digit random numbers

```
#!/bin/bash
# rnd.sh: Outputs a 10-digit random number

# Script by Stephane Chazelas.

head -c4 /dev/urandom | od -N4 -tu4 | sed -ne '1s/.* //p'


# ===================================================================== #

# Analysis
# --------

# head:
# -c4 option takes first 4 bytes.

# od:
# -N4 option limits output to 4 bytes.
# -tu4 option selects unsigned decimal format for output.

# sed:
# -n option, in combination with "p" flag to the "s" command,
# outputs only matched lines.



# The author of this script explains the action of 'sed', as follows.

# head -c4 /dev/urandom | od -N4 -tu4 | sed -ne '1s/.* //p'
# ----------------------------------> |

# Assume output up to "sed" --------> |
# is 0000000 1198195154\n

#  sed begins reading characters: 0000000 1198195154\n.
#  Here it finds a newline character,
#+ so it is ready to process the first line (0000000 1198195154).
#  It looks at its <range><action>s. The first and only one is

#   range     action
#   1         s/.* //p

#  The line number is in the range, so it executes the action:
#+ tries to substitute the longest string ending with a space in the line
#  ("0000000 ") with nothing (//), and if it succeeds, prints the result
#  ("p" is a flag to the "s" command here, this is different
#+ from the "p" command).

#  sed is now ready to continue reading its input. (Note that before
#+ continuing, if -n option had not been passed, sed would have printed
#+ the line once again).

#  Now, sed reads the remainder of the characters, and finds the
#+ end of the file.
```

```
#  It is now ready to process its 2nd line (which is also numbered '$' as
#+ it's the last one).
#  It sees it is not matched by any <range>, so its job is done.

#  In few word this sed commmand means:
#  "On the first line only, remove any character up to the right-most space,
#+ then print it."

# A better way to do this would have been:
#           sed -e 's/.* //;q'

# Here, two <range><action>s (could have been written
#           sed -e 's/.* //' -e q):

#   range                    action
#   nothing (matches line)   s/.* //
#   nothing (matches line)   q (quit)

#  Here, sed only reads its first line of input.
#  It performs both actions, and prints the line (substituted) before
#+ quitting (because of the "q" action) since the "-n" option is not passed.

# ================================================================== #

# An even simpler altenative to the above one-line script would be:
#           head -c4 /dev/urandom| od -An -tu4

exit
```

See also Example 16-39.

tail

lists the (tail) end of a file to `stdout`. The default is `10` lines, but this can be changed with the `-n` option. Commonly used to keep track of changes to a system logfile, using the `-f` option, which outputs lines appended to the file.

Example 16-15. Using *tail* to monitor the system log

```
#!/bin/bash

filename=sys.log

cat /dev/null > $filename; echo "Creating / cleaning out file."
#  Creates the file if it does not already exist,
#+ and truncates it to zero length if it does.
#  : > filename   and   > filename also work.

tail /var/log/messages > $filename
# /var/log/messages must have world read permission for this to work.

echo "$filename contains tail end of system log."

exit 0
```

To list a specific line of a text file, pipe the output of **head** to **tail -n 1**. For example **`head -n 8 database.txt | tail -n 1`** lists the 8th line of the file `database.txt`.

To set a variable to a given block of a text file:

```
var=$(head -n $m $filename | tail -n $n)

# filename = name of file
# m = from beginning of file, number of lines to end of block
# n = number of lines to set variable to (trim from end of block)
```

Newer implementations of **tail** deprecate the older **tail -$LINES filename** usage. The standard **tail -n $LINES filename** is correct.

See also Example 16-5, Example 16-39 and Example 32-6.

grep

A multi-purpose file search tool that uses Regular Expressions. It was originally a command/filter in the venerable **ed** line editor: **`g/re/p`** -- *global - regular expression - print*.

grep *`pattern`* [*`file`*...]

Search the target file(s) for occurrences of *`pattern`*, where *`pattern`* may be literal text or a Regular Expression.

```
bash$ grep '[rst]ystem.$' osinfo.txt
The GPL governs the distribution of the Linux operating system.
```

If no target file(s) specified, **grep** works as a filter on `stdout`, as in a pipe.

```
bash$ ps ax | grep clock
765 tty1     S      0:00 xclock
 901 pts/1    S      0:00 grep clock
```

The `-i` option causes a case-insensitive search.

The `-w` option matches only whole words.

The `-l` option lists only the files in which matches were found, but not the matching lines.

The `-r` (recursive) option searches files in the current working directory and all subdirectories below it.

The `-n` option lists the matching lines, together with line numbers.

```
bash$ grep -n Linux osinfo.txt
2:This is a file containing information about Linux.
 6:The GPL governs the distribution of the Linux operating system.
```

The `-v` (or `--invert-match`) option *filters out* matches.

```
grep pattern1 *.txt | grep -v pattern2

# Matches all lines in "*.txt" files containing "pattern1",
# but ***not*** "pattern2".
```

The `-c` (`--count`) option gives a numerical count of matches, rather than actually listing the matches.

```
grep -c txt *.sgml   # (number of occurrences of "txt" in "*.sgml" files)
```

```
#   grep -cz .
#            ^ dot
# means count (-c) zero-separated (-z) items matching "."
# that is, non-empty ones (containing at least 1 character).
#
printf 'a b\nc  d\n\n\n\n\n\000\n\000e\000\000\nf' | grep -cz .     # 3
printf 'a b\nc  d\n\n\n\n\n\000\n\000e\000\000\nf' | grep -cz '$'   # 5
printf 'a b\nc  d\n\n\n\n\n\000\n\000e\000\000\nf' | grep -cz '^'   # 5
#
printf 'a b\nc  d\n\n\n\n\n\000\n\000e\000\000\nf' | grep -c '$'    # 9
# By default, newline chars (\n) separate items to match.

# Note that the -z option is GNU "grep" specific.


# Thanks, S.C.
```

The `--color` (or `--colour`) option marks the matching string in color (on the console or in an *xterm* window). Since *grep* prints out each entire line containing the matching pattern, this lets you see exactly *what* is being matched. See also the `-o` option, which shows only the matching portion of the line(s).

Example 16-16. Printing out the *From* lines in stored e-mail messages

```
#!/bin/bash
# from.sh

#  Emulates the useful 'from' utility in Solaris, BSD, etc.
#  Echoes the "From" header line in all messages
#+ in your e-mail directory.


MAILDIR=~/mail/*               #  No quoting of variable. Why?
# Maybe check if-exists $MAILDIR:   if [ -d $MAILDIR ] . . .
GREP_OPTS="-H -A 5 --color"    #  Show file, plus extra context lines
                               #+ and display "From" in color.
TARGETSTR="^From"              # "From" at beginning of line.

for file in $MAILDIR           #  No quoting of variable.
do
  grep $GREP_OPTS "$TARGETSTR" "$file"
  #    ^^^^^^^^^^              #  Again, do not quote this variable.
  echo
done

exit $?

#  You might wish to pipe the output of this script to 'more'
#+ or redirect it to a file . . .
```

When invoked with more than one target file given, **grep** specifies which file contains matches.

```
bash$ grep Linux osinfo.txt misc.txt
osinfo.txt:This is a file containing information about Linux.
 osinfo.txt:The GPL governs the distribution of the Linux operating system.
 misc.txt:The Linux operating system is steadily gaining in popularity.
```

To force **grep** to show the filename when searching only one target file, simply give `/dev/null` as the second file.

```
bash$ grep Linux osinfo.txt /dev/null
osinfo.txt:This is a file containing information about Linux.
 osinfo.txt:The GPL governs the distribution of the Linux operating system.
```

If there is a successful match, **grep** returns an exit status of 0, which makes it useful in a condition test in a script, especially in combination with the `-q` option to suppress output.

```
SUCCESS=0                      # if grep lookup succeeds
word=Linux
filename=data.file

grep -q "$word" "$filename"    #  The "-q" option
                               #+ causes nothing to echo to stdout.
if [ $? -eq $SUCCESS ]
# if grep -q "$word" "$filename"   can replace lines 5 - 7.
then
  echo "$word found in $filename"
else
  echo "$word not found in $filename"
fi
```

Example 32-6 demonstrates how to use **grep** to search for a word pattern in a system logfile.

Example 16-17. Emulating *grep* in a script

```
#!/bin/bash
# grp.sh: Rudimentary reimplementation of grep.

E_BADARGS=85

if [ -z "$1" ]    # Check for argument to script.
then
  echo "Usage: `basename $0` pattern"
  exit $E_BADARGS
fi

echo

for file in *     # Traverse all files in $PWD.
do
  output=$(sed -n /"$1"/p $file)  # Command substitution.

  if [ ! -z "$output" ]           # What happens if "$output" is not quoted?
  then
    echo -n "$file: "
    echo "$output"
  fi              #  sed -ne "/$1/s|^|${file}: |p"  is equivalent to above.

  echo
done

echo

exit 0

# Exercises:
```

```
# ---------
# 1) Add newlines to output, if more than one match in any given file.
# 2) Add features.
```

How can **grep** search for two (or more) separate patterns? What if you want **grep** to display all lines in a file or files that contain both "pattern1" *and* "pattern2"?

One method is to pipe the result of **grep pattern1** to **grep pattern2**.

For example, given the following file:

```
# Filename: tstfile

This is a sample file.
This is an ordinary text file.
This file does not contain any unusual text.
This file is not unusual.
Here is some text.
```

Now, let's search this file for lines containing *both* "file" and "text" . . .

```
bash$ grep file tstfile
# Filename: tstfile
 This is a sample file.
 This is an ordinary text file.
 This file does not contain any unusual text.
 This file is not unusual.

bash$ grep file tstfile | grep text
This is an ordinary text file.
 This file does not contain any unusual text.
```

Now, for an interesting recreational use of *grep* . . .

Example 16-18. Crossword puzzle solver

```
#!/bin/bash
# cw-solver.sh
# This is actually a wrapper around a one-liner (line 46).

#  Crossword puzzle and anagramming word game solver.
#  You know *some* of the letters in the word you're looking for,
#+ so you need a list of all valid words
#+ with the known letters in given positions.
#  For example: w...i....n
#               1???5????10
# w in position 1, 3 unknowns, i in the 5th, 4 unknowns, n at the end.
# (See comments at end of script.)


E_NOPATT=71
DICT=/usr/share/dict/word.lst
#                    ^^^^^^^^   Looks for word list here.
#  ASCII word list, one word per line.
#  If you happen to need an appropriate list,
#+ download the author's "yawl" word list package.
#  http://ibiblio.org/pub/Linux/libs/yawl-0.3.2.tar.gz
#  or
#  http://bash.deta.in/yawl-0.3.2.tar.gz

```

```
if [ -z "$1" ]   #  If no word pattern specified
then             #+ as a command-line argument . . .
  echo           #+ . . . then . . .
  echo "Usage:"  #+ Usage message.
  echo
  echo ""$0" \"pattern,\""
  echo "where \"pattern\" is in the form"
  echo "xxx..x.x..."
  echo
  echo "The x's represent known letters,"
  echo "and the periods are unknown letters (blanks)."
  echo "Letters and periods can be in any position."
  echo "For example, try:   sh cw-solver.sh w...i....n"
  echo
  exit $E_NOPATT
fi

echo
# ===============================================
# This is where all the work gets done.
grep ^"$1"$ "$DICT"   # Yes, only one line!
#    |    |
# ^ is start-of-word regex anchor.
# $ is end-of-word regex anchor.

#  From _Stupid Grep Tricks_, vol. 1,
#+ a book the ABS Guide author may yet get around
#+ to writing . . . one of these days . . .
# ===============================================
echo


exit $?  # Script terminates here.
#  If there are too many words generated,
#+ redirect the output to a file.

$ sh cw-solver.sh w...i....n

wellington
workingman
workingmen
```

egrep -- *extended grep* -- is the same as **grep -E**. This uses a somewhat different, extended set of Regular Expressions, which can make the search a bit more flexible. It also allows the boolean | (*or*) operator.

```
bash $ egrep 'matches|Matches' file.txt
Line 1 matches.
 Line 3 Matches.
 Line 4 contains matches, but also Matches
```

fgrep -- *fast grep* -- is the same as **grep -F**. It does a literal string search (no Regular Expressions), which generally speeds things up a bit.

On some Linux distros, **egrep** and **fgrep** are symbolic links to, or aliases for **grep**, but invoked with the `-E` and `-F` options, respectively.

Example 16-19. Looking up definitions in *Webster's 1913 Dictionary*

```
#!/bin/bash
# dict-lookup.sh

#  This script looks up definitions in the 1913 Webster's Dictionary.
#  This Public Domain dictionary is available for download
#+ from various sites, including
#+ Project Gutenberg (http://www.gutenberg.org/etext/247).
#
#  Convert it from DOS to UNIX format (with only LF at end of line)
#+ before using it with this script.
#  Store the file in plain, uncompressed ASCII text.
#  Set DEFAULT_DICTFILE variable below to path/filename.


E_BADARGS=85
MAXCONTEXTLINES=50                        # Maximum number of lines to show.
DEFAULT_DICTFILE="/usr/share/dict/webster1913-dict.txt"
                                          # Default dictionary file pathname.
                                          # Change this as necessary.
#  Note:
#  ----
#  This particular edition of the 1913 Webster's
#+ begins each entry with an uppercase letter
#+ (lowercase for the remaining characters).
#  Only the *very first line* of an entry begins this way,
#+ and that's why the search algorithm below works.



if [[ -z $(echo "$1" | sed -n '/^[A-Z]/p') ]]
#  Must at least specify word to look up, and
#+ it must start with an uppercase letter.
then
  echo "Usage: `basename $0` Word-to-define [dictionary-file]"
  echo
  echo "Note: Word to look up must start with capital letter,"
  echo "with the rest of the word in lowercase."
  echo "--------------------------------------------"
  echo "Examples: Abandon, Dictionary, Marking, etc."
  exit $E_BADARGS
fi


if [ -z "$2" ]                            #  May specify different dictionary
                                          #+ as an argument to this script.
then
  dictfile=$DEFAULT_DICTFILE
else
  dictfile="$2"
fi

# ---------------------------------------------------------
Definition=$(fgrep -A $MAXCONTEXTLINES "$1 \\" "$dictfile")
#                  Definitions in form "Word \..."
#
#  And, yes, "fgrep" is fast enough
#+ to search even a very large text file.


# Now, snip out just the definition block.

echo "$Definition" |
```

```
sed -n '1,/^[A-Z]/p' |
#  Print from first line of output
#+ to the first line of the next entry.
sed '$d' | sed '$d'
#  Delete last two lines of output
#+ (blank line and first line of next entry).
# ---------------------------------------------------------

exit $?

# Exercises:
# ---------
# 1)  Modify the script to accept any type of alphabetic input
#   + (uppercase, lowercase, mixed case), and convert it
#   + to an acceptable format for processing.
#
# 2)  Convert the script to a GUI application,
#   + using something like 'gdialog' or 'zenity' . . .
#     The script will then no longer take its argument(s)
#   + from the command-line.
#
# 3)  Modify the script to parse one of the other available
#   + Public Domain Dictionaries, such as the U.S. Census Bureau Gazetteer.
```

See also Example A-41 for an example of speedy *fgrep* lookup on a large text file.

agrep (*approximate grep*) extends the capabilities of **grep** to approximate matching. The search string may differ by a specified number of characters from the resulting matches. This utility is not part of the core Linux distribution.

To search compressed files, use **zgrep**, **zegrep**, or **zfgrep**. These also work on non-compressed files, though slower than plain **grep**, **egrep**, **fgrep**. They are handy for searching through a mixed set of files, some compressed, some not.

To search bzipped files, use **bzgrep**.

look

The command **look** works like **grep**, but does a lookup on a "dictionary," a sorted word list. By default, **look** searches for a match in `/usr/dict/words`, but a different dictionary file may be specified.

Example 16-20. Checking words in a list for validity

```
#!/bin/bash
# lookup: Does a dictionary lookup on each word in a data file.

file=words.data  # Data file from which to read words to test.

echo
echo "Testing file $file"
echo

while [ "$word" != end ]  # Last word in data file.
do                 # ^^^
```

```
  read word      # From data file, because of redirection at end of loop.
  look $word > /dev/null  # Don't want to display lines in dictionary file.
  #  Searches for words in the file /usr/share/dict/words
  #+ (usually a link to linux.words).
  lookup=$?      # Exit status of 'look' command.

  if [ "$lookup" -eq 0 ]
  then
    echo "\"$word\" is valid."
  else
    echo "\"$word\" is invalid."
  fi

done <"$file"    # Redirects stdin to $file, so "reads" come from there.

echo

exit 0

# ----------------------------------------------------------------
# Code below line will not execute because of "exit" command above.


# Stephane Chazelas proposes the following, more concise alternative:

while read word && [[ $word != end ]]
do if look "$word" > /dev/null
   then echo "\"$word\" is valid."
   else echo "\"$word\" is invalid."
   fi
done <"$file"

exit 0
```

sed, **awk**

Scripting languages especially suited for parsing text files and command output. May be embedded singly or in combination in pipes and shell scripts.

sed

Non-interactive "stream editor", permits using many **ex** commands in batch mode. It finds many uses in shell scripts.

awk

Programmable file extractor and formatter, good for manipulating and/or extracting fields (columns) in structured text files. Its syntax is similar to C.

wc

wc gives a "word count" on a file or I/O stream:

```
bash $ wc /usr/share/doc/sed-4.1.2/README
13  70  447 README
[13 lines  70 words  447 characters]
```

wc -w gives only the word count.

wc -l gives only the line count.

wc -c gives only the byte count.

wc -m gives only the character count.

wc -L gives only the length of the longest line.

Using **wc** to count how many `.txt` files are in current working directory:

```
$ ls *.txt | wc -l
#  Will work as long as none of the "*.txt" files
#+ have a linefeed embedded in their name.

#  Alternative ways of doing this are:
#      find . -maxdepth 1 -name \*.txt -print0 | grep -cz .
#      (shopt -s nullglob; set -- *.txt; echo $#)

#  Thanks, S.C.
```

Using **wc** to total up the size of all the files whose names begin with letters in the range d - h

```
bash$ wc [d-h]* | grep total | awk '{print $3}'
71832
```

Using **wc** to count the instances of the word "Linux" in the main source file for this book.

```
bash$ grep Linux abs-book.sgml | wc -l
138
```

See also Example 16-39 and Example 20-8.

Certain commands include some of the functionality of **wc** as options.

```
... | grep foo | wc -l
# This frequently used construct can be more concisely rendered.

... | grep -c foo
# Just use the "-c" (or "--count") option of grep.

# Thanks, S.C.
```

tr

character translation filter.

Must use quoting and/or brackets, as appropriate. Quotes prevent the shell from reinterpreting the special characters in **tr** command sequences. Brackets should be quoted to prevent expansion by the shell.

Either **`tr "A-Z" "*" <filename`** or **`tr A-Z \* <filename`** changes all the uppercase letters in `filename` to asterisks (writes to `stdout`). On some systems this may not work, but **`tr A-Z '[**]'`** will.

The `-d` option deletes a range of characters.

```
echo "abcdef"                 # abcdef
echo "abcdef" | tr -d b-d     # aef


tr -d 0-9 <filename
# Deletes all digits from the file "filename".
```

The `--squeeze-repeats` (or `-s`) option deletes all but the first instance of a string of consecutive characters. This option is useful for removing excess whitespace.

```
bash$ echo "XXXXX" | tr --squeeze-repeats 'X'
X
```

The `-c` "complement" option *inverts* the character set to match. With this option, **tr** acts only upon those characters *not* matching the specified set.

```
bash$ echo "acfdeb123" | tr -c b-d +
+c+d+b++++
```

Note that **tr** recognizes POSIX character classes. [74]

```
bash$ echo "abcd2ef1" | tr '[:alpha:]' -
----2--1
```

Example 16-21. *toupper*: Transforms a file to all uppercase.

```
#!/bin/bash
# Changes a file to all uppercase.

E_BADARGS=85

if [ -z "$1" ]  # Standard check for command-line arg.
then
  echo "Usage: `basename $0` filename"
  exit $E_BADARGS
fi

tr a-z A-Z <"$1"

# Same effect as above, but using POSIX character set notation:
#        tr '[:lower:]' '[:upper:]' <"$1"
# Thanks, S.C.

#     Or even . . .
#     cat "$1" | tr a-z A-Z
#     Or dozens of other ways . . .

exit 0

#  Exercise:
#  Rewrite this script to give the option of changing a file
#+ to *either* upper or lowercase.
#  Hint: Use either the "case" or "select" command.
```

Example 16-22. *lowercase*: Changes all filenames in working directory to lowercase.

```
#!/bin/bash
#
#  Changes every filename in working directory to all lowercase.
#
#  Inspired by a script of John Dubois,
#+ which was translated into Bash by Chet Ramey,
#+ and considerably simplified by the author of the ABS Guide.


for filename in *                # Traverse all files in directory.
do
   fname=`basename $filename`
   n=`echo $fname | tr A-Z a-z`  # Change name to lowercase.
```

```
   if [ "$fname" != "$n" ]       # Rename only files not already lowercase.
   then
     mv $fname $n
   fi
done

exit $?


# Code below this line will not execute because of "exit".
#--------------------------------------------------------#
# To run it, delete script above line.

# The above script will not work on filenames containing blanks or newlines.
# Stephane Chazelas therefore suggests the following alternative:


for filename in *    # Not necessary to use basename,
                     # since "*" won't return any file containing "/".
do n=`echo "$filename/" | tr '[:upper:]' '[:lower:]'`
#                             POSIX char set notation.
#                    Slash added so that trailing newlines are not
#                    removed by command substitution.
   # Variable substitution:
   n=${n%/}          # Removes trailing slash, added above, from filename.
   [[ $filename == $n ]] || mv "$filename" "$n"
                     # Checks if filename already lowercase.
done

exit $?
```

Example 16-23. *du*: DOS to UNIX text file conversion.

```
#!/bin/bash
# Du.sh: DOS to UNIX text file converter.

E_WRONGARGS=85

if [ -z "$1" ]
then
  echo "Usage: `basename $0` filename-to-convert"
  exit $E_WRONGARGS
fi

NEWFILENAME=$1.unx

CR='\015'  # Carriage return.
           # 015 is octal ASCII code for CR.
           # Lines in a DOS text file end in CR-LF.
           # Lines in a UNIX text file end in LF only.

tr -d $CR < $1 > $NEWFILENAME
# Delete CR's and write to new file.

echo "Original DOS text file is \"$1\"."
echo "Converted UNIX text file is \"$NEWFILENAME\"."

exit 0

# Exercise:
```

```
# --------
# Change the above script to convert from UNIX to DOS.
```

Example 16-24. *rot13*: ultra-weak encryption.

```
#!/bin/bash
# rot13.sh: Classic rot13 algorithm,
#           encryption that might fool a 3-year old
#           for about 10 minutes.

# Usage: ./rot13.sh filename
# or     ./rot13.sh <filename
# or     ./rot13.sh and supply keyboard input (stdin)

cat "$@" | tr 'a-zA-Z' 'n-za-mN-ZA-M'   # "a" goes to "n", "b" to "o" ...
#  The   cat "$@"   construct
#+ permits input either from stdin or from files.

exit 0
```

Example 16-25. Generating "Crypto-Quote" Puzzles

```
#!/bin/bash
# crypto-quote.sh: Encrypt quotes

#  Will encrypt famous quotes in a simple monoalphabetic substitution.
#  The result is similar to the "Crypto Quote" puzzles
#+ seen in the Op Ed pages of the Sunday paper.


key=ETAOINSHRDLUBCFGJMQPVWZYXK
# The "key" is nothing more than a scrambled alphabet.
# Changing the "key" changes the encryption.

# The 'cat "$@"' construction gets input either from stdin or from files.
# If using stdin, terminate input with a Control-D.
# Otherwise, specify filename as command-line parameter.

cat "$@" | tr "a-z" "A-Z" | tr "A-Z" "$key"
#        |  to uppercase  |     encrypt
# Will work on lowercase, uppercase, or mixed-case quotes.
# Passes non-alphabetic characters through unchanged.


# Try this script with something like:
# "Nothing so needs reforming as other people's habits."
# --Mark Twain
#
# Output is:
# "CFPHRCS QF CIIOQ MINFMBRCS EQ FPHIM GIFGUI'Q HETRPQ."
# --BEML PZERC

# To reverse the encryption:
# cat "$@" | tr "$key" "A-Z"


#  This simple-minded cipher can be broken by an average 12-year old
#+ using only pencil and paper.

exit 0
```

```
#  Exercise:
#  --------
#  Modify the script so that it will either encrypt or decrypt,
#+ depending on command-line argument(s).
```

Of course, *tr* lends itself to *code obfuscation*.

```
#!/bin/bash
# jabh.sh

x="wftedskaebjgdBstbdbsmnjgz"
echo $x | tr "a-z" 'oh, turtleneck Phrase Jar!'

# Based on the Wikipedia "Just another Perl hacker" article.
```

***tr* variants**

The **tr** utility has two historic variants. The BSD version does not use brackets (**`tr a-z A-Z`**), but the SysV one does (**`tr '[a-z]' '[A-Z]'`**). The GNU version of **tr** resembles the BSD one.

fold

A filter that wraps lines of input to a specified width. This is especially useful with the `-s` option, which breaks lines at word spaces (see Example 16-26 and Example A-1).

fmt

Simple-minded file formatter, used as a filter in a pipe to "wrap" long lines of text output.

Example 16-26. Formatted file listing.

```
#!/bin/bash

WIDTH=40                    # 40 columns wide.

b=`ls /usr/local/bin`       # Get a file listing...

echo $b | fmt -w $WIDTH

# Could also have been done by
#    echo $b | fold - -s -w $WIDTH

exit 0
```

See also Example 16-5.

A powerful alternative to **fmt** is Kamil Toman's **par** utility, available from http://www.cs.berkeley.edu/~amc/Par/.

col

This deceptively named filter removes reverse line feeds from an input stream. It also attempts to replace whitespace with equivalent tabs. The chief use of **col** is in filtering the output from certain text processing utilities, such as **groff** and **tbl**.

column

Column formatter. This filter transforms list-type text output into a "pretty-printed" table by inserting tabs at appropriate places.

Example 16-27. Using *column* to format a directory listing

```
#!/bin/bash
# colms.sh
# A minor modification of the example file in the "column" man page.


(printf "PERMISSIONS LINKS OWNER GROUP SIZE MONTH DAY HH:MM PROG-NAME\n" \
; ls -l | sed 1d) | column -t
#         ^^^^^^           ^^

#  The "sed 1d" in the pipe deletes the first line of output,
#+ which would be "total        N",
#+ where "N" is the total number of files found by "ls -l".

# The -t option to "column" pretty-prints a table.

exit 0
```

colrm

Column removal filter. This removes columns (characters) from a file and writes the file, lacking the range of specified columns, back to `stdout`. **`colrm 2 4 <filename`** removes the second through fourth characters from each line of the text file `filename`.

If the file contains tabs or nonprintable characters, this may cause unpredictable behavior. In such cases, consider using expand and **unexpand** in a pipe preceding **colrm**.

nl

Line numbering filter: **`nl filename`** lists `filename` to `stdout`, but inserts consecutive numbers at the beginning of each non-blank line. If `filename` omitted, operates on `stdin`.

The output of **nl** is very similar to **`cat -b`**, since, by default **nl** does not list blank lines.

Example 16-28. *nl*: A self-numbering script.

```
#!/bin/bash
# line-number.sh

# This script echoes itself twice to stdout with its lines numbered.

echo "     line number = $LINENO" # 'nl' sees this as line 4
#                                   (nl does not number blank lines).
#                                   'cat -n' sees it correctly as line #6.

nl `basename $0`

echo; echo  # Now, let's try it with 'cat -n'

cat -n `basename $0`
# The difference is that 'cat -n' numbers the blank lines.
# Note that 'nl -ba' will also do so.

exit 0
# -----------------------------------------------------------------
```

pr

Print formatting filter. This will paginate files (or `stdout`) into sections suitable for hard copy printing or viewing on screen. Various options permit row and column manipulation, joining lines, setting margins, numbering lines, adding page headers, and merging files, among other things. The **pr** command combines much of the functionality of **nl**, **paste**, **fold**, **column**, and **expand**.

`pr -o 5 --width=65 fileZZZ | more` gives a nice paginated listing to screen of `fileZZZ` with margins set at 5 and 65.

A particularly useful option is `-d`, forcing double-spacing (same effect as **sed -G**).

gettext

The GNU **gettext** package is a set of utilities for localizing and translating the text output of programs into foreign languages. While originally intended for C programs, it now supports quite a number of programming and scripting languages.

The **gettext** *program* works on shell scripts. See the *`info page`*.

msgfmt

A program for generating binary message catalogs. It is used for localization.

iconv

A utility for converting file(s) to a different encoding (character set). Its chief use is for localization.

```
# Convert a string from UTF-8 to UTF-16 and print to the BookList
function write_utf8_string {
    STRING=$1
    BOOKLIST=$2
    echo -n "$STRING" | iconv -f UTF8 -t UTF16 | \
    cut -b 3- | tr -d \\n >> "$BOOKLIST"
}

#  From Peter Knowles' "booklistgen.sh" script
#+ for converting files to Sony Librie/PRS-50X format.
#  (http://booklistgensh.peterknowles.com)
```

recode

Consider this a fancier version of **iconv**, above. This very versatile utility for converting a file to a different encoding scheme. Note that *recode* is not part of the standard Linux installation.

TeX, **gs**

TeX and **Postscript** are text markup languages used for preparing copy for printing or formatted video display.

TeX is Donald Knuth's elaborate typsetting system. It is often convenient to write a shell script encapsulating all the options and arguments passed to one of these markup languages.

Ghostscript (**gs**) is a GPL-ed Postscript interpreter.

texexec

Utility for processing *TeX* and *pdf* files. Found in `/usr/bin` on many Linux distros, it is actually a shell wrapper that calls Perl to invoke *Tex*.

```
texexec --pdfarrange --result=Concatenated.pdf *pdf

#  Concatenates all the pdf files in the current working directory
#+ into the merged file, Concatenated.pdf . . .
#  (The --pdfarrange option repaginates a pdf file. See also --pdfcombine.)
#  The above command-line could be parameterized and put into a shell script.
```

enscript

Utility for converting plain text file to PostScript

For example, **enscript filename.txt -p filename.ps** produces the PostScript output file `filename.ps`.

groff, **tbl**, **eqn**

Yet another text markup and display formatting language is **groff**. This is the enhanced GNU version of the venerable UNIX **roff/troff** display and typesetting package. Manpages use **groff**.

The **tbl** table processing utility is considered part of **groff**, as its function is to convert table markup into **groff** commands.

The **eqn** equation processing utility is likewise part of **groff**, and its function is to convert equation markup into **groff** commands.

Example 16-29. *manview*: Viewing formatted manpages

```
#!/bin/bash
# manview.sh: Formats the source of a man page for viewing.

#  This script is useful when writing man page source.
#  It lets you look at the intermediate results on the fly
#+ while working on it.

E_WRONGARGS=85

if [ -z "$1" ]
then
  echo "Usage: `basename $0` filename"
  exit $E_WRONGARGS
fi

# ---------------------------
groff -Tascii -man $1 | less
# From the man page for groff.
# ---------------------------

#  If the man page includes tables and/or equations,
#+ then the above code will barf.
#  The following line can handle such cases.
#
#   gtbl < "$1" | geqn -Tlatin1 | groff -Tlatin1 -mtty-char -man
#
#   Thanks, S.C.

exit $?   # See also the "maned.sh" script.
```

See also Example A-39.

lex, **yacc**

The **lex** lexical analyzer produces programs for pattern matching. This has been replaced by the nonproprietary **flex** on Linux systems.

The **yacc** utility creates a parser based on a set of specifications. This has been replaced by the nonproprietary **bison** on Linux systems.

16.5. File and Archiving Commands

Archiving

tar

The standard UNIX archiving utility. [75] Originally a *Tape ARchiving* program, it has developed into a general purpose package that can handle all manner of archiving with all types of destination devices, ranging from tape drives to regular files to even `stdout` (see Example 3-4). GNU *tar* has been patched to accept various compression filters, for example: **tar czvf archive_name.tar.gz ***, which recursively archives and gzips all files in a directory tree except dotfiles in the current working directory ($PWD). [76]

Some useful **tar** options:

1. `-c` create (a new archive)
2. `-x` extract (files from existing archive)
3. `--delete` delete (files from existing archive)

   This option will not work on magnetic tape devices.
4. `-r` append (files to existing archive)
5. `-A` append (*tar* files to existing archive)
6. `-t` list (contents of existing archive)
7. `-u` update archive
8. `-d` compare archive with specified filesystem
9. `--after-date` only process files with a date stamp *after* specified date
10. `-z` gzip the archive

    (compress or uncompress, depending on whether combined with the `-c` or `-x`) option
11. `-j` bzip2 the archive

It may be difficult to recover data from a corrupted *gzipped* tar archive. When archiving important files, make multiple backups.

shar

Shell archiving utility. The text and/or binary files in a shell archive are concatenated without compression, and the resultant archive is essentially a shell script, complete with #!/bin/sh header, containing all the necessary unarchiving commands, as well as the files themselves. Unprintable binary characters in the target file(s) are converted to printable ASCII characters in the output *shar* file. *Shar archives* still show up in Usenet newsgroups, but otherwise **shar** has been replaced by **tar/gzip**. The **unshar** command unpacks *shar* archives.

The **mailshar** command is a Bash script that uses **shar** to concatenate multiple files into a single one for e-mailing. This script supports compression and uuencoding.

ar

Creation and manipulation utility for archives, mainly used for binary object file libraries.

rpm

The *Red Hat Package Manager*, or **rpm** utility provides a wrapper for source or binary archives. It includes commands for installing and checking the integrity of packages, among other things.

A simple **rpm -i package_name.rpm** usually suffices to install a package, though there are many more options available.

`rpm -qf` identifies which package a file originates from.

```
bash$ rpm -qf /bin/ls
coreutils-5.2.1-31
```

`rpm -qa` gives a complete list of all installed *rpm* packages on a given system. An **`rpm -qa package_name`** lists only the package(s) corresponding to `package_name`.

```
bash$ rpm -qa
redhat-logos-1.1.3-1
 glibc-2.2.4-13
 cracklib-2.7-12
 dosfstools-2.7-1
 gdbm-1.8.0-10
 ksymoops-2.4.1-1
 mktemp-1.5-11
 perl-5.6.0-17
 reiserfs-utils-3.x.0j-2
 ...


bash$ rpm -qa docbook-utils
docbook-utils-0.6.9-2


bash$ rpm -qa docbook | grep docbook
docbook-dtd31-sgml-1.0-10
 docbook-style-dsssl-1.64-3
 docbook-dtd30-sgml-1.0-10
 docbook-dtd40-sgml-1.0-11
 docbook-utils-pdf-0.6.9-2
 docbook-dtd41-sgml-1.0-10
 docbook-utils-0.6.9-2
```

cpio

This specialized archiving copy command (**co**p**y** **i**nput and **o**utput) is rarely seen any more, having been supplanted by **tar/gzip**. It still has its uses, such as moving a directory tree. With an appropriate block size (for copying) specified, it can be appreciably faster than **tar**.

Example 16-30. Using *cpio* to move a directory tree

```
#!/bin/bash

# Copying a directory tree using cpio.

# Advantages of using 'cpio':
#   Speed of copying. It's faster than 'tar' with pipes.
#   Well suited for copying special files (named pipes, etc.)
#+  that 'cp' may choke on.

ARGS=2
E_BADARGS=65
```

```
if [ $# -ne "$ARGS" ]
then
  echo "Usage: `basename $0` source destination"
  exit $E_BADARGS
fi

source="$1"
destination="$2"

###################################################################
find "$source" -depth | cpio -admvp "$destination"
#               ^^^^^         ^^^^^
#  Read the 'find' and 'cpio' info pages to decipher these options.
#  The above works only relative to $PWD (current directory) . . .
#+ full pathnames are specified.
###################################################################


# Exercise:
# --------

#  Add code to check the exit status ($?) of the 'find | cpio' pipe
#+ and output appropriate error messages if anything went wrong.

exit $?
```

rpm2cpio

This command extracts a **cpio** archive from an rpm one.

Example 16-31. Unpacking an *rpm* archive

```
#!/bin/bash
# de-rpm.sh: Unpack an 'rpm' archive

: ${1?"Usage: `basename $0` target-file"}
# Must specify 'rpm' archive name as an argument.


TEMPFILE=$$.cpio                         #  Tempfile with "unique" name.
                                         #  $$ is process ID of script.

rpm2cpio < $1 > $TEMPFILE                #  Converts rpm archive into
                                         #+ cpio archive.
cpio --make-directories -F $TEMPFILE -i  #  Unpacks cpio archive.
rm -f $TEMPFILE                          #  Deletes cpio archive.

exit 0

#  Exercise:
#  Add check for whether 1) "target-file" exists and
#+                        2) it is an rpm archive.
#  Hint:                     Parse output of 'file' command.
```

pax

The *pax* **p**ortable **a**rchive e**x**change toolkit facilitates periodic file backups and is designed to be cross-compatible between various flavors of UNIX. It was designed to replace tar and cpio.

```
pax -wf daily_backup.pax ~/linux-server/files
#  Creates a tar archive of all files in the target directory.
```

```
#  Note that the options to pax must be in the correct order --
#+ pax -fw     has an entirely different effect.

pax -f daily_backup.pax
#  Lists the files in the archive.

pax -rf daily_backup.pax ~/bsd-server/files
#  Restores the backed-up files from the Linux machine
#+ onto a BSD one.
```

Note that *pax* handles many of the standard archiving and compression commands.

Compression

gzip

The standard GNU/UNIX compression utility, replacing the inferior and proprietary **compress**. The corresponding decompression command is **gunzip**, which is the equivalent of **gzip -d**.

The `-c` option sends the output of **gzip** to `stdout`. This is useful when piping to other commands.

The **zcat** filter decompresses a *gzipped* file to `stdout`, as possible input to a pipe or redirection. This is, in effect, a **cat** command that works on compressed files (including files processed with the older compress utility). The **zcat** command is equivalent to **gzip -dc**.

On some commercial UNIX systems, **zcat** is a synonym for **uncompress -c**, and will not work on *gzipped* files.

See also Example 7-7.

bzip2

An alternate compression utility, usually more efficient (but slower) than **gzip**, especially on large files. The corresponding decompression command is **bunzip2**.

Similar to the **zcat** command, **bzcat** decompresses a *bzipped2-ed* file to `stdout`.

Newer versions of tar have been patched with **bzip2** support.

compress, **uncompress**

This is an older, proprietary compression utility found in commercial UNIX distributions. The more efficient **gzip** has largely replaced it. Linux distributions generally include a **compress** workalike for compatibility, although **gunzip** can unarchive files treated with **compress**.

The **znew** command transforms *compressed* files into *gzipped* ones.

sq

Yet another compression (**sq**ueeze) utility, a filter that works only on sorted ASCII word lists. It uses the standard invocation syntax for a filter, **sq < input-file > output-file**. Fast, but not nearly as efficient as gzip. The corresponding uncompression filter is **unsq**, invoked like **sq**.

The output of **sq** may be piped to **gzip** for further compression.

zip, **unzip**

Cross-platform file archiving and compression utility compatible with DOS *pkzip.exe*. "Zipped" archives seem to be a more common medium of file exchange on the Internet than "tarballs."

unarc, **unarj**, **unrar**

These Linux utilities permit unpacking archives compressed with the DOS *arc.exe*, *arj.exe*, and *rar.exe* programs.

lzma, **unlzma**, **lzcat**

Highly efficient Lempel-Ziv-Markov compression. The syntax of *lzma* is similar to that of *gzip*. The 7-zip Website has more information.

xz, **unxz**, **xzcat**

A new high-efficiency compression tool, backward compatible with *lzma*, and with an invocation syntax similar to *gzip*. For more information, see the Wikipedia entry.

File Information

file

A utility for identifying file types. The command **`file file-name`** will return a file specification for `file-name`, such as `ascii text` or `data`. It references the magic numbers found in `/usr/share/magic`, `/etc/magic`, or `/usr/lib/magic`, depending on the Linux/UNIX distribution.

The `-f` option causes **file** to run in batch mode, to read from a designated file a list of filenames to analyze. The `-z` option, when used on a compressed target file, forces an attempt to analyze the uncompressed file type.

```
bash$ file test.tar.gz
test.tar.gz: gzip compressed data, deflated,
 last modified: Sun Sep 16 13:34:51 2001, os: Unix

bash file -z test.tar.gz
test.tar.gz: GNU tar archive (gzip compressed data, deflated,
 last modified: Sun Sep 16 13:34:51 2001, os: Unix)
```

```
# Find sh and Bash scripts in a given directory:

DIRECTORY=/usr/local/bin
KEYWORD=Bourne
# Bourne and Bourne-Again shell scripts

file $DIRECTORY/* | fgrep $KEYWORD

# Output:

# /usr/local/bin/burn-cd:          Bourne-Again shell script text executable
# /usr/local/bin/burnit:           Bourne-Again shell script text executable
# /usr/local/bin/cassette.sh:      Bourne shell script text executable
# /usr/local/bin/copy-cd:          Bourne-Again shell script text executable
# . . .
```

Example 16-32. Stripping comments from C program files

```
#!/bin/bash
# strip-comment.sh: Strips out the comments (/* COMMENT */) in a C program.

E_NOARGS=0
E_ARGERROR=66
E_WRONG_FILE_TYPE=67

if [ $# -eq "$E_NOARGS" ]
```

```
then
  echo "Usage: `basename $0` C-program-file" >&2 # Error message to stderr.
  exit $E_ARGERROR
fi

# Test for correct file type.
type=`file $1 | awk '{ print $2, $3, $4, $5 }'`
# "file $1" echoes file type . . .
# Then awk removes the first field, the filename . . .
# Then the result is fed into the variable "type."
correct_type="ASCII C program text"

if [ "$type" != "$correct_type" ]
then
  echo
  echo "This script works on C program files only."
  echo
  exit $E_WRONG_FILE_TYPE
fi


# Rather cryptic sed script:
#--------
sed '
/^\/\*/d
/.*\*\//d
' $1
#--------
# Easy to understand if you take several hours to learn sed fundamentals.


#  Need to add one more line to the sed script to deal with
#+ case where line of code has a comment following it on same line.
#  This is left as a non-trivial exercise.

#  Also, the above code deletes non-comment lines with a "*/" . . .
#+ not a desirable result.

exit 0


# ----------------------------------------------------------------
# Code below this line will not execute because of 'exit 0' above.

# Stephane Chazelas suggests the following alternative:

usage() {
  echo "Usage: `basename $0` C-program-file" >&2
  exit 1
}

WEIRD=`echo -n -e '\377'`   # or WEIRD=$'\377'
[[ $# -eq 1 ]] || usage
case `file "$1"` in
  *"C program text"*) sed -e "s%/\*%${WEIRD}%g;s%\*/%${WEIRD}%g" "$1" \
     | tr '\377\n' '\n\377' \
     | sed -ne 'p;n' \
     | tr -d '\n' | tr '\377' '\n';;
  *) usage;;
esac

#  This is still fooled by things like:
```

```
#  printf("/*");
#  or
#  /*  /* buggy embedded comment */
#
#  To handle all special cases (comments in strings, comments in string
#+ where there is a \", \\" ...),
#+ the only way is to write a C parser (using lex or yacc perhaps?).

exit 0
```

which

which command gives the full path to "command." This is useful for finding out whether a particular command or utility is installed on the system.

`$bash which rm`

```
/usr/bin/rm
```

For an interesting use of this command, see Example 36-16.

whereis

Similar to **which**, above, **whereis command** gives the full path to "command," but also to its manpage.

`$bash whereis rm`

```
rm: /bin/rm /usr/share/man/man1/rm.1.bz2
```

whatis

whatis command looks up "command" in the *`whatis`* database. This is useful for identifying system commands and important configuration files. Consider it a simplified **man** command.

`$bash whatis whatis`

```
whatis               (1)  - search the whatis database for complete words
```

Example 16-33. Exploring `/usr/X11R6/bin`

```
#!/bin/bash

# What are all those mysterious binaries in /usr/X11R6/bin?

DIRECTORY="/usr/X11R6/bin"
# Try also "/bin", "/usr/bin", "/usr/local/bin", etc.

for file in $DIRECTORY/*
do
  whatis `basename $file`   # Echoes info about the binary.
done

exit 0

#  Note: For this to work, you must create a "whatis" database
#+ with /usr/sbin/makewhatis.
#  You may wish to redirect output of this script, like so:
#    ./what.sh >>whatis.db
#  or view it a page at a time on stdout,
#    ./what.sh | less
```

See also Example 11-3.

vdir

Show a detailed directory listing. The effect is similar to ls -lb.

This is one of the GNU *fileutils*.

```
bash$ vdir
total 10
 -rw-r--r--    1 bozo  bozo      4034 Jul 18 22:04 data1.xrolo
 -rw-r--r--    1 bozo  bozo      4602 May 25 13:58 data1.xrolo.bak
 -rw-r--r--    1 bozo  bozo       877 Dec 17  2000 employment.xrolo

bash ls -l
total 10
 -rw-r--r--    1 bozo  bozo      4034 Jul 18 22:04 data1.xrolo
 -rw-r--r--    1 bozo  bozo      4602 May 25 13:58 data1.xrolo.bak
 -rw-r--r--    1 bozo  bozo       877 Dec 17  2000 employment.xrolo
```

locate, **slocate**

The **locate** command searches for files using a database stored for just that purpose. The **slocate** command is the secure version of **locate** (which may be aliased to **slocate**).

`$bash locate hickson`

```
/usr/lib/xephem/catalogs/hickson.edb
```

getfacl, **setfacl**

These commands *retrieve* or *set* the **f**ile **a**ccess **c**ontrol **l**ist -- the *owner*, *group*, and file permissions.

```
bash$ getfacl *
# file: test1.txt
 # owner: bozo
 # group: bozgrp
 user::rw-
 group::rw-
 other::r--

 # file: test2.txt
 # owner: bozo
 # group: bozgrp
 user::rw-
 group::rw-
 other::r--



bash$ setfacl -m u:bozo:rw yearly_budget.csv
bash$ getfacl yearly_budget.csv
# file: yearly_budget.csv
 # owner: accountant
 # group: budgetgrp
 user::rw-
 user:bozo:rw-
 user:accountant:rw-
 group::rw-
 mask::rw-
 other::r--
```

readlink

Disclose the file that a symbolic link points to.

```
bash$ readlink /usr/bin/awk
../../bin/gawk
```

strings

Use the **strings** command to find printable strings in a binary or data file. It will list sequences of printable characters found in the target file. This might be handy for a quick 'n dirty examination of a core dump or for looking at an unknown graphic image file (**`strings image-file | more`** might show something like *JFIF*, which would identify the file as a *jpeg* graphic). In a script, you would probably parse the output of **strings** with grep or sed. See Example 11-8 and Example 11-10.

Example 16-34. An "improved" *strings* command

```
#!/bin/bash
# wstrings.sh: "word-strings" (enhanced "strings" command)
#
#  This script filters the output of "strings" by checking it
#+ against a standard word list file.
#  This effectively eliminates gibberish and noise,
#+ and outputs only recognized words.

# ===========================================================
#                 Standard Check for Script Argument(s)
ARGS=1
E_BADARGS=85
E_NOFILE=86

if [ $# -ne $ARGS ]
then
  echo "Usage: `basename $0` filename"
  exit $E_BADARGS
fi

if [ ! -f "$1" ]                      # Check if file exists.
then
    echo "File \"$1\" does not exist."
    exit $E_NOFILE
fi
# ===========================================================


MINSTRLEN=3                           #  Minimum string length.
WORDFILE=/usr/share/dict/linux.words  #  Dictionary file.
#  May specify a different word list file
#+ of one-word-per-line format.
#  For example, the "yawl" word-list package,
#  http://bash.deta.in/yawl-0.3.2.tar.gz


wlist=`strings "$1" | tr A-Z a-z | tr '[:space:]' Z | \
       tr -cs '[:alpha:]' Z | tr -s '\173-\377' Z | tr Z ' '`

# Translate output of 'strings' command with multiple passes of 'tr'.
#  "tr A-Z a-z"  converts to lowercase.
#  "tr '[:space:]'"  converts whitespace characters to Z's.
#  "tr -cs '[:alpha:]' Z"  converts non-alphabetic characters to Z's,
#+ and squeezes multiple consecutive Z's.
#  "tr -s '\173-\377' Z"  converts all characters past 'z' to Z's
#+ and squeezes multiple consecutive Z's,
#+ which gets rid of all the weird characters that the previous
```

```
#+ translation failed to deal with.
#  Finally, "tr Z ' '" converts all those Z's to whitespace,
#+ which will be seen as word separators in the loop below.

#  ***********************************************************************
#  Note the technique of feeding/piping the output of 'tr' back to itself,
#+ but with different arguments and/or options on each successive pass.
#  ***********************************************************************


for word in $wlist                     #  Important:
                                       #  $wlist must not be quoted here.
                                       # "$wlist" does not work.
                                       #  Why not?
do
  strlen=${#word}                      #  String length.
  if [ "$strlen" -lt "$MINSTRLEN" ]    #  Skip over short strings.
  then
    continue
  fi

  grep -Fw $word "$WORDFILE"           #   Match whole words only.
#      ^^^                             #  "Fixed strings" and
                                       #+ "whole words" options.
done

exit $?
```

Comparison

diff, **patch**

diff: flexible file comparison utility. It compares the target files line-by-line sequentially. In some applications, such as comparing word dictionaries, it may be helpful to filter the files through sort and **uniq** before piping them to **diff**. **`diff file-1 file-2`** outputs the lines in the files that differ, with carets showing which file each particular line belongs to.

The `--side-by-side` option to **diff** outputs each compared file, line by line, in separate columns, with non-matching lines marked. The `-c` and `-u` options likewise make the output of the command easier to interpret.

There are available various fancy frontends for **diff**, such as **sdiff**, **wdiff**, **xdiff**, and **mgdiff**.

The **diff** command returns an exit status of 0 if the compared files are identical, and 1 if they differ (or 2 when *binary* files are being compared). This permits use of **diff** in a test construct within a shell script (see below).

A common use for **diff** is generating difference files to be used with **patch** The `-e` option outputs files suitable for **ed** or **ex** scripts.

patch: flexible versioning utility. Given a difference file generated by **diff**, **patch** can upgrade a previous version of a package to a newer version. It is much more convenient to distribute a relatively small "diff" file than the entire body of a newly revised package. Kernel "patches" have become the preferred method of distributing the frequent releases of the Linux kernel.

```
patch -p1 <patch-file
```

```
# Takes all the changes listed in 'patch-file'
# and applies them to the files referenced therein.
# This upgrades to a newer version of the package.
```

Patching the kernel:

```
cd /usr/src
gzip -cd patchXX.gz | patch -p0
# Upgrading kernel source using 'patch'.
# From the Linux kernel docs "README",
# by anonymous author (Alan Cox?).
```

The **diff** command can also recursively compare directories (for the filenames present).

```
bash$ diff -r ~/notes1 ~/notes2
Only in /home/bozo/notes1: file02
 Only in /home/bozo/notes1: file03
 Only in /home/bozo/notes2: file04
```

Use **zdiff** to compare *gzipped* files.

Use **diffstat** to create a histogram (point-distribution graph) of output from **diff**.

diff3, **merge**

An extended version of **diff** that compares three files at a time. This command returns an exit value of 0 upon successful execution, but unfortunately this gives no information about the results of the comparison.

```
bash$ diff3 file-1 file-2 file-3
====
 1:1c
   This is line 1 of "file-1".
 2:1c
   This is line 1 of "file-2".
 3:1c
   This is line 1 of "file-3"
```

The **merge** (3-way file merge) command is an interesting adjunct to *diff3*. Its syntax is **`merge Mergefile file1 file2`**. The result is to output to `Mergefile` the changes that lead from `file1` to `file2`. Consider this command a stripped-down version of *patch*.

sdiff

Compare and/or edit two files in order to merge them into an output file. Because of its interactive nature, this command would find little use in a script.

cmp

The **cmp** command is a simpler version of **diff**, above. Whereas **diff** reports the differences between two files, **cmp** merely shows at what point they differ.

Like **diff**, **cmp** returns an exit status of 0 if the compared files are identical, and 1 if they differ. This permits use in a test construct within a shell script.

Example 16-35. Using *cmp* to compare two files within a script.

```
#!/bin/bash
# file-comparison.sh

ARGS=2  # Two args to script expected.
E_BADARGS=85
E_UNREADABLE=86

if [ $# -ne "$ARGS" ]
then
  echo "Usage: `basename $0` file1 file2"
  exit $E_BADARGS
fi

if [[ ! -r "$1" || ! -r "$2" ]]
then
  echo "Both files to be compared must exist and be readable."
  exit $E_UNREADABLE
fi

cmp $1 $2 &> /dev/null
#   Redirection to /dev/null buries the output of the "cmp" command.
#   cmp -s $1 $2  has same result ("-s" silent flag to "cmp")
#   Thank you  Anders Gustavsson for pointing this out.
#
#  Also works with 'diff', i.e.,
#+ diff $1 $2 &> /dev/null

if [ $? -eq 0 ]         # Test exit status of "cmp" command.
then
  echo "File \"$1\" is identical to file \"$2\"."
else
  echo "File \"$1\" differs from file \"$2\"."
fi

exit 0
```

Use **zcmp** on *gzipped* files.

comm

Versatile file comparison utility. The files must be sorted for this to be useful.

comm *`-options`* *`first-file`* *`second-file`*

`comm file-1 file-2` outputs three columns:

- ◊ column 1 = lines unique to `file-1`
- ◊ column 2 = lines unique to `file-2`
- ◊ column 3 = lines common to both.

The options allow suppressing output of one or more columns.

- ◊ `-1` suppresses column `1`
- ◊ `-2` suppresses column `2`
- ◊ `-3` suppresses column `3`
- ◊ `-12` suppresses both columns `1` and `2`, etc.

This command is useful for comparing "dictionaries" or *word lists* -- sorted text files with one word per line.

Utilities

basename

Strips the path information from a file name, printing only the file name. The construction **`basename $0`** lets the script know its name, that is, the name it was invoked by. This can be used for "usage" messages if, for example a script is called with missing arguments:

```
echo "Usage: `basename $0` arg1 arg2 ... argn"
```

dirname

Strips the **basename** from a filename, printing only the path information.

basename and **dirname** can operate on any arbitrary string. The argument does not need to refer to an existing file, or even be a filename for that matter (see Example A-7).

Example 16-36. *basename* and *dirname*

```
#!/bin/bash

address=/home/bozo/daily-journal.txt

echo "Basename of /home/bozo/daily-journal.txt = `basename $address`"
echo "Dirname of /home/bozo/daily-journal.txt = `dirname $address`"
echo
echo "My own home is `basename ~/`."         # `basename ~` also works.
echo "The home of my home is `dirname ~/`."  # `dirname ~`  also works.

exit 0
```

split, **csplit**

These are utilities for splitting a file into smaller chunks. Their usual use is for splitting up large files in order to back them up on floppies or preparatory to e-mailing or uploading them.

The **csplit** command splits a file according to *context*, the split occuring where patterns are matched.

Example 16-37. A script that copies itself in sections

```
#!/bin/bash
# splitcopy.sh

#  A script that splits itself into chunks,
#+ then reassembles the chunks into an exact copy
#+ of the original script.

CHUNKSIZE=4    #  Size of first chunk of split files.
OUTPREFIX=xx   #  csplit prefixes, by default,
               #+ files with "xx" ...

csplit "$0" "$CHUNKSIZE"

# Some comment lines for padding . . .
# Line 15
# Line 16
# Line 17
# Line 18
# Line 19
```

```
# Line 20

cat "$OUTPREFIX"* > "$0.copy"  # Concatenate the chunks.
rm "$OUTPREFIX"*               # Get rid of the chunks.

exit $?
```

Encoding and Encryption

sum, **cksum**, **md5sum**, **sha1sum**

These are utilities for generating *checksums*. A *checksum* is a number [77] mathematically calculated from the contents of a file, for the purpose of checking its integrity. A script might refer to a list of checksums for security purposes, such as ensuring that the contents of key system files have not been altered or corrupted. For security applications, use the **md5sum** (**m**essage **d**igest **5** check**sum**) command, or better yet, the newer **sha1sum** (Secure Hash Algorithm). [78]

```
bash$ cksum /boot/vmlinuz
1670054224 804083 /boot/vmlinuz

bash$ echo -n "Top Secret" | cksum
3391003827 10



bash$ md5sum /boot/vmlinuz
0f43eccea8f09e0a0b2b5cf1dcf333ba  /boot/vmlinuz

bash$ echo -n "Top Secret" | md5sum
8babc97a6f62a4649716f4df8d61728f  -
```

The **cksum** command shows the size, in bytes, of its target, whether file or `stdout`.

The **md5sum** and **sha1sum** commands display a dash when they receive their input from `stdout`.

Example 16-38. Checking file integrity

```
#!/bin/bash
# file-integrity.sh: Checking whether files in a given directory
#                    have been tampered with.

E_DIR_NOMATCH=80
E_BAD_DBFILE=81

dbfile=File_record.md5
# Filename for storing records (database file).


set_up_database ()
{
  echo ""$directory"" > "$dbfile"
  # Write directory name to first line of file.
  md5sum "$directory"/* >> "$dbfile"
  # Append md5 checksums and filenames.
}
```

```
check_database ()
{
  local n=0
  local filename
  local checksum

  # ------------------------------------------- #
  #  This file check should be unnecessary,
  #+ but better safe than sorry.

  if [ ! -r "$dbfile" ]
  then
    echo "Unable to read checksum database file!"
    exit $E_BAD_DBFILE
  fi
  # ------------------------------------------- #

  while read record[n]
  do

    directory_checked="${record[0]}"
    if [ "$directory_checked" != "$directory" ]
    then
      echo "Directories do not match up!"
      # Tried to use file for a different directory.
      exit $E_DIR_NOMATCH
    fi

    if [ "$n" -gt 0 ]   # Not directory name.
    then
      filename[n]=$( echo ${record[$n]} | awk '{ print $2 }' )
      #  md5sum writes records backwards,
      #+ checksum first, then filename.
      checksum[n]=$( md5sum "${filename[n]}" )


      if [ "${record[n]}" = "${checksum[n]}" ]
      then
        echo "${filename[n]} unchanged."

        elif [ "`basename ${filename[n]}`" != "$dbfile" ]
               #  Skip over checksum database file,
               #+ as it will change with each invocation of script.
               #  ---
               #  This unfortunately means that when running
               #+ this script on $PWD, tampering with the
               #+ checksum database file will not be detected.
               #  Exercise: Fix this.
        then
          echo "${filename[n]} : CHECKSUM ERROR!"
        # File has been changed since last checked.
        fi

      fi



    let "n+=1"
  done <"$dbfile"       # Read from checksum database file.

}
```

```
# ================================================== #
# main ()

if [ -z  "$1" ]
then
  directory="$PWD"      #  If not specified,
else                    #+ use current working directory.
  directory="$1"
fi

clear                   # Clear screen.
echo " Running file integrity check on $directory"
echo

# ------------------------------------------------------------------ #
  if [ ! -r "$dbfile" ] # Need to create database file?
  then
    echo "Setting up database file, \""$directory"/"$dbfile"\"."; echo
    set_up_database
  fi
# ------------------------------------------------------------------ #

check_database          # Do the actual work.

echo

#  You may wish to redirect the stdout of this script to a file,
#+ especially if the directory checked has many files in it.

exit 0

#  For a much more thorough file integrity check,
#+ consider the "Tripwire" package,
#+ http://sourceforge.net/projects/tripwire/.
```

Also see Example A-19, Example 36-16, and Example 10-2 for creative uses of the **md5sum** command.

There have been reports that the 128-bit **md5sum** can be cracked, so the more secure 160-bit **sha1sum** is a welcome new addition to the checksum toolkit.

```
bash$ md5sum testfile
e181e2c8720c60522c4c4c981108e367  testfile


bash$ sha1sum testfile
5d7425a9c08a66c3177f1e31286fa40986ffc996  testfile
```

Security consultants have demonstrated that even **sha1sum** can be compromised. Fortunately, newer Linux distros include longer bit-length **sha224sum**, **sha256sum**, **sha384sum**, and **sha512sum** commands.

uuencode

This utility encodes binary files (images, sound files, compressed files, etc.) into ASCII characters, making them suitable for transmission in the body of an e-mail message or in a newsgroup posting. This is especially useful where MIME (multimedia) encoding is not available.

uudecode

This reverses the encoding, decoding *uuencoded* files back into the original binaries.

Example 16-39. Uudecoding encoded files

```
#!/bin/bash
# Uudecodes all uuencoded files in current working directory.

lines=35        # Allow 35 lines for the header (very generous).

for File in *   # Test all the files in $PWD.
do
  search1=`head -n $lines $File | grep begin | wc -w`
  search2=`tail -n $lines $File | grep end | wc -w`
  #  Uuencoded files have a "begin" near the beginning,
  #+ and an "end" near the end.
  if [ "$search1" -gt 0 ]
  then
    if [ "$search2" -gt 0 ]
    then
      echo "uudecoding - $File -"
      uudecode $File
    fi
  fi
done

#  Note that running this script upon itself fools it
#+ into thinking it is a uuencoded file,
#+ because it contains both "begin" and "end".

#  Exercise:
#  --------
#  Modify this script to check each file for a newsgroup header,
#+ and skip to next if not found.

exit 0
```

The fold -s command may be useful (possibly in a pipe) to process long uudecoded text messages downloaded from Usenet newsgroups.

mimencode, **mmencode**

The **mimencode** and **mmencode** commands process multimedia-encoded e-mail attachments. Although *mail user agents* (such as *pine* or *kmail*) normally handle this automatically, these particular utilities permit manipulating such attachments manually from the command-line or in batch processing mode by means of a shell script.

crypt

At one time, this was the standard UNIX file encryption utility. [79] Politically-motivated government regulations prohibiting the export of encryption software resulted in the disappearance of **crypt** from much of the UNIX world, and it is still missing from most Linux distributions. Fortunately, programmers have come up with a number of decent alternatives to it, among them the author's very own cruft (see Example A-4).

openssl

This is an Open Source implementation of *Secure Sockets Layer* encryption.

```
# To encrypt a file:
openssl aes-128-ecb -salt -in file.txt -out file.encrypted \
-pass pass:my_password
#          ^^^^^^^^^^^   User-selected password.
#       aes-128-ecb      is the encryption method chosen.

# To decrypt an openssl-encrypted file:
```

```
openssl aes-128-ecb -d -salt -in file.encrypted -out file.txt \
-pass pass:my_password
#          ^^^^^^^^^^^   User-selected password.
```

Piping *openssl* to/from tar makes it possible to encrypt an entire directory tree.

```
# To encrypt a directory:

sourcedir="/home/bozo/testfiles"
encrfile="encr-dir.tar.gz"
password=my_secret_password

tar czvf - "$sourcedir" |
openssl des3 -salt -out "$encrfile" -pass pass:"$password"
#       ^^^^   Uses des3 encryption.
# Writes encrypted file "encr-dir.tar.gz" in current working directory.

# To decrypt the resulting tarball:
openssl des3 -d -salt -in "$encrfile" -pass pass:"$password" |
tar -xzv
# Decrypts and unpacks into current working directory.
```

Of course, *openssl* has many other uses, such as obtaining signed *certificates* for Web sites. See the info page.

shred

Securely erase a file by overwriting it multiple times with random bit patterns before deleting it. This command has the same effect as Example 16-61, but does it in a more thorough and elegant manner.

This is one of the GNU *fileutils*.

Advanced forensic technology may still be able to recover the contents of a file, even after application of **shred**.

Miscellaneous

mktemp

Create a *temporary file* [80] with a "unique" filename. When invoked from the command-line without additional arguments, it creates a zero-length file in the /tmp directory.

```
bash$ mktemp
/tmp/tmp.zzsvql3154
```

```
PREFIX=filename
tempfile=`mktemp $PREFIX.XXXXXX`
#                        ^^^^^^ Need at least 6 placeholders
#+                              in the filename template.
#   If no filename template supplied,
#+ "tmp.XXXXXXXXXX" is the default.

echo "tempfile name = $tempfile"
# tempfile name = filename.QA2ZpY
#                 or something similar...

#  Creates a file of that name in the current working directory
#+ with 600 file permissions.
#  A "umask 177" is therefore unnecessary,
#+ but it's good programming practice nevertheless.
```

make

Utility for building and compiling binary packages. This can also be used for any set of operations triggered by incremental changes in source files.

The *make* command checks a `Makefile`, a list of file dependencies and operations to be carried out.

The *make* utility is, in effect, a powerful scripting language similar in many ways to *Bash*, but with the capability of recognizing *dependencies*. For in-depth coverage of this useful tool set, see the GNU software documentation site.

install

Special purpose file copying command, similar to cp, but capable of setting permissions and attributes of the copied files. This command seems tailormade for installing software packages, and as such it shows up frequently in `Makefiles` (in the *`make install :`* section). It could likewise prove useful in installation scripts.

dos2unix

This utility, written by Benjamin Lin and collaborators, converts DOS-formatted text files (lines terminated by CR-LF) to UNIX format (lines terminated by LF only), and vice-versa.

ptx

The **ptx [targetfile]** command outputs a permuted index (cross-reference list) of the targetfile. This may be further filtered and formatted in a pipe, if necessary.

more, **less**

Pagers that display a text file or stream to `stdout`, one screenful at a time. These may be used to filter the output of `stdout` . . . or of a script.

An interesting application of *more* is to "test drive" a command sequence, to forestall potentially unpleasant consequences.

```
ls /home/bozo | awk '{print "rm -rf " $1}' | more
#                                            ^^^^

# Testing the effect of the following (disastrous) command-line:
#      ls /home/bozo | awk '{print "rm -rf " $1}' | sh
#      Hand off to the shell to execute . . .       ^^
```

The *less* pager has the interesting property of doing a formatted display of *man page* source. See Example A-39.

16.6. Communications Commands

Certain of the following commands find use in network data transfer and analysis, as well as in chasing spammers.

Information and Statistics

host

Searches for information about an Internet host by name or IP address, using DNS.

```
bash$ host surfacemail.com
surfacemail.com. has address 202.92.42.236
```

ipcalc

Displays IP information for a host. With the `-h` option, **ipcalc** does a reverse DNS lookup, finding the name of the host (server) from the IP address.

```
bash$ ipcalc -h 202.92.42.236
HOSTNAME=surfacemail.com
```

nslookup

Do an Internet "name server lookup" on a host by IP address. This is essentially equivalent to **ipcalc -h** or **dig -x** . The command may be run either interactively or noninteractively, i.e., from within a script.

The **nslookup** command has allegedly been "deprecated," but it is still useful.

```
bash$ nslookup -sil 66.97.104.180
nslookup kuhleersparnis.ch
 Server:         135.116.137.2
 Address:        135.116.137.2#53

 Non-authoritative answer:
 Name:   kuhleersparnis.ch
```

dig

Domain **I**nformation **G**roper. Similar to **nslookup**, *dig* does an Internet *name server lookup* on a host. May be run from the command-line or from within a script.

Some interesting options to *dig* are `+time=N` for setting a query timeout to *N* seconds, `+nofail` for continuing to query servers until a reply is received, and `-x` for doing a reverse address lookup.

Compare the output of **dig -x** with **ipcalc -h** and **nslookup**.

```
bash$ dig -x 81.9.6.2
;; Got answer:
 ;; ->>HEADER<<- opcode: QUERY, status: NXDOMAIN, id: 11649
 ;; flags: qr rd ra; QUERY: 1, ANSWER: 0, AUTHORITY: 1, ADDITIONAL: 0

 ;; QUESTION SECTION:
 ;2.6.9.81.in-addr.arpa.         IN      PTR

 ;; AUTHORITY SECTION:
 6.9.81.in-addr.arpa.    3600    IN      SOA     ns.eltel.net. noc.eltel.net.
 2002031705 900 600 86400 3600

 ;; Query time: 537 msec
 ;; SERVER: 135.116.137.2#53(135.116.137.2)
 ;; WHEN: Wed Jun 26 08:35:24 2002
 ;; MSG SIZE  rcvd: 91
```

Example 16-40. Finding out where to report a spammer

```
#!/bin/bash
# spam-lookup.sh: Look up abuse contact to report a spammer.
# Thanks, Michael Zick.

# Check for command-line arg.
```

```
ARGCOUNT=1
E_WRONGARGS=85
if [ $# -ne "$ARGCOUNT" ]
then
  echo "Usage: `basename $0` domain-name"
  exit $E_WRONGARGS
fi


dig +short $1.contacts.abuse.net -c in -t txt
# Also try:
#     dig +nssearch $1
#     Tries to find "authoritative name servers" and display SOA records.

# The following also works:
#     whois -h whois.abuse.net $1
#           ^^ ^^^^^^^^^^^^^^^  Specify host.
#     Can even lookup multiple spammers with this, i.e."
#     whois -h whois.abuse.net $spamdomain1 $spamdomain2 . . .


#  Exercise:
#  --------
#  Expand the functionality of this script
#+ so that it automatically e-mails a notification
#+ to the responsible ISP's contact address(es).
#  Hint: use the "mail" command.

exit $?

# spam-lookup.sh chinatietong.com
#                A known spam domain.

# "crnet_mgr@chinatietong.com"
# "crnet_tec@chinatietong.com"
# "postmaster@chinatietong.com"


#  For a more elaborate version of this script,
#+ see the SpamViz home page, http://www.spamviz.net/index.html.
```

Example 16-41. Analyzing a spam domain

```
#! /bin/bash
# is-spammer.sh: Identifying spam domains

# $Id: is-spammer, v 1.4 2004/09/01 19:37:52 mszick Exp $
# Above line is RCS ID info.
#
#  This is a simplified version of the "is_spammer.bash
#+ script in the Contributed Scripts appendix.

# is-spammer <domain.name>

# Uses an external program: 'dig'
# Tested with version: 9.2.4rc5

# Uses functions.
# Uses IFS to parse strings by assignment into arrays.
# And even does something useful: checks e-mail blacklists.
```

```
# Use the domain.name(s) from the text body:
# http://www.good_stuff.spammer.biz/just_ignore_everything_else
#                        ^^^^^^^^^^^
# Or the domain.name(s) from any e-mail address:
# Really_Good_Offer@spammer.biz
#
# as the only argument to this script.
#(PS: have your Inet connection running)
#
# So, to invoke this script in the above two instances:
#       is-spammer.sh spammer.biz


# Whitespace == :Space:Tab:Line Feed:Carriage Return:
WSP_IFS=$'\x20'$'\x09'$'\x0A'$'\x0D'

# No Whitespace == Line Feed:Carriage Return
No_WSP=$'\x0A'$'\x0D'

# Field separator for dotted decimal ip addresses
ADR_IFS=${No_WSP}'.'

# Get the dns text resource record.
# get_txt <error_code> <list_query>
get_txt() {

    # Parse $1 by assignment at the dots.
    local -a dns
    IFS=$ADR_IFS
    dns=( $1 )
    IFS=$WSP_IFS
    if [ "${dns[0]}" == '127' ]
    then
        # See if there is a reason.
        echo $(dig +short $2 -t txt)
    fi
}

# Get the dns address resource record.
# chk_adr <rev_dns> <list_server>
chk_adr() {
    local reply
    local server
    local reason

    server=${1}${2}
    reply=$( dig +short ${server} )

    # If reply might be an error code . . .
    if [ ${#reply} -gt 6 ]
    then
        reason=$(get_txt ${reply} ${server} )
        reason=${reason:-${reply}}
    fi
    echo ${reason:-' not blacklisted.'}
}

# Need to get the IP address from the name.
echo 'Get address of: '$1
ip_adr=$(dig +short $1)
dns_reply=${ip_adr:-' no answer '}
```

```
echo ' Found address: '${dns_reply}

# A valid reply is at least 4 digits plus 3 dots.
if [ ${#ip_adr} -gt 6 ]
then
    echo
    declare query

    # Parse by assignment at the dots.
    declare -a dns
    IFS=$ADR_IFS
    dns=( ${ip_adr} )
    IFS=$WSP_IFS

    # Reorder octets into dns query order.
    rev_dns="${dns[3]}"'.'"${dns[2]}"'.'"${dns[1]}"'.'"${dns[0]}"'.'

# See: http://www.spamhaus.org (Conservative, well maintained)
    echo -n 'spamhaus.org says: '
    echo $(chk_adr ${rev_dns} 'sbl-xbl.spamhaus.org')

# See: http://ordb.org (Open mail relays)
    echo -n '   ordb.org  says: '
    echo $(chk_adr ${rev_dns} 'relays.ordb.org')

# See: http://www.spamcop.net/ (You can report spammers here)
    echo -n ' spamcop.net says: '
    echo $(chk_adr ${rev_dns} 'bl.spamcop.net')

# # # other blacklist operations # # #

# See: http://cbl.abuseat.org.
    echo -n ' abuseat.org says: '
    echo $(chk_adr ${rev_dns} 'cbl.abuseat.org')

# See: http://dsbl.org/usage (Various mail relays)
    echo
    echo 'Distributed Server Listings'
    echo -n '       list.dsbl.org says: '
    echo $(chk_adr ${rev_dns} 'list.dsbl.org')

    echo -n '   multihop.dsbl.org says: '
    echo $(chk_adr ${rev_dns} 'multihop.dsbl.org')

    echo -n 'unconfirmed.dsbl.org says: '
    echo $(chk_adr ${rev_dns} 'unconfirmed.dsbl.org')

else
    echo
    echo 'Could not use that address.'
fi

exit 0

# Exercises:
# --------

# 1) Check arguments to script,
#    and exit with appropriate error message if necessary.

# 2) Check if on-line at invocation of script,
#    and exit with appropriate error message if necessary.
```

```
# 3) Substitute generic variables for "hard-coded" BHL domains.

# 4) Set a time-out for the script using the "+time=" option
     to the 'dig' command.
```

For a much more elaborate version of the above script, see Example A-28.

traceroute

Trace the route taken by packets sent to a remote host. This command works within a LAN, WAN, or over the Internet. The remote host may be specified by an IP address. The output of this command may be filtered by grep or sed in a pipe.

```
bash$ traceroute 81.9.6.2
traceroute to 81.9.6.2 (81.9.6.2), 30 hops max, 38 byte packets
 1  tc43.xjbnnbrb.com (136.30.178.8)  191.303 ms  179.400 ms  179.767 ms
 2  or0.xjbnnbrb.com (136.30.178.1)  179.536 ms  179.534 ms  169.685 ms
 3  192.168.11.101 (192.168.11.101)  189.471 ms  189.556 ms *
 ...
```

ping

Broadcast an `ICMP ECHO_REQUEST` packet to another machine, either on a local or remote network. This is a diagnostic tool for testing network connections, and it should be used with caution.

```
bash$ ping localhost
PING localhost.localdomain (127.0.0.1) from 127.0.0.1 : 56(84) bytes of data.
 64 bytes from localhost.localdomain (127.0.0.1): icmp_seq=0 ttl=255 time=709 usec
 64 bytes from localhost.localdomain (127.0.0.1): icmp_seq=1 ttl=255 time=286 usec

 --- localhost.localdomain ping statistics ---
 2 packets transmitted, 2 packets received, 0% packet loss
 round-trip min/avg/max/mdev = 0.286/0.497/0.709/0.212 ms
```

A successful *ping* returns an exit status of 0. This can be tested for in a script.

```
  HNAME=news-15.net  # Notorious spammer.
# HNAME=$HOST     # Debug: test for localhost.
  count=2  # Send only two pings.

if [[ `ping -c $count "$HNAME"` ]]
then
  echo ""$HNAME" still up and broadcasting spam your way."
else
  echo ""$HNAME" seems to be down. Pity."
fi
```

whois

Perform a DNS (Domain Name System) lookup. The `-h` option permits specifying which particular *whois* server to query. See Example 4-6 and Example 16-40.

finger

Retrieve information about users on a network. Optionally, this command can display a user's `~/.plan`, `~/.project`, and `~/.forward` files, if present.

```
bash$ finger
Login  Name           Tty      Idle  Login Time   Office     Office Phone
 bozo   Bozo Bozeman   tty1        8  Jun 25 16:59                (:0)
 bozo   Bozo Bozeman   ttyp0          Jun 25 16:59                (:0.0)
 bozo   Bozo Bozeman   ttyp1          Jun 25 17:07                (:0.0)
```

```
bash$ finger bozo
Login: bozo                              Name: Bozo Bozeman
 Directory: /home/bozo                    Shell: /bin/bash
 Office: 2355 Clown St., 543-1234
 On since Fri Aug 31 20:13 (MST) on tty1    1 hour 38 minutes idle
 On since Fri Aug 31 20:13 (MST) on pts/0   12 seconds idle
 On since Fri Aug 31 20:13 (MST) on pts/1
 On since Fri Aug 31 20:31 (MST) on pts/2   1 hour 16 minutes idle
 Mail last read Tue Jul  3 10:08 2007 (MST)
 No Plan.
```

Out of security considerations, many networks disable **finger** and its associated daemon. [81]

chfn

Change information disclosed by the **finger** command.

vrfy

Verify an Internet e-mail address.

This command seems to be missing from newer Linux distros.

Remote Host Access

sx, **rx**

The **sx** and **rx** command set serves to transfer files to and from a remote host using the *xmodem* protocol. These are generally part of a communications package, such as **minicom**.

sz, **rz**

The **sz** and **rz** command set serves to transfer files to and from a remote host using the *zmodem* protocol. *Zmodem* has certain advantages over *xmodem*, such as faster transmission rate and resumption of interrupted file transfers. Like **sx** and **rx**, these are generally part of a communications package.

ftp

Utility and protocol for uploading / downloading files to or from a remote host. An ftp session can be automated in a script (see Example 19-6 and Example A-4).

uucp, **uux**, **cu**

uucp: *UNIX to UNIX copy*. This is a communications package for transferring files between UNIX servers. A shell script is an effective way to handle a **uucp** command sequence.

Since the advent of the Internet and e-mail, **uucp** seems to have faded into obscurity, but it still exists and remains perfectly workable in situations where an Internet connection is not available or appropriate. The advantage of **uucp** is that it is fault-tolerant, so even if there is a service interruption the copy operation will resume where it left off when the connection is restored.

---

uux: *UNIX to UNIX execute*. Execute a command on a remote system. This command is part of the **uucp** package.

---

cu: **C**all **U**p a remote system and connect as a simple terminal. It is a sort of dumbed-down version of telnet. This command is part of the **uucp** package.

telnet

Utility and protocol for connecting to a remote host.

The *telnet* protocol contains security holes and should therefore probably be avoided. Its use within a shell script is *not* recommended.

wget

The **wget** utility *noninteractively* retrieves or downloads files from a Web or ftp site. It works well in a script.

```
wget -p http://www.xyz23.com/file01.html
#  The -p or --page-requisite option causes wget to fetch all files
#+ required to display the specified page.

wget -r ftp://ftp.xyz24.net/~bozo/project_files/ -O $SAVEFILE
#  The -r option recursively follows and retrieves all links
#+ on the specified site.

wget -c ftp://ftp.xyz25.net/bozofiles/filename.tar.bz2
#  The -c option lets wget resume an interrupted download.
#  This works with ftp servers and many HTTP sites.
```

Example 16-42. Getting a stock quote

```
#!/bin/bash
# quote-fetch.sh: Download a stock quote.


E_NOPARAMS=86

if [ -z "$1" ]  # Must specify a stock (symbol) to fetch.
  then echo "Usage: `basename $0` stock-symbol"
  exit $E_NOPARAMS
fi

stock_symbol=$1

file_suffix=.html
# Fetches an HTML file, so name it appropriately.
URL='http://finance.yahoo.com/q?s='
# Yahoo finance board, with stock query suffix.

# -----------------------------------------------------------
wget -O ${stock_symbol}${file_suffix} "${URL}${stock_symbol}"
# -----------------------------------------------------------


# To look up stuff on http://search.yahoo.com:
# -----------------------------------------------------------
# URL="http://search.yahoo.com/search?fr=ush-news&p=${query}"
# wget -O "$savefilename" "${URL}"
# -----------------------------------------------------------
# Saves a list of relevant URLs.

exit $?

# Exercises:
# ---------
#
# 1) Add a test to ensure the user running the script is on-line.
```

```
#    (Hint: parse the output of 'ps -ax' for "ppp" or "connect."
#
# 2) Modify this script to fetch the local weather report,
#+   taking the user's zip code as an argument.
```

See also Example A-30 and Example A-31.

lynx

The **lynx** Web and file browser can be used inside a script (with the `-dump` option) to retrieve a file from a Web or ftp site noninteractively.

```
lynx -dump http://www.xyz23.com/file01.html >$SAVEFILE
```

With the `-traversal` option, **lynx** starts at the HTTP URL specified as an argument, then "crawls" through all links located on that particular server. Used together with the `-crawl` option, outputs page text to a log file.

rlogin

Remote login, initates a session on a remote host. This command has security issues, so use ssh instead.

rsh

Remote shell, executes command(s) on a remote host. This has security issues, so use **ssh** instead.

rcp

Remote copy, copies files between two different networked machines.

rsync

Remote synchronize, updates (synchronizes) files between two different networked machines.

```
bash$ rsync -a ~/sourcedir/*txt /node1/subdirectory/
```

Example 16-43. Updating FC4

```
#!/bin/bash
# fc4upd.sh

# Script author: Frank Wang.
# Slight stylistic modifications by ABS Guide author.
# Used in ABS Guide with permission.


#  Download Fedora Core 4 update from mirror site using rsync.
#  Should also work for newer Fedora Cores -- 5, 6, . . .
#  Only download latest package if multiple versions exist,
#+ to save space.

URL=rsync://distro.ibiblio.org/fedora-linux-core/updates/
# URL=rsync://ftp.kddilabs.jp/fedora/core/updates/
# URL=rsync://rsync.planetmirror.com/fedora-linux-core/updates/

DEST=${1:-/var/www/html/fedora/updates/}
LOG=/tmp/repo-update-$(/bin/date +%Y-%m-%d).txt
PID_FILE=/var/run/${0##*/}.pid

E_RETURN=85        # Something unexpected happened.


# General rsync options
# -r: recursive download
# -t: reserve time
```

```
# -v: verbose

OPTS="-rtv --delete-excluded --delete-after --partial"

# rsync include pattern
# Leading slash causes absolute path name match.
INCLUDE=(
    "/4/i386/kde-i18n-Chinese*"
#   ^                         ^
# Quoting is necessary to prevent globbing.
)


# rsync exclude pattern
# Temporarily comment out unwanted pkgs using "#" . . .
EXCLUDE=(
    /1
    /2
    /3
    /testing
    /4/SRPMS
    /4/ppc
    /4/x86_64
    /4/i386/debug
   "/4/i386/kde-i18n-*"
   "/4/i386/openoffice.org-langpack-*"
   "/4/i386/*i586.rpm"
   "/4/i386/GFS-*"
   "/4/i386/cman-*"
   "/4/i386/dlm-*"
   "/4/i386/gnbd-*"
   "/4/i386/kernel-smp*"
#  "/4/i386/kernel-xen*"
#  "/4/i386/xen-*"
)


init () {
    # Let pipe command return possible rsync error, e.g., stalled network.
    set -o pipefail                  # Newly introduced in Bash, version 3.

    TMP=${TMPDIR:-/tmp}/${0##*/}.$$  # Store refined download list.
    trap "{
        rm -f $TMP 2>/dev/null
    }" EXIT                          # Clear temporary file on exit.
}


check_pid () {
# Check if process exists.
    if [ -s "$PID_FILE" ]; then
        echo "PID file exists. Checking ..."
        PID=$(/bin/egrep -o "^[[:digit:]]+" $PID_FILE)
        if /bin/ps --pid $PID &>/dev/null; then
            echo "Process $PID found. ${0##*/} seems to be running!"
           /usr/bin/logger -t ${0##*/} \
                 "Process $PID found. ${0##*/} seems to be running!"
            exit $E_RETURN
        fi
        echo "Process $PID not found. Start new process . . ."
    fi
}
```

```
#  Set overall file update range starting from root or $URL,
#+ according to above patterns.
set_range () {
    include=
    exclude=
    for p in "${INCLUDE[@]}"; do
        include="$include --include \"$p\""
    done

    for p in "${EXCLUDE[@]}"; do
        exclude="$exclude --exclude \"$p\""
    done
}


# Retrieve and refine rsync update list.
get_list () {
    echo $$ > $PID_FILE || {
        echo "Can't write to pid file $PID_FILE"
        exit $E_RETURN
    }

    echo -n "Retrieving and refining update list . . ."

    # Retrieve list -- 'eval' is needed to run rsync as a single command.
    # $3 and $4 is the date and time of file creation.
    # $5 is the full package name.
    previous=
    pre_file=
    pre_date=0
    eval /bin/nice /usr/bin/rsync \
        -r $include $exclude $URL | \
        egrep '^dr.x|^-r' | \
        awk '{print $3, $4, $5}' | \
        sort -k3 | \
        { while read line; do
            # Get seconds since epoch, to filter out obsolete pkgs.
            cur_date=$(date -d "$(echo $line | awk '{print $1, $2}')" +%s)
            #  echo $cur_date

            # Get file name.
            cur_file=$(echo $line | awk '{print $3}')
            #  echo $cur_file

            # Get rpm pkg name from file name, if possible.
            if [[ $cur_file == *rpm ]]; then
                pkg_name=$(echo $cur_file | sed -r -e \
                    's/(^([^_-]+[_-])+)[[:digit:]]+\..*[_-].*$/\1/')
            else
                pkg_name=
            fi
            # echo $pkg_name

            if [ -z "$pkg_name" ]; then   #  If not a rpm file,
                echo $cur_file >> $TMP    #+ then append to download list.
            elif [ "$pkg_name" != "$previous" ]; then   # A new pkg found.
                echo $pre_file >> $TMP                  # Output latest file.
                previous=$pkg_name                      # Save current.
                pre_date=$cur_date
                pre_file=$cur_file
```

```
            elif [ "$cur_date" -gt "$pre_date" ]; then
                                                #  If same pkg, but newer,
                pre_date=$cur_date              #+ then update latest pointer.
                pre_file=$cur_file
            fi
            done
            echo $pre_file >> $TMP              #  TMP contains ALL
                                                #+ of refined list now.
            # echo "subshell=$BASH_SUBSHELL"

    }       # Bracket required here to let final "echo $pre_file >> $TMP"
            # Remained in the same subshell ( 1 ) with the entire loop.

    RET=$?  # Get return code of the pipe command.

    [ "$RET" -ne 0 ] && {
        echo "List retrieving failed with code $RET"
        exit $E_RETURN
    }

    echo "done"; echo
}

# Real rsync download part.
get_file () {

    echo "Downloading..."
    /bin/nice /usr/bin/rsync \
        $OPTS \
        --filter "merge,+/ $TMP" \
        --exclude '*'  \
        $URL $DEST     \
        | /usr/bin/tee $LOG

    RET=$?

   #  --filter merge,+/ is crucial for the intention.
   #  + modifier means include and / means absolute path.
   #  Then sorted list in $TMP will contain ascending dir name and
   #+ prevent the following --exclude '*' from "shortcutting the circuit."

    echo "Done"

    rm -f $PID_FILE 2>/dev/null

    return $RET
}

# -------
# Main
init
check_pid
set_range
get_list
get_file
RET=$?
# -------

if [ "$RET" -eq 0 ]; then
    /usr/bin/logger -t ${0##*/} "Fedora update mirrored successfully."
else
    /usr/bin/logger -t ${0##*/} \
```

```
    "Fedora update mirrored with failure code: $RET"
fi

exit $RET
```

See also Example A-32.

Using rcp, rsync, and similar utilities with security implications in a shell script may not be advisable. Consider, instead, using **ssh**, scp, or an **expect** script.

ssh

`Secure shell`, logs onto a remote host and executes commands there. This secure replacement for **telnet**, **rlogin**, **rcp**, and **rsh** uses identity authentication and encryption. See its manpage for details.

Example 16-44. Using *ssh*

```
#!/bin/bash
# remote.bash: Using ssh.

# This example by Michael Zick.
# Used with permission.


#   Presumptions:
#   ------------
#   fd-2 isn't being captured ( '2>/dev/null' ).
#   ssh/sshd presumes stderr ('2') will display to user.
#
#   sshd is running on your machine.
#   For any 'standard' distribution, it probably is,
#+  and without any funky ssh-keygen having been done.

# Try ssh to your machine from the command-line:
#
# $ ssh $HOSTNAME
# Without extra set-up you'll be asked for your password.
#   enter password
#   when done,  $ exit
#
# Did that work? If so, you're ready for more fun.

# Try ssh to your machine as 'root':
#
#   $  ssh -l root $HOSTNAME
#   When asked for password, enter root's, not yours.
#          Last login: Tue Aug 10 20:25:49 2004 from localhost.localdomain
#   Enter 'exit' when done.

#  The above gives you an interactive shell.
#  It is possible for sshd to be set up in a 'single command' mode,
#+ but that is beyond the scope of this example.
#  The only thing to note is that the following will work in
#+ 'single command' mode.


# A basic, write stdout (local) command.

ls -l

# Now the same basic command on a remote machine.
```

```
# Pass a different 'USERNAME' 'HOSTNAME' if desired:
USER=${USERNAME:-$(whoami)}
HOST=${HOSTNAME:-$(hostname)}

#  Now excute the above command-line on the remote host,
#+ with all transmissions encrypted.

ssh -l ${USER} ${HOST} " ls -l "

#  The expected result is a listing of your username's home
#+ directory on the remote machine.
#  To see any difference, run this script from somewhere
#+ other than your home directory.

#  In other words, the Bash command is passed as a quoted line
#+ to the remote shell, which executes it on the remote machine.
#  In this case, sshd does  ' bash -c "ls -l" '   on your behalf.

#  For information on topics such as not having to enter a
#+ password/passphrase for every command-line, see
#+    man ssh
#+    man ssh-keygen
#+    man sshd_config.

exit 0
```

Within a loop, **ssh** may cause unexpected behavior. According to a Usenet post in the comp.unix shell archives, **ssh** inherits the loop's `stdin`. To remedy this, pass **ssh** either the `-n` or `-f` option.

Thanks, Jason Bechtel, for pointing this out.

scp

`Secure copy`, similar in function to **rcp**, copies files between two different networked machines, but does so using authentication, and with a security level similar to **ssh**.

Local Network

write

This is a utility for terminal-to-terminal communication. It allows sending lines from your terminal (console or *xterm*) to that of another user. The mesg command may, of course, be used to disable write access to a terminal

Since **write** is interactive, it would not normally find use in a script.

netconfig

A command-line utility for configuring a network adapter (using *DHCP*). This command is native to Red Hat centric Linux distros.

Mail

mail

Send or read e-mail messages.

This stripped-down command-line mail client works fine as a command embedded in a script.

Example 16-45. A script that mails itself

```
#!/bin/sh
# self-mailer.sh: Self-mailing script

adr=${1:-`whoami`}     # Default to current user, if not specified.
#  Typing 'self-mailer.sh wiseguy@superdupergenius.com'
#+ sends this script to that addressee.
#  Just 'self-mailer.sh' (no argument) sends the script
#+ to the person invoking it, for example, bozo@localhost.localdomain.
#
#  For more on the ${parameter:-default} construct,
#+ see the "Parameter Substitution" section
#+ of the "Variables Revisited" chapter.

# ============================================================================
  cat $0 | mail -s "Script \"`basename $0`\" has mailed itself to you." "$adr"
# ============================================================================

# --------------------------------------------
#  Greetings from the self-mailing script.
#  A mischievous person has run this script,
#+ which has caused it to mail itself to you.
#  Apparently, some people have nothing better
#+ to do with their time.
# --------------------------------------------

echo "At `date`, script \"`basename $0`\" mailed to "$adr"."

exit 0

#  Note that the "mailx" command (in "send" mode) may be substituted
#+ for "mail" ... but with somewhat different options.
```

mailto

Similar to the **mail** command, **mailto** sends e-mail messages from the command-line or in a script. However, **mailto** also permits sending MIME (multimedia) messages.

mailstats

Show *mail statistics*. This command may be invoked only by *root*.

```
root# mailstats
Statistics from Tue Jan  1 20:32:08 2008
  M   msgsfr  bytes_from   msgsto    bytes_to  msgsrej msgsdis msgsqur  Mailer
  4     1682      24118K        0          0K        0       0       0  esmtp
  9      212        640K     1894      25131K        0       0       0  local
 =====================================================================
  T     1894      24758K     1894      25131K        0       0       0
  C      414                    0
```

vacation

This utility automatically replies to e-mails that the intended recipient is on vacation and temporarily unavailable. It runs on a network, in conjunction with **sendmail**, and is not applicable to a dial-up POPmail account.

16.7. Terminal Control Commands

Command affecting the console or terminal

tput

Initialize terminal and/or fetch information about it from terminfo data. Various options permit certain terminal operations: **tput clear** is the equivalent of clear; **tput reset** is the equivalent of reset.

```
bash$ tput longname
xterm terminal emulator (X Window System)
```

Issuing a **tput cup X Y** moves the cursor to the (X,Y) coordinates in the current terminal. A **clear** to erase the terminal screen would normally precede this.

Some interesting options to *tput* are:

◊ `bold`, for high-intensity text
◊ `smul`, to underline text in the terminal
◊ `smso`, to render text in reverse
◊ `sgr0`, to reset the terminal parameters (to normal), without clearing the screen

Example scripts using *tput*:

1. Example 36-15
2. Example 36-13
3. Example A-44
4. Example A-42
5. Example 27-2

Note that stty offers a more powerful command set for controlling a terminal.

infocmp

This command prints out extensive information about the current terminal. It references the *terminfo* database.

```
bash$ infocmp
#       Reconstructed via infocmp from file:
 /usr/share/terminfo/r/rxvt
 rxvt|rxvt terminal emulator (X Window System),
         am, bce, eo, km, mir, msgr, xenl, xon,
         colors#8, cols#80, it#8, lines#24, pairs#64,
         acsc=``aaffggjjkkllmmnnooppqqrrssttuuvvwwxxyyzz{{||}}~~,
         bel=^G, blink=\E[5m, bold=\E[1m,
         civis=\E[?25l,
         clear=\E[H\E[2J, cnorm=\E[?25h, cr=^M,
         ...
```

reset

Reset terminal parameters and clear text screen. As with **clear**, the cursor and prompt reappear in the upper lefthand corner of the terminal.

clear

The **clear** command simply clears the text screen at the console or in an *xterm*. The prompt and cursor reappear at the upper lefthand corner of the screen or xterm window. This command may be used either at the command line or in a script. See Example 11-26.

resize

Echoes commands necessary to set `$TERM` and `$TERMCAP` to duplicate the *size* (dimensions) of the current terminal.

```
bash$ resize
set noglob;
 setenv COLUMNS '80';
 setenv LINES '24';
 unset noglob;
```

script

This utility records (saves to a file) all the user keystrokes at the command-line in a console or an xterm window. This, in effect, creates a record of a session.

16.8. Math Commands

"Doing the numbers"

factor

Decompose an integer into prime factors.

```
bash$ factor 27417
27417: 3 13 19 37
```

Example 16-46. Generating prime numbers

```
#!/bin/bash
# primes2.sh

#  Generating prime numbers the quick-and-easy way,
#+ without resorting to fancy algorithms.

CEILING=10000   # 1 to 10000
PRIME=0
E_NOTPRIME=

is_prime ()
{
  local factors
  factors=( $(factor $1) )  # Load output of `factor` into array.

if [ -z "${factors[2]}" ]
#  Third element of "factors" array:
#+ ${factors[2]} is 2nd factor of argument.
#  If it is blank, then there is no 2nd factor,
#+ and the argument is therefore prime.
then
  return $PRIME             # 0
else
  return $E_NOTPRIME        # null
fi
}

echo
for n in $(seq $CEILING)
do
  if is_prime $n
```

```
  then
    printf %5d $n
  fi   #    ^  Five positions per number suffices.
done   #       For a higher $CEILING, adjust upward, as necessary.

echo

exit
```

bc

Bash can't handle floating point calculations, and it lacks operators for certain important mathematical functions. Fortunately, **bc** gallops to the rescue.

Not just a versatile, arbitrary precision calculation utility, **bc** offers many of the facilities of a programming language. It has a syntax vaguely resembling **C**.

Since it is a fairly well-behaved UNIX utility, and may therefore be used in a pipe, **bc** comes in handy in scripts.

Here is a simple template for using **bc** to calculate a script variable. This uses command substitution.

```
variable=$(echo "OPTIONS; OPERATIONS" | bc)
```

Example 16-47. Monthly Payment on a Mortgage

```
#!/bin/bash
# monthlypmt.sh: Calculates monthly payment on a mortgage.


#  This is a modification of code in the
#+ "mcalc" (mortgage calculator) package,
#+ by Jeff Schmidt
#+ and
#+ Mendel Cooper (yours truly, the ABS Guide author).
#   http://www.ibiblio.org/pub/Linux/apps/financial/mcalc-1.6.tar.gz

echo
echo "Given the principal, interest rate, and term of a mortgage,"
echo "calculate the monthly payment."

bottom=1.0

echo
echo -n "Enter principal (no commas) "
read principal
echo -n "Enter interest rate (percent) "  # If 12%, enter "12", not ".12".
read interest_r
echo -n "Enter term (months) "
read term


 interest_r=$(echo "scale=9; $interest_r/100.0" | bc) # Convert to decimal.
                 #           ^^^^^^^^^^^^^^^^^  Divide by 100.
                 # "scale" determines how many decimal places.

 interest_rate=$(echo "scale=9; $interest_r/12 + 1.0" | bc)
```

```
 top=$(echo "scale=9; $principal*$interest_rate^$term" | bc)
          #           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
          #           Standard formula for figuring interest.

 echo; echo "Please be patient. This may take a while."

 let "months = $term - 1"
# ====================================================================
 for ((x=$months; x > 0; x--))
 do
   bot=$(echo "scale=9; $interest_rate^$x" | bc)
   bottom=$(echo "scale=9; $bottom+$bot" | bc)
#  bottom = $(($bottom + $bot"))
 done
# ====================================================================

# --------------------------------------------------------------------
#  Rick Boivie pointed out a more efficient implementation
#+ of the above loop, which decreases computation time by 2/3.

# for ((x=1; x <= $months; x++))
# do
#   bottom=$(echo "scale=9; $bottom * $interest_rate + 1" | bc)
# done


#  And then he came up with an even more efficient alternative,
#+ one that cuts down the run time by about 95%!

# bottom=`{
#     echo "scale=9; bottom=$bottom; interest_rate=$interest_rate"
#     for ((x=1; x <= $months; x++))
#     do
#          echo 'bottom = bottom * interest_rate + 1'
#     done
#     echo 'bottom'
#     } | bc`       # Embeds a 'for loop' within command substitution.
# --------------------------------------------------------------------------
#  On the other hand, Frank Wang suggests:
#  bottom=$(echo "scale=9; ($interest_rate^$term-1)/($interest_rate-1)" | bc)

#  Because . . .
#  The algorithm behind the loop
#+ is actually a sum of geometric proportion series.
#  The sum formula is e0(1-q^n)/(1-q),
#+ where e0 is the first element and q=e(n+1)/e(n)
#+ and n is the number of elements.
# --------------------------------------------------------------------------


 # let "payment = $top/$bottom"
 payment=$(echo "scale=2; $top/$bottom" | bc)
 # Use two decimal places for dollars and cents.

 echo
 echo "monthly payment = \$$payment"  # Echo a dollar sign in front of amount.
 echo


 exit 0
```

```
# Exercises:
#   1) Filter input to permit commas in principal amount.
#   2) Filter input to permit interest to be entered as percent or decimal.
#   3) If you are really ambitious,
#+     expand this script to print complete amortization tables.
```

Example 16-48. Base Conversion

```
#!/bin/bash
###########################################################################
# Shellscript:  base.sh - print number to different bases (Bourne Shell)
# Author     :  Heiner Steven (heiner.steven@odn.de)
# Date       :  07-03-95
# Category   :  Desktop
# $Id: base.sh,v 1.2 2000/02/06 19:55:35 heiner Exp $
# ==> Above line is RCS ID info.
###########################################################################
# Description
#
# Changes
# 21-03-95 stv  fixed error occuring with 0xb as input (0.2)
###########################################################################

# ==> Used in ABS Guide with the script author's permission.
# ==> Comments added by ABS Guide author.

NOARGS=85
PN=`basename "$0"`                               # Program name
VER=`echo '$Revision: 1.2 $' | cut -d' ' -f2`   # ==> VER=1.2

Usage () {
    echo "$PN - print number to different bases, $VER (stv '95)
usage: $PN [number ...]

If no number is given, the numbers are read from standard input.
A number may be
    binary (base 2)             starting with 0b (i.e. 0b1100)
    octal (base 8)              starting with 0  (i.e. 014)
    hexadecimal (base 16)       starting with 0x (i.e. 0xc)
    decimal                     otherwise (i.e. 12)" >&2
    exit $NOARGS
}   # ==> Prints usage message.

Msg () {
    for i   # ==> in [list] missing. Why?
    do echo "$PN: $i" >&2
    done
}

Fatal () { Msg "$@"; exit 66; }

PrintBases () {
    # Determine base of the number
    for i      # ==> in [list] missing...
    do         # ==> so operates on command-line arg(s).
        case "$i" in
            0b*)                ibase=2;;       # binary
            0x*|[a-f]*|[A-F]*)  ibase=16;;      # hexadecimal
```

```
            0*)                 ibase=8;;       # octal
            [1-9]*)             ibase=10;;      # decimal
            *)
                Msg "illegal number $i - ignored"
                continue;;
        esac

        # Remove prefix, convert hex digits to uppercase (bc needs this).
        number=`echo "$i" | sed -e 's:^0[bBxX]::' | tr '[a-f]' '[A-F]'`
        # ==> Uses ":" as sed separator, rather than "/".

        # Convert number to decimal
        dec=`echo "ibase=$ibase; $number" | bc`  # ==> 'bc' is calculator utility.
        case "$dec" in
            [0-9]*)     ;;                         # number ok
            *)          continue;;                 # error: ignore
        esac

        # Print all conversions in one line.
        # ==> 'here document' feeds command list to 'bc'.
        echo `bc <<!
            obase=16; "hex="; $dec
            obase=10; "dec="; $dec
            obase=8;  "oct="; $dec
            obase=2;  "bin="; $dec
!
    ` | sed -e 's: :    :g'

    done
}

while [ $# -gt 0 ]
# ==>  Is a "while loop" really necessary here,
# ==>+ since all the cases either break out of the loop
# ==>+ or terminate the script.
# ==> (Above comment by Paulo Marcel Coelho Aragao.)
do
    case "$1" in
        --)     shift; break;;
        -h)     Usage;;                 # ==> Help message.
        -*)     Usage;;
         *)     break;;                 # First number
    esac   # ==> Error checking for illegal input might be appropriate.
    shift
done

if [ $# -gt 0 ]
then
    PrintBases "$@"
else                                    # Read from stdin.
    while read line
    do
        PrintBases $line
    done
fi


exit
```

An alternate method of invoking **bc** involves using a here document embedded within a command substitution block. This is especially appropriate when a script needs to pass a list of options and commands to **bc**.

```
variable=`bc << LIMIT_STRING
options
statements
operations
LIMIT_STRING
`

...or...


variable=$(bc << LIMIT_STRING
options
statements
operations
LIMIT_STRING
)
```

Example 16-49. Invoking *bc* using a *here document*

```
#!/bin/bash
# Invoking 'bc' using command substitution
# in combination with a 'here document'.


var1=`bc << EOF
18.33 * 19.78
EOF
`
echo $var1       # 362.56


#  $( ... ) notation also works.
v1=23.53
v2=17.881
v3=83.501
v4=171.63

var2=$(bc << EOF
scale = 4
a = ( $v1 + $v2 )
b = ( $v3 * $v4 )
a * b + 15.35
EOF
)
echo $var2       # 593487.8452


var3=$(bc -l << EOF
scale = 9
s ( 1.7 )
EOF
)
# Returns the sine of 1.7 radians.
# The "-l" option calls the 'bc' math library.
echo $var3       # .991664810


# Now, try it in a function...
hypotenuse ()    # Calculate hypotenuse of a right triangle.
```

```
{                 # c = sqrt( a^2 + b^2 )
hyp=$(bc -l << EOF
scale = 9
sqrt ( $1 * $1 + $2 * $2 )
EOF
)
# Can't directly return floating point values from a Bash function.
# But, can echo-and-capture:
echo "$hyp"
}

hyp=$(hypotenuse 3.68 7.31)
echo "hypotenuse = $hyp"    # 8.184039344


exit 0
```

Example 16-50. Calculating PI

```
#!/bin/bash
# cannon.sh: Approximating PI by firing cannonballs.

# Author: Mendel Cooper
# License: Public Domain
# Version 2.2, reldate 13oct08.

# This is a very simple instance of a "Monte Carlo" simulation:
#+ a mathematical model of a real-life event,
#+ using pseudorandom numbers to emulate random chance.

#  Consider a perfectly square plot of land, 10000 units on a side.
#  This land has a perfectly circular lake in its center,
#+ with a diameter of 10000 units.
#  The plot is actually mostly water, except for land in the four corners.
#  (Think of it as a square with an inscribed circle.)
#
#  We will fire iron cannonballs from an old-style cannon
#+ at the square.
#  All the shots impact somewhere on the square,
#+ either in the lake or on the dry corners.
#  Since the lake takes up most of the area,
#+ most of the shots will SPLASH! into the water.
#  Just a few shots will THUD! into solid ground
#+ in the four corners of the square.
#
#  If we take enough random, unaimed shots at the square,
#+ Then the ratio of SPLASHES to total shots will approximate
#+ the value of PI/4.
#
#  The simplified explanation is that the cannon is actually
#+ shooting only at the upper right-hand quadrant of the square,
#+ i.e., Quadrant I of the Cartesian coordinate plane.
#
#
#  Theoretically, the more shots taken, the better the fit.
#  However, a shell script, as opposed to a compiled language
#+ with floating-point math built in, requires some compromises.
#  This decreases the accuracy of the simulation.
```

```
DIMENSION=10000  # Length of each side of the plot.
                 # Also sets ceiling for random integers generated.

MAXSHOTS=1000    # Fire this many shots.
                 # 10000 or more would be better, but would take too long.
PMULTIPLIER=4.0  # Scaling factor.

declare -r M_PI=3.141592654
                 # Actual 9-place value of PI, for comparison purposes.

get_random ()
{
SEED=$(head -n 1 /dev/urandom | od -N 1 | awk '{ print $2 }')
RANDOM=$SEED                                  #  From "seeding-random.sh"
                                              #+ example script.
let "rnum = $RANDOM % $DIMENSION"             #  Range less than 10000.
echo $rnum
}

distance=        # Declare global variable.
hypotenuse ()    # Calculate hypotenuse of a right triangle.
{                # From "alt-bc.sh" example.
distance=$(bc -l << EOF
scale = 0
sqrt ( $1 * $1 + $2 * $2 )
EOF
)
#  Setting "scale" to zero rounds down result to integer value,
#+ a necessary compromise in this script.
#  It decreases the accuracy of this simulation.
}


# ==========================================================
# main() {
# "Main" code block, mimicking a C-language main() function.

# Initialize variables.
shots=0
splashes=0
thuds=0
Pi=0
error=0

while [ "$shots" -lt  "$MAXSHOTS" ]           # Main loop.
do

  xCoord=$(get_random)                        # Get random X and Y coords.
  yCoord=$(get_random)
  hypotenuse $xCoord $yCoord                  #  Hypotenuse of
                                              #+ right-triangle = distance.
  ((shots++))

  printf "#%4d   " $shots
  printf "Xc = %4d  " $xCoord
  printf "Yc = %4d  " $yCoord
  printf "Distance = %5d  " $distance         #   Distance from
                                              #+  center of lake
                                              #+  -- the "origin" --
                                              #+  coordinate (0,0).

  if [ "$distance" -le "$DIMENSION" ]
```

```
  then
    echo -n "SPLASH!  "
    ((splashes++))
  else
    echo -n "THUD!    "
    ((thuds++))
  fi

  Pi=$(echo "scale=9; $PMULTIPLIER*$splashes/$shots" | bc)
  # Multiply ratio by 4.0.
  echo -n "PI ~ $Pi"
  echo

done

echo
echo "After $shots shots, PI looks like approximately   $Pi"
#  Tends to run a bit high,
#+ possibly due to round-off error and imperfect randomness of $RANDOM.
#  But still usually within plus-or-minus 5% . . .
#+ a pretty fair rough approximation.
error=$(echo "scale=9; $Pi - $M_PI" | bc)
pct_error=$(echo "scale=2; 100.0 * $error / $M_PI" | bc)
echo -n "Deviation from mathematical value of PI =        $error"
echo " ($pct_error% error)"
echo

# End of "main" code block.
# }
# ==========================================================

exit 0

#  One might well wonder whether a shell script is appropriate for
#+ an application as complex and computation-intensive as a simulation.
#
#  There are at least two justifications.
#  1) As a proof of concept: to show it can be done.
#  2) To prototype and test the algorithms before rewriting
#+    it in a compiled high-level language.
```

See also Example A-37.

dc

The **dc** (**d**esk **c**alculator) utility is stack-oriented and uses RPN (*Reverse Polish Notation*). Like **bc**, it has much of the power of a programming language.

Similar to the procedure with **bc**, echo a command-string to **dc**.

```
echo "[Printing a string ... ]P" | dc
# The P command prints the string between the preceding brackets.

# And now for some simple arithmetic.
echo "7 8 * p" | dc     # 56
#  Pushes 7, then 8 onto the stack,
#+ multiplies ("*" operator), then prints the result ("p" operator).
```

Most persons avoid **dc**, because of its non-intuitive input and rather cryptic operators. Yet, it has its uses.

Example 16-51. Converting a decimal number to hexadecimal

```
#!/bin/bash
# hexconvert.sh: Convert a decimal number to hexadecimal.

E_NOARGS=85 # Command-line arg missing.
BASE=16     # Hexadecimal.

if [ -z "$1" ]
then        # Need a command-line argument.
  echo "Usage: $0 number"
  exit $E_NOARGS
fi          # Exercise: add argument validity checking.


hexcvt ()
{
if [ -z "$1" ]
then
  echo 0
  return    # "Return" 0 if no arg passed to function.
fi

echo ""$1" "$BASE" o p" | dc
#                  o    sets radix (numerical base) of output.
#                    p  prints the top of stack.
# For other options: 'man dc' ...
return
}

hexcvt "$1"

exit
```

Studying the info page for **dc** is a painful path to understanding its intricacies. There seems to be a small, select group of *dc wizards* who delight in showing off their mastery of this powerful, but arcane utility.

```
bash$ echo "16i[q]sa[ln0=aln100%Pln100/snlbx]sbA0D68736142snlbxq" | dc
Bash
```

```
dc <<< 10k5v1+2/p # 1.6180339887
#  ^^^            Feed operations to dc using a Here String.
#      ^^^        Pushes 10 and sets that as the precision (10k).
#         ^^      Pushes 5 and takes its square root
#                 (5v, v = square root).
#           ^^    Pushes 1 and adds it to the running total (1+).
#             ^^  Pushes 2 and divides the running total by that (2/).
#               ^ Pops and prints the result (p)
#  The result is  1.6180339887 ...
#  ... which happens to be the Pythagorean Golden Ratio, to 10 places.
```

Example 16-52. Factoring

```
#!/bin/bash
# factr.sh: Factor a number

MIN=2       # Will not work for number smaller than this.
E_NOARGS=85
E_TOOSMALL=86
```

```
if [ -z $1 ]
then
  echo "Usage: $0 number"
  exit $E_NOARGS
fi

if [ "$1" -lt "$MIN" ]
then
  echo "Number to factor must be $MIN or greater."
  exit $E_TOOSMALL
fi

# Exercise: Add type checking (to reject non-integer arg).

echo "Factors of $1:"
# -------------------------------------------------------
echo  "$1[p]s2[lip/dli%0=1dvsr]s12sid2%0=13sidvsr[dli%0=\
1lrli2+dsi!>.]ds.xd1<2" | dc
# -------------------------------------------------------
#  Above code written by Michel Charpentier <charpov@cs.unh.edu>
#  (as a one-liner, here broken into two lines for display purposes).
#  Used in ABS Guide with permission (thanks!).

 exit

 # $ sh factr.sh 270138
 # 2
 # 3
 # 11
 # 4093
```

awk

Yet another way of doing floating point math in a script is using awk's built-in math functions in a shell wrapper.

Example 16-53. Calculating the hypotenuse of a triangle

```
#!/bin/bash
# hypotenuse.sh: Returns the "hypotenuse" of a right triangle.
#                (square root of sum of squares of the "legs")

ARGS=2                # Script needs sides of triangle passed.
E_BADARGS=85          # Wrong number of arguments.

if [ $# -ne "$ARGS" ] # Test number of arguments to script.
then
  echo "Usage: `basename $0` side_1 side_2"
  exit $E_BADARGS
fi


AWKSCRIPT=' { printf( "%3.7f\n", sqrt($1*$1 + $2*$2) ) } '
#             command(s) / parameters passed to awk


# Now, pipe the parameters to awk.
    echo -n "Hypotenuse of $1 and $2 = "
    echo $1 $2 | awk "$AWKSCRIPT"
#   ^^^^^^^^^^^^
# An echo-and-pipe is an easy way of passing shell parameters to awk.
```

```
exit

# Exercise: Rewrite this script using 'bc' rather than awk.
#           Which method is more intuitive?
```

16.9. Miscellaneous Commands

Command that fit in no special category

jot, **seq**

These utilities emit a sequence of integers, with a user-selectable increment.

The default separator character between each integer is a newline, but this can be changed with the `-s` option.

```
bash$ seq 5
1
 2
 3
 4
 5



bash$ seq -s : 5
1:2:3:4:5
```

Both **jot** and **seq** come in handy in a for loop.

Example 16-54. Using *seq* to generate loop arguments

```
#!/bin/bash
# Using "seq"

echo

for a in `seq 80`  # or   for a in $( seq 80 )
# Same as   for a in 1 2 3 4 5 ... 80   (saves much typing!).
# May also use 'jot' (if present on system).
do
  echo -n "$a "
done      # 1 2 3 4 5 ... 80
# Example of using the output of a command to generate
# the [list] in a "for" loop.

echo; echo


COUNT=80  # Yes, 'seq' also accepts a replaceable parameter.

for a in `seq $COUNT`  # or   for a in $( seq $COUNT )
do
  echo -n "$a "
done      # 1 2 3 4 5 ... 80
```

```
echo; echo

BEGIN=75
END=80

for a in `seq $BEGIN $END`
#  Giving "seq" two arguments starts the count at the first one,
#+ and continues until it reaches the second.
do
  echo -n "$a "
done      # 75 76 77 78 79 80

echo; echo

BEGIN=45
INTERVAL=5
END=80

for a in `seq $BEGIN $INTERVAL $END`
#  Giving "seq" three arguments starts the count at the first one,
#+ uses the second for a step interval,
#+ and continues until it reaches the third.
do
  echo -n "$a "
done      # 45 50 55 60 65 70 75 80

echo; echo

exit 0
```

A simpler example:

```
#  Create a set of 10 files,
#+ named file.1, file.2 . . . file.10.
COUNT=10
PREFIX=file

for filename in `seq $COUNT`
do
  touch $PREFIX.$filename
  #  Or, can do other operations,
  #+ such as rm, grep, etc.
done
```

Example 16-55. Letter Count"

```
#!/bin/bash
# letter-count.sh: Counting letter occurrences in a text file.
# Written by Stefano Palmeri.
# Used in ABS Guide with permission.
# Slightly modified by document author.

MINARGS=2          # Script requires at least two arguments.
E_BADARGS=65
FILE=$1

let LETTERS=$#-1   # How many letters specified (as command-line args).
                   # (Subtract 1 from number of command-line args.)


show_help(){
```

```
            echo
            echo Usage: `basename $0` file letters
            echo Note: `basename $0` arguments are case sensitive.
            echo Example: `basename $0` foobar.txt G n U L i N U x.
            echo
}

# Checks number of arguments.
if [ $# -lt $MINARGS ]; then
   echo
   echo "Not enough arguments."
   echo
   show_help
   exit $E_BADARGS
fi


# Checks if file exists.
if [ ! -f $FILE ]; then
    echo "File \"$FILE\" does not exist."
    exit $E_BADARGS
fi



# Counts letter occurrences .
for n in `seq $LETTERS`; do
      shift
      if [[ `echo -n "$1" | wc -c` -eq 1 ]]; then             #  Checks arg.
             echo "$1" -\> `cat $FILE | tr -cd  "$1" | wc -c` #  Counting.
      else
             echo "$1 is not a  single char."
      fi
done

exit $?

#  This script has exactly the same functionality as letter-count2.sh,
#+ but executes faster.
#  Why?
```

Somewhat more capable than *seq*, **jot** is a classic UNIX utility that is not normally included in a standard Linux distro. However, the source *rpm* is available for download from the MIT repository.

Unlike *seq*, **jot** can generate a sequence of random numbers, using the `-r` option.

```
bash$ jot -r 3 999
1069
 1272
 1428
```

getopt

The **getopt** command parses command-line options preceded by a dash. This external command corresponds to the getopts Bash builtin. Using **getopt** permits handling long options by means of the `-l` flag, and this also allows parameter reshuffling.

Example 16-56. Using *getopt* to parse command-line options

```
#!/bin/bash
# Using getopt

# Try the following when invoking this script:
#   sh ex33a.sh -a
#   sh ex33a.sh -abc
#   sh ex33a.sh -a -b -c
#   sh ex33a.sh -d
#   sh ex33a.sh -dXYZ
#   sh ex33a.sh -d XYZ
#   sh ex33a.sh -abcd
#   sh ex33a.sh -abcdZ
#   sh ex33a.sh -z
#   sh ex33a.sh a
# Explain the results of each of the above.

E_OPTERR=65

if [ "$#" -eq 0 ]
then   # Script needs at least one command-line argument.
  echo "Usage $0 -[options a,b,c]"
  exit $E_OPTERR
fi

set -- `getopt "abcd:" "$@"`
# Sets positional parameters to command-line arguments.
# What happens if you use "$*" instead of "$@"?

while [ ! -z "$1" ]
do
  case "$1" in
    -a) echo "Option \"a\"";;
    -b) echo "Option \"b\"";;
    -c) echo "Option \"c\"";;
    -d) echo "Option \"d\" $2";;
     *) break;;
  esac

  shift
done

#  It is usually better to use the 'getopts' builtin in a script.
#  See "ex33.sh."

exit 0
```

As *Peggy Russell* points out:

It is often necessary to include an eval to correctly process whitespace and *quotes*.

```
args=$(getopt -o a:bc:d -- "$@")
eval set -- "$args"
```

See Example 10-5 for a simplified emulation of **getopt**.

run-parts

The **run-parts** command [82] executes all the scripts in a target directory, sequentially in ASCII-sorted filename order. Of course, the scripts need to have execute permission.

The cron daemon invokes **run-parts** to run the scripts in the `/etc/cron.*` directories.

yes

In its default behavior the **yes** command feeds a continuous string of the character `y` followed by a line feed to `stdout`. A **control-C** terminates the run. A different output string may be specified, as in **`yes different string`**, which would continually output `different string` to `stdout`.

One might well ask the purpose of this. From the command-line or in a script, the output of **yes** can be redirected or piped into a program expecting user input. In effect, this becomes a sort of poor man's version of *expect*.

`yes | fsck /dev/hda1` runs **fsck** non-interactively (careful!).

`yes | rm -r dirname` has same effect as **`rm -rf dirname`** (careful!).

Caution advised when piping *yes* to a potentially dangerous system command, such as fsck or fdisk. It might have unintended consequences.

The *yes* command parses variables, or more accurately, it echoes parsed variables. For example:

```
bash$ yes $BASH_VERSION
3.1.17(1)-release
 3.1.17(1)-release
 3.1.17(1)-release
 3.1.17(1)-release
 3.1.17(1)-release
 . . .
```

This particular "feature" may be used to create a *very large* ASCII file on the fly:

```
bash$ yes $PATH > huge_file.txt
Ctl-C
```

Hit **`Ctl-C`** *very quickly*, or you just might get more than you bargained for. . . .

The *yes* command may be emulated in a very simple script function.

```
yes ()
{ # Trivial emulation of "yes" ...
  local DEFAULT_TEXT="y"
  while [ true ]   # Endless loop.
  do
    if [ -z "$1" ]
    then
      echo "$DEFAULT_TEXT"
    else           # If argument ...
      echo "$1"    # ... expand and echo it.
    fi
  done             #  The only things missing are the
}                  #+ --help and --version options.
```

banner

Prints arguments as a large vertical banner to `stdout`, using an ASCII character (default '#'). This may be redirected to a printer for hardcopy.

Note that *banner* has been dropped from many Linux distros, presumably because it is no longer considered useful.

printenv

Show all the environmental variables set for a particular user.

```
bash$ printenv | grep HOME
HOME=/home/bozo
```

lp

The **lp** and **lpr** commands send file(s) to the print queue, to be printed as hard copy. [83] These commands trace the origin of their names to the line printers of another era. [84]

`bash$ lp file1.txt` or `bash lp <file1.txt`

It is often useful to pipe the formatted output from **pr** to **lp**.

`bash$ pr -options file1.txt | lp`

Formatting packages, such as groff and *Ghostscript* may send their output directly to **lp**.

`bash$ groff -Tascii file.tr | lp`

`bash$ gs -options | lp file.ps`

Related commands are **lpq**, for viewing the print queue, and **lprm**, for removing jobs from the print queue.

tee

[UNIX borrows an idea from the plumbing trade.]

This is a redirection operator, but with a difference. Like the plumber's *tee,* it permits "siphoning off" *to a file* the output of a command or commands within a pipe, but without affecting the result. This is useful for printing an ongoing process to a file or paper, perhaps to keep track of it for debugging purposes.

```
                              (redirection)
                             |----> to file
                             |
  ==========================|====================
  command ---> command ---> |tee ---> command ---> ---> output of pipe
  ===============================================
```

```
cat listfile* | sort | tee check.file | uniq > result.file
#                      ^^^^^^^^^^^^^^   ^^^^

#  The file "check.file" contains the concatenated sorted "listfiles,"
#+ before the duplicate lines are removed by 'uniq.'
```

mkfifo

This obscure command creates a *named pipe*, a temporary *first-in-first-out buffer* for transferring data between processes. [85] Typically, one process writes to the FIFO, and the other reads from it. See Example A-14.

```
#!/bin/bash
```

```
# This short script by Omair Eshkenazi.
# Used in ABS Guide with permission (thanks!).

mkfifo pipe1   # Yes, pipes can be given names.
mkfifo pipe2   # Hence the designation "named pipe."

(cut -d' ' -f1 | tr "a-z" "A-Z") >pipe2 <pipe1 &
ls -l | tr -s ' ' | cut -d' ' -f3,9- | tee pipe1 |
cut -d' ' -f2 | paste - pipe2

rm -f pipe1
rm -f pipe2

# No need to kill background processes when script terminates (why not?).

exit $?

Now, invoke the script and explain the output:
sh mkfifo-example.sh

4830.tar.gz          BOZO
pipe1   BOZO
pipe2   BOZO
mkfifo-example.sh    BOZO
Mixed.msg BOZO
```

pathchk

This command checks the validity of a filename. If the filename exceeds the maximum allowable length (255 characters) or one or more of the directories in its path is not searchable, then an error message results.

Unfortunately, **pathchk** does not return a recognizable error code, and it is therefore pretty much useless in a script. Consider instead the file test operators.

dd

Though this somewhat obscure and much feared **d**ata **d**uplicator command originated as a utility for exchanging data on magnetic tapes between UNIX minicomputers and IBM mainframes, it still has its uses. The **dd** command simply copies a file (or `stdin/stdout`), but with conversions. Possible conversions include ASCII/EBCDIC, [86] upper/lower case, swapping of byte pairs between input and output, and skipping and/or truncating the head or tail of the input file.

```
# Converting a file to all uppercase:

dd if=$filename conv=ucase > $filename.uppercase
#                    lcase   # For lower case conversion
```

Some basic options to **dd** are:

◊ if=INFILE

INFILE is the *source* file.

◊ of=OUTFILE

OUTFILE is the *target* file, the file that will have the data written to it.

◊ bs=BLOCKSIZE

This is the size of each block of data being read and written, usually a power of 2.

◊ skip=BLOCKS

How many blocks of data to skip in INFILE before starting to copy. This is useful when the INFILE has "garbage" or garbled data in its header or when it is desirable to copy only a portion of the INFILE.

◊ seek=BLOCKS

How many blocks of data to skip in OUTFILE before starting to copy, leaving blank data at beginning of OUTFILE.

◊ count=BLOCKS

Copy only this many blocks of data, rather than the entire INFILE.

◊ conv=CONVERSION

Type of conversion to be applied to INFILE data before copying operation.

A **`dd --help`** lists all the options this powerful utility takes.

Example 16-57. A script that copies itself

```
#!/bin/bash
# self-copy.sh

# This script copies itself.

file_subscript=copy

dd if=$0 of=$0.$file_subscript 2>/dev/null
# Suppress messages from dd:   ^^^^^^^^^^^

exit $?

#  A program whose only output is its own source code
#+ is called a "quine" per Willard Quine.
#  Does this script qualify as a quine?
```

Example 16-58. Exercising *dd*

```
#!/bin/bash
# exercising-dd.sh

# Script by Stephane Chazelas.
# Somewhat modified by ABS Guide author.

infile=$0           # This script.
outfile=log.txt     # Output file left behind.
n=8
p=11

dd if=$infile of=$outfile bs=1 skip=$((n-1)) count=$((p-n+1)) 2> /dev/null
# Extracts characters n to p (8 to 11) from this script ("bash").

# -----------------------------------------------------------------

echo -n "hello vertical world" | dd cbs=1 conv=unblock 2> /dev/null
# Echoes "hello vertical world" vertically downward.
# Why? A newline follows each character dd emits.

exit $?
```

To demonstrate just how versatile **dd** is, let's use it to capture keystrokes.

Example 16-59. Capturing Keystrokes

```
#!/bin/bash
# dd-keypress.sh: Capture keystrokes without needing to press ENTER.


keypresses=4                      # Number of keypresses to capture.


old_tty_setting=$(stty -g)        # Save old terminal settings.

echo "Press $keypresses keys."
stty -icanon -echo                # Disable canonical mode.
                                  # Disable local echo.
keys=$(dd bs=1 count=$keypresses 2> /dev/null)
# 'dd' uses stdin, if "if" (input file) not specified.

stty "$old_tty_setting"           # Restore old terminal settings.

echo "You pressed the \"$keys\" keys."

# Thanks, Stephane Chazelas, for showing the way.
exit 0
```

The **dd** command can do random access on a data stream.

```
echo -n . | dd bs=1 seek=4 of=file conv=notrunc
#  The "conv=notrunc" option means that the output file
#+ will not be truncated.

# Thanks, S.C.
```

The **dd** command can copy raw data and disk images to and from devices, such as floppies and tape drives (Example A-5). A common use is creating boot floppies.

`dd if=kernel-image of=/dev/fd0H1440`

Similarly, **dd** can copy the entire contents of a floppy, even one formatted with a "foreign" OS, to the hard drive as an image file.

`dd if=/dev/fd0 of=/home/bozo/projects/floppy.img`

Likewise, **dd** can create bootable flash drives and SD cards.

`dd if=image.iso of=/dev/sdb`

Example 16-60. Preparing a bootable SD card for the *Raspberry Pi*

```
#!/bin/bash
# rp.sdcard.sh
```

```
# Preparing an SD card with a bootable image for the Raspberry Pi.

# $1 = imagefile name
# $2 = sdcard (device file)
# Otherwise defaults to the defaults, see below.

DEFAULTbs=4M                                 # Block size, 4 mb default.
DEFAULTif="2013-07-26-wheezy-raspbian.img"   # Commonly used distro.
DEFAULTsdcard="/dev/mmcblk0"                 # May be different. Check!
ROOTUSER_NAME=root                           # Must run as root!
E_NOTROOT=81
E_NOIMAGE=82

username=$(id -nu)                           # Who is running this script?
if [ "$username" != "$ROOTUSER_NAME" ]
then
  echo "This script must run as root or with root privileges."
  exit $E_NOTROOT
fi

if [ -n "$1" ]
then
  imagefile="$1"
else
  imagefile="$DEFAULTif"
fi

if [ -n "$2" ]
then
  sdcard="$2"
else
  sdcard="$DEFAULTsdcard"
fi

if [ ! -e $imagefile ]
then
  echo "Image file \"$imagefile\" not found!"
  exit $E_NOIMAGE
fi

echo "Last chance to change your mind!"; echo
read -s -n1 -p "Hit a key to write $imagefile to $sdcard [Ctl-c to exit]."
echo; echo

echo "Writing $imagefile to $sdcard ..."
dd bs=$DEFAULTbs if=$imagefile of=$sdcard

exit $?

# Exercises:
# ---------
# 1) Provide additional error checking.
# 2) Have script autodetect device file for SD card (difficult!).
# 3) Have script sutodetect image file (*img) in $PWD.
```

Other applications of **dd** include initializing temporary swap files (Example 31-2) and ramdisks (Example 31-3). It can even do a low-level copy of an entire hard drive partition, although this is not necessarily recommended.

People (with presumably nothing better to do with their time) are constantly thinking of interesting applications of **dd**.

Example 16-61. Securely deleting a file

```
#!/bin/bash
# blot-out.sh: Erase "all" traces of a file.

#  This script overwrites a target file alternately
#+ with random bytes, then zeros before finally deleting it.
#  After that, even examining the raw disk sectors by conventional methods
#+ will not reveal the original file data.

PASSES=7         #  Number of file-shredding passes.
                 #  Increasing this slows script execution,
                 #+ especially on large target files.
BLOCKSIZE=1      #  I/O with /dev/urandom requires unit block size,
                 #+ otherwise you get weird results.
E_BADARGS=70     #  Various error exit codes.
E_NOT_FOUND=71
E_CHANGED_MIND=72

if [ -z "$1" ]   # No filename specified.
then
  echo "Usage: `basename $0` filename"
  exit $E_BADARGS
fi

file=$1

if [ ! -e "$file" ]
then
  echo "File \"$file\" not found."
  exit $E_NOT_FOUND
fi

echo; echo -n "Are you absolutely sure you want to blot out \"$file\" (y/n)? "
read answer
case "$answer" in
[nN]) echo "Changed your mind, huh?"
      exit $E_CHANGED_MIND
      ;;
*)    echo "Blotting out file \"$file\".";;
esac


flength=$(ls -l "$file" | awk '{print $5}')  # Field 5 is file length.
pass_count=1

chmod u+w "$file"   # Allow overwriting/deleting the file.

echo

while [ "$pass_count" -le "$PASSES" ]
do
  echo "Pass #$pass_count"
  sync         # Flush buffers.
  dd if=/dev/urandom of=$file bs=$BLOCKSIZE count=$flength
```

```
                # Fill with random bytes.
  sync          # Flush buffers again.
  dd if=/dev/zero of=$file bs=$BLOCKSIZE count=$flength
                # Fill with zeros.
  sync          # Flush buffers yet again.
  let "pass_count += 1"
  echo
done


rm -f $file    # Finally, delete scrambled and shredded file.
sync           # Flush buffers a final time.

echo "File \"$file\" blotted out and deleted."; echo


exit 0

#  This is a fairly secure, if inefficient and slow method
#+ of thoroughly "shredding" a file.
#  The "shred" command, part of the GNU "fileutils" package,
#+ does the same thing, although more efficiently.

#  The file cannot not be "undeleted" or retrieved by normal methods.
#  However . . .
#+ this simple method would *not* likely withstand
#+ sophisticated forensic analysis.

#  This script may not play well with a journaled file system.
#  Exercise (difficult): Fix it so it does.



#  Tom Vier's "wipe" file-deletion package does a much more thorough job
#+ of file shredding than this simple script.
#     http://www.ibiblio.org/pub/Linux/utils/file/wipe-2.0.0.tar.bz2

#  For an in-depth analysis on the topic of file deletion and security,
#+ see Peter Gutmann's paper,
#+     "Secure Deletion of Data From Magnetic and Solid-State Memory".
#       http://www.cs.auckland.ac.nz/~pgut001/pubs/secure_del.html
```

See also the dd thread entry in the bibliography.

od

The **od**, or *octal dump* filter converts input (or files) to octal (base-8) or other bases. This is useful for viewing or processing binary data files or otherwise unreadable system device files, such as `/dev/urandom`, and as a filter for binary data.

```
head -c4 /dev/urandom | od -N4 -tu4 | sed -ne '1s/.* //p'
# Sample output: 1324725719, 3918166450, 2989231420, etc.

# From rnd.sh example script, by Stéphane Chazelas
```

See also Example 9-16 and Example A-36.

hexdump

Performs a hexadecimal, octal, decimal, or ASCII dump of a binary file. This command is the rough equivalent of **od**, above, but not nearly as useful. May be used to view the contents of a binary file, in combination with dd and less.

```
dd if=/bin/ls | hexdump -C | less
# The -C option nicely formats the output in tabular form.
```

objdump

Displays information about an object file or binary executable in either hexadecimal form or as a disassembled listing (with the `-d` option).

```
bash$ objdump -d /bin/ls
/bin/ls:     file format elf32-i386

 Disassembly of section .init:

 080490bc <.init>:
  80490bc:       55                          push   %ebp
  80490bd:       89 e5                       mov    %esp,%ebp
  . . .
```

mcookie

This command generates a "magic cookie," a 128-bit (32-character) pseudorandom hexadecimal number, normally used as an authorization "signature" by the X server. This also available for use in a script as a "quick 'n dirty" random number.

```
random000=$(mcookie)
```

Of course, a script could use md5sum for the same purpose.

```
# Generate md5 checksum on the script itself.
random001=`md5sum $0 | awk '{print $1}'`
# Uses 'awk' to strip off the filename.
```

The **mcookie** command gives yet another way to generate a "unique" filename.

Example 16-62. Filename generator

```
#!/bin/bash
# tempfile-name.sh:  temp filename generator

BASE_STR=`mcookie`   # 32-character magic cookie.
POS=11               # Arbitrary position in magic cookie string.
LEN=5                # Get $LEN consecutive characters.

prefix=temp          #  This is, after all, a "temp" file.
                     #  For more "uniqueness," generate the
                     #+ filename prefix using the same method
                     #+ as the suffix, below.

suffix=${BASE_STR:POS:LEN}
                     #  Extract a 5-character string,
                     #+ starting at position 11.

temp_filename=$prefix.$suffix
                     # Construct the filename.

echo "Temp filename = "$temp_filename""

# sh tempfile-name.sh
# Temp filename = temp.e19ea

#  Compare this method of generating "unique" filenames
#+ with the 'date' method in ex51.sh.

exit 0
```

units

This utility converts between different *units of measure*. While normally invoked in interactive mode, **units** may find use in a script.

Example 16-63. Converting meters to miles

```
#!/bin/bash
# unit-conversion.sh
# Must have 'units' utility installed.


convert_units ()  # Takes as arguments the units to convert.
{
  cf=$(units "$1" "$2" | sed --silent -e '1p' | awk '{print $2}')
  # Strip off everything except the actual conversion factor.
  echo "$cf"
}

Unit1=miles
Unit2=meters
cfactor=`convert_units $Unit1 $Unit2`
quantity=3.73

result=$(echo $quantity*$cfactor | bc)

echo "There are $result $Unit2 in $quantity $Unit1."

#  What happens if you pass incompatible units,
#+ such as "acres" and "miles" to the function?

exit 0

# Exercise: Edit this script to accept command-line parameters,
#           with appropriate error checking, of course.
```

m4

A hidden treasure, **m4** is a powerful macro [87] processing filter, virtually a complete language. Although originally written as a pre-processor for *RatFor*, **m4** turned out to be useful as a stand-alone utility. In fact, **m4** combines some of the functionality of eval, tr, and awk, in addition to its extensive macro expansion facilities.

The April, 2002 issue of *Linux Journal* has a very nice article on **m4** and its uses.

Example 16-64. Using *m4*

```
#!/bin/bash
# m4.sh: Using the m4 macro processor

# Strings
string=abcdA01
echo "len($string)" | m4                            #   7
echo "substr($string,4)" | m4                       # A01
echo "regexp($string,[0-1][0-1],\&Z)" | m4      # 01Z

# Arithmetic
var=99
echo "incr($var)" | m4                              #  100
```

```
echo "eval($var / 3)" | m4                          #   33

exit
```

xmessage

This X-based variant of echo pops up a message/query window on the desktop.

```
xmessage Left click to continue -button okay
```

zenity

The zenity utility is adept at displaying *GTK+* dialog widgets and very suitable for scripting purposes.

doexec

The **doexec** command enables passing an arbitrary list of arguments to a *binary executable*. In particular, passing `argv[0]` (which corresponds to $0 in a script) lets the executable be invoked by various names, and it can then carry out different sets of actions, according to the name by which it was called. What this amounts to is roundabout way of passing options to an executable.

For example, the `/usr/local/bin` directory might contain a binary called "aaa". Invoking **doexec /usr/local/bin/aaa list** would *list* all those files in the current working directory beginning with an "a", while invoking (the same executable with) **doexec /usr/local/bin/aaa delete** would *delete* those files.

The various behaviors of the executable must be defined within the code of the executable itself, analogous to something like the following in a shell script:

```
case `basename $0` in
"name1" ) do_something;;
"name2" ) do_something_else;;
"name3" ) do_yet_another_thing;;
*       ) bail_out;;
esac
```

dialog

The dialog family of tools provide a method of calling interactive "dialog" boxes from a script. The more elaborate variations of **dialog** -- **gdialog**, **Xdialog**, and **kdialog** -- actually invoke X-Windows widgets.

sox

The **sox**, or "**so**und e**x**change" command plays and performs transformations on sound files. In fact, the `/usr/bin/play` executable (now deprecated) is nothing but a shell wrapper for *sox*.

For example, **sox soundfile.wav soundfile.au** changes a WAV sound file into a (Sun audio format) AU sound file.

Shell scripts are ideally suited for batch-processing **sox** operations on sound files. For examples, see the Linux Radio Timeshift HOWTO and the MP3do Project.

Chapter 17. System and Administrative Commands

The startup and shutdown scripts in `/etc/rc.d` illustrate the uses (and usefulness) of many of these comands. These are usually invoked by *root* and used for system maintenance or emergency filesystem repairs. Use with caution, as some of these commands may damage your system if misused.

Users and Groups

users
: Show all logged on users. This is the approximate equivalent of **who -q**.

groups
: Lists the current user and the groups she belongs to. This corresponds to the $GROUPS internal variable, but gives the group names, rather than the numbers.

```
bash$ groups
bozita cdrom cdwriter audio xgrp

bash$ echo $GROUPS
501
```

chown, **chgrp**
: The **chown** command changes the ownership of a file or files. This command is a useful method that *root* can use to shift file ownership from one user to another. An ordinary user may not change the ownership of files, not even her own files. [88]

```
root# chown bozo *.txt
```

The **chgrp** command changes the *`group`* ownership of a file or files. You must be owner of the file(s) as well as a member of the destination group (or *root*) to use this operation.

```
chgrp --recursive dunderheads *.data
#  The "dunderheads" group will now own all the "*.data" files
#+ all the way down the $PWD directory tree (that's what "recursive" means).
```

useradd, **userdel**
: The **useradd** administrative command adds a user account to the system and creates a home directory for that particular user, if so specified. The corresponding **userdel** command removes a user account from the system [89] and deletes associated files.

The **adduser** command is a synonym for **useradd** and is usually a symbolic link to it.

usermod
: Modify a user account. Changes may be made to the password, group membership, expiration date, and other attributes of a given user's account. With this command, a user's password may be locked, which has the effect of disabling the account.

groupmod
: Modify a given group. The group name and/or ID number may be changed using this command.

id
: The **id** command lists the real and effective user IDs and the group IDs of the user associated with the current process. This is the counterpart to the $UID, $EUID, and $GROUPS internal Bash variables.

```
bash$ id
uid=501(bozo) gid=501(bozo) groups=501(bozo),22(cdrom),80(cdwriter),81(audio)
```

```
bash$ echo $UID
501
```

The **id** command shows the *effective* IDs only when they differ from the *real* ones.

Also see Example 9-5.

lid

The *lid* (list ID) command shows the group(s) that a given user belongs to, or alternately, the users belonging to a given group. May be invoked only by root.

```
root# lid bozo
 bozo(gid=500)


root# lid daemon
 bin(gid=1)
  daemon(gid=2)
  adm(gid=4)
  lp(gid=7)
```

who

Show all users logged on to the system.

```
bash$ who
bozo  tty1     Apr 27 17:45
 bozo  pts/0    Apr 27 17:46
 bozo  pts/1    Apr 27 17:47
 bozo  pts/2    Apr 27 17:49
```

The `-m` gives detailed information about only the current user. Passing any two arguments to **who** is the equivalent of **who -m**, as in **who am i** or **who The Man**.

```
bash$ who -m
localhost.localdomain!bozo  pts/2    Apr 27 17:49
```

whoami is similar to **who -m**, but only lists the user name.

```
bash$ whoami
bozo
```

w

Show all logged on users and the processes belonging to them. This is an extended version of **who**. The output of **w** may be piped to grep to find a specific user and/or process.

```
bash$ w | grep startx
bozo  tty1     -                 4:22pm  6:41   4.47s  0.45s  startx
```

logname

Show current user's login name (as found in `/var/run/utmp`). This is a near-equivalent to whoami, above.

```
bash$ logname
bozo

bash$ whoami
bozo
```

However . . .

```
bash$ su
Password: ......

bash# whoami
root
bash# logname
bozo
```

While **logname** prints the name of the logged in user, **whoami** gives the name of the user attached to the current process. As we have just seen, sometimes these are not the same.

su

Runs a program or script as a **s**ubstitute **u**ser. **su rjones** starts a shell as user *rjones*. A naked **su** defaults to *root*. See Example A-14.

sudo

Runs a command as *root* (or another user). This may be used in a script, thus permitting a *regular user* to run the script.

```
#!/bin/bash

# Some commands.
sudo cp /root/secretfile /home/bozo/secret
# Some more commands.
```

The file `/etc/sudoers` holds the names of users permitted to invoke **sudo**.

passwd

Sets, changes, or manages a user's password.

The **passwd** command can be used in a script, but probably *should not* be.

Example 17-1. Setting a new password

```
#!/bin/bash
#  setnew-password.sh: For demonstration purposes only.
#                      Not a good idea to actually run this script.
#  This script must be run as root.

ROOT_UID=0         # Root has $UID 0.
E_WRONG_USER=65    # Not root?

E_NOSUCHUSER=70
SUCCESS=0


if [ "$UID" -ne "$ROOT_UID" ]
then
  echo; echo "Only root can run this script."; echo
  exit $E_WRONG_USER
else
  echo
  echo "You should know better than to run this script, root."
  echo "Even root users get the blues... "
  echo
fi
```

```
username=bozo
NEWPASSWORD=security_violation

# Check if bozo lives here.
grep -q "$username" /etc/passwd
if [ $? -ne $SUCCESS ]
then
  echo "User $username does not exist."
  echo "No password changed."
  exit $E_NOSUCHUSER
fi

echo "$NEWPASSWORD" | passwd --stdin "$username"
#  The '--stdin' option to 'passwd' permits
#+ getting a new password from stdin (or a pipe).

echo; echo "User $username's password changed!"

# Using the 'passwd' command in a script is dangerous.

exit 0
```

The **passwd** command's `-l`, `-u`, and `-d` options permit locking, unlocking, and deleting a user's password. Only *root* may use these options.

ac

Show users' logged in time, as read from `/var/log/wtmp`. This is one of the GNU accounting utilities.

```
bash$ ac
        total       68.08
```

last

List *last* logged in users, as read from `/var/log/wtmp`. This command can also show remote logins.

For example, to show the last few times the system rebooted:

```
bash$ last reboot
reboot   system boot  2.6.9-1.667      Fri Feb  4 18:18          (00:02)
 reboot   system boot  2.6.9-1.667      Fri Feb  4 15:20          (01:27)
 reboot   system boot  2.6.9-1.667      Fri Feb  4 12:56          (00:49)
 reboot   system boot  2.6.9-1.667      Thu Feb  3 21:08          (02:17)
 . . .

 wtmp begins Tue Feb  1 12:50:09 2005
```

newgrp

Change user's *group ID* without logging out. This permits access to the new group's files. Since users may be members of multiple groups simultaneously, this command finds only limited use.

Kurt Glaesemann points out that the *newgrp* command could prove helpful in setting the default group permissions for files a user writes. However, the chgrp command might be more convenient for this purpose.

Terminals

tty

Echoes the name (filename) of the current user's terminal. Note that each separate *xterm* window counts as a different terminal.

```
bash$ tty
/dev/pts/1
```

stty

Shows and/or changes terminal settings. This complex command, used in a script, can control terminal behavior and the way output displays. See the info page, and study it carefully.

Example 17-2. Setting an *erase* character

```
#!/bin/bash
# erase.sh: Using "stty" to set an erase character when reading input.

echo -n "What is your name? "
read name                      #  Try to backspace
                               #+ to erase characters of input.
                               #  Problems?
echo "Your name is $name."

stty erase '#'                 #  Set "hashmark" (#) as erase character.
echo -n "What is your name? "
read name                      #  Use # to erase last character typed.
echo "Your name is $name."

exit 0

# Even after the script exits, the new key value remains set.
# Exercise: How would you reset the erase character to the default value?
```

Example 17-3. *secret password*: Turning off terminal echoing

```
#!/bin/bash
# secret-pw.sh: secret password

echo
echo -n "Enter password "
read passwd
echo "password is $passwd"
echo -n "If someone had been looking over your shoulder, "
echo "your password would have been compromised."

echo && echo  # Two line-feeds in an "and list."


stty -echo    # Turns off screen echo.
#   May also be done with
#   read -sp passwd
#   A big Thank You to Leigh James for pointing this out.

echo -n "Enter password again "
read passwd
echo
echo "password is $passwd"
echo

stty echo     # Restores screen echo.
```

```
exit 0

# Do an 'info stty' for more on this useful-but-tricky command.
```

A creative use of **stty** is detecting a user keypress (without hitting **ENTER**).

Example 17-4. Keypress detection

```
#!/bin/bash
# keypress.sh: Detect a user keypress ("hot keys").

echo

old_tty_settings=$(stty -g)   # Save old settings (why?).
stty -icanon
Keypress=$(head -c1)          # or $(dd bs=1 count=1 2> /dev/null)
                              # on non-GNU systems

echo
echo "Key pressed was \""$Keypress"\"."
echo

stty "$old_tty_settings"      # Restore old settings.

# Thanks, Stephane Chazelas.

exit 0
```

Also see Example 9-3 and Example A-43.

terminals and modes

Normally, a terminal works in the *canonical* mode. When a user hits a key, the resulting character does not immediately go to the program actually running in this terminal. A buffer local to the terminal stores keystrokes. When the user hits the **ENTER** key, this sends all the stored keystrokes to the program running. There is even a basic line editor inside the terminal.

```
bash$ stty -a
speed 9600 baud; rows 36; columns 96; line = 0;
 intr = ^C; quit = ^\; erase = ^H; kill = ^U; eof = ^D; eol = <undef>; eol2 = <undef>;
 start = ^Q; stop = ^S; susp = ^Z; rprnt = ^R; werase = ^W; lnext = ^V; flush = ^O;
 ...
 isig icanon iexten echo echoe echok -echonl -noflsh -xcase -tostop -echoprt
```

Using canonical mode, it is possible to redefine the special keys for the local terminal line editor.

```
bash$ cat > filexxx
wha<ctl-W>I<ctl-H>foo bar<ctl-U>hello world<ENTER>
<ctl-D>
bash$ cat filexxx
hello world
bash$ wc -c < filexxx
12
```

The process controlling the terminal receives only 12 characters (11 alphabetic ones, plus a newline), although the user hit 26 keys.

In non-canonical ("raw") mode, every key hit (including special editing keys such as **ctl-H**) sends a character immediately to the controlling process.

The Bash prompt disables both `icanon` and `echo`, since it replaces the basic terminal line editor with its own more elaborate one. For example, when you hit **ctl-A** at the Bash prompt, there's no **^A** echoed by the terminal, but Bash gets a **\1** character, interprets it, and moves the cursor to the begining of the line.

Stéphane Chazelas

setterm

Set certain terminal attributes. This command writes to its terminal's `stdout` a string that changes the behavior of that terminal.

```
bash$ setterm -cursor off
bash$
```

The **setterm** command can be used within a script to change the appearance of text written to `stdout`, although there are certainly better tools available for this purpose.

```
setterm -bold on
echo bold hello

setterm -bold off
echo normal hello
```

tset

Show or initialize terminal settings. This is a less capable version of **stty**.

```
bash$ tset -r
Terminal type is xterm-xfree86.
 Kill is control-U (^U).
 Interrupt is control-C (^C).
```

setserial

Set or display serial port parameters. This command must be run by *root* and is usually found in a system setup script.

```
# From /etc/pcmcia/serial script:

IRQ=`setserial /dev/$DEVICE | sed -e 's/.*IRQ: //'`
setserial /dev/$DEVICE irq 0 ; setserial /dev/$DEVICE irq $IRQ
```

getty, **agetty**

The initialization process for a terminal uses **getty** or **agetty** to set it up for login by a user. These commands are not used within user shell scripts. Their scripting counterpart is **stty**.

mesg

Enables or disables write access to the current user's terminal. Disabling access would prevent another user on the network to write to the terminal.

It can be quite annoying to have a message about ordering pizza suddenly appear in the middle of the text file you are editing. On a multi-user network, you might therefore wish to disable write access to your terminal when you need to avoid interruptions.

wall

This is an acronym for "write all," i.e., sending a message to all users at every terminal logged into the network. It is primarily a system administrator's tool, useful, for example, when warning everyone that the system will shortly go down due to a problem (see Example 19-1).

```
bash$ wall System going down for maintenance in 5 minutes!
Broadcast message from bozo (pts/1) Sun Jul  8 13:53:27 2001...

 System going down for maintenance in 5 minutes!
```

If write access to a particular terminal has been disabled with **mesg**, then **wall** cannot send a message to that terminal.

Information and Statistics

uname

Output system specifications (OS, kernel version, etc.) to `stdout`. Invoked with the `-a` option, gives verbose system info (see Example 16-5). The `-s` option shows only the OS type.

```
bash$ uname
Linux

bash$ uname -s
Linux


bash$ uname -a
Linux iron.bozo 2.6.15-1.2054_FC5 #1 Tue Mar 14 15:48:33 EST 2006
 i686 i686 i386 GNU/Linux
```

arch

Show system architecture. Equivalent to **uname -m**. See Example 11-27.

```
bash$ arch
i686

bash$ uname -m
i686
```

lastcomm

Gives information about previous commands, as stored in the `/var/account/pacct` file. Command name and user name can be specified by options. This is one of the GNU accounting utilities.

lastlog

List the last login time of all system users. This references the `/var/log/lastlog` file.

```
bash$ lastlog
root          tty1                      Fri Dec  7 18:43:21 -0700 2001
 bin                                     **Never logged in**
 daemon                                  **Never logged in**
 ...
 bozo          tty1                      Sat Dec  8 21:14:29 -0700 2001
```

```
bash$ lastlog | grep root
root          tty1                      Fri Dec  7 18:43:21 -0700 2001
```

This command will fail if the user invoking it does not have read permission for the `/var/log/lastlog` file.

lsof

List open files. This command outputs a detailed table of all currently open files and gives information about their owner, size, the processes associated with them, and more. Of course, **lsof** may be piped to grep and/or awk to parse and analyze its results.

```
bash$ lsof
COMMAND    PID    USER   FD   TYPE     DEVICE    SIZE     NODE NAME
 init        1     root  mem    REG        3,5   30748    30303 /sbin/init
 init        1     root  mem    REG        3,5   73120     8069 /lib/ld-2.1.3.so
 init        1     root  mem    REG        3,5  931668     8075 /lib/libc-2.1.3.so
 cardmgr   213     root  mem    REG        3,5   36956    30357 /sbin/cardmgr
 ...
```

The **lsof** command is a useful, if complex administrative tool. If you are unable to dismount a filesystem and get an error message that it is still in use, then running *lsof* helps determine which files are still open on that filesystem. The `-i` option lists open network socket files, and this can help trace intrusion or hack attempts.

```
bash$ lsof -an -i tcp
COMMAND  PID USER  FD  TYPE DEVICE SIZE NODE NAME
 firefox 2330 bozo  32u IPv4   9956       TCP 66.0.118.137:57596->67.112.7.104:http ...
 firefox 2330 bozo  38u IPv4  10535       TCP 66.0.118.137:57708->216.79.48.24:http ...
```

See Example 30-2 for an effective use of **lsof**.

strace

System **trace**: diagnostic and debugging tool for tracing *system calls* and signals. This command and **ltrace**, following, are useful for diagnosing why a given program or package fails to run . . . perhaps due to missing libraries or related causes.

```
bash$ strace df
execve("/bin/df", ["df"], [/* 45 vars */]) = 0
 uname({sys="Linux", node="bozo.localdomain", ...}) = 0
 brk(0)                                  = 0x804f5e4

 ...
```

This is the Linux equivalent of the Solaris **truss** command.

ltrace

Library **trace**: diagnostic and debugging tool that traces *library calls* invoked by a given command.

```
bash$ ltrace df
__libc_start_main(0x804a910, 1, 0xbfb589a4, 0x804fb70, 0x804fb68 <unfinished ...>:
 setlocale(6, "")                                 = "en_US.UTF-8"
bindtextdomain("coreutils", "/usr/share/locale") = "/usr/share/locale"
textdomain("coreutils")                          = "coreutils"
__cxa_atexit(0x804b650, 0, 0, 0x8052bf0, 0xbfb58908) = 0
getenv("DF_BLOCK_SIZE")                          = NULL

 ...
```

nc

The **nc** (*netcat*) utility is a complete toolkit for connecting to and listening to TCP and UDP ports. It is useful as a diagnostic and testing tool and as a component in simple script-based HTTP clients and servers.

```
bash$ nc localhost.localdomain 25
220 localhost.localdomain ESMTP Sendmail 8.13.1/8.13.1;
 Thu, 31 Mar 2005 15:41:35 -0700
```

A real-life usage example.

Example 17-5. Checking a remote server for *identd*

```
#! /bin/sh
## Duplicate DaveG's ident-scan thingie using netcat. Oooh, he'll be p*ssed.
## Args: target port [port port port ...]
## Hose stdout _and_ stderr together.
##
##  Advantages: runs slower than ident-scan, giving remote inetd less cause
##+ for alarm, and only hits the few known daemon ports you specify.
##  Disadvantages: requires numeric-only port args, the output sleazitude,
##+ and won't work for r-services when coming from high source ports.
# Script author: Hobbit <hobbit@avian.org>
# Used in ABS Guide with permission.

# ---------------------------------------------------
E_BADARGS=65       # Need at least two args.
TWO_WINKS=2        # How long to sleep.
THREE_WINKS=3
IDPORT=113         # Authentication "tap ident" port.
RAND1=999
RAND2=31337
TIMEOUT0=9
TIMEOUT1=8
TIMEOUT2=4
# ---------------------------------------------------

case "${2}" in
  "" ) echo "Need HOST and at least one PORT." ; exit $E_BADARGS ;;
esac

# Ping 'em once and see if they *are* running identd.
nc -z -w $TIMEOUT0 "$1" $IDPORT || \
{ echo "Oops, $1 isn't running identd." ; exit 0 ; }
#  -z scans for listening daemons.
#     -w $TIMEOUT = How long to try to connect.

# Generate a randomish base port.
RP=`expr $$ % $RAND1 + $RAND2`

TRG="$1"
shift

while test "$1" ; do
  nc -v -w $TIMEOUT1 -p ${RP} "$TRG" ${1} < /dev/null > /dev/null &
  PROC=$!
  sleep $THREE_WINKS
  echo "${1},${RP}" | nc -w $TIMEOUT2 -r "$TRG" $IDPORT 2>&1
  sleep $TWO_WINKS
```

```
# Does this look like a lamer script or what . . . ?
# ABS Guide author comments: "Ain't really all that bad . . .
#+                            kinda clever, actually."

  kill -HUP $PROC
  RP=`expr ${RP} + 1`
  shift
done

exit $?

#  Notes:
#  -----

#  Try commenting out line 30 and running this script
#+ with "localhost.localdomain 25" as arguments.

#  For more of Hobbit's 'nc' example scripts,
#+ look in the documentation:
#+ the /usr/share/doc/nc-X.XX/scripts directory.
```

And, of course, there's Dr. Andrew Tridgell's notorious one-line script in the BitKeeper Affair:

```
echo clone | nc thunk.org 5000 > e2fsprogs.dat
```

free

Shows memory and cache usage in tabular form. The output of this command lends itself to parsing, using grep, awk or **Perl**. The **procinfo** command shows all the information that **free** does, and much more.

```
bash$ free
                total       used       free     shared    buffers     cached
   Mem:         30504      28624       1880      15820       1608      16376
   -/+ buffers/cache:      10640      19864
   Swap:        68540       3128      65412
```

To show unused RAM memory:

```
bash$ free | grep Mem | awk '{ print $4 }'
1880
```

procinfo

Extract and list information and statistics from the `/proc` pseudo-filesystem. This gives a very extensive and detailed listing.

```
bash$ procinfo | grep Bootup
Bootup: Wed Mar 21 15:15:50 2001    Load average: 0.04 0.21 0.34 3/47 6829
```

lsdev

List devices, that is, show installed hardware.

```
bash$ lsdev
Device            DMA   IRQ  I/O Ports
 ------------------------------------------------
 cascade             4     2
 dma                          0080-008f
 dma1                         0000-001f
 dma2                         00c0-00df
 fpu                          00f0-00ff
 ide0                     14  01f0-01f7 03f6-03f6
 ...
```

du

Show (disk) file usage, recursively. Defaults to current working directory, unless otherwise specified.

```
bash$ du -ach
1.0k    ./wi.sh
 1.0k    ./tst.sh
 1.0k    ./random.file
 6.0k    .
 6.0k    total
```

df

Shows filesystem usage in tabular form.

```
bash$ df
Filesystem           1k-blocks      Used Available Use% Mounted on
 /dev/hda5               273262     92607    166547  36% /
 /dev/hda8               222525    123951     87085  59% /home
 /dev/hda7              1408796   1075744    261488  80% /usr
```

dmesg

Lists all system bootup messages to `stdout`. Handy for debugging and ascertaining which device drivers were installed and which system interrupts in use. The output of **dmesg** may, of course, be parsed with grep, sed, or awk from within a script.

```
bash$ dmesg | grep hda
Kernel command line: ro root=/dev/hda2
 hda: IBM-DLGA-23080, ATA DISK drive
 hda: 6015744 sectors (3080 MB) w/96KiB Cache, CHS=746/128/63
 hda: hda1 hda2 hda3 < hda5 hda6 hda7 > hda4
```

stat

Gives detailed and verbose *stat*istics on a given file (even a directory or device file) or set of files.

```
bash$ stat test.cru
  File: "test.cru"
   Size: 49970        Allocated Blocks: 100          Filetype: Regular File
   Mode: (0664/-rw-rw-r--)         Uid: (  501/ bozo)  Gid: (  501/ bozo)
 Device:  3,8   Inode: 18185     Links: 1
 Access: Sat Jun  2 16:40:24 2001
 Modify: Sat Jun  2 16:40:24 2001
 Change: Sat Jun  2 16:40:24 2001
```

If the target file does not exist, **stat** returns an error message.

```
bash$ stat nonexistent-file
nonexistent-file: No such file or directory
```

In a script, you can use **stat** to extract information about files (and filesystems) and set variables accordingly.

```
#!/bin/bash
# fileinfo2.sh

# Per suggestion of Joël Bourquard and . . .
# http://www.linuxquestions.org/questions/showthread.php?t=410766


FILENAME=testfile.txt
```

```
file_name=$(stat -c%n "$FILENAME")   # Same as "$FILENAME" of course.
file_owner=$(stat -c%U "$FILENAME")
file_size=$(stat -c%s "$FILENAME")
#  Certainly easier than using "ls -l $FILENAME"
#+ and then parsing with sed.
file_inode=$(stat -c%i "$FILENAME")
file_type=$(stat -c%F "$FILENAME")
file_access_rights=$(stat -c%A "$FILENAME")

echo "File name:          $file_name"
echo "File owner:         $file_owner"
echo "File size:          $file_size"
echo "File inode:         $file_inode"
echo "File type:          $file_type"
echo "File access rights: $file_access_rights"

exit 0

sh fileinfo2.sh

File name:          testfile.txt
File owner:         bozo
File size:          418
File inode:         1730378
File type:          regular file
File access rights: -rw-rw-r--
```

vmstat

Display virtual memory statistics.

```
bash$ vmstat
   procs                      memory    swap          io system         cpu
 r  b  w   swpd   free   buff  cache  si  so    bi    bo   in    cs  us  sy id
 0  0  0      0  11040   2636  38952   0   0    33     7  271    88   8   3 89
```

uptime

Shows how long the system has been running, along with associated statistics.

```
bash$ uptime
10:28pm  up  1:57,  3 users,  load average: 0.17, 0.34, 0.27
```

A *load average* of 1 or less indicates that the system handles processes immediately. A load average greater than 1 means that processes are being queued. When the load average gets above 3 (on a single-core processor), then system performance is significantly degraded.

hostname

Lists the system's host name. This command sets the host name in an `/etc/rc.d` setup script (`/etc/rc.d/rc.sysinit` or similar). It is equivalent to **uname -n**, and a counterpart to the $HOSTNAME internal variable.

```
bash$ hostname
localhost.localdomain

bash$ echo $HOSTNAME
localhost.localdomain
```

Similar to the **hostname** command are the **domainname**, **dnsdomainname**, **nisdomainname**, and **ypdomainname** commands. Use these to display or set the system DNS or NIS/YP domain name. Various options to **hostname** also perform these functions.

hostid

Echo a 32-bit hexadecimal numerical identifier for the host machine.

```
bash$ hostid
7f0100
```

This command allegedly fetches a "unique" serial number for a particular system. Certain product registration procedures use this number to brand a particular user license. Unfortunately, **hostid** only returns the machine network address in hexadecimal, with pairs of bytes transposed.

The network address of a typical non-networked Linux machine, is found in `/etc/hosts`.

```
bash$ cat /etc/hosts
127.0.0.1               localhost.localdomain localhost
```

As it happens, transposing the bytes of **127.0.0.1**, we get **0.127.1.0**, which translates in hex to **007f0100**, the exact equivalent of what **hostid** returns, above. There exist only a few million other Linux machines with this identical *hostid*.

sar

Invoking **sar** (System Activity Reporter) gives a very detailed rundown on system statistics. The Santa Cruz Operation ("Old" SCO) released **sar** as Open Source in June, 1999.

This command is not part of the base Linux distribution, but may be obtained as part of the sysstat utilities package, written by Sebastien Godard.

```
bash$ sar
Linux 2.4.9 (brooks.seringas.fr)        09/26/03

10:30:00          CPU     %user     %nice   %system   %iowait     %idle
10:40:00          all      2.21     10.90     65.48      0.00     21.41
10:50:00          all      3.36      0.00     72.36      0.00     24.28
11:00:00          all      1.12      0.00     80.77      0.00     18.11
Average:          all      2.23      3.63     72.87      0.00     21.27

14:32:30          LINUX RESTART

15:00:00          CPU     %user     %nice   %system   %iowait     %idle
15:10:00          all      8.59      2.40     17.47      0.00     71.54
15:20:00          all      4.07      1.00     11.95      0.00     82.98
15:30:00          all      0.79      2.94      7.56      0.00     88.71
Average:          all      6.33      1.70     14.71      0.00     77.26
```

readelf

Show information and statistics about a designated *elf* binary. This is part of the *binutils* package.

```
bash$ readelf -h /bin/bash
ELF Header:
   Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
   Class:                             ELF32
   Data:                              2's complement, little endian
   Version:                           1 (current)
   OS/ABI:                            UNIX - System V
   ABI Version:                       0
   Type:                              EXEC (Executable file)
```

```
   . . .
```

size

The **size [/path/to/binary]** command gives the segment sizes of a binary executable or archive file. This is mainly of use to programmers.

```
bash$ size /bin/bash
   text    data     bss     dec     hex filename
  495971   22496   17392  535859   82d33 /bin/bash
```

System Logs

logger

Appends a user-generated message to the system log (`/var/log/messages`). You do not have to be *root* to invoke **logger**.

```
logger Experiencing instability in network connection at 23:10, 05/21.
# Now, do a 'tail /var/log/messages'.
```

By embedding a **logger** command in a script, it is possible to write debugging information to `/var/log/messages`.

```
logger -t $0 -i Logging at line "$LINENO".
# The "-t" option specifies the tag for the logger entry.
# The "-i" option records the process ID.

# tail /var/log/message
# ...
# Jul  7 20:48:58 localhost ./test.sh[1712]: Logging at line 3.
```

logrotate

This utility manages the system log files, rotating, compressing, deleting, and/or e-mailing them, as appropriate. This keeps the `/var/log` from getting cluttered with old log files. Usually cron runs **logrotate** on a daily basis.

Adding an appropriate entry to `/etc/logrotate.conf` makes it possible to manage personal log files, as well as system-wide ones.

Stefano Falsetto has created rottlog, which he considers to be an improved version of **logrotate**.

Job Control

ps

*P*rocess *S*tatistics: lists currently executing processes by owner and PID (process ID). This is usually invoked with `ax` or `aux` options, and may be piped to grep or sed to search for a specific process (see Example 15-14 and Example 29-3).

```
bash$  ps ax | grep sendmail
295 ?      S      0:00 sendmail: accepting connections on port 25
```

To display system processes in graphical "tree" format: **ps afjx** or **ps ax --forest**.

pgrep, **pkill**

Combining the **ps** command with grep or kill.

```
bash$ ps a | grep mingetty
2212 tty2     Ss+    0:00 /sbin/mingetty tty2
 2213 tty3     Ss+    0:00 /sbin/mingetty tty3
 2214 tty4     Ss+    0:00 /sbin/mingetty tty4
 2215 tty5     Ss+    0:00 /sbin/mingetty tty5
 2216 tty6     Ss+    0:00 /sbin/mingetty tty6
 4849 pts/2    S+     0:00 grep mingetty


bash$ pgrep mingetty
2212 mingetty
 2213 mingetty
 2214 mingetty
 2215 mingetty
 2216 mingetty
```

Compare the action of **pkill** with killall.

pstree

Lists currently executing processes in "tree" format. The `-p` option shows the PIDs, as well as the process names.

top

Continuously updated display of most cpu-intensive processes. The `-b` option displays in text mode, so that the output may be parsed or accessed from a script.

```
bash$ top -b
  8:30pm  up 3 min,  3 users,  load average: 0.49, 0.32, 0.13
 45 processes: 44 sleeping, 1 running, 0 zombie, 0 stopped
 CPU states: 13.6% user,  7.3% system,  0.0% nice, 78.9% idle
 Mem:    78396K av,   65468K used,   12928K free,       0K shrd,    2352K buff
 Swap:  157208K av,       0K used,  157208K free                   37244K cached

   PID USER     PRI  NI  SIZE  RSS SHARE STAT %CPU %MEM   TIME COMMAND
   848 bozo      17   0   996  996   800 R     5.6  1.2   0:00 top
     1 root       8   0   512  512   444 S     0.0  0.6   0:04 init
     2 root       9   0     0    0     0 SW    0.0  0.0   0:00 keventd
   ...
```

nice

Run a background job with an altered priority. Priorities run from 19 (lowest) to -20 (highest). Only *root* may set the negative (higher) priorities. Related commands are **renice** and **snice**, which change the priority of a running process or processes, and **skill**, which sends a kill signal to a process or processes.

nohup

Keeps a command running even after user logs off. The command will run as a foreground process unless followed by &. If you use **nohup** within a script, consider coupling it with a wait to avoid creating an *orphan* or zombie process.

pidof

Identifies *process ID (PID)* of a running job. Since job control commands, such as kill and renice act on the *PID* of a process (not its name), it is sometimes necessary to identify that *PID*. The **pidof** command is the approximate counterpart to the $PPID internal variable.

```
bash$ pidof xclock
880
```

Example 17-6. *pidof* helps kill a process

```
#!/bin/bash
# kill-process.sh

NOPROCESS=2

process=xxxyyyzzz  # Use nonexistent process.
# For demo purposes only...
# ... don't want to actually kill any actual process with this script.
#
# If, for example, you wanted to use this script to logoff the Internet,
#     process=pppd

t=`pidof $process`       # Find pid (process id) of $process.
# The pid is needed by 'kill' (can't 'kill' by program name).

if [ -z "$t" ]           # If process not present, 'pidof' returns null.
then
  echo "Process $process was not running."
  echo "Nothing killed."
  exit $NOPROCESS
fi

kill $t                  # May need 'kill -9' for stubborn process.

# Need a check here to see if process allowed itself to be killed.
# Perhaps another " t=`pidof $process` " or ...


# This entire script could be replaced by
#        kill $(pidof -x process_name)
# or
#        killall process_name
# but it would not be as instructive.

exit 0
```

fuser

Identifies the processes (by PID) that are accessing a given file, set of files, or directory. May also be invoked with the `-k` option, which kills those processes. This has interesting implications for system security, especially in scripts preventing unauthorized users from accessing system services.

```
bash$ fuser -u /usr/bin/vim
/usr/bin/vim:         3207e(bozo)



bash$ fuser -u /dev/null
/dev/null:            3009(bozo)  3010(bozo)  3197(bozo)  3199(bozo)
```

One important application for **fuser** is when physically inserting or removing storage media, such as CD ROM disks or USB flash drives. Sometimes trying a umount fails with a device is busy error message. This means that some user(s) and/or process(es) are accessing the device. An **fuser -um /dev/device_name** will clear up the mystery, so you can kill any relevant processes.

```
bash$ umount /mnt/usbdrive
umount: /mnt/usbdrive: device is busy


```

```
bash$ fuser -um /dev/usbdrive
/mnt/usbdrive:        1772c(bozo)

bash$ kill -9 1772
bash$ umount /mnt/usbdrive
```

The **fuser** command, invoked with the `-n` option identifies the processes accessing a *port*. This is especially useful in combination with nmap.

```
root# nmap localhost.localdomain
PORT     STATE SERVICE
 25/tcp   open  smtp



root# fuser -un tcp 25
25/tcp:               2095(root)

root# ps ax | grep 2095 | grep -v grep
2095 ?        Ss     0:00 sendmail: accepting connections
```

cron

Administrative program scheduler, performing such duties as cleaning up and deleting system log files and updating the slocate database. This is the *superuser* version of at (although each user may have their own `crontab` file which can be changed with the **crontab** command). It runs as a daemon and executes scheduled entries from `/etc/crontab`.

Some flavors of Linux run **crond**, Matthew Dillon's version of **cron**.

Process Control and Booting

init

The **init** command is the parent of all processes. Called in the final step of a bootup, **init** determines the runlevel of the system from `/etc/inittab`. Invoked by its alias **telinit**, and by *root* only.

telinit

Symlinked to **init**, this is a means of changing the system runlevel, usually done for system maintenance or emergency filesystem repairs. Invoked only by *root*. This command can be dangerous -- be certain you understand it well before using!

runlevel

Shows the current and last runlevel, that is, whether the system is halted (runlevel `0`), in single-user mode (`1`), in multi-user mode (`2` or `3`), in X Windows (`5`), or rebooting (`6`). This command accesses the `/var/run/utmp` file.

halt, **shutdown**, **reboot**

Command set to shut the system down, usually just prior to a power down.

On some Linux distros, the **halt** command has 755 permissions, so it can be invoked by a non-root user. A careless *halt* in a terminal or a script may shut down the system!

service

Starts or stops a system *service*. The startup scripts in `/etc/init.d` and `/etc/rc.d` use this command to start services at bootup.

```
root# /sbin/service iptables stop
Flushing firewall rules:                                      [  OK  ]
 Setting chains to policy ACCEPT: filter                      [  OK  ]
 Unloading iptables modules:                                  [  OK  ]
```

Network

nmap

Network **map**per and port scanner. This command scans a server to locate open ports and the services associated with those ports. It can also report information about packet filters and firewalls. This is an important security tool for locking down a network against hacking attempts.

```
#!/bin/bash

SERVER=$HOST                           # localhost.localdomain (127.0.0.1).
PORT_NUMBER=25                         # SMTP port.

nmap $SERVER | grep -w "$PORT_NUMBER"  # Is that particular port open?
#              grep -w matches whole words only,
#+             so this wouldn't match port 1025, for example.

exit 0

# 25/tcp     open        smtp
```

ifconfig

Network *interface configuration* and tuning utility.

```
bash$ ifconfig -a
lo        Link encap:Local Loopback
           inet addr:127.0.0.1  Mask:255.0.0.0
           UP LOOPBACK RUNNING  MTU:16436  Metric:1
           RX packets:10 errors:0 dropped:0 overruns:0 frame:0
           TX packets:10 errors:0 dropped:0 overruns:0 carrier:0
           collisions:0 txqueuelen:0
           RX bytes:700 (700.0 b)  TX bytes:700 (700.0 b)
```

The **ifconfig** command is most often used at bootup to set up the interfaces, or to shut them down when rebooting.

```
# Code snippets from /etc/rc.d/init.d/network

# ...

# Check that networking is up.
[ ${NETWORKING} = "no" ] && exit 0

[ -x /sbin/ifconfig ] || exit 0

# ...

for i in $interfaces ; do
  if ifconfig $i 2>/dev/null | grep -q "UP" >/dev/null 2>&1 ; then
    action "Shutting down interface $i: " ./ifdown $i boot
  fi
#  The GNU-specific "-q" option to "grep" means "quiet", i.e.,
#+ producing no output.
#  Redirecting output to /dev/null is therefore not strictly necessary.
```

```
# ...

echo "Currently active devices:"
echo `/sbin/ifconfig | grep ^[a-z] | awk '{print $1}'`
#                            ^^^^^  should be quoted to prevent globbing.
#  The following also work.
#    echo $(/sbin/ifconfig | awk '/^[a-z]/ { print $1 })'
#    echo $(/sbin/ifconfig | sed -e 's/ .*//')
#  Thanks, S.C., for additional comments.
```

See also Example 32-6.

netstat

Show current network statistics and information, such as routing tables and active connections. This utility accesses information in `/proc/net` (Chapter 29). See Example 29-4.

netstat -r is equivalent to route.

```
bash$ netstat
Active Internet connections (w/o servers)
 Proto Recv-Q Send-Q Local Address           Foreign Address         State
 Active UNIX domain sockets (w/o servers)
 Proto RefCnt Flags       Type       State         I-Node Path
 unix  11     [ ]         DGRAM                    906    /dev/log
 unix  3      [ ]         STREAM     CONNECTED     4514   /tmp/.X11-unix/X0
 unix  3      [ ]         STREAM     CONNECTED     4513
 . . .
```

A **netstat -lptu** shows sockets that are listening to ports, and the associated processes. This can be useful for determining whether a computer has been hacked or compromised.

iwconfig

This is the command set for configuring a wireless network. It is the wireless equivalent of **ifconfig**, above.

ip

General purpose utility for setting up, changing, and analyzing *IP* (Internet Protocol) networks and attached devices. This command is part of the *iproute2* package.

```
bash$ ip link show
1: lo: <LOOPBACK,UP> mtu 16436 qdisc noqueue
     link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
 2: eth0: <BROADCAST,MULTICAST> mtu 1500 qdisc pfifo_fast qlen 1000
     link/ether 00:d0:59:ce:af:da brd ff:ff:ff:ff:ff:ff
 3: sit0: <NOARP> mtu 1480 qdisc noop
     link/sit 0.0.0.0 brd 0.0.0.0


bash$ ip route list
169.254.0.0/16 dev lo  scope link
```

Or, in a script:

```
#!/bin/bash
# Script by Juan Nicolas Ruiz
# Used with his kind permission.

# Setting up (and stopping) a GRE tunnel.
```

```
# --- start-tunnel.sh ---

LOCAL_IP="192.168.1.17"
REMOTE_IP="10.0.5.33"
OTHER_IFACE="192.168.0.100"
REMOTE_NET="192.168.3.0/24"

/sbin/ip tunnel add netb mode gre remote $REMOTE_IP \
  local $LOCAL_IP ttl 255
/sbin/ip addr add $OTHER_IFACE dev netb
/sbin/ip link set netb up
/sbin/ip route add $REMOTE_NET dev netb

exit 0  #############################################

# --- stop-tunnel.sh ---

REMOTE_NET="192.168.3.0/24"

/sbin/ip route del $REMOTE_NET dev netb
/sbin/ip link set netb down
/sbin/ip tunnel del netb

exit 0
```

route

Show info about or make changes to the kernel routing table.

```
bash$ route
Destination     Gateway         Genmask         Flags   MSS Window  irtt Iface
 pm3-67.bozosisp *               255.255.255.255 UH       40 0          0 ppp0
 127.0.0.0       *               255.0.0.0       U        40 0          0 lo
 default         pm3-67.bozosisp 0.0.0.0         UG       40 0          0 ppp0
```

iptables

The **iptables** command set is a packet filtering tool used mainly for such security purposes as setting up network firewalls. This is a complex tool, and a detailed explanation of its use is beyond the scope of this document. Oskar Andreasson's tutorial is a reasonable starting point.

See also shutting down *iptables* and Example 30-2.

chkconfig

Check network and system configuration. This command lists and manages the network and system services started at bootup in the `/etc/rc?.d` directory.

Originally a port from IRIX to Red Hat Linux, **chkconfig** may not be part of the core installation of some Linux flavors.

```
bash$ chkconfig --list
atd             0:off   1:off   2:off   3:on    4:on    5:on    6:off
 rwhod           0:off   1:off   2:off   3:off   4:off   5:off   6:off
 ...
```

tcpdump

Network packet "sniffer." This is a tool for analyzing and troubleshooting traffic on a network by dumping packet headers that match specified criteria.

Dump ip packet traffic between hosts *bozoville* and *caduceus*:

```
bash$ tcpdump ip host bozoville and caduceus
```

Of course, the output of **tcpdump** can be parsed with certain of the previously discussed text processing utilities.

Filesystem

mount

Mount a filesystem, usually on an external device, such as a floppy or CDROM. The file `/etc/fstab` provides a handy listing of available filesystems, partitions, and devices, including options, that may be automatically or manually mounted. The file `/etc/mtab` shows the currently mounted filesystems and partitions (including the virtual ones, such as `/proc`).

mount -a mounts all filesystems and partitions listed in `/etc/fstab`, except those with a `noauto` option. At bootup, a startup script in `/etc/rc.d` (`rc.sysinit` or something similar) invokes this to get everything mounted.

```
mount -t iso9660 /dev/cdrom /mnt/cdrom
# Mounts CD ROM. ISO 9660 is a standard CD ROM filesystem.
mount /mnt/cdrom
# Shortcut, if /mnt/cdrom listed in /etc/fstab
```

The versatile *mount* command can even mount an ordinary file on a block device, and the file will act as if it were a filesystem. *Mount* accomplishes that by associating the file with a loopback device. One application of this is to mount and examine an ISO9660 filesystem image before burning it onto a CDR. [90]

Example 17-7. Checking a CD image

```
# As root...

mkdir /mnt/cdtest  # Prepare a mount point, if not already there.

mount -r -t iso9660 -o loop cd-image.iso /mnt/cdtest   # Mount the image.
#                  "-o loop" option equivalent to "losetup /dev/loop0"
cd /mnt/cdtest     # Now, check the image.
ls -alR            # List the files in the directory tree there.
                   # And so forth.
```

umount

Unmount a currently mounted filesystem. Before physically removing a previously mounted floppy or CDROM disk, the device must be **umount**ed, else filesystem corruption may result.

```
umount /mnt/cdrom
# You may now press the eject button and safely remove the disk.
```

The **automount** utility, if properly installed, can mount and unmount floppies or CDROM disks as they are accessed or removed. On "multispindle" laptops with swappable floppy and optical drives, this can cause problems, however.

gnome-mount

The newer Linux distros have deprecated **mount** and **umount**. The successor, for command-line mounting of removable storage devices, is **gnome-mount**. It can take the `-d` option to mount a device file by its listing in `/dev`.

For example, to mount a USB flash drive:

```
bash$ gnome-mount -d /dev/sda1
gnome-mount 0.4


bash$ df
. . .
 /dev/sda1                63584     12034     51550  19% /media/disk
```

sync

Forces an immediate write of all updated data from buffers to hard drive (synchronize drive with buffers). While not strictly necessary, a **sync** assures the sys admin or user that the data just changed will survive a sudden power failure. In the olden days, a **`sync; sync`** (twice, just to make absolutely sure) was a useful precautionary measure before a system reboot.

At times, you may wish to force an immediate buffer flush, as when securely deleting a file (see Example 16-61) or when the lights begin to flicker.

losetup

Sets up and configures loopback devices.

Example 17-8. Creating a filesystem in a file

```
SIZE=1000000  # 1 meg

head -c $SIZE < /dev/zero > file  # Set up file of designated size.
losetup /dev/loop0 file           # Set it up as loopback device.
mke2fs /dev/loop0                 # Create filesystem.
mount -o loop /dev/loop0 /mnt     # Mount it.

# Thanks, S.C.
```

mkswap

Creates a swap partition or file. The swap area must subsequently be enabled with **swapon**.

swapon, **swapoff**

Enable / disable swap partitition or file. These commands usually take effect at bootup and shutdown.

mke2fs

Create a Linux *ext2* filesystem. This command must be invoked as *root*.

Example 17-9. Adding a new hard drive

```
#!/bin/bash

# Adding a second hard drive to system.
# Software configuration. Assumes hardware already mounted.
# From an article by the author of the ABS Guide.
# In issue #38 of _Linux Gazette_, http://www.linuxgazette.com.

ROOT_UID=0     # This script must be run as root.
E_NOTROOT=67   # Non-root exit error.
```

```
if [ "$UID" -ne "$ROOT_UID" ]
then
  echo "Must be root to run this script."
  exit $E_NOTROOT
fi

# Use with extreme caution!
# If something goes wrong, you may wipe out your current filesystem.


NEWDISK=/dev/hdb         # Assumes /dev/hdb vacant. Check!
MOUNTPOINT=/mnt/newdisk  # Or choose another mount point.


fdisk $NEWDISK
mke2fs -cv $NEWDISK1   # Check for bad blocks (verbose output).
#  Note:           ^     /dev/hdb1, *not* /dev/hdb!
mkdir $MOUNTPOINT
chmod 777 $MOUNTPOINT  # Makes new drive accessible to all users.


# Now, test ...
# mount -t ext2 /dev/hdb1 /mnt/newdisk
# Try creating a directory.
# If it works, umount it, and proceed.

# Final step:
# Add the following line to /etc/fstab.
# /dev/hdb1  /mnt/newdisk  ext2  defaults  1 1

exit
```

See also Example 17-8 and Example 31-3.

mkdosfs

Create a DOS *FAT* filesystem.

tune2fs

Tune *ext2* filesystem. May be used to change filesystem parameters, such as maximum mount count. This must be invoked as *root*.

This is an extremely dangerous command. Use it at your own risk, as you may inadvertently destroy your filesystem.

dumpe2fs

Dump (list to `stdout`) very verbose filesystem info. This must be invoked as *root*.

```
root# dumpe2fs /dev/hda7 | grep 'ount count'
dumpe2fs 1.19, 13-Jul-2000 for EXT2 FS 0.5b, 95/08/09
 Mount count:              6
 Maximum mount count:      20
```

hdparm

List or change hard disk parameters. This command must be invoked as *root*, and it may be dangerous if misused.

fdisk

Create or change a partition table on a storage device, usually a hard drive. This command must be invoked as *root*.

Use this command with extreme caution. If something goes wrong, you may destroy an existing filesystem.

fsck, **e2fsck**, **debugfs**

Filesystem check, repair, and debug command set.

fsck: a front end for checking a UNIX filesystem (may invoke other utilities). The actual filesystem type generally defaults to *ext2*.

e2fsck: ext2 filesystem checker.

debugfs: ext2 filesystem debugger. One of the uses of this versatile, but dangerous command is to (attempt to) recover deleted files. For advanced users only!

All of these should be invoked as *root*, and they can damage or destroy a filesystem if misused.

badblocks

Checks for bad blocks (physical media flaws) on a storage device. This command finds use when formatting a newly installed hard drive or testing the integrity of backup media. [91] As an example, **badblocks /dev/fd0** tests a floppy disk.

The **badblocks** command may be invoked destructively (overwrite all data) or in non-destructive read-only mode. If *root user* owns the device to be tested, as is generally the case, then *root* must invoke this command.

lsusb, **usbmodules**

The **lsusb** command lists all USB (Universal Serial Bus) buses and the devices hooked up to them.

The **usbmodules** command outputs information about the driver modules for connected USB devices.

```
bash$ lsusb
Bus 001 Device 001: ID 0000:0000
 Device Descriptor:
   bLength                18
   bDescriptorType         1
   bcdUSB               1.00
   bDeviceClass            9 Hub
   bDeviceSubClass         0
   bDeviceProtocol         0
   bMaxPacketSize0         8
   idVendor           0x0000
   idProduct          0x0000

   . . .
```

lspci

Lists *pci* busses present.

```
bash$ lspci
00:00.0 Host bridge: Intel Corporation 82845 845
 (Brookdale) Chipset Host Bridge (rev 04)
 00:01.0 PCI bridge: Intel Corporation 82845 845
 (Brookdale) Chipset AGP Bridge (rev 04)
 00:1d.0 USB Controller: Intel Corporation 82801CA/CAM USB (Hub #1) (rev 02)
 00:1d.1 USB Controller: Intel Corporation 82801CA/CAM USB (Hub #2) (rev 02)
 00:1d.2 USB Controller: Intel Corporation 82801CA/CAM USB (Hub #3) (rev 02)
 00:1e.0 PCI bridge: Intel Corporation 82801 Mobile PCI Bridge (rev 42)
```

```
 . . .
```

mkbootdisk

Creates a boot floppy which can be used to bring up the system if, for example, the MBR (master boot record) becomes corrupted. Of special interest is the `--iso` option, which uses **mkisofs** to create a bootable *ISO9660* filesystem image suitable for burning a bootable CDR.

The **mkbootdisk** command is actually a Bash script, written by Erik Troan, in the `/sbin` directory.

mkisofs

Creates an *ISO9660* filesystem suitable for a CDR image.

chroot

CHange ROOT directory. Normally commands are fetched from $PATH, relative to `/`, the default *root directory*. This changes the *root* directory to a different one (and also changes the working directory to there). This is useful for security purposes, for instance when the system administrator wishes to restrict certain users, such as those telnetting in, to a secured portion of the filesystem (this is sometimes referred to as confining a guest user to a "chroot jail"). Note that after a **chroot**, the execution path for system binaries is no longer valid.

A **`chroot /opt`** would cause references to `/usr/bin` to be translated to `/opt/usr/bin`. Likewise, **`chroot /aaa/bbb /bin/ls`** would redirect future instances of **ls** to `/aaa/bbb` as the base directory, rather than `/` as is normally the case. An **alias XX 'chroot /aaa/bbb ls'** in a user's `~/.bashrc` effectively restricts which portion of the filesystem she may run command "XX" on.

The **chroot** command is also handy when running from an emergency boot floppy (**chroot** to `/dev/fd0`), or as an option to **lilo** when recovering from a system crash. Other uses include installation from a different filesystem (an rpm option) or running a readonly filesystem from a CD ROM. Invoke only as *root*, and use with care.

It might be necessary to copy certain system files to a *chrooted* directory, since the normal `$PATH` can no longer be relied upon.

lockfile

This utility is part of the **procmail** package (www.procmail.org). It creates a *lock file*, a *semaphore* that controls access to a file, device, or resource.

`Definition:` A *semaphore* is a flag or signal. (The usage originated in railroading, where a colored flag, lantern, or striped movable arm *semaphore* indicated whether a particular track was in use and therefore unavailable for another train.) A UNIX process can check the appropriate semaphore to determine whether a particular resource is available/accessible.

The lock file serves as a flag that this particular file, device, or resource is in use by a process (and is therefore "busy"). The presence of a lock file permits only restricted access (or no access) to other processes.

```
lockfile /home/bozo/lockfiles/$0.lock
# Creates a write-protected lockfile prefixed with the name of the script.

lockfile /home/bozo/lockfiles/${0##*/}.lock
# A safer version of the above, as pointed out by E. Choroba.
```

Lock files are used in such applications as protecting system mail folders from simultaneously being changed by multiple users, indicating that a modem port is being accessed, and showing that an instance of Firefox is using its cache. Scripts may check for the existence of a lock file created by a certain process to check if that process is running. Note that if a script attempts to create a lock file that already exists, the script will likely hang.

Normally, applications create and check for lock files in the `/var/lock` directory. [92] A script can test for the presence of a lock file by something like the following.

```
appname=xyzip
# Application "xyzip" created lock file "/var/lock/xyzip.lock".

if [ -e "/var/lock/$appname.lock" ]
then   #+ Prevent other programs & scripts
       #  from accessing files/resources used by xyzip.
  ...
```

flock

Much less useful than the **lockfile** command is **flock**. It sets an "advisory" lock on a file and then executes a command while the lock is on. This is to prevent any other process from setting a lock on that file until completion of the specified command.

```
flock $0 cat $0 > lockfile__$0
#  Set a lock on the script the above line appears in,
#+ while listing the script to stdout.
```

Unlike **lockfile**, **flock** does *not* automatically create a lock file.

mknod

Creates block or character device files (may be necessary when installing new hardware on the system). The **MAKEDEV** utility has virtually all of the functionality of **mknod**, and is easier to use.

MAKEDEV

Utility for creating device files. It must be run as *root*, and in the `/dev` directory. It is a sort of advanced version of **mknod**.

tmpwatch

Automatically deletes files which have not been accessed within a specified period of time. Usually invoked by cron to remove stale log files.

Backup

dump, **restore**

The **dump** command is an elaborate filesystem backup utility, generally used on larger installations and networks. [93] It reads raw disk partitions and writes a backup file in a binary format. Files to be backed up may be saved to a variety of storage media, including disks and tape drives. The **restore** command restores backups made with **dump**.

fdformat

Perform a low-level format on a floppy disk (`/dev/fd0*`).

System Resources

ulimit

Sets an *upper limit* on use of system resources. Usually invoked with the `-f` option, which sets a limit on file size (**ulimit -f 1000** limits files to 1 meg maximum). [94] The `-t` option limits the coredump size (**ulimit -c 0** eliminates coredumps). Normally, the value of **ulimit** would be set in `/etc/profile` and/or `~/.bash_profile` (see Appendix H).

Judicious use of **ulimit** can protect a system against the dreaded *fork bomb*.

```
#!/bin/bash
# This script is for illustrative purposes only.
# Run it at your own peril -- it WILL freeze your system.

while true  #  Endless loop.
do
  $0 &      #  This script invokes itself . . .
            #+ forks an infinite number of times . . .
            #+ until the system freezes up because all resources exhausted.
done        #  This is the notorious "sorcerer's appentice" scenario.

exit 0      #  Will not exit here, because this script will never terminate.
```

A **ulimit -Hu XX** (where *XX* is the user process limit) in `/etc/profile` would abort this script when it exceeded the preset limit.

quota
: Display user or group disk quotas.

setquota
: Set user or group disk quotas from the command-line.

umask
: User file creation permissions *mask*. Limit the default file attributes for a particular user. All files created by that user take on the attributes specified by **umask**. The (octal) value passed to **umask** defines the file permissions *disabled*. For example, **umask 022** ensures that new files will have at most 755 permissions (777 NAND 022). [95] Of course, the user may later change the attributes of particular files with chmod. The usual practice is to set the value of **umask** in `/etc/profile` and/or `~/.bash_profile` (see Appendix H).

Example 17-10. Using *umask* to hide an output file from prying eyes

```
#!/bin/bash
# rot13a.sh: Same as "rot13.sh" script, but writes output to "secure" file.

# Usage: ./rot13a.sh filename
# or     ./rot13a.sh <filename
# or     ./rot13a.sh and supply keyboard input (stdin)

umask 177               #  File creation mask.
                        #  Files created by this script
                        #+ will have 600 permissions.

OUTFILE=decrypted.txt   #  Results output to file "decrypted.txt"
                        #+ which can only be read/written
                        #  by invoker of script (or root).

cat "$@" | tr 'a-zA-Z' 'n-za-mN-ZA-M' > $OUTFILE
#    ^^ Input from stdin or a file.   ^^^^^^^^^^ Output redirected to file.

exit 0
```

rdev
: Get info about or make changes to root device, swap space, or video mode. The functionality of **rdev** has generally been taken over by **lilo**, but **rdev** remains useful for setting up a ram disk. This is a dangerous command, if misused.

Modules

lsmod

List installed kernel modules.

```
bash$ lsmod
Module                  Size  Used by
 autofs                  9456   2  (autoclean)
 opl3                   11376   0
 serial_cs               5456   0  (unused)
 sb                     34752   0
 uart401                 6384   0  [sb]
 sound                  58368   0  [opl3 sb uart401]
 soundlow                 464   0  [sound]
 soundcore               2800   6  [sb sound]
 ds                      6448   2  [serial_cs]
 i82365                 22928   2
 pcmcia_core            45984   0  [serial_cs ds i82365]
```

Doing a **cat /proc/modules** gives the same information.

insmod

Force installation of a kernel module (use **modprobe** instead, when possible). Must be invoked as *root*.

rmmod

Force unloading of a kernel module. Must be invoked as *root*.

modprobe

Module loader that is normally invoked automatically in a startup script. Must be invoked as *root*.

depmod

Creates module dependency file. Usually invoked from a startup script.

modinfo

Output information about a loadable module.

```
bash$ modinfo hid
filename:    /lib/modules/2.4.20-6/kernel/drivers/usb/hid.o
 description: "USB HID support drivers"
 author:      "Andreas Gal, Vojtech Pavlik <vojtech@suse.cz>"
 license:     "GPL"
```

Miscellaneous

env

Runs a program or script with certain environmental variables set or changed (without changing the overall system environment). The `[varname=xxx]` permits changing the environmental variable `varname` for the duration of the script. With no options specified, this command lists all the environmental variable settings. [96]

The first line of a script (the "sha-bang" line) may use **env** when the path to the shell or interpreter is unknown.

```
#! /usr/bin/env perl

print "This Perl script will run,\n";
print "even when I don't know where to find Perl.\n";
```

```
# Good for portable cross-platform scripts,
# where the Perl binaries may not be in the expected place.
# Thanks, S.C.
```

Or even ...

```
#!/bin/env bash
# Queries the $PATH enviromental variable for the location of bash.
# Therefore ...
# This script will run where Bash is not in its usual place, in /bin.
...
```

ldd

Show shared lib dependencies for an executable file.

```
bash$ ldd /bin/ls
libc.so.6 => /lib/libc.so.6 (0x4000c000)
/lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x80000000)
```

watch

Run a command repeatedly, at specified time intervals.

The default is two-second intervals, but this may be changed with the `-n` option.

```
watch -n 5 tail /var/log/messages
# Shows tail end of system log, /var/log/messages, every five seconds.
```

Unfortunately, piping the output of **watch command** to grep does not work.

strip

Remove the debugging symbolic references from an executable binary. This decreases its size, but makes debugging it impossible.

This command often occurs in a Makefile, but rarely in a shell script.

nm

List symbols in an unstripped compiled binary.

xrandr

Command-line tool for manipulating the root window of the screen.

Example 17-11. *Backlight*: changes the brightness of the (laptop) screen backlight

```
#!/bin/bash
# backlight.sh
# reldate 02dec2011

#  A bug in Fedora Core 16/17 messes up the keyboard backlight controls.
#  This script is a quick-n-dirty workaround, essentially a shell wrapper
#+ for xrandr. It gives more control than on-screen sliders and widgets.

OUTPUT=$(xrandr | grep LV | awk '{print $1}')   # Get display name!
INCR=.05      # For finer-grained control, set INCR to .03 or .02.

old_brightness=$(xrandr --verbose | grep rightness | awk '{ print $2 }')


if [ -z "$1" ]
then
```

```
  bright=1    # If no command-line arg, set brightness to 1.0 (default).

  else
    if [ "$1" = "+" ]
    then
      bright=$(echo "scale=2; $old_brightness + $INCR" | bc)   # +.05

  else
    if [ "$1" = "-" ]
    then
      bright=$(echo "scale=2; $old_brightness - $INCR" | bc)   # -.05

  else
    if [ "$1" = "#" ]   # Echoes current brightness; does not change it.
    then
      bright=$old_brightness

  else
    if [[ "$1" = "h" || "$1" = "H" ]]
    then
      echo
      echo "Usage:"
      echo "$0 [No args]    Sets/resets brightness to default (1.0)."
      echo "$0 +            Increments brightness by 0.5."
      echo "$0 -            Decrements brightness by 0.5."
      echo "$0 #            Echoes current brightness without changing it."
      echo "$0 N (number)   Sets brightness to N (useful range .7 - 1.2)."
      echo "$0 h [H]        Echoes this help message."
      echo "$0 any-other    Gives xrandr usage message."

      bright=$old_brightness

  else
    bright="$1"

      fi
     fi
    fi
  fi
fi


xrandr --output "$OUTPUT" --brightness "$bright"   # See xrandr manpage.
                                                   # As root!
E_CHANGE0=$?
echo "Current brightness = $bright"

exit $E_CHANGE0


# =========== Or, alternately . . . ==================== #

#!/bin/bash
# backlight2.sh
# reldate 20jun2012

#  A bug in Fedora Core 16/17 messes up the keyboard backlight controls.
#  This is a quick-n-dirty workaround, an alternate to backlight.sh.

target_dir=\
/sys/devices/pci0000:00/0000:00:01.0/0000:01:00.0/backlight/acpi_video0
# Hardware directory.
```

```
actual_brightness=$(cat $target_dir/actual_brightness)
max_brightness=$(cat $target_dir/max_brightness)
Brightness=$target_dir/brightness

let "req_brightness = actual_brightness"   # Requested brightness.

if [ "$1" = "-" ]
then     # Decrement brightness 1 notch.
  let "req_brightness = $actual_brightness - 1"
else
  if [ "$1" = "+" ]
  then   # Increment brightness 1 notch.
    let "req_brightness = $actual_brightness + 1"
   fi
fi

if [ $req_brightness -gt $max_brightness ]
then
  req_brightness=$max_brightness
fi   # Do not exceed max. hardware design brightness.

echo

echo "Old brightness = $actual_brightness"
echo "Max brightness = $max_brightness"
echo "Requested brightness = $req_brightness"
echo

# =====================================
echo $req_brightness > $Brightness
# Must be root for this to take effect.
E_CHANGE1=$?   # Successful?
# =====================================

if [ "$?" -eq 0 ]
then
  echo "Changed brightness!"
else
  echo "Failed to change brightness!"
fi

act_brightness=$(cat $Brightness)
echo "Actual brightness = $act_brightness"

scale0=2
sf=100 # Scale factor.
pct=$(echo "scale=$scale0; $act_brightness / $max_brightness * $sf" | bc)
echo "Percentage brightness = $pct%"

exit $E_CHANGE1
```

rdist

Remote distribution client: synchronizes, clones, or backs up a file system on a remote server.

17.1. Analyzing a System Script

Using our knowledge of administrative commands, let us examine a system script. One of the shortest and simplest to understand scripts is "killall," [97] used to suspend running processes at system shutdown.

Example 17-12. ***killall*****, from `/etc/rc.d/init.d`**

```
#!/bin/sh

# --> Comments added by the author of this document marked by "# -->".

# --> This is part of the 'rc' script package
# --> by Miquel van Smoorenburg, <miquels@drinkel.nl.mugnet.org>.

# --> This particular script seems to be Red Hat / FC specific
# --> (may not be present in other distributions).

#  Bring down all unneeded services that are still running
#+ (there shouldn't be any, so this is just a sanity check)

for i in /var/lock/subsys/*; do
        # --> Standard for/in loop, but since "do" is on same line,
        # --> it is necessary to add ";".
        # Check if the script is there.
        [ ! -f $i ] && continue
        # --> This is a clever use of an "and list", equivalent to:
        # --> if [ ! -f "$i" ]; then continue

        # Get the subsystem name.
        subsys=${i#/var/lock/subsys/}
        # --> Match variable name, which, in this case, is the file name.
        # --> This is the exact equivalent of subsys=`basename $i`.

        # -->  It gets it from the lock file name
        # -->+ (if there is a lock file,
        # -->+ that's proof the process has been running).
        # -->  See the "lockfile" entry, above.


        # Bring the subsystem down.
        if [ -f /etc/rc.d/init.d/$subsys.init ]; then
           /etc/rc.d/init.d/$subsys.init stop
        else
           /etc/rc.d/init.d/$subsys stop
        # -->  Suspend running jobs and daemons.
        # -->  Note that "stop" is a positional parameter,
        # -->+ not a shell builtin.
        fi
done
```

That wasn't so bad. Aside from a little fancy footwork with variable matching, there is no new material there.

Exercise 1. In `/etc/rc.d/init.d`, analyze the **halt** script. It is a bit longer than **killall**, but similar in concept. Make a copy of this script somewhere in your home directory and experiment with it (do *not* run it as *root*). Do a simulated run with the `-vn` flags (**`sh -vn scriptname`**). Add extensive comments. Change the commands to echos.

Exercise 2. Look at some of the more complex scripts in `/etc/rc.d/init.d`. Try to understand at least portions of them. Follow the above procedure to analyze them. For some additional insight, you might also examine the file `sysvinitfiles` in `/usr/share/doc/initscripts-?.??`, which is part of the "initscripts" documentation.

Part 5. Advanced Topics

At this point, we are ready to delve into certain of the difficult and unusual aspects of scripting. Along the way, we will attempt to "push the envelope" in various ways and examine *boundary conditions* (what happens when we move into uncharted territory?).

Table of Contents

Chapter 18. Regular Expressions

. . . the intellectual activity associated with software development is largely one of gaining insight.

--Stowe Boyd

To fully utilize the power of shell scripting, you need to master Regular Expressions. Certain commands and utilities commonly used in scripts, such as grep, expr, sed and awk, interpret and use REs. As of version 3, Bash has acquired its own RE-match operator: **=~**.

18.1. A Brief Introduction to Regular Expressions

An expression is a string of characters. Those characters having an interpretation above and beyond their literal meaning are called *metacharacters*. A quote symbol, for example, may denote speech by a person, *ditto*, or a meta-meaning [98] for the symbols that follow. Regular Expressions are sets of characters and/or metacharacters that match (or specify) patterns.

A Regular Expression contains one or more of the following:

- *A character set*. These are the characters retaining their literal meaning. The simplest type of Regular Expression consists *only* of a character set, with no metacharacters.
- An anchor. These designate (*anchor*) the position in the line of text that the RE is to match. For example, ^, and $ are anchors.
- *Modifiers*. These expand or narrow (*modify*) the range of text the RE is to match. Modifiers include the asterisk, brackets, and the backslash.

The main uses for Regular Expressions (*RE*s) are text searches and string manipulation. An RE *matches* a single character or a set of characters -- a string or a part of a string.

- The asterisk -- * -- matches any number of repeats of the character string or RE preceding it, including *zero* instances.

  "1133*" matches `11 + one or more 3's`: `113`, `1133`, `1133333`, and so forth.
- The *dot* -- . -- matches any one character, except a newline. [99]

  "13." matches `13 + at least one of any character (including a space)`: `1133`, `11333`, but not `13` (additional character missing).

  See Example 16-18 for a demonstration of *dot single-character* matching.
- The caret -- ^ -- matches the beginning of a line, but sometimes, depending on context, negates the meaning of a set of characters in an RE.
- The dollar sign -- $ -- at the end of an RE matches the end of a line.

  "XXX$" matches XXX at the end of a line.

"^$" matches blank lines.

- Brackets -- [...] -- enclose a set of characters to match in a single RE.

  "[xyz]" matches any one of the characters `x`, `y`, or `z`.

  "[c-n]" matches any one of the characters in the range `c` to `n`.

  "[B-Pk-y]" matches any one of the characters in the ranges `B` to `P` and `k` to `y`.

  "[a-z0-9]" matches any single lowercase letter or any digit.

  "[^b-d]" matches any character *except* those in the range `b` to `d`. This is an instance of ^ negating or inverting the meaning of the following RE (taking on a role similar to ! in a different context).

  Combined sequences of bracketed characters match common word patterns. "[Yy][Ee][Ss]" matches `yes`, `Yes`, `YES`, `yEs`, and so forth. "[0-9][0-9][0-9]-[0-9][0-9]-[0-9][0-9][0-9][0-9]" matches any Social Security number.

- The backslash -- \ -- escapes a special character, which means that character gets interpreted literally (and is therefore no longer *special*).

  A "\$" reverts back to its literal meaning of "$", rather than its RE meaning of end-of-line. Likewise a "\\" has the literal meaning of "\".

- Escaped "angle brackets" -- \<...\> -- mark word boundaries.

  The angle brackets must be escaped, since otherwise they have only their literal character meaning.

  "\<the\>" matches the word "the," but not the words "them," "there," "other," etc.

```
bash$ cat textfile
This is line 1, of which there is only one instance.
 This is the only instance of line 2.
 This is line 3, another line.
 This is line 4.


bash$ grep 'the' textfile
This is line 1, of which there is only one instance.
 This is the only instance of line 2.
 This is line 3, another line.


bash$ grep '\<the\>' textfile
This is the only instance of line 2.
```

The only way to be certain that a particular RE works is to test it.

```
TEST FILE: tstfile                          # No match.
                                            # No match.
```

```
Run   grep "1133*"  on this file.            # Match.
                                             # No match.
                                             # No match.
This line contains the number 113.           # Match.
This line contains the number 13.            # No match.
This line contains the number 133.           # No match.
This line contains the number 1133.          # Match.
This line contains the number 113312.        # Match.
This line contains the number 1112.          # No match.
This line contains the number 113312312.     # Match.
This line contains no numbers at all.        # No match.
```

```
bash$ grep "1133*" tstfile
Run   grep "1133*"  on this file.            # Match.
 This line contains the number 113.           # Match.
 This line contains the number 1133.          # Match.
 This line contains the number 113312.        # Match.
 This line contains the number 113312312.     # Match.
```

- **Extended REs.** Additional metacharacters added to the basic set. Used in egrep, awk, and Perl.
- The question mark -- ? -- matches zero or one of the previous RE. It is generally used for matching single characters.
- The plus -- + -- matches one or more of the previous RE. It serves a role similar to the *, but does *not* match zero occurrences.

  ```
  # GNU versions of sed and awk can use "+",
  # but it needs to be escaped.

  echo a111b | sed -ne '/a1\+b/p'
  echo a111b | grep 'a1\+b'
  echo a111b | gawk '/a1+b/'
  # All of above are equivalent.

  # Thanks, S.C.
  ```
- Escaped "curly brackets" -- \{ \} -- indicate the number of occurrences of a preceding RE to match.

  It is necessary to escape the curly brackets since they have only their literal character meaning otherwise. This usage is technically not part of the basic RE set.

  "[0-9]\{5\}" matches exactly five digits (characters in the range of 0 to 9).

  Curly brackets are not available as an RE in the "classic" (non-POSIX compliant) version of awk. However, the GNU extended version of *awk*, **gawk**, has the `--re-interval` option that permits them (without being escaped).

  ```
  bash$ echo 2222 | gawk --re-interval '/2{3}/'
  2222
  ```

  Perl and some **egrep** versions do not require escaping the curly brackets.
- Parentheses -- **( )** -- enclose a group of REs. They are useful with the following "|" operator and in substring extraction using expr.

* The -- | -- "or" RE operator matches any of a set of alternate characters.

```
bash$ egrep 're(a|e)d' misc.txt
People who read seem to be better informed than those who do not.
 The clarinet produces sound by the vibration of its reed.
```

Some versions of **sed**, **ed**, and **ex** support escaped versions of the extended Regular Expressions described above, as do the GNU utilities.

* **POSIX Character Classes. `[:class:]`**

  This is an alternate method of specifying a range of characters to match.
* **`[:alnum:]`** matches alphabetic or numeric characters. This is equivalent to **`A-Za-z0-9`**.
* **`[:alpha:]`** matches alphabetic characters. This is equivalent to **`A-Za-z`**.
* **`[:blank:]`** matches a space or a tab.
* **`[:cntrl:]`** matches control characters.
* **`[:digit:]`** matches (decimal) digits. This is equivalent to **`0-9`**.
* **`[:graph:]`** (graphic printable characters). Matches characters in the range of ASCII 33 - 126. This is the same as **`[:print:]`**, below, but excluding the space character.
* **`[:lower:]`** matches lowercase alphabetic characters. This is equivalent to **`a-z`**.
* **`[:print:]`** (printable characters). Matches characters in the range of ASCII 32 - 126. This is the same as **`[:graph:]`**, above, but adding the space character.
* **`[:space:]`** matches whitespace characters (space and horizontal tab).
* **`[:upper:]`** matches uppercase alphabetic characters. This is equivalent to **`A-Z`**.
* **`[:xdigit:]`** matches hexadecimal digits. This is equivalent to **`0-9A-Fa-f`**.

  POSIX character classes generally require quoting or double brackets ([[ ]]).

```
bash$ grep [[:digit:]] test.file
abc=723
```

```
# ...
if [[ $arow =~ [[:digit:]] ]]   #  Numerical input?
then       #  POSIX char class
  if [[ $acol =~ [[:alpha:]] ]] # Number followed by a letter? Illegal!
# ...
# From ktour.sh example script.
```

These character classes may even be used with globbing, to a limited extent.

```
bash$ ls -l ?[[:digit:]][[:digit:]]?
-rw-rw-r--    1 bozo  bozo         0 Aug 21 14:47 a33b
```

POSIX character classes are used in Example 16-21 and Example 16-22.

Sed, awk, and Perl, used as filters in scripts, take REs as arguments when "sifting" or transforming files or I/O streams. See Example A-12 and Example A-16 for illustrations of this.

The standard reference on this complex topic is Friedl's *Mastering Regular Expressions*. *Sed & Awk*, by Dougherty and Robbins, also gives a very lucid treatment of REs. See the *Bibliography* for more information on these books.

18.2. Globbing

Bash itself cannot recognize Regular Expressions. Inside scripts, it is commands and utilities -- such as sed and awk -- that interpret RE's.

Bash *does* carry out *filename expansion* [100] -- a process known as *globbing* -- but this does *not* use the standard RE set. Instead, globbing recognizes and expands *wild cards*. Globbing interprets the standard wild card characters [101] -- * and ?, character lists in square brackets, and certain other special characters (such as ^ for negating the sense of a match). There are important limitations on wild card characters in globbing, however. Strings containing *** will not match filenames that start with a dot, as, for example, `.bashrc`. [102] Likewise, the *?* has a different meaning in globbing than as part of an RE.

```
bash$ ls -l
total 2
 -rw-rw-r--    1 bozo  bozo         0 Aug  6 18:42 a.1
 -rw-rw-r--    1 bozo  bozo         0 Aug  6 18:42 b.1
 -rw-rw-r--    1 bozo  bozo         0 Aug  6 18:42 c.1
 -rw-rw-r--    1 bozo  bozo       466 Aug  6 17:48 t2.sh
 -rw-rw-r--    1 bozo  bozo       758 Jul 30 09:02 test1.txt

bash$ ls -l t?.sh
-rw-rw-r--    1 bozo  bozo       466 Aug  6 17:48 t2.sh

bash$ ls -l [ab]*
-rw-rw-r--    1 bozo  bozo         0 Aug  6 18:42 a.1
 -rw-rw-r--    1 bozo  bozo         0 Aug  6 18:42 b.1

bash$ ls -l [a-c]*
-rw-rw-r--    1 bozo  bozo         0 Aug  6 18:42 a.1
 -rw-rw-r--    1 bozo  bozo         0 Aug  6 18:42 b.1
 -rw-rw-r--    1 bozo  bozo         0 Aug  6 18:42 c.1

bash$ ls -l [^ab]*
-rw-rw-r--    1 bozo  bozo         0 Aug  6 18:42 c.1
 -rw-rw-r--    1 bozo  bozo       466 Aug  6 17:48 t2.sh
 -rw-rw-r--    1 bozo  bozo       758 Jul 30 09:02 test1.txt

bash$ ls -l {b*,c*,*est*}
-rw-rw-r--    1 bozo  bozo         0 Aug  6 18:42 b.1
 -rw-rw-r--    1 bozo  bozo         0 Aug  6 18:42 c.1
 -rw-rw-r--    1 bozo  bozo       758 Jul 30 09:02 test1.txt
```

Bash performs filename expansion on unquoted command-line arguments. The echo command demonstrates this.

```
bash$ echo *
a.1 b.1 c.1 t2.sh test1.txt

bash$ echo t*
t2.sh test1.txt

bash$ echo t?.sh
t2.sh
```

It is possible to modify the way Bash interprets special characters in globbing. A **set -f** command disables globbing, and the `nocaseglob` and `nullglob` options to shopt change globbing behavior.

See also Example 11-5.

Filenames with embedded whitespace can cause *globbing* to choke. David Wheeler shows how to avoid many such pitfalls.

```
IFS="$(printf '\n\t')"   # Remove space.

#  Correct glob use:
#  Always use for-loop, prefix glob, check if exists file.
for file in ./* ; do         # Use ./* ... NEVER bare *
  if [ -e "$file" ] ; then   # Check whether file exists.
     COMMAND ... "$file" ...
  fi
done

# This example taken from David Wheeler's site, with permission.
```

Chapter 19. Here Documents

Here and now, boys.

--Aldous Huxley, Island

A *here document* is a special-purpose code block. It uses a form of I/O redirection to feed a command list to an interactive program or a command, such as ftp, cat, or the *ex* text editor.

```
COMMAND <<InputComesFromHERE
...
...
...
InputComesFromHERE
```

A *limit string* delineates (frames) the command list. The special symbol << precedes the limit string. This has the effect of redirecting the output of a command block into the `stdin` of the program or command. It is similar to **`interactive-program < command-file`**, where `command-file` contains

```
command #1
command #2
...
```

The *here document* equivalent looks like this:

```
interactive-program <<LimitString
command #1
command #2
...
LimitString
```

Choose a *limit string* sufficiently unusual that it will not occur anywhere in the command list and confuse matters.

Note that *here documents* may sometimes be used to good effect with non-interactive utilities and commands, such as, for example, wall.

Example 19-1. *broadcast*: Sends message to everyone logged in

```
#!/bin/bash

wall <<zzz23EndOfMessagezzz23
E-mail your noontime orders for pizza to the system administrator.
    (Add an extra dollar for anchovy or mushroom topping.)
# Additional message text goes here.
# Note: 'wall' prints comment lines.
zzz23EndOfMessagezzz23

# Could have been done more efficiently by
#         wall <message-file
#  However, embedding the message template in a script
#+ is a quick-and-dirty one-off solution.

exit
```

Even such unlikely candidates as the *vi* text editor lend themselves to *here documents*.

Example 19-2. *dummyfile*: Creates a 2-line dummy file

```
#!/bin/bash

# Noninteractive use of 'vi' to edit a file.
# Emulates 'sed'.

E_BADARGS=85

if [ -z "$1" ]
then
  echo "Usage: `basename $0` filename"
  exit $E_BADARGS
fi

TARGETFILE=$1

# Insert 2 lines in file, then save.
#--------Begin here document-----------#
vi $TARGETFILE <<x23LimitStringx23
i
This is line 1 of the example file.
This is line 2 of the example file.
^[
ZZ
x23LimitStringx23
#----------End here document-----------#

#  Note that ^[ above is a literal escape
#+ typed by Control-V <Esc>.

#  Bram Moolenaar points out that this may not work with 'vim'
#+ because of possible problems with terminal interaction.

exit
```

The above script could just as effectively have been implemented with **ex**, rather than **vi**. *Here documents* containing a list of **ex** commands are common enough to form their own category, known as *ex scripts*.

```
#!/bin/bash
#  Replace all instances of "Smith" with "Jones"
#+ in files with a ".txt" filename suffix.

ORIGINAL=Smith
REPLACEMENT=Jones

for word in $(fgrep -l $ORIGINAL *.txt)
do
  # -------------------------------------
  ex $word <<EOF
  :%s/$ORIGINAL/$REPLACEMENT/g
  :wq
EOF
  # :%s is the "ex" substitution command.
  # :wq is write-and-quit.
  # -------------------------------------
done
```

Analogous to "ex scripts" are *cat scripts*.

Example 19-3. Multi-line message using *cat*

```
#!/bin/bash

#  'echo' is fine for printing single line messages,
#+  but somewhat problematic for for message blocks.
#   A 'cat' here document overcomes this limitation.

cat <<End-of-message
-------------------------------------
This is line 1 of the message.
This is line 2 of the message.
This is line 3 of the message.
This is line 4 of the message.
This is the last line of the message.
-------------------------------------
End-of-message

#  Replacing line 7, above, with
#+   cat > $Newfile <<End-of-message
#+       ^^^^^^^^^^
#+ writes the output to the file $Newfile, rather than to stdout.

exit 0


#--------------------------------------------
# Code below disabled, due to "exit 0" above.

# S.C. points out that the following also works.
echo "-------------------------------------
This is line 1 of the message.
This is line 2 of the message.
This is line 3 of the message.
This is line 4 of the message.
This is the last line of the message.
-------------------------------------"
# However, text may not include double quotes unless they are escaped.
```

The − option to mark a here document limit string (**`<<-LimitString`**) suppresses leading tabs (but not spaces) in the output. This may be useful in making a script more readable.

Example 19-4. Multi-line message, with tabs suppressed

```
#!/bin/bash
# Same as previous example, but...

#  The - option to a here document <<-
#+ suppresses leading tabs in the body of the document,
#+ but *not* spaces.

cat <<-ENDOFMESSAGE
	This is line 1 of the message.
	This is line 2 of the message.
	This is line 3 of the message.
```

```
	This is line 4 of the message.
	This is the last line of the message.
ENDOFMESSAGE
# The output of the script will be flush left.
# Leading tab in each line will not show.

# Above 5 lines of "message" prefaced by a tab, not spaces.
# Spaces not affected by   <<-  .

# Note that this option has no effect on *embedded* tabs.

exit 0
```

A *here document* supports parameter and command substitution. It is therefore possible to pass different parameters to the body of the here document, changing its output accordingly.

Example 19-5. Here document with replaceable parameters

```
#!/bin/bash
# Another 'cat' here document, using parameter substitution.

# Try it with no command-line parameters,   ./scriptname
# Try it with one command-line parameter,   ./scriptname Mortimer
# Try it with one two-word quoted command-line parameter,
#                           ./scriptname "Mortimer Jones"

CMDLINEPARAM=1     #  Expect at least command-line parameter.

if [ $# -ge $CMDLINEPARAM ]
then
  NAME=$1          #  If more than one command-line param,
                   #+ then just take the first.
else
  NAME="John Doe"  #  Default, if no command-line parameter.
fi

RESPONDENT="the author of this fine script"


cat <<Endofmessage

Hello, there, $NAME.
Greetings to you, $NAME, from $RESPONDENT.

# This comment shows up in the output (why?).

Endofmessage

# Note that the blank lines show up in the output.
# So does the comment.

exit
```

This is a useful script containing a *here document* with parameter substitution.

Example 19-6. Upload a file pair to *Sunsite* incoming directory

```
#!/bin/bash
# upload.sh

#  Upload file pair (Filename.lsm, Filename.tar.gz)
#+ to incoming directory at Sunsite/UNC (ibiblio.org).
#  Filename.tar.gz is the tarball itself.
#  Filename.lsm is the descriptor file.
#  Sunsite requires "lsm" file, otherwise will bounce contributions.


E_ARGERROR=85

if [ -z "$1" ]
then
  echo "Usage: `basename $0` Filename-to-upload"
  exit $E_ARGERROR
fi


Filename=`basename $1`           # Strips pathname out of file name.

Server="ibiblio.org"
Directory="/incoming/Linux"
#  These need not be hard-coded into script,
#+ but may instead be changed to command-line argument.

Password="your.e-mail.address"   # Change above to suit.

ftp -n $Server <<End-Of-Session
# -n option disables auto-logon

user anonymous "$Password"       #  If this doesn't work, then try:
                                 #  quote user anonymous "$Password"
binary
bell                             # Ring 'bell' after each file transfer.
cd $Directory
put "$Filename.lsm"
put "$Filename.tar.gz"
bye
End-Of-Session

exit 0
```

Quoting or escaping the "limit string" at the head of a here document disables parameter substitution within its body. The reason for this is that *quoting/escaping the limit string* effectively escapes the $, `, and \ special characters, and causes them to be interpreted literally. (Thank you, Allen Halsey, for pointing this out.)

Example 19-7. Parameter substitution turned off

```
#!/bin/bash
#  A 'cat' here-document, but with parameter substitution disabled.

NAME="John Doe"
RESPONDENT="the author of this fine script"

cat <<'Endofmessage'

Hello, there, $NAME.
Greetings to you, $NAME, from $RESPONDENT.
```

```
Endofmessage

#   No parameter substitution when the "limit string" is quoted or escaped.
#   Either of the following at the head of the here document would have
#+  the same effect.
#   cat <<"Endofmessage"
#   cat <<\Endofmessage



#   And, likewise:

cat <<"SpecialCharTest"

Directory listing would follow
if limit string were not quoted.
`ls -l`

Arithmetic expansion would take place
if limit string were not quoted.
$((5 + 3))

A a single backslash would echo
if limit string were not quoted.
\\

SpecialCharTest


exit
```

Disabling parameter substitution permits outputting literal text. Generating scripts or even program code is one use for this.

Example 19-8. A script that generates another script

```
#!/bin/bash
# generate-script.sh
# Based on an idea by Albert Reiner.

OUTFILE=generated.sh         # Name of the file to generate.


# -----------------------------------------------------------
# 'Here document containing the body of the generated script.
(
cat <<'EOF'
#!/bin/bash

echo "This is a generated shell script."
#  Note that since we are inside a subshell,
#+ we can't access variables in the "outside" script.

echo "Generated file will be named: $OUTFILE"
#  Above line will not work as normally expected
#+ because parameter expansion has been disabled.
#  Instead, the result is literal output.
```

```
a=7
b=3

let "c = $a * $b"
echo "c = $c"

exit 0
EOF
) > $OUTFILE
# -----------------------------------------------------------

#  Quoting the 'limit string' prevents variable expansion
#+ within the body of the above 'here document.'
#  This permits outputting literal strings in the output file.

if [ -f "$OUTFILE" ]
then
  chmod 755 $OUTFILE
  # Make the generated file executable.
else
  echo "Problem in creating file: \"$OUTFILE\""
fi

#  This method also works for generating
#+ C programs, Perl programs, Python programs, Makefiles,
#+ and the like.

exit 0
```

It is possible to set a variable from the output of a here document. This is actually a devious form of command substitution.

```
variable=$(cat <<SETVAR
This variable
runs over multiple lines.
SETVAR
)

echo "$variable"
```

A here document can supply input to a function in the same script.

Example 19-9. Here documents and functions

```
#!/bin/bash
# here-function.sh

GetPersonalData ()
{
  read firstname
  read lastname
  read address
  read city
  read state
  read zipcode
} # This certainly appears to be an interactive function, but . . .
```

```
# Supply input to the above function.
GetPersonalData <<RECORD001
Bozo
Bozeman
2726 Nondescript Dr.
Bozeman
MT
21226
RECORD001


echo
echo "$firstname $lastname"
echo "$address"
echo "$city, $state $zipcode"
echo

exit 0
```

It is possible to use : as a dummy command accepting output from a here document. This, in effect, creates an "anonymous" here document.

Example 19-10. "Anonymous" Here Document

```
#!/bin/bash

: <<TESTVARIABLES
${HOSTNAME?}${USER?}${MAIL?}  # Print error message if one of the variables not set.
TESTVARIABLES

exit $?
```

A variation of the above technique permits "commenting out" blocks of code.

Example 19-11. Commenting out a block of code

```
#!/bin/bash
# commentblock.sh

: <<COMMENTBLOCK
echo "This line will not echo."
This is a comment line missing the "#" prefix.
This is another comment line missing the "#" prefix.

&*@!!++=
The above line will cause no error message,
because the Bash interpreter will ignore it.
COMMENTBLOCK

echo "Exit value of above \"COMMENTBLOCK\" is $?."   # 0
# No error shown.
echo


#  The above technique also comes in useful for commenting out
#+ a block of working code for debugging purposes.
#  This saves having to put a "#" at the beginning of each line,
```

```
#+ then having to go back and delete each "#" later.
#  Note that the use of of colon, above, is optional.

echo "Just before commented-out code block."
#  The lines of code between the double-dashed lines will not execute.
#  ===================================================================
: <<DEBUGXXX
for file in *
do
 cat "$file"
done
DEBUGXXX
#  ===================================================================
echo "Just after commented-out code block."

exit 0



######################################################################
#  Note, however, that if a bracketed variable is contained within
#+ the commented-out code block,
#+ then this could cause problems.
#  for example:


#/!/bin/bash

  : <<COMMENTBLOCK
  echo "This line will not echo."
  &*@!!++=
  ${foo_bar_bazz?}
  $(rm -rf /tmp/foobar/)
  $(touch my_build_directory/cups/Makefile)
COMMENTBLOCK


$ sh commented-bad.sh
commented-bad.sh: line 3: foo_bar_bazz: parameter null or not set

# The remedy for this is to strong-quote the 'COMMENTBLOCK' in line 49, above.

  : <<'COMMENTBLOCK'

# Thank you, Kurt Pfeifle, for pointing this out.
```

Yet another twist of this nifty trick makes "self-documenting" scripts possible.

Example 19-12. A self-documenting script

```
#!/bin/bash
# self-document.sh: self-documenting script
# Modification of "colm.sh".

DOC_REQUEST=70

if [ "$1" = "-h"  -o "$1" = "--help" ]     # Request help.
then
  echo; echo "Usage: $0 [directory-name]"; echo
  sed --silent -e '/DOCUMENTATIONXX$/,/^DOCUMENTATIONXX$/p' "$0" |
```

```
  sed -e '/DOCUMENTATIONXX$/d'; exit $DOC_REQUEST; fi


: <<DOCUMENTATIONXX
List the statistics of a specified directory in tabular format.
---------------------------------------------------------------
The command-line parameter gives the directory to be listed.
If no directory specified or directory specified cannot be read,
then list the current working directory.

DOCUMENTATIONXX

if [ -z "$1" -o ! -r "$1" ]
then
  directory=.
else
  directory="$1"
fi

echo "Listing of "$directory":"; echo
(printf "PERMISSIONS LINKS OWNER GROUP SIZE MONTH DAY HH:MM PROG-NAME\n" \
; ls -l "$directory" | sed 1d) | column -t

exit 0
```

Using a cat script is an alternate way of accomplishing this.

```
DOC_REQUEST=70

if [ "$1" = "-h"  -o "$1" = "--help" ]     # Request help.
then                                       # Use a "cat script" . . .
  cat <<DOCUMENTATIONXX
List the statistics of a specified directory in tabular format.
---------------------------------------------------------------
The command-line parameter gives the directory to be listed.
If no directory specified or directory specified cannot be read,
then list the current working directory.

DOCUMENTATIONXX
exit $DOC_REQUEST
fi
```

See also Example A-28, Example A-40, Example A-41, and Example A-42 for more examples of self-documenting scripts.

Here documents create temporary files, but these files are deleted after opening and are not accessible to any other process.

```
bash$ bash -c 'lsof -a -p $$ -d0' << EOF
> EOF
lsof    1213 bozo    0r   REG    3,5    0 30386 /tmp/t1213-0-sh (deleted)
```

Some utilities will not work inside a *here document*.

The closing *limit string*, on the final line of a here document, must start in the *first* character position. There can be *no leading whitespace*. Trailing whitespace after the limit string likewise causes unexpected behavior. The whitespace prevents the limit string from being recognized. [103]

```
#!/bin/bash

echo "----------------------------------------------------------------------"

cat <<LimitString
echo "This is line 1 of the message inside the here document."
echo "This is line 2 of the message inside the here document."
echo "This is the final line of the message inside the here document."
     LimitString
#^^^^Indented limit string. Error! This script will not behave as expected.

echo "----------------------------------------------------------------------"

#  These comments are outside the 'here document',
#+ and should not echo.

echo "Outside the here document."

exit 0

echo "This line had better not echo."  # Follows an 'exit' command.
```

Some people very cleverly use a single ! as a limit string. But, that's not necessarily a good idea.

```
# This works.
cat <<!
Hello!
! Three more exclamations !!!
!


# But . . .
cat <<!
Hello!
Single exclamation point follows!
!
!
# Crashes with an error message.


# However, the following will work.
cat <<EOF
Hello!
Single exclamation point follows!
!
EOF
# It's safer to use a multi-character limit string.
```

For those tasks too complex for a *here document*, consider using the `expect` scripting language, which was specifically designed for feeding input into interactive programs.

19.1. Here Strings

A *here string* can be considered as a stripped-down form of a *here document*.

```
It consists of nothing more than COMMAND <<< $WORD,
where $WORD is expanded and fed to the stdin of COMMAND.
```

As a simple example, consider this alternative to the echo-grep construction.

```
# Instead of:
if echo "$VAR" | grep -q txt   # if [[ $VAR = *txt* ]]
# etc.

# Try:
if grep -q "txt" <<< "$VAR"
then   #         ^^^
   echo "$VAR contains the substring sequence \"txt\""
fi
# Thank you, Sebastian Kaminski, for the suggestion.
```

Or, in combination with read:

```
String="This is a string of words."

read -r -a Words <<< "$String"
#  The -a option to "read"
#+ assigns the resulting values to successive members of an array.

echo "First word in String is:    ${Words[0]}"   # This
echo "Second word in String is:   ${Words[1]}"   # is
echo "Third word in String is:    ${Words[2]}"   # a
echo "Fourth word in String is:   ${Words[3]}"   # string
echo "Fifth word in String is:    ${Words[4]}"   # of
echo "Sixth word in String is:    ${Words[5]}"   # words.
echo "Seventh word in String is:  ${Words[6]}"   # (null)
                                                 # Past end of $String.

# Thank you, Francisco Lobo, for the suggestion.
```

It is, of course, possible to feed the output of a *here string* into the `stdin` of a loop.

```
# As Seamus points out . . .

ArrayVar=( element0 element1 element2 {A..D} )

while read element ; do
  echo "$element" 1>&2
done <<< $(echo ${ArrayVar[*]})

# element0 element1 element2 A B C D
```

Example 19-13. Prepending a line to a file

```
#!/bin/bash
# prepend.sh: Add text at beginning of file.
#
#  Example contributed by Kenny Stauffer,
#+ and slightly modified by document author.


E_NOSUCHFILE=85
```

```
read -p "File: " file   # -p arg to 'read' displays prompt.
if [ ! -e "$file" ]
then   # Bail out if no such file.
  echo "File $file not found."
  exit $E_NOSUCHFILE
fi

read -p "Title: " title
cat - $file <<<$title > $file.new

echo "Modified file is $file.new"

exit  # Ends script execution.

  from 'man bash':
  Here Strings
        A variant of here documents, the format is:

                <<<word

        The word is expanded and supplied to the command on its standard input.


  Of course, the following also works:
   sed -e '1i\
   Title: ' $file
```

Example 19-14. Parsing a mailbox

```
#!/bin/bash
#  Script by Francisco Lobo,
#+ and slightly modified and commented by ABS Guide author.
#  Used in ABS Guide with permission. (Thank you!)

# This script will not run under Bash versions -lt 3.0.


E_MISSING_ARG=87
if [ -z "$1" ]
then
  echo "Usage: $0 mailbox-file"
  exit $E_MISSING_ARG
fi

mbox_grep()  # Parse mailbox file.
{
    declare -i body=0 match=0
    declare -a date sender
    declare mail header value


    while IFS= read -r mail
#         ^^^^                 Reset $IFS.
#  Otherwise "read" will strip leading & trailing space from its input.

   do
       if [[ $mail =~ ^From  ]]   # Match "From" field in message.
       then
          (( body  = 0 ))           # "Zero out" variables.
          (( match = 0 ))
```

```
          unset date

       elif (( body ))
       then
            (( match ))
            #  echo "$mail"
            #  Uncomment above line if you want entire body
            #+ of message to display.

   elif [[ $mail ]]; then
      IFS=: read -r header value <<< "$mail"
      #                          ^^^  "here string"

      case "$header" in
      [Ff][Rr][Oo][Mm] ) [[ $value =~ "$2" ]] && (( match++ )) ;;
      # Match "From" line.
      [Dd][Aa][Tt][Ee] ) read -r -a date <<< "$value" ;;
      #                                  ^^^
      # Match "Date" line.
      [Rr][Ee][Cc][Ee][Ii][Vv][Ee][Dd] ) read -r -a sender <<< "$value" ;;
      #                                                    ^^^
      # Match IP Address (may be spoofed).
      esac

       else
          (( body++ ))
          (( match  )) &&
          echo "MESSAGE ${date:+of: ${date[*]} }"
       #    Entire $date array             ^
          echo "IP address of sender: ${sender[1]}"
       #    Second field of "Received" line    ^

       fi


    done < "$1" # Redirect stdout of file into loop.
}


mbox_grep "$1"  # Send mailbox file to function.

exit $?

# Exercises:
# ---------
# 1) Break the single function, above, into multiple functions,
#+   for the sake of readability.
# 2) Add additional parsing to the script, checking for various keywords.



$ mailbox_grep.sh scam_mail
  MESSAGE of Thu, 5 Jan 2006 08:00:56 -0500 (EST)
  IP address of sender: 196.3.62.4
```

Exercise: Find other uses for *here strings*, such as, for example, feeding input to *dc*.

Chapter 20. I/O Redirection

There are always three default *files* [104] open, `stdin` (the keyboard), `stdout` (the screen), and `stderr` (error messages output to the screen). These, and any other open files, can be redirected. Redirection simply means capturing output from a file, command, program, script, or even code block within a script (see Example 3-1 and Example 3-2) and sending it as input to another file, command, program, or script.

Each open file gets assigned a file descriptor. [105] The file descriptors for `stdin`, `stdout`, and `stderr` are 0, 1, and 2, respectively. For opening additional files, there remain descriptors 3 to 9. It is sometimes useful to assign one of these additional file descriptors to `stdin`, `stdout`, or `stderr` as a temporary duplicate link. [106] This simplifies restoration to normal after complex redirection and reshuffling (see Example 20-1).

```
COMMAND_OUTPUT >
   # Redirect stdout to a file.
   # Creates the file if not present, otherwise overwrites it.

   ls -lR > dir-tree.list
   # Creates a file containing a listing of the directory tree.

: > filename
   # The > truncates file "filename" to zero length.
   # If file not present, creates zero-length file (same effect as 'touch').
   # The : serves as a dummy placeholder, producing no output.

> filename
   # The > truncates file "filename" to zero length.
   # If file not present, creates zero-length file (same effect as 'touch').
   # (Same result as ": >", above, but this does not work with some shells.)

COMMAND_OUTPUT >>
   # Redirect stdout to a file.
   # Creates the file if not present, otherwise appends to it.


   # Single-line redirection commands (affect only the line they are on):
   # --------------------------------------------------------------------

1>filename
   # Redirect stdout to file "filename."
1>>filename
   # Redirect and append stdout to file "filename."
2>filename
   # Redirect stderr to file "filename."
2>>filename
   # Redirect and append stderr to file "filename."
&>filename
   # Redirect both stdout and stderr to file "filename."
   # This operator is now functional, as of Bash 4, final release.

M>N
  # "M" is a file descriptor, which defaults to 1, if not explicitly set.
  # "N" is a filename.
  # File descriptor "M" is redirect to file "N."
M>&N
  # "M" is a file descriptor, which defaults to 1, if not set.
```

```
   # "N" is another file descriptor.

    #==============================================================================

    # Redirecting stdout, one line at a time.
    LOGFILE=script.log

    echo "This statement is sent to the log file, \"$LOGFILE\"." 1>$LOGFILE
    echo "This statement is appended to \"$LOGFILE\"." 1>>$LOGFILE
    echo "This statement is also appended to \"$LOGFILE\"." 1>>$LOGFILE
    echo "This statement is echoed to stdout, and will not appear in \"$LOGFILE\"."
    # These redirection commands automatically "reset" after each line.



    # Redirecting stderr, one line at a time.
    ERRORFILE=script.errors

    bad_command1 2>$ERRORFILE       #  Error message sent to $ERRORFILE.
    bad_command2 2>>$ERRORFILE      #  Error message appended to $ERRORFILE.
    bad_command3                    #  Error message echoed to stderr,
                                    #+ and does not appear in $ERRORFILE.
    # These redirection commands also automatically "reset" after each line.
    #=======================================================================
```

```
 2>&1
    # Redirects stderr to stdout.
    # Error messages get sent to same place as standard output.
      >>filename 2>&1
          bad_command >>filename 2>&1
          # Appends both stdout and stderr to the file "filename" ...
      2>&1 | [command(s)]
          bad_command 2>&1 | awk '{print $5}'   # found
          # Sends stderr through a pipe.
          # |& was added to Bash 4 as an abbreviation for 2>&1 |.

 i>&j
    # Redirects file descriptor i to j.
    # All output of file pointed to by i gets sent to file pointed to by j.

 >&j
    # Redirects, by default, file descriptor 1 (stdout) to j.
    # All stdout gets sent to file pointed to by j.
```

```
 0< FILENAME
  < FILENAME
    # Accept input from a file.
    # Companion command to ">", and often used in combination with it.
    #
    # grep search-word <filename


 [j]<>filename
    #  Open file "filename" for reading and writing,
    #+ and assign file descriptor "j" to it.
    #  If "filename" does not exist, create it.
    #  If file descriptor "j" is not specified, default to fd 0, stdin.
    #
    #  An application of this is writing at a specified place in a file.
    echo 1234567890 > File    # Write string to "File".
    exec 3<> File             # Open "File" and assign fd 3 to it.
```

```
      read -n 4 <&3             # Read only 4 characters.
      echo -n . >&3             # Write a decimal point there.
      exec 3>&-                 # Close fd 3.
      cat File                  # ==> 1234.67890
      #  Random access, by golly.
```

```
   |
      # Pipe.
      # General purpose process and command chaining tool.
      # Similar to ">", but more general in effect.
      # Useful for chaining commands, scripts, files, and programs together.
      cat *.txt | sort | uniq > result-file
      # Sorts the output of all the .txt files and deletes duplicate lines,
      # finally saves results to "result-file".
```

Multiple instances of input and output redirection and/or pipes can be combined in a single command line.

```
command < input-file > output-file
# Or the equivalent:
< input-file command > output-file   # Although this is non-standard.

command1 | command2 | command3 > output-file
```

See Example 16-31 and Example A-14.

Multiple output streams may be redirected to one file.

```
ls -yz >> command.log 2>&1
#  Capture result of illegal options "yz" in file "command.log."
#  Because stderr is redirected to the file,
#+ any error messages will also be there.

#  Note, however, that the following does *not* give the same result.
ls -yz 2>&1 >> command.log
#  Outputs an error message, but does not write to file.
#  More precisely, the command output (in this case, null)
#+ writes to the file, but the error message goes only to stdout.

#  If redirecting both stdout and stderr,
#+ the order of the commands makes a difference.
```

Closing File Descriptors

n<&-
: Close input file descriptor *n*.

0<&-, <&-
: Close `stdin`.

n>&-
: Close output file descriptor *n*.

1>&-, >&-
: Close `stdout`.

Child processes inherit open file descriptors. This is why pipes work. To prevent an fd from being inherited, close it.

```
# Redirecting only stderr to a pipe.

exec 3>&1                              # Save current "value" of stdout.
ls -l 2>&1 >&3 3>&- | grep bad 3>&-    # Close fd 3 for 'grep' (but not 'ls').
#              ^^^^   ^^^^
exec 3>&-                              # Now close it for the remainder of the script.

# Thanks, S.C.
```

For a more detailed introduction to I/O redirection see Appendix F.

20.1. Using *exec*

An **exec <filename** command redirects `stdin` to a file. From that point on, all `stdin` comes from that file, rather than its normal source (usually keyboard input). This provides a method of reading a file line by line and possibly parsing each line of input using sed and/or awk.

Example 20-1. Redirecting `stdin` using *exec*

```
#!/bin/bash
# Redirecting stdin using 'exec'.


exec 6<&0          # Link file descriptor #6 with stdin.
                   # Saves stdin.

exec < data-file   # stdin replaced by file "data-file"

read a1            # Reads first line of file "data-file".
read a2            # Reads second line of file "data-file."

echo
echo "Following lines read from file."
echo "-------------------------------"
echo $a1
echo $a2

echo; echo; echo

exec 0<&6 6<&-
#  Now restore stdin from fd #6, where it had been saved,
#+ and close fd #6 ( 6<&- ) to free it for other processes to use.
#
# <&6 6<&-    also works.

echo -n "Enter data  "
read b1  # Now "read" functions as expected, reading from normal stdin.
echo "Input read from stdin."
echo "----------------------"
echo "b1 = $b1"

echo

exit 0
```

Similarly, an **exec >filename** command redirects `stdout` to a designated file. This sends all command output that would normally go to `stdout` to that file.

exec N > filename affects the entire script or *current shell*. Redirection in the PID of the script or shell from that point on has changed. However . . .

N > filename affects only the newly-forked process, not the entire script or shell.

Thank you, Ahmed Darwish, for pointing this out.

Example 20-2. Redirecting `stdout` using *exec*

```
#!/bin/bash
# reassign-stdout.sh

LOGFILE=logfile.txt

exec 6>&1           # Link file descriptor #6 with stdout.
                    # Saves stdout.

exec > $LOGFILE     # stdout replaced with file "logfile.txt".

# ----------------------------------------------------------- #
# All output from commands in this block sent to file $LOGFILE.

echo -n "Logfile: "
date
echo "-------------------------------------"
echo

echo "Output of \"ls -al\" command"
echo
ls -al
echo; echo
echo "Output of \"df\" command"
echo
df

# ----------------------------------------------------------- #

exec 1>&6 6>&-      # Restore stdout and close file descriptor #6.

echo
echo "== stdout now restored to default == "
echo
ls -al
echo

exit 0
```

Example 20-3. Redirecting both `stdin` and `stdout` in the same script with *exec*

```
#!/bin/bash
# upperconv.sh
# Converts a specified input file to uppercase.

E_FILE_ACCESS=70
E_WRONG_ARGS=71

if [ ! -r "$1" ]     # Is specified input file readable?
then
  echo "Can't read from input file!"
```

```
  echo "Usage: $0 input-file output-file"
  exit $E_FILE_ACCESS
fi                     #  Will exit with same error
                       #+ even if input file ($1) not specified (why?).

if [ -z "$2" ]
then
  echo "Need to specify output file."
  echo "Usage: $0 input-file output-file"
  exit $E_WRONG_ARGS
fi


exec 4<&0
exec < $1              # Will read from input file.

exec 7>&1
exec > $2              # Will write to output file.
                       # Assumes output file writable (add check?).

# -----------------------------------------------
    cat - | tr a-z A-Z   # Uppercase conversion.
#   ^^^^^                # Reads from stdin.
#           ^^^^^^^^^^   # Writes to stdout.
# However, both stdin and stdout were redirected.
# Note that the 'cat' can be omitted.
# -----------------------------------------------

exec 1>&7 7>&-       # Restore stout.
exec 0<&4 4<&-       # Restore stdin.

# After restoration, the following line prints to stdout as expected.
echo "File \"$1\" written to \"$2\" as uppercase conversion."

exit 0
```

I/O redirection is a clever way of avoiding the dreaded inaccessible variables within a subshell problem.

Example 20-4. Avoiding a subshell

```
#!/bin/bash
# avoid-subshell.sh
# Suggested by Matthew Walker.

Lines=0

echo

cat myfile.txt | while read line;
                 do {
                   echo $line
                   (( Lines++ ));  #  Incremented values of this variable
                                   #+ inaccessible outside loop.
                                   #  Subshell problem.
                 }
                 done

echo "Number of lines read = $Lines"     # 0
                                         # Wrong!

echo "------------------------"
```

```
exec 3<> myfile.txt
while read line <&3
do {
  echo "$line"
  (( Lines++ ));                   #  Incremented values of this variable
                                   #+ accessible outside loop.
                                   #  No subshell, no problem.
}
done
exec 3>&-

echo "Number of lines read = $Lines"     # 8

echo

exit 0

# Lines below not seen by script.

$ cat myfile.txt

Line 1.
Line 2.
Line 3.
Line 4.
Line 5.
Line 6.
Line 7.
Line 8.
```

20.2. Redirecting Code Blocks

Blocks of code, such as while, until, and for loops, even if/then test blocks can also incorporate redirection of `stdin`. Even a function may use this form of redirection (see Example 24-11). The < operator at the end of the code block accomplishes this.

Example 20-5. Redirected *while* loop

```
#!/bin/bash
# redir2.sh

if [ -z "$1" ]
then
  Filename=names.data       # Default, if no filename specified.
else
  Filename=$1
fi
#+ Filename=${1:-names.data}
#  can replace the above test (parameter substitution).

count=0

echo

while [ "$name" != Smith ]  # Why is variable $name in quotes?
do
```

```
  read name                  # Reads from $Filename, rather than stdin.
  echo $name
  let "count += 1"
done <"$Filename"            # Redirects stdin to file $Filename.
#    ^^^^^^^^^^^^

echo; echo "$count names read"; echo

exit 0

#  Note that in some older shell scripting languages,
#+ the redirected loop would run as a subshell.
#  Therefore, $count would return 0, the initialized value outside the loop.
#  Bash and ksh avoid starting a subshell *whenever possible*,
#+ so that this script, for example, runs correctly.
#  (Thanks to Heiner Steven for pointing this out.)

#  However . . .
#  Bash *can* sometimes start a subshell in a PIPED "while-read" loop,
#+ as distinct from a REDIRECTED "while" loop.

abc=hi
echo -e "1\n2\n3" | while read l
     do abc="$l"
        echo $abc
     done
echo $abc

#  Thanks, Bruno de Oliveira Schneider, for demonstrating this
#+ with the above snippet of code.
#  And, thanks, Brian Onn, for correcting an annotation error.
```

Example 20-6. Alternate form of redirected *while* loop

```
#!/bin/bash

# This is an alternate form of the preceding script.

#  Suggested by Heiner Steven
#+ as a workaround in those situations when a redirect loop
#+ runs as a subshell, and therefore variables inside the loop
# +do not keep their values upon loop termination.


if [ -z "$1" ]
then
  Filename=names.data     # Default, if no filename specified.
else
  Filename=$1
fi


exec 3<&0                 # Save stdin to file descriptor 3.
exec 0<"$Filename"        # Redirect standard input.

count=0
echo


while [ "$name" != Smith ]
do
```

```
  read name                 # Reads from redirected stdin ($Filename).
  echo $name
  let "count += 1"
done                        #  Loop reads from file $Filename
                            #+ because of line 20.

#  The original version of this script terminated the "while" loop with
#+      done <"$Filename"
#  Exercise:
#  Why is this unnecessary?


exec 0<&3                   # Restore old stdin.
exec 3<&-                   # Close temporary fd 3.

echo; echo "$count names read"; echo

exit 0
```

Example 20-7. Redirected *until* loop

```
#!/bin/bash
# Same as previous example, but with "until" loop.

if [ -z "$1" ]
then
  Filename=names.data         # Default, if no filename specified.
else
  Filename=$1
fi

# while [ "$name" != Smith ]
until [ "$name" = Smith ]     # Change  !=  to =.
do
  read name                   # Reads from $Filename, rather than stdin.
  echo $name
done <"$Filename"             # Redirects stdin to file $Filename.
#    ^^^^^^^^^^^^

# Same results as with "while" loop in previous example.

exit 0
```

Example 20-8. Redirected *for* loop

```
#!/bin/bash

if [ -z "$1" ]
then
  Filename=names.data          # Default, if no filename specified.
else
  Filename=$1
fi

line_count=`wc $Filename | awk '{ print $1 }'`
#           Number of lines in target file.
#
#  Very contrived and kludgy, nevertheless shows that
#+ it's possible to redirect stdin within a "for" loop...
#+ if you're clever enough.
```

```
#
# More concise is     line_count=$(wc -l < "$Filename")


for name in `seq $line_count`  # Recall that "seq" prints sequence of numbers.
# while [ "$name" != Smith ]   --   more complicated than a "while" loop   --
do
  read name                    # Reads from $Filename, rather than stdin.
  echo $name
  if [ "$name" = Smith ]       # Need all this extra baggage here.
  then
    break
  fi
done <"$Filename"              # Redirects stdin to file $Filename.
#    ^^^^^^^^^^^^

exit 0
```

We can modify the previous example to also redirect the output of the loop.

Example 20-9. Redirected *for* loop (both `stdin` and `stdout` redirected)

```
#!/bin/bash

if [ -z "$1" ]
then
  Filename=names.data          # Default, if no filename specified.
else
  Filename=$1
fi

Savefile=$Filename.new         # Filename to save results in.
FinalName=Jonah                # Name to terminate "read" on.

line_count=`wc $Filename | awk '{ print $1 }'`  # Number of lines in target file.


for name in `seq $line_count`
do
  read name
  echo "$name"
  if [ "$name" = "$FinalName" ]
  then
    break
  fi
done < "$Filename" > "$Savefile"     # Redirects stdin to file $Filename,
#    ^^^^^^^^^^^^^^^^^^^^^^^^^^^       and saves it to backup file.

exit 0
```

Example 20-10. Redirected *if/then* test

```
#!/bin/bash

if [ -z "$1" ]
then
  Filename=names.data   # Default, if no filename specified.
else
  Filename=$1
fi
```

```
TRUE=1

if [ "$TRUE" ]          # if true    and   if :   also work.
then
 read name
 echo $name
fi <"$Filename"
#  ^^^^^^^^^^^^

# Reads only first line of file.
# An "if/then" test has no way of iterating unless embedded in a loop.

exit 0
```

Example 20-11. Data file *names.data* for above examples

```
Aristotle
Arrhenius
Belisarius
Capablanca
Dickens
Euler
Goethe
Hegel
Jonah
Laplace
Maroczy
Purcell
Schmidt
Schopenhauer
Semmelweiss
Smith
Steinmetz
Tukhashevsky
Turing
Venn
Warshawski
Znosko-Borowski

#  This is a data file for
#+ "redir2.sh", "redir3.sh", "redir4.sh", "redir4a.sh", "redir5.sh".
```

Redirecting the `stdout` of a code block has the effect of saving its output to a file. See Example 3-2.

Here documents are a special case of redirected code blocks. That being the case, it should be possible to feed the output of a *here document* into the `stdin` for a *while loop*.

```
# This example by Albert Siersema
# Used with permission (thanks!).

function doesOutput()
 # Could be an external command too, of course.
 # Here we show you can use a function as well.
{
  ls -al *.jpg | awk '{print $5,$9}'
}


nr=0          #  We want the while loop to be able to manipulate these and
totalSize=0   #+ to be able to see the changes after the 'while' finished.
```

```
while read fileSize fileName ; do
  echo "$fileName is $fileSize bytes"
  let nr++
  totalSize=$((totalSize+fileSize))   # Or: "let totalSize+=fileSize"
done<<EOF
$(doesOutput)
EOF

echo "$nr files totaling $totalSize bytes"
```

20.3. Applications

Clever use of I/O redirection permits parsing and stitching together snippets of command output (see Example 15-7). This permits generating report and log files.

Example 20-12. Logging events

```
#!/bin/bash
# logevents.sh
# Author: Stephane Chazelas.
# Used in ABS Guide with permission.

# Event logging to a file.
# Must be run as root (for write access in /var/log).

ROOT_UID=0     # Only users with $UID 0 have root privileges.
E_NOTROOT=67   # Non-root exit error.


if [ "$UID" -ne "$ROOT_UID" ]
then
  echo "Must be root to run this script."
  exit $E_NOTROOT
fi


FD_DEBUG1=3
FD_DEBUG2=4
FD_DEBUG3=5

# === Uncomment one of the two lines below to activate script. ===
# LOG_EVENTS=1
# LOG_VARS=1


log()  # Writes time and date to log file.
{
echo "$(date)  $*" >&7     # This *appends* the date to the file.
#     ^^^^^^^  command substitution
                           # See below.
}



case $LOG_LEVEL in
 1) exec 3>&2         4> /dev/null 5> /dev/null;;
 2) exec 3>&2         4>&2         5> /dev/null;;
```

```
 3) exec 3>&2         4>&2         5>&2;;
 *) exec 3> /dev/null 4> /dev/null 5> /dev/null;;
esac

FD_LOGVARS=6
if [[ $LOG_VARS ]]
then exec 6>> /var/log/vars.log
else exec 6> /dev/null                     # Bury output.
fi

FD_LOGEVENTS=7
if [[ $LOG_EVENTS ]]
then
  # exec 7 >(exec gawk '{print strftime(), $0}' >> /var/log/event.log)
  # Above line fails in versions of Bash more recent than 2.04. Why?
  exec 7>> /var/log/event.log              # Append to "event.log".
  log                                      # Write time and date.
else exec 7> /dev/null                     # Bury output.
fi

echo "DEBUG3: beginning" >&${FD_DEBUG3}

ls -l >&5 2>&4                             # command1 >&5 2>&4

echo "Done"                                # command2

echo "sending mail" >&${FD_LOGEVENTS}
# Writes "sending mail" to file descriptor #7.


exit 0
```

Chapter 21. Subshells

Running a shell script launches a new process, a *subshell*.

> **Definition:** A *subshell* is a child process launched by a shell (or *shell script*).

A subshell is a separate instance of the command processor -- the *shell* that gives you the prompt at the console or in an *xterm* window. Just as your commands are interpreted at the command-line prompt, similarly does a script batch-process a list of commands. Each shell script running is, in effect, a subprocess (*child process*) of the parent shell.

A shell script can itself launch subprocesses. These *subshells* let the script do parallel processing, in effect executing multiple subtasks simultaneously.

```
#!/bin/bash
# subshell-test.sh

(
# Inside parentheses, and therefore a subshell . . .
while [ 1 ]   # Endless loop.
do
  echo "Subshell running . . ."
done
)

#  Script will run forever,
#+ or at least until terminated by a Ctl-C.

exit $?  # End of script (but will never get here).



Now, run the script:
sh subshell-test.sh

And, while the script is running, from a different xterm:
ps -ef | grep subshell-test.sh

UID       PID   PPID  C STIME TTY      TIME     CMD
500       2698  2502  0 14:26 pts/4    00:00:00 sh subshell-test.sh
500       2699  2698 21 14:26 pts/4    00:00:24 sh subshell-test.sh

          ^^^^

Analysis:
PID 2698, the script, launched PID 2699, the subshell.

Note: The "UID ..." line would be filtered out by the "grep" command,
but is shown here for illustrative purposes.
```

In general, an external command in a script forks off a subprocess, [107] whereas a Bash builtin does not. For this reason, builtins execute more quickly and use fewer system resources than their external command equivalents.

Command List within Parentheses

(command1; command2; command3; ...)

A command list embedded between `parentheses` runs as a subshell.

Variables in a subshell are *not* visible outside the block of code in the subshell. They are not accessible to the parent process, to the shell that launched the subshell. These are, in effect, variables local to the *child process*.

Example 21-1. Variable scope in a subshell

```
#!/bin/bash
# subshell.sh

echo

echo "We are outside the subshell."
echo "Subshell level OUTSIDE subshell = $BASH_SUBSHELL"
# Bash, version 3, adds the new         $BASH_SUBSHELL variable.
echo; echo

outer_variable=Outer
global_variable=
#  Define global variable for "storage" of
#+ value of subshell variable.

(
echo "We are inside the subshell."
echo "Subshell level INSIDE subshell = $BASH_SUBSHELL"
inner_variable=Inner

echo "From inside subshell, \"inner_variable\" = $inner_variable"
echo "From inside subshell, \"outer\" = $outer_variable"

global_variable="$inner_variable"   #  Will this allow "exporting"
                                    #+ a subshell variable?
)

echo; echo
echo "We are outside the subshell."
echo "Subshell level OUTSIDE subshell = $BASH_SUBSHELL"
echo

if [ -z "$inner_variable" ]
then
  echo "inner_variable undefined in main body of shell"
else
  echo "inner_variable defined in main body of shell"
fi

echo "From main body of shell, \"inner_variable\" = $inner_variable"
#  $inner_variable will show as blank (uninitialized)
#+ because variables defined in a subshell are "local variables".
#  Is there a remedy for this?
echo "global_variable = "$global_variable""  # Why doesn't this work?

echo

# =======================================================================

# Additionally ...
```

```
echo "-----------------"; echo

var=41                                                 # Global variable.

( let "var+=1"; echo "\$var INSIDE subshell = $var" )  # 42

echo "\$var OUTSIDE subshell = $var"                   # 41
#  Variable operations inside a subshell, even to a GLOBAL variable
#+ do not affect the value of the variable outside the subshell!


exit 0

#  Question:
#  --------
#  Once having exited a subshell,
#+ is there any way to reenter that very same subshell
#+ to modify or access the subshell variables?
```

See also $BASHPID and Example 34-2.

Definition: The *scope* of a variable is the context in which it has meaning, in which it has a *value* that can be referenced. For example, the scope of a local variable lies only within the function, block of code, or subshell within which it is defined, while the scope of a *global* variable is the entire script in which it appears.

While the $BASH_SUBSHELL internal variable indicates the nesting level of a subshell, the $SHLVL variable *shows no change* within a subshell.

```
echo " \$BASH_SUBSHELL outside subshell       = $BASH_SUBSHELL"           # 0
  ( echo " \$BASH_SUBSHELL inside subshell        = $BASH_SUBSHELL" )     # 1
  ( ( echo " \$BASH_SUBSHELL inside nested subshell = $BASH_SUBSHELL" ) ) # 2
# ^ ^                           *** nested ***                        ^ ^

echo

echo " \$SHLVL outside subshell = $SHLVL"       # 3
( echo " \$SHLVL inside subshell  = $SHLVL" )   # 3 (No change!)
```

Directory changes made in a subshell do not carry over to the parent shell.

Example 21-2. List User Profiles

```
#!/bin/bash
# allprofs.sh: Print all user profiles.

# This script written by Heiner Steven, and modified by the document author.

FILE=.bashrc  #  File containing user profile,
              #+ was ".profile" in original script.

for home in `awk -F: '{print $6}' /etc/passwd`
do
  [ -d "$home" ] || continue    # If no home directory, go to next.
  [ -r "$home" ] || continue    # If not readable, go to next.
  (cd $home; [ -e $FILE ] && less $FILE)
```

```
done

#  When script terminates, there is no need to 'cd' back to original directory,
#+ because 'cd $home' takes place in a subshell.

exit 0
```

A subshell may be used to set up a "dedicated environment" for a command group.

```
COMMAND1
COMMAND2
COMMAND3
(
  IFS=:
  PATH=/bin
  unset TERMINFO
  set -C
  shift 5
  COMMAND4
  COMMAND5
  exit 3 # Only exits the subshell!
)
# The parent shell has not been affected, and the environment is preserved.
COMMAND6
COMMAND7
```

As seen here, the exit command only terminates the subshell in which it is running, *not* the parent shell or script.

One application of such a "dedicated environment" is testing whether a variable is defined.

```
if (set -u; : $variable) 2> /dev/null
then
  echo "Variable is set."
fi     #  Variable has been set in current script,
       #+ or is an an internal Bash variable,
       #+ or is present in environment (has been exported).

# Could also be written [[ ${variable-x} != x || ${variable-y} != y ]]
# or                    [[ ${variable-x} != x$variable ]]
# or                    [[ ${variable+x} = x ]]
# or                    [[ ${variable-x} != x ]]
```

Another application is checking for a lock file:

```
if (set -C; : > lock_file) 2> /dev/null
then
  :   # lock_file didn't exist: no user running the script
else
  echo "Another user is already running that script."
exit 65
fi

#  Code snippet by Stéphane Chazelas,
#+ with modifications by Paulo Marcel Coelho Aragao.
```

+

Processes may execute in parallel within different subshells. This permits breaking a complex task into subcomponents processed concurrently.

Example 21-3. Running parallel processes in subshells

```
        (cat list1 list2 list3 | sort | uniq > list123) &
        (cat list4 list5 list6 | sort | uniq > list456) &
        # Merges and sorts both sets of lists simultaneously.
        # Running in background ensures parallel execution.
        #
        # Same effect as
        #   cat list1 list2 list3 | sort | uniq > list123 &
        #   cat list4 list5 list6 | sort | uniq > list456 &

        wait   # Don't execute the next command until subshells finish.

        diff list123 list456
```

Redirecting I/O to a subshell uses the "|" pipe operator, as in **`ls -al | (command)`**.

A code block between curly brackets does *not* launch a subshell.

{ command1; command2; command3; . . . commandN; }

```
var1=23
echo "$var1"   # 23

{ var1=76; }
echo "$var1"   # 76
```

Chapter 22. Restricted Shells

Disabled commands in restricted shells

. Running a script or portion of a script in *restricted mode* disables certain commands that would otherwise be available. This is a security measure intended to limit the privileges of the script user and to minimize possible damage from running the script.

The following commands and actions are disabled:

- Using *cd* to change the working directory.
- Changing the values of the *$PATH*, *$SHELL*, *$BASH_ENV*, or *$ENV* environmental variables.
- Reading or changing the *$SHELLOPTS*, shell environmental options.
- Output redirection.
- Invoking commands containing one or more /'s.
- Invoking exec to substitute a different process for the shell.
- Various other commands that would enable monkeying with or attempting to subvert the script for an unintended purpose.
- Getting out of restricted mode within the script.

Example 22-1. Running a script in restricted mode

```
#!/bin/bash

#  Starting the script with "#!/bin/bash -r"
#+ runs entire script in restricted mode.

echo

echo "Changing directory."
cd /usr/local
echo "Now in `pwd`"
echo "Coming back home."
cd
echo "Now in `pwd`"
echo

# Everything up to here in normal, unrestricted mode.

set -r
# set --restricted    has same effect.
echo "==> Now in restricted mode. <=="

echo
echo

echo "Attempting directory change in restricted mode."
cd ..
echo "Still in `pwd`"

echo
echo

echo "\$SHELL = $SHELL"
echo "Attempting to change shell in restricted mode."
```

```
SHELL="/bin/ash"
echo
echo "\$SHELL= $SHELL"

echo
echo

echo "Attempting to redirect output in restricted mode."
ls -l /usr/bin > bin.files
ls -l bin.files    # Try to list attempted file creation effort.

echo

exit 0
```

Chapter 23. Process Substitution

Piping the `stdout` of a command into the `stdin` of another is a powerful technique. But, what if you need to pipe the `stdout` of *multiple* commands? This is where *process substitution* comes in.

Process substitution feeds the output of a process (or processes) into the `stdin` of another process.

Template

Command list enclosed within parentheses

>(command_list)

<(command_list)

Process substitution uses `/dev/fd/<n>` files to send the results of the process(es) within parentheses to another process. [108]

There is *no* space between the the "<" or ">" and the parentheses. Space there would give an error message.

```
bash$ echo >(true)
/dev/fd/63

bash$ echo <(true)
/dev/fd/63

bash$ echo >(true) <(true)
/dev/fd/63 /dev/fd/62



bash$ wc <(cat /usr/share/dict/linux.words)
 483523  483523 4992010 /dev/fd/63

bash$ grep script /usr/share/dict/linux.words | wc
    262     262    3601

bash$ wc <(grep script /usr/share/dict/linux.words)
    262     262    3601 /dev/fd/63
```

Bash creates a pipe with two file descriptors, `--fIn` and `fOut--`. The `stdin` of true connects to `fOut` (dup2(fOut, 0)), then Bash passes a `/dev/fd/fIn` argument to **echo**. On systems lacking `/dev/fd/<n>` files, Bash may use temporary files. (Thanks, S.C.)

Process substitution can compare the output of two different commands, or even the output of different options to the same command.

```
bash$ comm <(ls -l) <(ls -al)
total 12
-rw-rw-r--    1 bozo bozo       78 Mar 10 12:58 File0
-rw-rw-r--    1 bozo bozo       42 Mar 10 12:58 File2
-rw-rw-r--    1 bozo bozo      103 Mar 10 12:58 t2.sh
        total 20
        drwxrwxrwx    2 bozo bozo     4096 Mar 10 18:10 .
```

```
        drwx------   72 bozo bozo     4096 Mar 10 17:58 ..
        -rw-rw-r--    1 bozo bozo       78 Mar 10 12:58 File0
        -rw-rw-r--    1 bozo bozo       42 Mar 10 12:58 File2
        -rw-rw-r--    1 bozo bozo      103 Mar 10 12:58 t2.sh
```

Process substitution can compare the contents of two directories -- to see which filenames are in one, but not the other.

```
diff <(ls $first_directory) <(ls $second_directory)
```

Some other usages and uses of process substitution:

```
read -a list < <( od -Ad -w24 -t u2 /dev/urandom )
#  Read a list of random numbers from /dev/urandom,
#+ process with "od"
#+ and feed into stdin of "read" . . .

#  From "insertion-sort.bash" example script.
#  Courtesy of JuanJo Ciarlante.
```

```
PORT=6881   # bittorrent

# Scan the port to make sure nothing nefarious is going on.
netcat -l $PORT | tee>(md5sum ->mydata-orig.md5) |
gzip | tee>(md5sum - | sed 's/-$/mydata.lz2/'>mydata-gz.md5)>mydata.gz

# Check the decompression:
  gzip -d<mydata.gz | md5sum -c mydata-orig.md5)
# The MD5sum of the original checks stdin and detects compression issues.

#  Bill Davidsen contributed this example
#+ (with light edits by the ABS Guide author).
```

```
cat <(ls -l)
# Same as     ls -l | cat

sort -k 9 <(ls -l /bin) <(ls -l /usr/bin) <(ls -l /usr/X11R6/bin)
# Lists all the files in the 3 main 'bin' directories, and sorts by filename.
# Note that three (count 'em) distinct commands are fed to 'sort'.


diff <(command1) <(command2)    # Gives difference in command output.

tar cf >(bzip2 -c > file.tar.bz2) $directory_name
# Calls "tar cf /dev/fd/?? $directory_name", and "bzip2 -c > file.tar.bz2".
#
# Because of the /dev/fd/<n> system feature,
# the pipe between both commands does not need to be named.
#
# This can be emulated.
#
bzip2 -c < pipe > file.tar.bz2&
tar cf pipe $directory_name
rm pipe
#        or
exec 3>&1
tar cf /dev/fd/4 $directory_name 4>&1 >&3 3>&- | bzip2 -c > file.tar.bz2 3>&-
exec 3>&-
```

```
# Thanks, Stéphane Chazelas
```

Here is a method of circumventing the problem of an *echo* piped to a *while-read loop* running in a subshell.

Example 23-1. Code block redirection without forking

```
#!/bin/bash
# wr-ps.bash: while-read loop with process substitution.

# This example contributed by Tomas Pospisek.
# (Heavily edited by the ABS Guide author.)

echo

echo "random input" | while read i
do
  global=3D": Not available outside the loop."
  # ... because it runs in a subshell.
done

echo "\$global (from outside the subprocess) = $global"
# $global (from outside the subprocess) =

echo; echo "--"; echo

while read i
do
  echo $i
  global=3D": Available outside the loop."
  # ... because it does NOT run in a subshell.
done < <( echo "random input" )
#    ^ ^

echo "\$global (using process substitution) = $global"
# Random input
# $global (using process substitution) = 3D: Available outside the loop.


echo; echo "##########"; echo



# And likewise . . .

declare -a inloop
index=0
cat $0 | while read line
do
  inloop[$index]="$line"
  ((index++))
  # It runs in a subshell, so ...
done
echo "OUTPUT = "
echo ${inloop[*]}           # ... nothing echoes.


echo; echo "--"; echo


declare -a outloop
```

```
index=0
while read line
do
  outloop[$index]="$line"
  ((index++))
  # It does NOT run in a subshell, so ...
done < <( cat $0 )
echo "OUTPUT = "
echo ${outloop[*]}          # ... the entire script echoes.

exit $?
```

This is a similar example.

Example 23-2. Redirecting the output of *process substitution* into a loop.

```
#!/bin/bash
# psub.bash

# As inspired by Diego Molina (thanks!).

declare -a array0
while read
do
  array0[${#array0[@]}]="$REPLY"
done < <( sed -e 's/bash/CRASH-BANG!/' $0 | grep bin | awk '{print $1}' )
#  Sets the default 'read' variable, $REPLY, by process substitution,
#+ then copies it into an array.

echo "${array0[@]}"

exit $?

# ====================================== #

bash psub.bash

#!/bin/CRASH-BANG! done #!/bin/CRASH-BANG!
```

A reader sent in the following interesting example of process substitution.

```
# Script fragment taken from SuSE distribution:

# --------------------------------------------------------------#
while read  des what mask iface; do
# Some commands ...
done < <(route -n)
#    ^ ^  First < is redirection, second is process substitution.

# To test it, let's make it do something.
while read  des what mask iface; do
  echo $des $what $mask $iface
done < <(route -n)

# Output:
# Kernel IP routing table
# Destination Gateway Genmask Flags Metric Ref Use Iface
# 127.0.0.0 0.0.0.0 255.0.0.0 U 0 0 0 lo
# --------------------------------------------------------------#

#  As Stéphane Chazelas points out,
```

```
#+ an easier-to-understand equivalent is:
route -n |
  while read des what mask iface; do   # Variables set from output of pipe.
    echo $des $what $mask $iface
  done  #  This yields the same output as above.
        #  However, as Ulrich Gayer points out . . .
        #+ this simplified equivalent uses a subshell for the while loop,
        #+ and therefore the variables disappear when the pipe terminates.

# -------------------------------------------------------------#

#  However, Filip Moritz comments that there is a subtle difference
#+ between the above two examples, as the following shows.

(
route -n | while read x; do ((y++)); done
echo $y # $y is still unset

while read x; do ((y++)); done < <(route -n)
echo $y # $y has the number of lines of output of route -n
)

More generally spoken
(
: | x=x
# seems to start a subshell like
: | ( x=x )
# while
x=x < <(:)
# does not
)

# This is useful, when parsing csv and the like.
# That is, in effect, what the original SuSE code fragment does.
```

Chapter 24. Functions

Like "real" programming languages, Bash has functions, though in a somewhat limited implementation. A function is a subroutine, a code block that implements a set of operations, a "black box" that performs a specified task. Wherever there is repetitive code, when a task repeats with only slight variations in procedure, then consider using a function.

function *function_name* {
command...
}

or

function_name () {
command...
}

This second form will cheer the hearts of C programmers (and is more portable).

As in C, the function's opening bracket may optionally appear on the second line.

function_name ()
{
command...
}

A function may be "compacted" into a single line.

```
fun () { echo "This is a function"; echo; }
#                                 ^     ^
```

In this case, however, a *semicolon* must follow the final command in the function.

```
fun () { echo "This is a function"; echo } # Error!
#                                       ^

fun2 () { echo "Even a single-command function? Yes!"; }
#                                                    ^
```

Functions are called, *triggered*, simply by invoking their names. *A function call is equivalent to a command.*

Example 24-1. Simple functions

```
#!/bin/bash
# ex59.sh: Exercising functions (simple).

JUST_A_SECOND=1

funky ()
{ # This is about as simple as functions get.
  echo "This is a funky function."
  echo "Now exiting funky function."
```

```
} # Function declaration must precede call.


fun ()
{ # A somewhat more complex function.
  i=0
  REPEATS=30

  echo
  echo "And now the fun really begins."
  echo

  sleep $JUST_A_SECOND    # Hey, wait a second!
  while [ $i -lt $REPEATS ]
  do
    echo "----------FUNCTIONS---------->"
    echo "<------------ARE------------->"
    echo "<------------FUN------------>"
    echo
    let "i+=1"
  done
}

  # Now, call the functions.

funky
fun

exit $?
```

The function definition must precede the first call to it. There is no method of "declaring" the function, as, for example, in C.

```
f1
# Will give an error message, since function "f1" not yet defined.

declare -f f1      # This doesn't help either.
f1                 # Still an error message.

# However...


f1 ()
{
  echo "Calling function \"f2\" from within function \"f1\"."
  f2
}

f2 ()
{
  echo "Function \"f2\"."
}

f1  #  Function "f2" is not actually called until this point,
    #+ although it is referenced before its definition.
    #  This is permissible.

    # Thanks, S.C.
```

Functions may not be empty!

```
#!/bin/bash
# empty-function.sh

empty ()
{
}

exit 0  # Will not exit here!

# $ sh empty-function.sh
# empty-function.sh: line 6: syntax error near unexpected token `}'
# empty-function.sh: line 6: `}'

# $ echo $?
# 2


# Note that a function containing only comments is empty.

func ()
{
  # Comment 1.
  # Comment 2.
  # This is still an empty function.
  # Thank you, Mark Bova, for pointing this out.
}
# Results in same error message as above.


# However ...

not_quite_empty ()
{
  illegal_command
} #  A script containing this function will *not* bomb
  #+ as long as the function is not called.

not_empty ()
{
  :
} # Contains a : (null command), and this is okay.


# Thank you, Dominick Geyer and Thiemo Kellner.
```

It is even possible to nest a function within another function, although this is not very useful.

```
f1 ()
{

  f2 () # nested
  {
    echo "Function \"f2\", inside \"f1\"."
  }

}

f2  #  Gives an error message.
    #  Even a preceding "declare -f f2" wouldn't help.

echo
```

```
f1  #  Does nothing, since calling "f1" does not automatically call "f2".
f2  #  Now, it's all right to call "f2",
    #+ since its definition has been made visible by calling "f1".

    # Thanks, S.C.
```

Function declarations can appear in unlikely places, even where a command would otherwise go.

```
ls -l | foo() { echo "foo"; }  # Permissible, but useless.



if [ "$USER" = bozo ]
then
  bozo_greet ()   # Function definition embedded in an if/then construct.
  {
    echo "Hello, Bozo."
  }
fi

bozo_greet        # Works only for Bozo, and other users get an error.



# Something like this might be useful in some contexts.
NO_EXIT=1   # Will enable function definition below.

[[ $NO_EXIT -eq 1 ]] && exit() { true; }     # Function definition in an "and-list".
# If $NO_EXIT is 1, declares "exit ()".
# This disables the "exit" builtin by aliasing it to "true".

exit  # Invokes "exit ()" function, not "exit" builtin.



# Or, similarly:
filename=file1

[ -f "$filename" ] &&
foo () { rm -f "$filename"; echo "File "$filename" deleted."; } ||
foo () { echo "File "$filename" not found."; touch bar; }

foo

# Thanks, S.C. and Christopher Head
```

Function names can take strange forms.

```
  _(){ for i in {1..10}; do echo -n "$FUNCNAME"; done; echo; }
# ^^^         No space between function name and parentheses.
#             This doesn't always work. Why not?

# Now, let's invoke the function.
  _         # __________
#             ^^^^^^^^^^   10 underscores (10 x function name)!
# A "naked" underscore is an acceptable function name.


# In fact, a colon is likewise an acceptable function name.

:(){ echo ":"; }; :

# Of what use is this?
```

```
# It's a devious way to obfuscate the code in a script.
```

See also Example A-56

What happens when different versions of the same function appear in a script?

```
#  As Yan Chen points out,
#  when a function is defined multiple times,
#  the final version is what is invoked.
#  This is not, however, particularly useful.

func ()
{
  echo "First version of func ()."
}

func ()
{
  echo "Second version of func ()."
}

func   # Second version of func ().

exit $?

#  It is even possible to use functions to override
#+ or preempt system commands.
#  Of course, this is *not* advisable.
```

24.1. Complex Functions and Function Complexities

Functions may process arguments passed to them and return an exit status to the script for further processing.

```
function_name $arg1 $arg2
```

The function refers to the passed arguments by position (as if they were positional parameters), that is, `$1`, `$2`, and so forth.

Example 24-2. Function Taking Parameters

```
#!/bin/bash
# Functions and parameters

DEFAULT=default                             # Default param value.

func2 () {
   if [ -z "$1" ]                           # Is parameter #1 zero length?
   then
     echo "-Parameter #1 is zero length.-"  # Or no parameter passed.
   else
     echo "-Parameter #1 is \"$1\".-"
   fi

   variable=${1-$DEFAULT}                   #  What does
   echo "variable = $variable"              #+ parameter substitution show?
                                            #  ---------------------------
                                            #  It distinguishes between
```

```
                                            #+ no param and a null param.

   if [ "$2" ]
   then
     echo "-Parameter #2 is \"$2\".-"
   fi

   return 0
}

echo

echo "Nothing passed."
func2                          # Called with no params
echo


echo "Zero-length parameter passed."
func2 ""                       # Called with zero-length param
echo

echo "Null parameter passed."
func2 "$uninitialized_param"   # Called with uninitialized param
echo

echo "One parameter passed."
func2 first           # Called with one param
echo

echo "Two parameters passed."
func2 first second    # Called with two params
echo

echo "\"\" \"second\" passed."
func2 "" second       # Called with zero-length first parameter
echo                  # and ASCII string as a second one.

exit 0
```

The shift command works on arguments passed to functions (see Example 36-18).

But, what about command-line arguments passed to the script? Does a function see them? Well, let's clear up the confusion.

Example 24-3. Functions and command-line args passed to the script

```
#!/bin/bash
# func-cmdlinearg.sh
#  Call this script with a command-line argument,
#+ something like $0 arg1.


func ()

{
echo "$1"   # Echoes first arg passed to the function.
}           # Does a command-line arg qualify?

echo "First call to function: no arg passed."
```

```
echo "See if command-line arg is seen."
func
# No! Command-line arg not seen.

echo "============================================================"
echo
echo "Second call to function: command-line arg passed explicitly."
func $1
# Now it's seen!

exit 0
```

In contrast to certain other programming languages, shell scripts normally pass only value parameters to functions. Variable names (which are actually *pointers*), if passed as parameters to functions, will be treated as string literals. *Functions interpret their arguments literally.*

Indirect variable references (see Example 37-2) provide a clumsy sort of mechanism for passing variable pointers to functions.

Example 24-4. Passing an indirect reference to a function

```
#!/bin/bash
# ind-func.sh: Passing an indirect reference to a function.

echo_var ()
{
echo "$1"
}

message=Hello
Hello=Goodbye

echo_var "$message"        # Hello
# Now, let's pass an indirect reference to the function.
echo_var "${!message}"     # Goodbye

echo "-------------"

# What happens if we change the contents of "hello" variable?
Hello="Hello, again!"
echo_var "$message"        # Hello
echo_var "${!message}"     # Hello, again!

exit 0
```

The next logical question is whether parameters can be dereferenced *after* being passed to a function.

Example 24-5. Dereferencing a parameter passed to a function

```
#!/bin/bash
# dereference.sh
# Dereferencing parameter passed to a function.
# Script by Bruce W. Clare.

dereference ()
{
     y=\$"$1"   # Name of variable (not value!).
```

```
     echo $y    # $Junk

     x=`eval "expr \"$y\" "`
     echo $1=$x
     eval "$1=\"Some Different Text \""  # Assign new value.
}

Junk="Some Text"
echo $Junk "before"    # Some Text before

dereference Junk
echo $Junk "after"     # Some Different Text after

exit 0
```

Example 24-6. Again, dereferencing a parameter passed to a function

```
#!/bin/bash
# ref-params.sh: Dereferencing a parameter passed to a function.
#                (Complex Example)

ITERATIONS=3  # How many times to get input.
icount=1

my_read () {
  #  Called with my_read varname,
  #+ outputs the previous value between brackets as the default value,
  #+ then asks for a new value.

  local local_var

  echo -n "Enter a value "
  eval 'echo -n "[$'$1'] "'  #  Previous value.
# eval echo -n "[\$$1] "     #  Easier to understand,
                             #+ but loses trailing space in user prompt.
  read local_var
  [ -n "$local_var" ] && eval $1=\$local_var

  # "And-list": if "local_var" then set "$1" to its value.
}

echo

while [ "$icount" -le "$ITERATIONS" ]
do
  my_read var
  echo "Entry #$icount = $var"
  let "icount += 1"
  echo
done


# Thanks to Stephane Chazelas for providing this instructive example.

exit 0
```

Exit and Return

exit status

Functions return a value, called an *exit status*. This is analogous to the exit status returned by a command. The exit status may be explicitly specified by a **return** statement, otherwise it is the exit status of the last command in the function (0 if successful, and a non-zero error code if not). This exit status may be used in the script by referencing it as $?. This mechanism effectively permits script functions to have a "return value" similar to C functions.

return

Terminates a function. A **return** command [109] optionally takes an *integer* argument, which is returned to the calling script as the "exit status" of the function, and this exit status is assigned to the variable $?.

Example 24-7. Maximum of two numbers

```
#!/bin/bash
# max.sh: Maximum of two integers.

E_PARAM_ERR=250    # If less than 2 params passed to function.
EQUAL=251          # Return value if both params equal.
#  Error values out of range of any
#+ params that might be fed to the function.

max2 ()             # Returns larger of two numbers.
{                   # Note: numbers compared must be less than 250.
if [ -z "$2" ]
then
  return $E_PARAM_ERR
fi

if [ "$1" -eq "$2" ]
then
  return $EQUAL
else
  if [ "$1" -gt "$2" ]
  then
    return $1
  else
    return $2
  fi
fi
}

max2 33 34
return_val=$?

if [ "$return_val" -eq $E_PARAM_ERR ]
then
  echo "Need to pass two parameters to the function."
elif [ "$return_val" -eq $EQUAL ]
  then
    echo "The two numbers are equal."
else
    echo "The larger of the two numbers is $return_val."
fi


exit 0

#  Exercise (easy):
```

```
#  ---------------
#  Convert this to an interactive script,
#+ that is, have the script ask for input (two numbers).
```

For a function to return a string or array, use a dedicated variable.

```
count_lines_in_etc_passwd()
{
  [[ -r /etc/passwd ]] && REPLY=$(echo $(wc -l < /etc/passwd))
  #  If /etc/passwd is readable, set REPLY to line count.
  #  Returns both a parameter value and status information.
  #  The 'echo' seems unnecessary, but . . .
  #+ it removes excess whitespace from the output.
}

if count_lines_in_etc_passwd
then
  echo "There are $REPLY lines in /etc/passwd."
else
  echo "Cannot count lines in /etc/passwd."
fi

# Thanks, S.C.
```

Example 24-8. Converting numbers to Roman numerals

```
#!/bin/bash

# Arabic number to Roman numeral conversion
# Range: 0 - 200
# It's crude, but it works.

# Extending the range and otherwise improving the script is left as an exercise.

# Usage: roman number-to-convert

LIMIT=200
E_ARG_ERR=65
E_OUT_OF_RANGE=66

if [ -z "$1" ]
then
  echo "Usage: `basename $0` number-to-convert"
  exit $E_ARG_ERR
fi

num=$1
if [ "$num" -gt $LIMIT ]
then
  echo "Out of range!"
  exit $E_OUT_OF_RANGE
fi

to_roman ()   # Must declare function before first call to it.
{
number=$1
factor=$2
rchar=$3
let "remainder = number - factor"
while [ "$remainder" -ge 0 ]
```

```
do
  echo -n $rchar
  let "number -= factor"
  let "remainder = number - factor"
done

return $number
       # Exercises:
       # ---------
       # 1) Explain how this function works.
       #    Hint: division by successive subtraction.
       # 2) Extend to range of the function.
       #    Hint: use "echo" and command-substitution capture.
}


to_roman $num 100 C
num=$?
to_roman $num 90 LXXXX
num=$?
to_roman $num 50 L
num=$?
to_roman $num 40 XL
num=$?
to_roman $num 10 X
num=$?
to_roman $num 9 IX
num=$?
to_roman $num 5 V
num=$?
to_roman $num 4 IV
num=$?
to_roman $num 1 I
# Successive calls to conversion function!
# Is this really necessary??? Can it be simplified?

echo

exit
```

See also Example 11-29.

The largest positive integer a function can return is 255. The **return** command is closely tied to the concept of exit status, which accounts for this particular limitation. Fortunately, there are various workarounds for those situations requiring a large integer return value from a function.

Example 24-9. Testing large return values in a function

```
#!/bin/bash
# return-test.sh

# The largest positive value a function can return is 255.

return_test ()         # Returns whatever passed to it.
{
  return $1
}
```

```
return_test 27          # o.k.
echo $?                 # Returns 27.

return_test 255         # Still o.k.
echo $?                 # Returns 255.

return_test 257         # Error!
echo $?                 # Returns 1 (return code for miscellaneous error).

# ===========================================================
return_test -151896     # Do large negative numbers work?
echo $?                 # Will this return -151896?
                        # No! It returns 168.
#  Version of Bash before 2.05b permitted
#+ large negative integer return values.
#  It happened to be a useful feature.
#  Newer versions of Bash unfortunately plug this loophole.
#  This may break older scripts.
#  Caution!
# ===========================================================

exit 0
```

A workaround for obtaining large integer "return values" is to simply assign the "return value" to a global variable.

```
Return_Val=   # Global variable to hold oversize return value of function.

alt_return_test ()
{
  fvar=$1
  Return_Val=$fvar
  return   # Returns 0 (success).
}

alt_return_test 1
echo $?                              # 0
echo "return value = $Return_Val"    # 1

alt_return_test 256
echo "return value = $Return_Val"    # 256

alt_return_test 257
echo "return value = $Return_Val"    # 257

alt_return_test 25701
echo "return value = $Return_Val"    #25701
```

A more elegant method is to have the function **echo** its "return value to `stdout`," and then capture it by command substitution. See the discussion of this in Section 36.7.

Example 24-10. Comparing two large integers

```
#!/bin/bash
# max2.sh: Maximum of two LARGE integers.

#  This is the previous "max.sh" example,
#+ modified to permit comparing large integers.
```

```
EQUAL=0              # Return value if both params equal.
E_PARAM_ERR=-99999   # Not enough params passed to function.
#           ^^^^^^    Out of range of any params that might be passed.

max2 ()              # "Returns" larger of two numbers.
{
if [ -z "$2" ]
then
  echo $E_PARAM_ERR
  return
fi

if [ "$1" -eq "$2" ]
then
  echo $EQUAL
  return
else
  if [ "$1" -gt "$2" ]
  then
    retval=$1
  else
    retval=$2
  fi
fi

echo $retval         # Echoes (to stdout), rather than returning value.
                     # Why?
}


return_val=$(max2 33001 33997)
#            ^^^^             Function name
#                 ^^^^^ ^^^^^ Params passed
#  This is actually a form of command substitution:
#+ treating a function as if it were a command,
#+ and assigning the stdout of the function to the variable "return_val."


# ========================= OUTPUT ========================
if [ "$return_val" -eq "$E_PARAM_ERR" ]
  then
  echo "Error in parameters passed to comparison function!"
elif [ "$return_val" -eq "$EQUAL" ]
  then
    echo "The two numbers are equal."
else
    echo "The larger of the two numbers is $return_val."
fi
# =========================================================

exit 0

#  Exercises:
#  ---------
#  1) Find a more elegant way of testing
#+    the parameters passed to the function.
#  2) Simplify the if/then structure at "OUTPUT."
#  3) Rewrite the script to take input from command-line parameters.
```

Here is another example of capturing a function "return value." Understanding it requires some knowledge of awk.

```
month_length ()  # Takes month number as an argument.
{                # Returns number of days in month.
monthD="31 28 31 30 31 30 31 31 30 31 30 31"  # Declare as local?
echo "$monthD" | awk '{ print $'"${1}"' }'    # Tricky.
#                             ^^^^^^^^^
# Parameter passed to function  ($1 -- month number), then to awk.
# Awk sees this as "print $1 . . . print $12" (depending on month number)
# Template for passing a parameter to embedded awk script:
#                                 $'"${script_parameter}"'

#    Here's a slightly simpler awk construct:
#    echo $monthD | awk -v month=$1 '{print $(month)}'
#    Uses the -v awk option, which assigns a variable value
#+   prior to execution of the awk program block.
#    Thank you, Rich.

#  Needs error checking for correct parameter range (1-12)
#+ and for February in leap year.
}

# ----------------------------------------------
# Usage example:
month=4        # April, for example (4th month).
days_in=$(month_length $month)
echo $days_in  # 30
# ----------------------------------------------
```

See also Example A-7 and Example A-37.

`Exercise:` Using what we have just learned, extend the previous Roman numerals example to accept arbitrarily large input.

Redirection

`Redirecting the stdin of a function`

A function is essentially a code block, which means its `stdin` can be redirected (as in Example 3-1).

Example 24-11. Real name from username

```
#!/bin/bash
# realname.sh
#
# From username, gets "real name" from /etc/passwd.


ARGCOUNT=1       # Expect one arg.
E_WRONGARGS=85

file=/etc/passwd
pattern=$1

if [ $# -ne "$ARGCOUNT" ]
then
  echo "Usage: `basename $0` USERNAME"
  exit $E_WRONGARGS
fi

file_excerpt ()    #  Scan file for pattern,
{                  #+ then print relevant portion of line.
```

```
  while read line  # "while" does not necessarily need [ condition ]
  do
    echo "$line" | grep $1 | awk -F":" '{ print $5 }'
    # Have awk use ":" delimiter.
  done
} <$file  # Redirect into function's stdin.

file_excerpt $pattern

# Yes, this entire script could be reduced to
#       grep PATTERN /etc/passwd | awk -F":" '{ print $5 }'
# or
#       awk -F: '/PATTERN/ {print $5}'
# or
#       awk -F: '($1 == "username") { print $5 }' # real name from username
# However, it might not be as instructive.

exit 0
```

There is an alternate, and perhaps less confusing method of redirecting a function's `stdin`. This involves redirecting the `stdin` to an embedded bracketed code block within the function.

```
# Instead of:
Function ()
{
 ...
 } < file

# Try this:
Function ()
{
  {
    ...
   } < file
}

# Similarly,

Function ()  # This works.
{
  {
   echo $*
  } | tr a b
}

Function ()  # This doesn't work.
{
  echo $*
} | tr a b   # A nested code block is mandatory here.


# Thanks, S.C.
```

Emmanuel Rouat's sample `bashrc` file contains some instructive examples of functions.

24.2. Local Variables

What makes a variable *local*?

local variables
: A variable declared as *local* is one that is visible only within the block of code in which it appears. It has local scope. In a function, a *local variable* has meaning only within that function block. [110]

Example 24-12. Local variable visibility

```
#!/bin/bash
# ex62.sh: Global and local variables inside a function.

func ()
{
  local loc_var=23       # Declared as local variable.
  echo                   # Uses the 'local' builtin.
  echo "\"loc_var\" in function = $loc_var"
  global_var=999         # Not declared as local.
                         # Therefore, defaults to global.
  echo "\"global_var\" in function = $global_var"
}

func

# Now, to see if local variable "loc_var" exists outside the function.

echo
echo "\"loc_var\" outside function = $loc_var"
                                      # $loc_var outside function =
                                      # No, $loc_var not visible globally.
echo "\"global_var\" outside function = $global_var"
                                      # $global_var outside function = 999
                                      # $global_var is visible globally.
echo

exit 0
#  In contrast to C, a Bash variable declared inside a function
#+ is local ONLY if declared as such.
```

Before a function is called, *all* variables declared within the function are invisible outside the body of the function, not just those explicitly declared as *local*.

```
#!/bin/bash

func ()
{
global_var=37    #  Visible only within the function block
                 #+ before the function has been called.
}                #  END OF FUNCTION

echo "global_var = $global_var"  # global_var =
                                 #  Function "func" has not yet been called,
                                 #+ so $global_var is not visible here.

func
```

```
echo "global_var = $global_var"  # global_var = 37
                                 # Has been set by function call.
```

As Evgeniy Ivanov points out, when declaring and setting a local variable in a single command, apparently the order of operations is to *first set the variable, and only afterwards restrict it to local scope*. This is reflected in the return value.

```
#!/bin/bash

echo "==OUTSIDE Function (global)=="
t=$(exit 1)
echo $?      # 1
             # As expected.
echo

function0 ()
{

echo "==INSIDE Function=="
echo "Global"
t0=$(exit 1)
echo $?      # 1
             # As expected.

echo
echo "Local declared & assigned in same command."
local t1=$(exit 1)
echo $?      # 0
             # Unexpected!
#  Apparently, the variable assignment takes place before
#+ the local declaration.
#+ The return value is for the latter.

echo
echo "Local declared, then assigned (separate commands)."
local t2
t2=$(exit 1)
echo $?      # 1
             # As expected.

}

function0
```

24.2.1. Local variables and recursion.

Recursion is an interesting and sometimes useful form of *self-reference*. Herbert Mayer defines it as ". . . expressing an algorithm by using a simpler version of that same algorithm . . ."

Consider a definition defined in terms of itself, [111] an expression implicit in its own expression, [112] *a snake swallowing its own tail*, [113] or . . . a function that calls itself. [114]

Example 24-13. Demonstration of a simple recursive function

```
#!/bin/bash
# recursion-demo.sh
# Demonstration of recursion.

RECURSIONS=9   # How many times to recurse.
r_count=0      # Must be global. Why?

recurse ()
{
  var="$1"

  while [ "$var" -ge 0 ]
  do
    echo "Recursion count = "$r_count"  +-+  \$var = "$var""
    (( var-- )); (( r_count++ ))
    recurse "$var"  #  Function calls itself (recurses)
  done              #+ until what condition is met?
}

recurse $RECURSIONS

exit $?
```

Example 24-14. Another simple demonstration

```
#!/bin/bash
# recursion-def.sh
# A script that defines "recursion" in a rather graphic way.

RECURSIONS=10
r_count=0
sp=" "

define_recursion ()
{
  ((r_count++))
  sp="$sp"" "
  echo -n "$sp"
  echo "\"The act of recurring ... \""   # Per 1913 Webster's dictionary.

  while [ $r_count -le $RECURSIONS ]
  do
    define_recursion
  done
}

echo
echo "Recursion: "
define_recursion
echo

exit $?
```

Local variables are a useful tool for writing recursive code, but this practice generally involves a great deal of computational overhead and is definitely *not* recommended in a shell script. [115]

Example 24-15. Recursion, using a local variable

```
#!/bin/bash

#               factorial
#               ---------


# Does bash permit recursion?
# Well, yes, but...
# It's so slow that you gotta have rocks in your head to try it.


MAX_ARG=5
E_WRONG_ARGS=85
E_RANGE_ERR=86


if [ -z "$1" ]
then
  echo "Usage: `basename $0` number"
  exit $E_WRONG_ARGS
fi

if [ "$1" -gt $MAX_ARG ]
then
  echo "Out of range ($MAX_ARG is maximum)."
  #  Let's get real now.
  #  If you want greater range than this,
  #+ rewrite it in a Real Programming Language.
  exit $E_RANGE_ERR
fi

fact ()
{
  local number=$1
  #  Variable "number" must be declared as local,
  #+ otherwise this doesn't work.
  if [ "$number" -eq 0 ]
  then
    factorial=1    # Factorial of 0 = 1.
  else
    let "decrnum = number - 1"
    fact $decrnum  # Recursive function call (the function calls itself).
    let "factorial = $number * $?"
  fi

  return $factorial
}

fact $1
echo "Factorial of $1 is $?."

exit 0
```

Also see Example A-15 for an example of recursion in a script. Be aware that recursion is resource-intensive and executes slowly, and is therefore generally not appropriate in a script.

24.3. Recursion Without Local Variables

A function may recursively call itself even without use of local variables.

Example 24-16. ***The Fibonacci Sequence***

```
#!/bin/bash
# fibo.sh : Fibonacci sequence (recursive)
# Author: M. Cooper
# License: GPL3

# ----------algorithm--------------
# Fibo(0) = 0
# Fibo(1) = 1
# else
#   Fibo(j) = Fibo(j-1) + Fibo(j-2)
# ---------------------------------

MAXTERM=15       # Number of terms (+1) to generate.
MINIDX=2         # If idx is less than 2, then Fibo(idx) = idx.

Fibonacci ()
{
  idx=$1   # Doesn't need to be local. Why not?
  if [ "$idx" -lt "$MINIDX" ]
  then
    echo "$idx"  # First two terms are 0 1 ... see above.
  else
    (( --idx ))  # j-1
    term1=$( Fibonacci $idx )   #  Fibo(j-1)

    (( --idx ))  # j-2
    term2=$( Fibonacci $idx )   #  Fibo(j-2)

    echo $(( term1 + term2 ))
  fi
  #  An ugly, ugly kludge.
  #  The more elegant implementation of recursive fibo in C
  #+ is a straightforward translation of the algorithm in lines 7 - 10.
}

for i in $(seq 0 $MAXTERM)
do  # Calculate $MAXTERM+1 terms.
  FIBO=$(Fibonacci $i)
  echo -n "$FIBO "
done
# 0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610
# Takes a while, doesn't it? Recursion in a script is slow.

echo

exit 0
```

Example 24-17. ***The Towers of Hanoi***

```
#! /bin/bash
#
# The Towers Of Hanoi
# Bash script
# Copyright (C) 2000 Amit Singh. All Rights Reserved.
# http://hanoi.kernelthread.com
#
# Tested under Bash version 2.05b.0(13)-release.
# Also works under Bash version 3.x.
#
#  Used in "Advanced Bash Scripting Guide"
#+ with permission of script author.
#  Slightly modified and commented by ABS author.

#==================================================================#
#  The Tower of Hanoi is a mathematical puzzle attributed to
#+ Edouard Lucas, a nineteenth-century French mathematician.
#
#  There are three vertical posts set in a base.
#  The first post has a set of annular rings stacked on it.
#  These rings are disks with a hole drilled out of the center,
#+ so they can slip over the posts and rest flat.
#  The rings have different diameters, and they stack in ascending
#+ order, according to size.
#  The smallest ring is on top, and the largest on the bottom.
#
#  The task is to transfer the stack of rings
#+ to one of the other posts.
#  You can move only one ring at a time to another post.
#  You are permitted to move rings back to the original post.
#  You may place a smaller ring atop a larger one,
#+ but *not* vice versa.
#  Again, it is forbidden to place a larger ring atop a smaller one.
#
#  For a small number of rings, only a few moves are required.
#+ For each additional ring,
#+ the required number of moves approximately doubles,
#+ and the "strategy" becomes increasingly complicated.
#
#  For more information, see http://hanoi.kernelthread.com
#+ or pp. 186-92 of _The Armchair Universe_ by A.K. Dewdney.
#
#
#         ...                   ...                    ...
#         | |                   | |                    | |
#        _|_|_                  | |                    | |
#       |_____|                 | |                    | |
#      |_______|                | |                    | |
#     |_________|               | |                    | |
#    |___________|              | |                    | |
#   |             |             | |                    | |
# .---------------------------------------------------------------.
# |***************************************************************|
#          #1                   #2                      #3
#
#==================================================================#


E_NOPARAM=66  # No parameter passed to script.
E_BADPARAM=67 # Illegal number of disks passed to script.
Moves=        # Global variable holding number of moves.
              # Modification to original script.
```

```
dohanoi() {   # Recursive function.
    case $1 in
    0)
        ;;
    *)
        dohanoi "$(($1-1))" $2 $4 $3
        echo move $2 "-->" $3
        ((Moves++))          # Modification to original script.
        dohanoi "$(($1-1))" $4 $3 $2
        ;;
    esac
}

case $# in
    1) case $(($1>0)) in     # Must have at least one disk.
       1)  # Nested case statement.
           dohanoi $1 1 3 2
           echo "Total moves = $Moves"   # 2^n - 1, where n = # of disks.
           exit 0;
           ;;
       *)
           echo "$0: illegal value for number of disks";
           exit $E_BADPARAM;
           ;;
       esac
    ;;
    *)
       echo "usage: $0 N"
       echo "       Where \"N\" is the number of disks."
       exit $E_NOPARAM;
       ;;
esac

# Exercises:
# ---------
# 1) Would commands beyond this point ever be executed?
#    Why not? (Easy)
# 2) Explain the workings of the workings of the "dohanoi" function.
#    (Difficult -- see the Dewdney reference, above.)
```

Chapter 25. Aliases

A Bash *alias* is essentially nothing more than a keyboard shortcut, an abbreviation, a means of avoiding typing a long command sequence. If, for example, we include **alias lm="ls -l | more"** in the `~/.bashrc` file, then each **`lm`** [116] typed at the command-line will automatically be replaced by a **ls -l | more**. This can save a great deal of typing at the command-line and avoid having to remember complex combinations of commands and options. Setting **alias rm="rm -i"** (interactive mode delete) may save a good deal of grief, since it can prevent inadvertently deleting important files.

In a script, aliases have very limited usefulness. It would be nice if aliases could assume some of the functionality of the **C** preprocessor, such as macro expansion, but unfortunately Bash does not expand arguments within the alias body. [117] Moreover, a script fails to expand an alias itself within "compound constructs," such as if/then statements, loops, and functions. An added limitation is that an alias will not expand recursively. Almost invariably, whatever we would like an alias to do could be accomplished much more effectively with a function.

Example 25-1. Aliases within a script

```
#!/bin/bash
# alias.sh

shopt -s expand_aliases
# Must set this option, else script will not expand aliases.


# First, some fun.
alias Jesse_James='echo "\"Alias Jesse James\" was a 1959 comedy starring Bob Hope."'
Jesse_James

echo; echo; echo;

alias ll="ls -l"
# May use either single (') or double (") quotes to define an alias.

echo "Trying aliased \"ll\":"
ll /usr/X11R6/bin/mk*   #* Alias works.

echo

directory=/usr/X11R6/bin/
prefix=mk*  # See if wild card causes problems.
echo "Variables \"directory\" + \"prefix\" = $directory$prefix"
echo

alias lll="ls -l $directory$prefix"

echo "Trying aliased \"lll\":"
lll         # Long listing of all files in /usr/X11R6/bin stating with mk.
# An alias can handle concatenated variables -- including wild card -- o.k.




TRUE=1
```

```
echo

if [ TRUE ]
then
  alias rr="ls -l"
  echo "Trying aliased \"rr\" within if/then statement:"
  rr /usr/X11R6/bin/mk*   #* Error message results!
  # Aliases not expanded within compound statements.
  echo "However, previously expanded alias still recognized:"
  ll /usr/X11R6/bin/mk*
fi

echo

count=0
while [ $count -lt 3 ]
do
  alias rrr="ls -l"
  echo "Trying aliased \"rrr\" within \"while\" loop:"
  rrr /usr/X11R6/bin/mk*   #* Alias will not expand here either.
                           #  alias.sh: line 57: rrr: command not found
  let count+=1
done

echo; echo

alias xyz='cat $0'   # Script lists itself.
                     # Note strong quotes.
xyz
#  This seems to work,
#+ although the Bash documentation suggests that it shouldn't.
#
#  However, as Steve Jacobson points out,
#+ the "$0" parameter expands immediately upon declaration of the alias.

exit 0
```

The **unalias** command removes a previously set *alias*.

Example 25-2. *unalias*: Setting and unsetting an alias

```
#!/bin/bash
# unalias.sh

shopt -s expand_aliases  # Enables alias expansion.

alias llm='ls -al | more'
llm

echo

unalias llm              # Unset alias.
llm
# Error message results, since 'llm' no longer recognized.

exit 0
```

```
bash$ ./unalias.sh
total 6
drwxrwxr-x    2 bozo     bozo         3072 Feb  6 14:04 .
```

```
drwxr-xr-x   40 bozo     bozo         2048 Feb  6 14:04 ..
-rwxr-xr-x    1 bozo     bozo          199 Feb  6 14:04 unalias.sh

./unalias.sh: llm: command not found
```

Chapter 26. List Constructs

The *and list* and *or list* constructs provide a means of processing a number of commands consecutively. These can effectively replace complex nested if/then or even case statements.

Chaining together commands

and list

```
command-1 && command-2 && command-3 && ... command-n
```

Each command executes in turn, provided that the previous command has given a return value of `true` (zero). At the first `false` (non-zero) return, the command chain terminates (the first command returning `false` is the last one to execute).

An interesting use of a two-condition *and list* from an early version of YongYe's Tetris game script:

```
equation()

{  # core algorithm used for doubling and halving the coordinates
   [[ ${cdx} ]] && ((y=cy+(ccy-cdy)${2}2))
   eval ${1}+=\"${x} ${y} \"
}
```

Example 26-1. Using an *and list* to test for command-line arguments

```
#!/bin/bash
# and list

if [ ! -z "$1" ] && echo "Argument #1 = $1" && [ ! -z "$2" ] && \
#                ^^                         ^^                 ^^
echo "Argument #2 = $2"
then
  echo "At least 2 arguments passed to script."
  # All the chained commands return true.
else
  echo "Fewer than 2 arguments passed to script."
  # At least one of the chained commands returns false.
fi
# Note that "if [ ! -z $1 ]" works, but its alleged equivalent,
#   "if [ -n $1 ]" does not.
#     However, quoting fixes this.
#  if "[ -n "$1" ]" works.
#           ^  ^    Careful!
# It is always best to QUOTE the variables being tested.


# This accomplishes the same thing, using "pure" if/then statements.
if [ ! -z "$1" ]
then
  echo "Argument #1 = $1"
fi
if [ ! -z "$2" ]
then
  echo "Argument #2 = $2"
  echo "At least 2 arguments passed to script."
```

```
else
  echo "Fewer than 2 arguments passed to script."
fi
# It's longer and more ponderous than using an "and list".


exit $?
```

Example 26-2. Another command-line arg test using an *and list*

```
#!/bin/bash

ARGS=1        # Number of arguments expected.
E_BADARGS=85  # Exit value if incorrect number of args passed.

test $# -ne $ARGS && \
#    ^^^^^^^^^^^^ condition #1
echo "Usage: `basename $0` $ARGS argument(s)" && exit $E_BADARGS
#                                             ^^
#  If condition #1 tests true (wrong number of args passed to script),
#+ then the rest of the line executes, and script terminates.

# Line below executes only if the above test fails.
echo "Correct number of arguments passed to this script."

exit 0

# To check exit value, do a "echo $?" after script termination.
```

Of course, an *and list* can also *set* variables to a default value.

```
arg1=$@ && [ -z "$arg1" ] && arg1=DEFAULT

              # Set $arg1 to command-line arguments, if any.
              # But . . . set to DEFAULT if not specified on command-line.
```

or list

```
command-1 || command-2 || command-3 || ... command-n
```

Each command executes in turn for as long as the previous command returns false. At the first true return, the command chain terminates (the first command returning true is the last one to execute). This is obviously the inverse of the "and list".

Example 26-3. Using *or lists* in combination with an *and list*

```
#!/bin/bash

#  delete.sh, a not-so-cunning file deletion utility.
#  Usage: delete filename

E_BADARGS=85

if [ -z "$1" ]
then
  echo "Usage: `basename $0` filename"
  exit $E_BADARGS  # No arg? Bail out.
else
  file=$1          # Set filename.
fi
```

```
[ ! -f "$file" ] && echo "File \"$file\" not found. \
Cowardly refusing to delete a nonexistent file."
# AND LIST, to give error message if file not present.
# Note echo message continuing on to a second line after an escape.

[ ! -f "$file" ] || (rm -f $file; echo "File \"$file\" deleted.")
# OR LIST, to delete file if present.

# Note logic inversion above.
# AND LIST executes on true, OR LIST on false.

exit $?
```

If the first command in an *or list* returns true, it `will` execute.

```
# ==> The following snippets from the /etc/rc.d/init.d/single
#+==> script by Miquel van Smoorenburg
#+==> illustrate use of "and" and "or" lists.
# ==> "Arrowed" comments added by document author.

[ -x /usr/bin/clear ] && /usr/bin/clear
  # ==> If /usr/bin/clear exists, then invoke it.
  # ==> Checking for the existence of a command before calling it
  #+==> avoids error messages and other awkward consequences.

  # ==> . . .

# If they want to run something in single user mode, might as well run it...
for i in /etc/rc1.d/S[0-9][0-9]* ; do
        # Check if the script is there.
        [ -x "$i" ] || continue
  # ==> If corresponding file in $PWD *not* found,
  #+==> then "continue" by jumping to the top of the loop.

        # Reject backup files and files generated by rpm.
        case "$1" in
                *.rpmsave|*.rpmorig|*.rpmnew|*~|*.orig)
                        continue;;
        esac
        [ "$i" = "/etc/rc1.d/S00single" ] && continue
  # ==> Set script name, but don't execute it yet.
        $i start
done

  # ==> . . .
```

The exit status of an **`and list`** or an **`or list`** is the exit status of the last command executed.

Clever combinations of *and* and *or* lists are possible, but the logic may easily become convoluted and require close attention to operator precedence rules, and possibly extensive debugging.

```
false && true || echo false         # false

# Same result as
( false && true ) || echo false     # false
# But NOT
false && ( true || echo false )     # (nothing echoed)
```

```
#  Note left-to-right grouping and evaluation of statements.
#  It's usually best to avoid such complexities.
#  Thanks, S.C.
```

See Example A-7 and Example 7-4 for illustrations of using **`and / or list`** constructs to test variables.

Chapter 27. Arrays

Newer versions of Bash support one-dimensional arrays. Array elements may be initialized with the **`variable[xx]`** notation. Alternatively, a script may introduce the entire array by an explicit **`declare -a variable`** statement. To dereference (retrieve the contents of) an array element, use *curly bracket* notation, that is, **`${element[xx]}`**.

Example 27-1. Simple array usage

```
#!/bin/bash


area[11]=23
area[13]=37
area[51]=UFOs

#  Array members need not be consecutive or contiguous.

#  Some members of the array can be left uninitialized.
#  Gaps in the array are okay.
#  In fact, arrays with sparse data ("sparse arrays")
#+ are useful in spreadsheet-processing software.


echo -n "area[11] = "
echo ${area[11]}    #  {curly brackets} needed.

echo -n "area[13] = "
echo ${area[13]}

echo "Contents of area[51] are ${area[51]}."

# Contents of uninitialized array variable print blank (null variable).
echo -n "area[43] = "
echo ${area[43]}
echo "(area[43] unassigned)"

echo

# Sum of two array variables assigned to third
area[5]=`expr ${area[11]} + ${area[13]}`
echo "area[5] = area[11] + area[13]"
echo -n "area[5] = "
echo ${area[5]}

area[6]=`expr ${area[11]} + ${area[51]}`
echo "area[6] = area[11] + area[51]"
echo -n "area[6] = "
echo ${area[6]}
# This fails because adding an integer to a string is not permitted.

echo; echo; echo

# -----------------------------------------------------------------
# Another array, "area2".
```

```
# Another way of assigning array variables...
# array_name=( XXX YYY ZZZ ... )

area2=( zero one two three four )

echo -n "area2[0] = "
echo ${area2[0]}
# Aha, zero-based indexing (first element of array is [0], not [1]).

echo -n "area2[1] = "
echo ${area2[1]}    # [1] is second element of array.
# -----------------------------------------------------------------

echo; echo; echo

# -----------------------------------------------
# Yet another array, "area3".
# Yet another way of assigning array variables...
# array_name=([xx]=XXX [yy]=YYY ...)

area3=([17]=seventeen [24]=twenty-four)

echo -n "area3[17] = "
echo ${area3[17]}

echo -n "area3[24] = "
echo ${area3[24]}
# -----------------------------------------------

exit 0
```

As we have seen, a convenient way of initializing an entire array is the `array=( element1 element2 ... elementN )` notation.

```
base64_charset=( {A..Z} {a..z} {0..9} + / = )
               #  Using extended brace expansion
               #+ to initialize the elements of the array.
               #  Excerpted from vladz's "base64.sh" script
               #+ in the "Contributed Scripts" appendix.
```

Bash permits array operations on variables, even if the variables are not explicitly declared as arrays.

```
string=abcABC123ABCabc
echo ${string[@]}               # abcABC123ABCabc
echo ${string[*]}               # abcABC123ABCabc
echo ${string[0]}               # abcABC123ABCabc
echo ${string[1]}               # No output!
                                # Why?
echo ${#string[@]}              # 1
                                # One element in the array.
                                # The string itself.

# Thank you, Michael Zick, for pointing this out.
```

Once again this demonstrates that Bash variables are untyped.

Example 27-2. Formatting a poem

```
#!/bin/bash
# poem.sh: Pretty-prints one of the ABS Guide author's favorite poems.

# Lines of the poem (single stanza).
Line[1]="I do not know which to prefer,"
Line[2]="The beauty of inflections"
Line[3]="Or the beauty of innuendoes,"
Line[4]="The blackbird whistling"
Line[5]="Or just after."
# Note that quoting permits embedding whitespace.

# Attribution.
Attrib[1]=" Wallace Stevens"
Attrib[2]="\"Thirteen Ways of Looking at a Blackbird\""
# This poem is in the Public Domain (copyright expired).

echo

tput bold   # Bold print.

for index in 1 2 3 4 5    # Five lines.
do
  printf "     %s\n" "${Line[index]}"
done

for index in 1 2          # Two attribution lines.
do
  printf "          %s\n" "${Attrib[index]}"
done

tput sgr0   # Reset terminal.
            # See 'tput' docs.

echo

exit 0

# Exercise:
# --------
# Modify this script to pretty-print a poem from a text data file.
```

Array variables have a syntax all their own, and even standard Bash commands and operators have special options adapted for array use.

Example 27-3. Various array operations

```
#!/bin/bash
# array-ops.sh: More fun with arrays.


array=( zero one two three four five )
# Element 0   1   2    3     4    5

echo ${array[0]}       #  zero
echo ${array:0}        #  zero
                       #  Parameter expansion of first element,
                       #+ starting at position # 0 (1st character).
echo ${array:1}        #  ero
                       #  Parameter expansion of first element,
```

```
                        #+ starting at position # 1 (2nd character).

echo "--------------"

echo ${#array[0]}       #  4
                        #  Length of first element of array.
echo ${#array}          #  4
                        #  Length of first element of array.
                        #  (Alternate notation)

echo ${#array[1]}       #  3
                        #  Length of second element of array.
                        #  Arrays in Bash have zero-based indexing.

echo ${#array[*]}       #  6
                        #  Number of elements in array.
echo ${#array[@]}       #  6
                        #  Number of elements in array.

echo "--------------"

array2=( [0]="first element" [1]="second element" [3]="fourth element" )
#            ^     ^       ^     ^      ^       ^     ^      ^       ^
# Quoting permits embedding whitespace within individual array elements.

echo ${array2[0]}       # first element
echo ${array2[1]}       # second element
echo ${array2[2]}       #
                        # Skipped in initialization, and therefore null.
echo ${array2[3]}       # fourth element
echo ${#array2[0]}      # 13    (length of first element)
echo ${#array2[*]}      # 3     (number of elements in array)

exit
```

Many of the standard string operations work on arrays.

Example 27-4. String operations on arrays

```
#!/bin/bash
# array-strops.sh: String operations on arrays.

# Script by Michael Zick.
# Used in ABS Guide with permission.
# Fixups: 05 May 08, 04 Aug 08.

#  In general, any string operation using the ${name ... } notation
#+ can be applied to all string elements in an array,
#+ with the ${name[@] ... } or ${name[*] ...} notation.


arrayZ=( one two three four five five )

echo

# Trailing Substring Extraction
echo ${arrayZ[@]:0}     # one two three four five five
#                ^        All elements.
```

```
echo ${arrayZ[@]:1}     # two three four five five
#                ^        All elements following element[0].

echo ${arrayZ[@]:1:2}   # two three
#                  ^      Only the two elements after element[0].

echo "---------"


# Substring Removal

# Removes shortest match from front of string(s).

echo ${arrayZ[@]#f*r}   # one two three five five
#               ^       # Applied to all elements of the array.
                        # Matches "four" and removes it.

# Longest match from front of string(s)
echo ${arrayZ[@]##t*e}  # one two four five five
#               ^^      # Applied to all elements of the array.
                        # Matches "three" and removes it.

# Shortest match from back of string(s)
echo ${arrayZ[@]%h*e}   # one two t four five five
#               ^       # Applied to all elements of the array.
                        # Matches "hree" and removes it.

# Longest match from back of string(s)
echo ${arrayZ[@]%%t*e}  # one two four five five
#               ^^      # Applied to all elements of the array.
                        # Matches "three" and removes it.

echo "----------------------"


# Substring Replacement

# Replace first occurrence of substring with replacement.
echo ${arrayZ[@]/fiv/XYZ}   # one two three four XYZe XYZe
#               ^           # Applied to all elements of the array.

# Replace all occurrences of substring.
echo ${arrayZ[@]//iv/YY}    # one two three four fYYe fYYe
                            # Applied to all elements of the array.

# Delete all occurrences of substring.
# Not specifing a replacement defaults to 'delete' ...
echo ${arrayZ[@]//fi/}      # one two three four ve ve
#               ^^          # Applied to all elements of the array.

# Replace front-end occurrences of substring.
echo ${arrayZ[@]/#fi/XY}    # one two three four XYve XYve
#                ^          # Applied to all elements of the array.

# Replace back-end occurrences of substring.
echo ${arrayZ[@]/%ve/ZZ}    # one two three four fiZZ fiZZ
#                ^          # Applied to all elements of the array.

echo ${arrayZ[@]/%o/XX}     # one twXX three four five five
#                ^          # Why?

echo "-----------------------------"
```

```
replacement() {
    echo -n "!!!"
}

echo ${arrayZ[@]/%e/$(replacement)}
#                 ^  ^^^^^^^^^^^^^^
# on!!! two thre!!! four fiv!!! fiv!!!
# The stdout of replacement() is the replacement string.
# Q.E.D: The replacement action is, in effect, an 'assignment.'

echo "------------------------------------"

#  Accessing the "for-each":
echo ${arrayZ[@]//*/$(replacement optional_arguments)}
#                 ^^ ^^^^^^^^^^^^^
# !!! !!! !!! !!! !!! !!!

#  Now, if Bash would only pass the matched string
#+ to the function being called . . .

echo

exit 0

#  Before reaching for a Big Hammer -- Perl, Python, or all the rest --
#  recall:
#    $( ... ) is command substitution.
#    A function runs as a sub-process.
#    A function writes its output (if echo-ed) to stdout.
#    Assignment, in conjunction with "echo" and command substitution,
#+   can read a function's stdout.
#    The name[@] notation specifies (the equivalent of) a "for-each"
#+   operation.
#  Bash is more powerful than you think!
```

Command substitution can construct the individual elements of an array.

Example 27-5. Loading the contents of a script into an array

```
#!/bin/bash
# script-array.sh: Loads this script into an array.
# Inspired by an e-mail from Chris Martin (thanks!).

script_contents=( $(cat "$0") )  #  Stores contents of this script ($0)
                                 #+ in an array.

for element in $(seq 0 $((${#script_contents[@]} - 1)))
  do                #  ${#script_contents[@]}
                    #+ gives number of elements in the array.
                    #
                    #  Question:
                    #  Why is  seq 0  necessary?
                    #  Try changing it to seq 1.
  echo -n "${script_contents[$element]}"
                    # List each field of this script on a single line.
# echo -n "${script_contents[element]}" also works because of ${ ... }.
  echo -n " -- "    # Use " -- " as a field separator.
done
```

```
echo

exit 0

# Exercise:
# --------
#  Modify this script so it lists itself
#+ in its original format,
#+ complete with whitespace, line breaks, etc.
```

In an array context, some Bash builtins have a slightly altered meaning. For example, unset deletes array elements, or even an entire array.

Example 27-6. Some special properties of arrays

```
#!/bin/bash

declare -a colors
#  All subsequent commands in this script will treat
#+ the variable "colors" as an array.

echo "Enter your favorite colors (separated from each other by a space)."

read -a colors    # Enter at least 3 colors to demonstrate features below.
#  Special option to 'read' command,
#+ allowing assignment of elements in an array.

echo

element_count=${#colors[@]}
# Special syntax to extract number of elements in array.
#     element_count=${#colors[*]} works also.
#
#  The "@" variable allows word splitting within quotes
#+ (extracts variables separated by whitespace).
#
#  This corresponds to the behavior of "$@" and "$*"
#+ in positional parameters.

index=0

while [ "$index" -lt "$element_count" ]
do    # List all the elements in the array.
  echo ${colors[$index]}
  #    ${colors[index]} also works because it's within ${ ... } brackets.
  let "index = $index + 1"
  # Or:
  #    ((index++))
done
# Each array element listed on a separate line.
# If this is not desired, use  echo -n "${colors[$index]} "
#
# Doing it with a "for" loop instead:
#   for i in "${colors[@]}"
#   do
#     echo "$i"
#   done
# (Thanks, S.C.)
```

```
echo

# Again, list all the elements in the array, but using a more elegant method.
  echo ${colors[@]}          # echo ${colors[*]} also works.

echo

# The "unset" command deletes elements of an array, or entire array.
unset colors[1]              # Remove 2nd element of array.
                             # Same effect as   colors[1]=
echo  ${colors[@]}           # List array again, missing 2nd element.

unset colors                 # Delete entire array.
                             #  unset colors[*] and
                             #+ unset colors[@] also work.
echo; echo -n "Colors gone."
echo ${colors[@]}            # List array again, now empty.

exit 0
```

As seen in the previous example, either **${array_name[@]}** or **${array_name[*]}** refers to *all* the elements of the array. Similarly, to get a count of the number of elements in an array, use either **${#array_name[@]}** or **${#array_name[*]}**. **${#array_name}** is the length (number of characters) of **${array_name[0]}**, the first element of the array.

Example 27-7. Of empty arrays and empty elements

```
#!/bin/bash
# empty-array.sh

#  Thanks to Stephane Chazelas for the original example,
#+ and to Michael Zick and Omair Eshkenazi, for extending it.
#  And to Nathan Coulter for clarifications and corrections.


# An empty array is not the same as an array with empty elements.

  array0=( first second third )
  array1=( '' )   # "array1" consists of one empty element.
  array2=( )      # No elements . . . "array2" is empty.
  array3=(   )    # What about this array?


echo
ListArray()
{
echo
echo "Elements in array0:  ${array0[@]}"
echo "Elements in array1:  ${array1[@]}"
echo "Elements in array2:  ${array2[@]}"
echo "Elements in array3:  ${array3[@]}"
echo
echo "Length of first element in array0 = ${#array0}"
echo "Length of first element in array1 = ${#array1}"
echo "Length of first element in array2 = ${#array2}"
echo "Length of first element in array3 = ${#array3}"
echo
echo "Number of elements in array0 = ${#array0[*]}"  # 3
```

```
echo "Number of elements in array1 = ${#array1[*]}"  # 1  (Surprise!)
echo "Number of elements in array2 = ${#array2[*]}"  # 0
echo "Number of elements in array3 = ${#array3[*]}"  # 0
}

# ===================================================================

ListArray

# Try extending those arrays.

# Adding an element to an array.
array0=( "${array0[@]}" "new1" )
array1=( "${array1[@]}" "new1" )
array2=( "${array2[@]}" "new1" )
array3=( "${array3[@]}" "new1" )

ListArray

# or
array0[${#array0[*]}]="new2"
array1[${#array1[*]}]="new2"
array2[${#array2[*]}]="new2"
array3[${#array3[*]}]="new2"

ListArray

# When extended as above, arrays are 'stacks' ...
# Above is the 'push' ...
# The stack 'height' is:
height=${#array2[@]}
echo
echo "Stack height for array2 = $height"

# The 'pop' is:
unset array2[${#array2[@]}-1]   #  Arrays are zero-based,
height=${#array2[@]}            #+ which means first element has index 0.
echo
echo "POP"
echo "New stack height for array2 = $height"

ListArray

# List only 2nd and 3rd elements of array0.
from=1              # Zero-based numbering.
to=2
array3=( ${array0[@]:1:2} )
echo
echo "Elements in array3:  ${array3[@]}"

# Works like a string (array of characters).
# Try some other "string" forms.

# Replacement:
array4=( ${array0[@]/second/2nd} )
echo
echo "Elements in array4:  ${array4[@]}"

# Replace all matching wildcarded string.
array5=( ${array0[@]//new?/old} )
echo
echo "Elements in array5:  ${array5[@]}"
```

```
# Just when you are getting the feel for this . . .
array6=( ${array0[@]#*new} )
echo # This one might surprise you.
echo "Elements in array6:  ${array6[@]}"

array7=( ${array0[@]#new1} )
echo # After array6 this should not be a surprise.
echo "Elements in array7:  ${array7[@]}"

# Which looks a lot like . . .
array8=( ${array0[@]/new1/} )
echo
echo "Elements in array8:  ${array8[@]}"

#  So what can one say about this?

#  The string operations are performed on
#+ each of the elements in var[@] in succession.
#  Therefore : Bash supports string vector operations.
#  If the result is a zero length string,
#+ that element disappears in the resulting assignment.
#  However, if the expansion is in quotes, the null elements remain.

#  Michael Zick:    Question, are those strings hard or soft quotes?
#  Nathan Coulter:  There is no such thing as "soft quotes."
#!    What's really happening is that
#!+   the pattern matching happens after
#!+   all the other expansions of [word]
#!+   in cases like ${parameter#word}.


zap='new*'
array9=( ${array0[@]/$zap/} )
echo
echo "Number of elements in array9:  ${#array9[@]}"
array9=( "${array0[@]/$zap/}" )
echo "Elements in array9:  ${array9[@]}"
# This time the null elements remain.
echo "Number of elements in array9:  ${#array9[@]}"


# Just when you thought you were still in Kansas . . .
array10=( ${array0[@]#$zap} )
echo
echo "Elements in array10:  ${array10[@]}"
# But, the asterisk in zap won't be interpreted if quoted.
array10=( ${array0[@]#"$zap"} )
echo
echo "Elements in array10:  ${array10[@]}"
# Well, maybe we _are_ still in Kansas . . .
# (Revisions to above code block by Nathan Coulter.)


#  Compare array7 with array10.
#  Compare array8 with array9.

#  Reiterating: No such thing as soft quotes!
#  Nathan Coulter explains:
#  Pattern matching of 'word' in ${parameter#word} is done after
#+ parameter expansion and *before* quote removal.
#  In the normal case, pattern matching is done *after* quote removal.
```

```
exit
```

The relationship of **${array_name[@]}** and **${array_name[*]}** is analogous to that between $@ and $*. This powerful array notation has a number of uses.

```
# Copying an array.
array2=( "${array1[@]}" )
# or
array2="${array1[@]}"
#
#  However, this fails with "sparse" arrays,
#+ arrays with holes (missing elements) in them,
#+ as Jochen DeSmet points out.
# ------------------------------------------
  array1[0]=0
# array1[1] not assigned
  array1[2]=2
  array2=( "${array1[@]}" )       # Copy it?

echo ${array2[0]}      # 0
echo ${array2[2]}      # (null), should be 2
# ------------------------------------------



# Adding an element to an array.
array=( "${array[@]}" "new element" )
# or
array[${#array[*]}]="new element"

# Thanks, S.C.
```

The **array=(element1 element2 ... elementN)** initialization operation, with the help of command substitution, makes it possible to load the contents of a text file into an array.

```
#!/bin/bash

filename=sample_file

#            cat sample_file
#
#            1 a b c
#            2 d e fg


declare -a array1

array1=( `cat "$filename"`)                #  Loads contents
#         List file to stdout              #+ of $filename into array1.
#
#  array1=( `cat "$filename" | tr '\n' ' '`)
#                            change linefeeds in file to spaces.
#  Not necessary because Bash does word splitting,
#+ changing linefeeds to spaces.

echo ${array1[@]}            # List the array.
#                              1 a b c 2 d e fg
#
```

```
#  Each whitespace-separated "word" in the file
#+ has been assigned to an element of the array.

element_count=${#array1[*]}
echo $element_count          # 8
```

Clever scripting makes it possible to add array operations.

Example 27-8. Initializing arrays

```
#! /bin/bash
# array-assign.bash

#  Array operations are Bash-specific,
#+ hence the ".bash" in the script name.

# Copyright (c) Michael S. Zick, 2003, All rights reserved.
# License: Unrestricted reuse in any form, for any purpose.
# Version: $ID$
#
# Clarification and additional comments by William Park.

#  Based on an example provided by Stephane Chazelas
#+ which appeared in an earlier version of the
#+ Advanced Bash Scripting Guide.

# Output format of the 'times' command:
# User CPU <space> System CPU
# User CPU of dead children <space> System CPU of dead children

#  Bash has two versions of assigning all elements of an array
#+ to a new array variable.
#  Both drop 'null reference' elements
#+ in Bash versions 2.04 and later.
#  An additional array assignment that maintains the relationship of
#+ [subscript]=value for arrays may be added to newer versions.

#  Constructs a large array using an internal command,
#+ but anything creating an array of several thousand elements
#+ will do just fine.

declare -a bigOne=( /dev/* )  # All the files in /dev . . .
echo
echo 'Conditions: Unquoted, default IFS, All-Elements-Of'
echo "Number of elements in array is ${#bigOne[@]}"

# set -vx



echo
echo '- - testing: =( ${array[@]} ) - -'
times
declare -a bigTwo=( ${bigOne[@]} )
# Note parens:    ^              ^
times


echo
```

```
echo '- - testing: =${array[@]} - -'
times
declare -a bigThree=${bigOne[@]}
# No parentheses this time.
times

#  Comparing the numbers shows that the second form, pointed out
#+ by Stephane Chazelas, is faster.
#
#  As William Park explains:
#+ The bigTwo array assigned element by element (because of parentheses),
#+ whereas bigThree assigned as a single string.
#  So, in essence, you have:
#                    bigTwo=( [0]="..." [1]="..." [2]="..." ... )
#                    bigThree=( [0]="... ... ..." )
#
#  Verify this by:  echo ${bigTwo[0]}
#                    echo ${bigThree[0]}


#  I will continue to use the first form in my example descriptions
#+ because I think it is a better illustration of what is happening.

#  The reusable portions of my examples will actual contain
#+ the second form where appropriate because of the speedup.

# MSZ: Sorry about that earlier oversight folks.


#  Note:
#  ----
#  The "declare -a" statements in lines 32 and 44
#+ are not strictly necessary, since it is implicit
#+ in the  Array=( ... )  assignment form.
#  However, eliminating these declarations slows down
#+ the execution of the following sections of the script.
#  Try it, and see.

exit 0
```

Adding a superfluous **declare -a** statement to an array declaration may speed up execution of subsequent operations on the array.

Example 27-9. Copying and concatenating arrays

```
#! /bin/bash
# CopyArray.sh
#
# This script written by Michael Zick.
# Used here with permission.

#  How-To "Pass by Name & Return by Name"
#+ or "Building your own assignment statement".


CpArray_Mac() {

# Assignment Command Statement Builder

```

```
    echo -n 'eval '
    echo -n "$2"                    # Destination name
    echo -n '=( ${'
    echo -n "$1"                    # Source name
    echo -n '[@]} )'

# That could all be a single command.
# Matter of style only.
}

declare -f CopyArray                # Function "Pointer"
CopyArray=CpArray_Mac               # Statement Builder

Hype()
{

# Hype the array named $1.
# (Splice it together with array containing "Really Rocks".)
# Return in array named $2.

    local -a TMP
    local -a hype=( Really Rocks )

    $($CopyArray $1 TMP)
    TMP=( ${TMP[@]} ${hype[@]} )
    $($CopyArray TMP $2)
}

declare -a before=( Advanced Bash Scripting )
declare -a after

echo "Array Before = ${before[@]}"

Hype before after

echo "Array After = ${after[@]}"

# Too much hype?

echo "What ${after[@]:3:2}?"

declare -a modest=( ${after[@]:2:1} ${after[@]:3:2} )
#                    ---- substring extraction ----

echo "Array Modest = ${modest[@]}"

# What happened to 'before' ?

echo "Array Before = ${before[@]}"

exit 0
```

Example 27-10. More on concatenating arrays

```
#! /bin/bash
# array-append.bash

# Copyright (c) Michael S. Zick, 2003, All rights reserved.
# License: Unrestricted reuse in any form, for any purpose.
# Version: $ID$
#
```

```
# Slightly modified in formatting by M.C.


# Array operations are Bash-specific.
# Legacy UNIX /bin/sh lacks equivalents.


#  Pipe the output of this script to 'more'
#+ so it doesn't scroll off the terminal.
#  Or, redirect output to a file.


declare -a array1=( zero1 one1 two1 )
# Subscript packed.
declare -a array2=( [0]=zero2 [2]=two2 [3]=three2 )
# Subscript sparse -- [1] is not defined.

echo
echo '- Confirm that the array is really subscript sparse. -'
echo "Number of elements: 4"        # Hard-coded for illustration.
for (( i = 0 ; i < 4 ; i++ ))
do
    echo "Element [$i]: ${array2[$i]}"
done
# See also the more general code example in basics-reviewed.bash.


declare -a dest

# Combine (append) two arrays into a third array.
echo
echo 'Conditions: Unquoted, default IFS, All-Elements-Of operator'
echo '- Undefined elements not present, subscripts not maintained. -'
# # The undefined elements do not exist; they are not being dropped.

dest=( ${array1[@]} ${array2[@]} )
# dest=${array1[@]}${array2[@]}     # Strange results, possibly a bug.

# Now, list the result.
echo
echo '- - Testing Array Append - -'
cnt=${#dest[@]}

echo "Number of elements: $cnt"
for (( i = 0 ; i < cnt ; i++ ))
do
    echo "Element [$i]: ${dest[$i]}"
done

# Assign an array to a single array element (twice).
dest[0]=${array1[@]}
dest[1]=${array2[@]}

# List the result.
echo
echo '- - Testing modified array - -'
cnt=${#dest[@]}

echo "Number of elements: $cnt"
for (( i = 0 ; i < cnt ; i++ ))
do
    echo "Element [$i]: ${dest[$i]}"
```

```
done

# Examine the modified second element.
echo
echo '- - Reassign and list second element - -'

declare -a subArray=${dest[1]}
cnt=${#subArray[@]}

echo "Number of elements: $cnt"
for (( i = 0 ; i < cnt ; i++ ))
do
    echo "Element [$i]: ${subArray[$i]}"
done

#  The assignment of an entire array to a single element
#+ of another array using the '=${ ... }' array assignment
#+ has converted the array being assigned into a string,
#+ with the elements separated by a space (the first character of IFS).

# If the original elements didn't contain whitespace . . .
# If the original array isn't subscript sparse . . .
# Then we could get the original array structure back again.

# Restore from the modified second element.
echo
echo '- - Listing restored element - -'

declare -a subArray=( ${dest[1]} )
cnt=${#subArray[@]}

echo "Number of elements: $cnt"
for (( i = 0 ; i < cnt ; i++ ))
do
    echo "Element [$i]: ${subArray[$i]}"
done
echo '- - Do not depend on this behavior. - -'
echo '- - This behavior is subject to change - -'
echo '- - in versions of Bash newer than version 2.05b - -'

# MSZ: Sorry about any earlier confusion folks.

exit 0
```

--

Arrays permit deploying old familiar algorithms as shell scripts. Whether this is necessarily a good idea is left for the reader to decide.

Example 27-11. The Bubble Sort

```
#!/bin/bash
# bubble.sh: Bubble sort, of sorts.

# Recall the algorithm for a bubble sort. In this particular version...

#  With each successive pass through the array to be sorted,
#+ compare two adjacent elements, and swap them if out of order.
#  At the end of the first pass, the "heaviest" element has sunk to bottom.
```

```
#  At the end of the second pass, the next "heaviest" one has sunk next to bottom.
#  And so forth.
#  This means that each successive pass needs to traverse less of the array.
#  You will therefore notice a speeding up in the printing of the later passes.


exchange()
{
  # Swaps two members of the array.
  local temp=${Countries[$1]} #  Temporary storage
                              #+ for element getting swapped out.
  Countries[$1]=${Countries[$2]}
  Countries[$2]=$temp

  return
}

declare -a Countries  #  Declare array,
                      #+ optional here since it's initialized below.

#  Is it permissable to split an array variable over multiple lines
#+ using an escape (\)?
#  Yes.

Countries=(Netherlands Ukraine Zaire Turkey Russia Yemen Syria \
Brazil Argentina Nicaragua Japan Mexico Venezuela Greece England \
Israel Peru Canada Oman Denmark Wales France Kenya \
Xanadu Qatar Liechtenstein Hungary)

# "Xanadu" is the mythical place where, according to Coleridge,
#+ Kubla Khan did a pleasure dome decree.


clear                      # Clear the screen to start with.

echo "0: ${Countries[*]}"  # List entire array at pass 0.

number_of_elements=${#Countries[@]}
let "comparisons = $number_of_elements - 1"

count=1 # Pass number.

while [ "$comparisons" -gt 0 ]          # Beginning of outer loop
do

  index=0  # Reset index to start of array after each pass.

  while [ "$index" -lt "$comparisons" ] # Beginning of inner loop
  do
    if [ ${Countries[$index]} \> ${Countries[`expr $index + 1`]} ]
    #  If out of order...
    #  Recalling that \> is ASCII comparison operator
    #+ within single brackets.

    #  if [[ ${Countries[$index]} > ${Countries[`expr $index + 1`]} ]]
    #+ also works.
    then
      exchange $index `expr $index + 1`  # Swap.
    fi
    let "index += 1"  # Or,   index+=1   on Bash, ver. 3.1 or newer.
  done # End of inner loop
```

```
# ----------------------------------------------------------------------
# Paulo Marcel Coelho Aragao suggests for-loops as a simpler altenative.
#
# for (( last = $number_of_elements - 1 ; last > 0 ; last-- ))
##                     Fix by C.Y. Hunt          ^   (Thanks!)
# do
#     for (( i = 0 ; i < last ; i++ ))
#     do
#         [[ "${Countries[$i]}" > "${Countries[$((i+1))]}" ]] \
#             && exchange $i $((i+1))
#     done
# done
# ----------------------------------------------------------------------


let "comparisons -= 1" #  Since "heaviest" element bubbles to bottom,
                       #+ we need do one less comparison each pass.

echo
echo "$count: ${Countries[@]}"  # Print resultant array at end of each pass.
echo
let "count += 1"                # Increment pass count.

done                            # End of outer loop
                                # All done.

exit 0
```

--

Is it possible to nest arrays within arrays?

```
#!/bin/bash
# "Nested" array.

#  Michael Zick provided this example,
#+ with corrections and clarifications by William Park.

AnArray=( $(ls --inode --ignore-backups --almost-all \
        --directory --full-time --color=none --time=status \
        --sort=time -l ${PWD} ) )  # Commands and options.

# Spaces are significant . . . and don't quote anything in the above.

SubArray=( ${AnArray[@]:11:1}  ${AnArray[@]:6:5} )
#  This array has six elements:
#+     SubArray=( [0]=${AnArray[11]} [1]=${AnArray[6]} [2]=${AnArray[7]}
#      [3]=${AnArray[8]} [4]=${AnArray[9]} [5]=${AnArray[10]} )
#
#  Arrays in Bash are (circularly) linked lists
#+ of type string (char *).
#  So, this isn't actually a nested array,
#+ but it's functionally similar.

echo "Current directory and date of last status change:"
echo "${SubArray[@]}"

exit 0
```

--

Embedded arrays in combination with indirect references create some fascinating possibilities

Example 27-12. Embedded arrays and indirect references

```
#!/bin/bash
# embedded-arrays.sh
# Embedded arrays and indirect references.

# This script by Dennis Leeuw.
# Used with permission.
# Modified by document author.


ARRAY1=(
        VAR1_1=value11
        VAR1_2=value12
        VAR1_3=value13
)

ARRAY2=(
        VARIABLE="test"
        STRING="VAR1=value1 VAR2=value2 VAR3=value3"
        ARRAY21=${ARRAY1[*]}
)       # Embed ARRAY1 within this second array.

function print () {
        OLD_IFS="$IFS"
        IFS=$'\n'       #  To print each array element
                        #+ on a separate line.
        TEST1="ARRAY2[*]"
        local ${!TEST1} # See what happens if you delete this line.
        #  Indirect reference.
        #  This makes the components of $TEST1
        #+ accessible to this function.


        #  Let's see what we've got so far.
        echo
        echo "\$TEST1 = $TEST1"       #  Just the name of the variable.
        echo; echo
        echo "{\$TEST1} = ${!TEST1}"  #  Contents of the variable.
                                      #  That's what an indirect
                                      #+ reference does.
        echo
        echo "-------------------------------------------"; echo
        echo


        # Print variable
        echo "Variable VARIABLE: $VARIABLE"

        # Print a string element
        IFS="$OLD_IFS"
        TEST2="STRING[*]"
        local ${!TEST2}      # Indirect reference (as above).
        echo "String element VAR2: $VAR2 from STRING"

        # Print an array element
        TEST2="ARRAY21[*]"
```

```
        local ${!TEST2}      # Indirect reference (as above).
        echo "Array element VAR1_1: $VAR1_1 from ARRAY21"
}

print
echo

exit 0

#   As the author of the script notes,
#+ "you can easily expand it to create named-hashes in bash."
#   (Difficult) exercise for the reader: implement this.
```

--

Arrays enable implementing a shell script version of the *Sieve of Eratosthenes*. Of course, a resource-intensive application of this nature should really be written in a compiled language, such as C. It runs excruciatingly slowly as a script.

Example 27-13. The Sieve of Eratosthenes

```
#!/bin/bash
# sieve.sh (ex68.sh)

# Sieve of Eratosthenes
# Ancient algorithm for finding prime numbers.

#  This runs a couple of orders of magnitude slower
#+ than the equivalent program written in C.

LOWER_LIMIT=1       # Starting with 1.
UPPER_LIMIT=1000    # Up to 1000.
# (You may set this higher . . . if you have time on your hands.)

PRIME=1
NON_PRIME=0

let SPLIT=UPPER_LIMIT/2
# Optimization:
# Need to test numbers only halfway to upper limit. Why?


declare -a Primes
# Primes[] is an array.


initialize ()
{
# Initialize the array.

i=$LOWER_LIMIT
until [ "$i" -gt "$UPPER_LIMIT" ]
do
  Primes[i]=$PRIME
  let "i += 1"
done
#  Assume all array members guilty (prime)
#+ until proven innocent.
}
```

```
print_primes ()
{
# Print out the members of the Primes[] array tagged as prime.

i=$LOWER_LIMIT

until [ "$i" -gt "$UPPER_LIMIT" ]
do

  if [ "${Primes[i]}" -eq "$PRIME" ]
  then
    printf "%8d" $i
    # 8 spaces per number gives nice, even columns.
  fi

  let "i += 1"

done

}

sift () # Sift out the non-primes.
{

let i=$LOWER_LIMIT+1
# Let's start with 2.

until [ "$i" -gt "$UPPER_LIMIT" ]
do

if [ "${Primes[i]}" -eq "$PRIME" ]
# Don't bother sieving numbers already sieved (tagged as non-prime).
then

  t=$i

  while [ "$t" -le "$UPPER_LIMIT" ]
  do
    let "t += $i "
    Primes[t]=$NON_PRIME
    # Tag as non-prime all multiples.
  done

fi

  let "i += 1"
done


}


# ==============================================
# main ()
# Invoke the functions sequentially.
initialize
sift
print_primes
# This is what they call structured programming.
# ==============================================
```

```
echo

exit 0



# ------------------------------------------------------------ #
# Code below line will not execute, because of 'exit.'

#  This improved version of the Sieve, by Stephane Chazelas,
#+ executes somewhat faster.

# Must invoke with command-line argument (limit of primes).

UPPER_LIMIT=$1                  # From command-line.
let SPLIT=UPPER_LIMIT/2         # Halfway to max number.

Primes=( '' $(seq $UPPER_LIMIT) )

i=1
until (( ( i += 1 ) > SPLIT ))  # Need check only halfway.
do
  if [[ -n ${Primes[i]} ]]
  then
    t=$i
    until (( ( t += i ) > UPPER_LIMIT ))
    do
      Primes[t]=
    done
  fi
done
echo ${Primes[*]}

exit $?
```

Example 27-14. The Sieve of Eratosthenes, Optimized

```
#!/bin/bash
# Optimized Sieve of Eratosthenes
# Script by Jared Martin, with very minor changes by ABS Guide author.
# Used in ABS Guide with permission (thanks!).

# Based on script in Advanced Bash Scripting Guide.
# http://tldp.org/LDP/abs/html/arrays.html#PRIMES0 (ex68.sh).

# http://www.cs.hmc.edu/~oneill/papers/Sieve-JFP.pdf (reference)
# Check results against http://primes.utm.edu/lists/small/1000.txt

# Necessary but not sufficient would be, e.g.,
#     (($(sieve 7919 | wc -w) == 1000)) && echo "7919 is the 1000th prime"

UPPER_LIMIT=${1:?"Need an upper limit of primes to search."}

Primes=( '' $(seq ${UPPER_LIMIT}) )

typeset -i i t
Primes[i=1]='' # 1 is not a prime.
until (( ( i += 1 ) > (${UPPER_LIMIT}/i) ))  # Need check only ith-way.
  do                                         # Why?
    if ((${Primes[t=i*(i-1), i]}))
    # Obscure, but instructive, use of arithmetic expansion in subscript.
```

```
    then
      until (( ( t += i ) > ${UPPER_LIMIT} ))
        do Primes[t]=; done
    fi
  done

# echo ${Primes[*]}
echo   # Change to original script for pretty-printing (80-col. display).
printf "%8d" ${Primes[*]}
echo; echo

exit $?
```

Compare these array-based prime number generators with alternatives that do not use arrays, Example A-15, and Example 16-46.

--

Arrays lend themselves, to some extent, to emulating data structures for which Bash has no native support.

Example 27-15. Emulating a push-down stack

```
#!/bin/bash
# stack.sh: push-down stack simulation

#  Similar to the CPU stack, a push-down stack stores data items
#+ sequentially, but releases them in reverse order, last-in first-out.


BP=100            #  Base Pointer of stack array.
                  #  Begin at element 100.

SP=$BP            #  Stack Pointer.
                  #  Initialize it to "base" (bottom) of stack.

Data=             #  Contents of stack location.
                  #  Must use global variable,
                  #+ because of limitation on function return range.


                  # 100     Base pointer       <-- Base Pointer
                  #  99     First data item
                  #  98     Second data item
                  # ...     More data
                  #         Last data item     <-- Stack pointer


declare -a stack


push()            # Push item on stack.
{
if [ -z "$1" ]    # Nothing to push?
then
  return
fi

let "SP -= 1"     # Bump stack pointer.
```

```
stack[$SP]=$1

return
}

pop()                    # Pop item off stack.
{
Data=                    # Empty out data item.

if [ "$SP" -eq "$BP" ]   # Stack empty?
then
  return
fi                       #  This also keeps SP from getting past 100,
                         #+ i.e., prevents a runaway stack.

Data=${stack[$SP]}
let "SP += 1"            # Bump stack pointer.
return
}

status_report()          # Find out what's happening.
{
echo "-------------------------------------"
echo "REPORT"
echo "Stack Pointer = $SP"
echo "Just popped \""$Data"\" off the stack."
echo "-------------------------------------"
echo
}


# =======================================================
# Now, for some fun.

echo

# See if you can pop anything off empty stack.
pop
status_report

echo

push garbage
pop
status_report     # Garbage in, garbage out.

value1=23;        push $value1
value2=skidoo;    push $value2
value3=LAST;      push $value3

pop               # LAST
status_report
pop               # skidoo
status_report
pop               # 23
status_report     # Last-in, first-out!

#  Notice how the stack pointer decrements with each push,
#+ and increments with each pop.

echo
```

```
exit 0

# ========================================================


# Exercises:
# ---------

# 1)  Modify the "push()" function to permit pushing
#   + multiple element on the stack with a single function call.

# 2)  Modify the "pop()" function to permit popping
#   + multiple element from the stack with a single function call.

# 3)  Add error checking to the critical functions.
#     That is, return an error code, depending on
#   + successful or unsuccessful completion of the operation,
#   + and take appropriate action.

# 4)  Using this script as a starting point,
#   + write a stack-based 4-function calculator.
```

--

Fancy manipulation of array "subscripts" may require intermediate variables. For projects involving this, again consider using a more powerful programming language, such as Perl or C.

Example 27-16. Complex array application: *Exploring a weird mathematical series*

```
#!/bin/bash

# Douglas Hofstadter's notorious "Q-series":

# Q(1) = Q(2) = 1
# Q(n) = Q(n - Q(n-1)) + Q(n - Q(n-2)), for n>2

#  This is a "chaotic" integer series with strange
#+ and unpredictable behavior.
#  The first 20 terms of the series are:
#  1 1 2 3 3 4 5 5 6 6 6 8 8 8 10 9 10 11 11 12

#  See Hofstadter's book, _Goedel, Escher, Bach: An Eternal Golden Braid_,
#+ p. 137, ff.


LIMIT=100     # Number of terms to calculate.
LINEWIDTH=20  # Number of terms printed per line.

Q[1]=1        # First two terms of series are 1.
Q[2]=1

echo
echo "Q-series [$LIMIT terms]:"
echo -n "${Q[1]} "             # Output first two terms.
echo -n "${Q[2]} "

for ((n=3; n <= $LIMIT; n++))  # C-like loop expression.
do   # Q[n] = Q[n - Q[n-1]] + Q[n - Q[n-2]]  for n>2
#    Need to break the expression into intermediate terms,
#+   since Bash doesn't handle complex array arithmetic very well.
```

```
  let "n1 = $n - 1"        # n-1
  let "n2 = $n - 2"        # n-2

  t0=`expr $n - ${Q[n1]}`  # n - Q[n-1]
  t1=`expr $n - ${Q[n2]}`  # n - Q[n-2]

  T0=${Q[t0]}              # Q[n - Q[n-1]]
  T1=${Q[t1]}              # Q[n - Q[n-2]]

Q[n]=`expr $T0 + $T1`      # Q[n - Q[n-1]] + Q[n - Q[n-2]]
echo -n "${Q[n]} "

if [ `expr $n % $LINEWIDTH` -eq 0 ]    # Format output.
then   #      ^ modulo
  echo # Break lines into neat chunks.
fi

done

echo

exit 0

#  This is an iterative implementation of the Q-series.
#  The more intuitive recursive implementation is left as an exercise.
#  Warning: calculating this series recursively takes a VERY long time
#+ via a script. C/C++ would be orders of magnitude faster.
```

--

Bash supports only one-dimensional arrays, though a little trickery permits simulating multi-dimensional ones.

Example 27-17. Simulating a two-dimensional array, then tilting it

```
#!/bin/bash
# twodim.sh: Simulating a two-dimensional array.

# A one-dimensional array consists of a single row.
# A two-dimensional array stores rows sequentially.

Rows=5
Columns=5
# 5 X 5 Array.

declare -a alpha     # char alpha [Rows] [Columns];
                     # Unnecessary declaration. Why?

load_alpha ()
{
local rc=0
local index

for i in A B C D E F G H I J K L M N O P Q R S T U V W X Y
do     # Use different symbols if you like.
  local row=`expr $rc / $Columns`
  local column=`expr $rc % $Rows`
  let "index = $row * $Rows + $column"
```

```
  alpha[$index]=$i
# alpha[$row][$column]
  let "rc += 1"
done

#  Simpler would be
#+   declare -a alpha=( A B C D E F G H I J K L M N O P Q R S T U V W X Y )
#+ but this somehow lacks the "flavor" of a two-dimensional array.
}

print_alpha ()
{
local row=0
local index

echo

while [ "$row" -lt "$Rows" ]   #  Print out in "row major" order:
do                             #+ columns vary,
                               #+ while row (outer loop) remains the same.
  local column=0

  echo -n "       "            #  Lines up "square" array with rotated one.

  while [ "$column" -lt "$Columns" ]
  do
    let "index = $row * $Rows + $column"
    echo -n "${alpha[index]} "  # alpha[$row][$column]
    let "column += 1"
  done

  let "row += 1"
  echo

done

# The simpler equivalent is
#     echo ${alpha[*]} | xargs -n $Columns

echo
}

filter ()     # Filter out negative array indices.
{

echo -n "  "  # Provides the tilt.
              # Explain how.

if [[ "$1" -ge 0 &&  "$1" -lt "$Rows" && "$2" -ge 0 && "$2" -lt "$Columns" ]]
then
    let "index = $1 * $Rows + $2"
    # Now, print it rotated.
    echo -n " ${alpha[index]}"
    #           alpha[$row][$column]
fi

}




rotate ()  #  Rotate the array 45 degrees --
```

```
{          #+ "balance" it on its lower lefthand corner.
local row
local column

for (( row = Rows; row > -Rows; row-- ))
  do       # Step through the array backwards. Why?

  for (( column = 0; column < Columns; column++ ))
  do

    if [ "$row" -ge 0 ]
    then
      let "t1 = $column - $row"
      let "t2 = $column"
    else
      let "t1 = $column"
      let "t2 = $column + $row"
    fi

    filter $t1 $t2   # Filter out negative array indices.
                     # What happens if you don't do this?
  done

  echo; echo

done

#  Array rotation inspired by examples (pp. 143-146) in
#+ "Advanced C Programming on the IBM PC," by Herbert Mayer
#+ (see bibliography).
#  This just goes to show that much of what can be done in C
#+ can also be done in shell scripting.

}


#--------------- Now, let the show begin. ------------#
load_alpha     # Load the array.
print_alpha    # Print it out.
rotate         # Rotate it 45 degrees counterclockwise.
#-----------------------------------------------------#

exit 0

# This is a rather contrived, not to mention inelegant simulation.

# Exercises:
# ---------
# 1)  Rewrite the array loading and printing functions
#     in a more intuitive and less kludgy fashion.
#
# 2)  Figure out how the array rotation functions work.
#     Hint: think about the implications of backwards-indexing an array.
#
# 3)  Rewrite this script to handle a non-square array,
#     such as a 6 X 4 one.
#     Try to minimize "distortion" when the array is rotated.
```

A two-dimensional array is essentially equivalent to a one-dimensional one, but with additional addressing modes for referencing and manipulating the individual elements by *row* and *column* position.

For an even more elaborate example of simulating a two-dimensional array, see Example A-10.

--

For more interesting scripts using arrays, see:

* Example 12-3
* Example 16-46
* Example A-22
* Example A-44
* Example A-41
* Example A-42

Chapter 28. Indirect References

We have seen that referencing a variable, `$var`, fetches its *value*. But, what about the *value of a value*? What about `$$var`?

The actual notation is *`\$$var`*, usually preceded by an eval (and sometimes an echo). This is called an *indirect reference*.

Example 28-1. Indirect Variable References

```
#!/bin/bash
# ind-ref.sh: Indirect variable referencing.
# Accessing the contents of the contents of a variable.

# First, let's fool around a little.

var=23

echo "\$var   = $var"           # $var   = 23
# So far, everything as expected. But ...

echo "\$\$var  = $$var"         # $$var  = 4570var
#  Not useful ...
#  \$\$ expanded to PID of the script
#  -- refer to the entry on the $$ variable --
#+ and "var" is echoed as plain text.
#  (Thank you, Jakob Bohm, for pointing this out.)

echo "\\\$\$var = \$$var"       # \$$var = $23
#  As expected. The first $ is escaped and pasted on to
#+ the value of var ($var = 23 ).
#  Meaningful, but still not useful.

# Now, let's start over and do it the right way.

# ============================================== #


a=letter_of_alphabet   # Variable "a" holds the name of another variable.
letter_of_alphabet=z

echo

# Direct reference.
echo "a = $a"          # a = letter_of_alphabet

# Indirect reference.
  eval a=\$$a
# ^^^        Forcing an eval(uation), and ...
#        ^   Escaping the first $ ...
# ------------------------------------------------------------------------
# The 'eval' forces an update of $a, sets it to the updated value of \$$a.
# So, we see why 'eval' so often shows up in indirect reference notation.
# ------------------------------------------------------------------------
  echo "Now a = $a"    # Now a = z
```

```
echo


# Now, let's try changing the second-order reference.

t=table_cell_3
table_cell_3=24
echo "\"table_cell_3\" = $table_cell_3"            # "table_cell_3" = 24
echo -n "dereferenced \"t\" = "; eval echo \$$t    # dereferenced "t" = 24
# In this simple case, the following also works (why?).
#         eval t=\$$t; echo "\"t\" = $t"

echo

t=table_cell_3
NEW_VAL=387
table_cell_3=$NEW_VAL
echo "Changing value of \"table_cell_3\" to $NEW_VAL."
echo "\"table_cell_3\" now $table_cell_3"
echo -n "dereferenced \"t\" now "; eval echo \$$t
# "eval" takes the two arguments "echo" and "\$$t" (set equal to $table_cell_3)


echo

# (Thanks, Stephane Chazelas, for clearing up the above behavior.)


#   A more straightforward method is the ${!t} notation, discussed in the
#+ "Bash, version 2" section.
#   See also ex78.sh.

exit 0
```

Indirect referencing in Bash is a multi-step process. First, take the name of a variable: `varname`. Then, reference it: `$varname`. Then, reference the reference: `$$varname`. Then, *escape* the first $: `\$$varname`. Finally, force a reevaluation of the expression and assign it: **eval newvar=\$$varname**.

Of what practical use is indirect referencing of variables? It gives Bash a little of the functionality of pointers in *C*, for instance, in table lookup. And, it also has some other very interesting applications. . . .

Nils Radtke shows how to build "dynamic" variable names and evaluate their contents. This can be useful when sourcing configuration files.

```
#!/bin/bash


# ---------------------------------------------
# This could be "sourced" from a separate file.
isdnMyProviderRemoteNet=172.16.0.100
isdnYourProviderRemoteNet=10.0.0.10
isdnOnlineService="MyProvider"
# ---------------------------------------------


remoteNet=$(eval "echo \$$(echo isdn${isdnOnlineService}RemoteNet)")
remoteNet=$(eval "echo \$$(echo isdnMyProviderRemoteNet)")
remoteNet=$(eval "echo \$isdnMyProviderRemoteNet")
```

```
remoteNet=$(eval "echo $isdnMyProviderRemoteNet")

echo "$remoteNet"    # 172.16.0.100

# ================================================================

#  And, it gets even better.

#  Consider the following snippet given a variable named getSparc,
#+ but no such variable getIa64:

chkMirrorArchs () {
  arch="$1";
  if [ "$(eval "echo \${$(echo get$(echo -ne $arch |
       sed 's/^\(.\).*/\1/g' | tr 'a-z' 'A-Z'; echo $arch |
       sed 's/^.\(.*\)/\1/g')):-false}")" = true ]
  then
     return 0;
  else
     return 1;
  fi;
}

getSparc="true"
unset getIa64
chkMirrorArchs sparc
echo $?        # 0
               # True

chkMirrorArchs Ia64
echo $?        # 1
               # False

# Notes:
# -----
# Even the to-be-substituted variable name part is built explicitly.
# The parameters to the chkMirrorArchs calls are all lower case.
# The variable name is composed of two parts: "get" and "Sparc" . . .
```

Example 28-2. Passing an indirect reference to *awk*

```
#!/bin/bash

#  Another version of the "column totaler" script
#+ that adds up a specified column (of numbers) in the target file.
#  This one uses indirect references.

ARGS=2
E_WRONGARGS=85

if [ $# -ne "$ARGS" ] # Check for proper number of command-line args.
then
   echo "Usage: `basename $0` filename column-number"
   exit $E_WRONGARGS
fi

filename=$1         # Name of file to operate on.
column_number=$2    # Which column to total up.

#===== Same as original script, up to this point =====#
```

```
# A multi-line awk script is invoked by
#   awk "
#   ...
#   ...
#   ...
#   "


# Begin awk script.
# -------------------------------------------------
awk "

{ total += \$${column_number} # Indirect reference
}
END {
     print total
     }

     " "$filename"
# Note that awk doesn't need an eval preceding \$$.
# -------------------------------------------------
# End awk script.

#  Indirect variable reference avoids the hassles
#+ of referencing a shell variable within the embedded awk script.
#  Thanks, Stephane Chazelas.


exit $?
```

This method of indirect referencing is a bit tricky. If the second order variable changes its value, then the first order variable must be properly dereferenced (as in the above example). Fortunately, the *${!variable}* notation introduced with version 2 of Bash (see Example 37-2 and Example A-22) makes indirect referencing more intuitive.

Bash does not support pointer arithmetic, and this severely limits the usefulness of indirect referencing. In fact, indirect referencing in a scripting language is, at best, something of an afterthought.

Chapter 29. `/dev` and `/proc`

A Linux or UNIX filesystem typically has the `/dev` and `/proc` special-purpose directories.

29.1. `/dev`

The `/dev` directory contains entries for the *physical devices* that may or may not be present in the hardware. [118] Appropriately enough, these are called *device files*. As an example, the hard drive partitions containing the mounted filesystem(s) have entries in `/dev`, as df shows.

```
bash$ df
Filesystem           1k-blocks      Used Available Use%
 Mounted on
 /dev/hda6              495876    222748    247527  48% /
 /dev/hda1               50755      3887     44248   9% /boot
 /dev/hda8              367013     13262    334803   4% /home
 /dev/hda5             1714416   1123624    503704  70% /usr
```

Among other things, the `/dev` directory contains *loopback* devices, such as `/dev/loop0`. A loopback device is a gimmick that allows an ordinary file to be accessed as if it were a block device. [119] This permits mounting an entire filesystem within a single large file. See Example 17-8 and Example 17-7.

A few of the pseudo-devices in `/dev` have other specialized uses, such as `/dev/null`, `/dev/zero`, `/dev/urandom`, `/dev/sda1` (hard drive partition), `/dev/udp` (*User Datagram Packet* port), and `/dev/tcp`.

For instance:

To manually mount a USB flash drive, append the following line to `/etc/fstab`. [120]

```
/dev/sda1    /mnt/flashdrive    auto    noauto,user,noatime    0 0
```

(See also Example A-23.)

Checking whether a disk is in the CD-burner (soft-linked to `/dev/hdc`):

```
head -1 /dev/hdc


#  head: cannot open '/dev/hdc' for reading: No medium found
#  (No disc in the drive.)

#  head: error reading '/dev/hdc': Input/output error
#  (There is a disk in the drive, but it can't be read;
#+  possibly it's an unrecorded CDR blank.)

#  Stream of characters and assorted gibberish
#  (There is a pre-recorded disk in the drive,
#+ and this is raw output -- a stream of ASCII and binary data.)
#  Here we see the wisdom of using 'head' to limit the output
#+ to manageable proportions, rather than 'cat' or something similar.


#  Now, it's just a matter of checking/parsing the output and taking
```

```
#+ appropriate action.
```

When executing a command on a `/dev/tcp/$host/$port` pseudo-device file, Bash opens a TCP connection to the associated *socket*.

A *socket* is a communications node associated with a specific I/O port. (This is analogous to a *hardware socket*, or *receptacle*, for a connecting cable.) It permits data transfer between hardware devices on the same machine, between machines on the same network, between machines across different networks, and, of course, between machines at different locations on the Internet.

The following examples assume an active Internet connection.

Getting the time from `nist.gov`:

```
bash$ cat </dev/tcp/time.nist.gov/13
53082 04-03-18 04:26:54 68 0 0 502.3 UTC(NIST) *
```

[Mark contributed this example.]

Generalizing the above into a script:

```
#!/bin/bash
# This script must run with root permissions.

URL="time.nist.gov/13"

Time=$(cat </dev/tcp/"$URL")
UTC=$(echo "$Time" | awk '{print$3}')   # Third field is UTC (GMT) time.
# Exercise: modify this for different time zones.

echo "UTC Time = "$UTC""
```

Downloading a URL:

```
bash$ exec 5<>/dev/tcp/www.net.cn/80
bash$ echo -e "GET / HTTP/1.0\n" >&5
bash$ cat <&5
```

[Thanks, Mark and Mihai Maties.]

Example 29-1. Using `/dev/tcp` for troubleshooting

```
#!/bin/bash
# dev-tcp.sh: /dev/tcp redirection to check Internet connection.

# Script by Troy Engel.
# Used with permission.

TCP_HOST=news-15.net       # A known spam-friendly ISP.
TCP_PORT=80                # Port 80 is http.

# Try to connect. (Somewhat similar to a 'ping' . . .)
echo "HEAD / HTTP/1.0" >/dev/tcp/${TCP_HOST}/${TCP_PORT}
MYEXIT=$?
```

```
: <<EXPLANATION
If bash was compiled with --enable-net-redirections, it has the capability of
using a special character device for both TCP and UDP redirections. These
redirections are used identically as STDIN/STDOUT/STDERR. The device entries
are 30,36 for /dev/tcp:

  mknod /dev/tcp c 30 36

>From the bash reference:
/dev/tcp/host/port
    If host is a valid hostname or Internet address, and port is an integer
port number or service name, Bash attempts to open a TCP connection to the
corresponding socket.
EXPLANATION


if [ "X$MYEXIT" = "X0" ]; then
  echo "Connection successful. Exit code: $MYEXIT"
else
  echo "Connection unsuccessful. Exit code: $MYEXIT"
fi

exit $MYEXIT
```

Example 29-2. Playing music

```
#!/bin/bash
# music.sh

# Music without external files

# Author: Antonio Macchi
# Used in ABS Guide with permission.


#  /dev/dsp default = 8000 frames per second, 8 bits per frame (1 byte),
#+ 1 channel (mono)

duration=2000       # If 8000 bytes = 1 second, then 2000 = 1/4 second.
volume=$'\xc0'      # Max volume = \xff (or \x00).
mute=$'\x80'        # No volume = \x80 (the middle).

function mknote ()  # $1=Note Hz in bytes (e.g. A = 440Hz ::
{                   #+ 8000 fps / 440 = 16 :: A = 16 bytes per second)
  for t in `seq 0 $duration`
  do
    test $(( $t % $1 )) = 0 && echo -n $volume || echo -n $mute
  done
}

e=`mknote 49`
g=`mknote 41`
a=`mknote 36`
b=`mknote 32`
c=`mknote 30`
cis=`mknote 29`
d=`mknote 27`
e2=`mknote 24`
n=`mknote 32767`
# European notation.
```

```
echo -n "$g$e2$d$c$d$c$a$g$n$g$e$n$g$e2$d$c$c$b$c$cis$n$cis$d \
$n$g$e2$d$c$d$c$a$g$n$g$e$n$g$a$d$c$b$a$b$c" > /dev/dsp
# dsp = Digital Signal Processor

exit      # A "bonny" example of an elegant shell script!
```

29.2. /proc

The /proc directory is actually a pseudo-filesystem. The files in /proc mirror currently running system and kernel processes and contain information and statistics about them.

```
bash$ cat /proc/devices
Character devices:
   1 mem
   2 pty
   3 ttyp
   4 ttyS
   5 cua
   7 vcs
  10 misc
  14 sound
  29 fb
  36 netlink
 128 ptm
 136 pts
 162 raw
 254 pcmcia

 Block devices:
   1 ramdisk
   2 fd
   3 ide0
   9 md



bash$ cat /proc/interrupts
           CPU0
   0:      84505          XT-PIC  timer
   1:       3375          XT-PIC  keyboard
   2:          0          XT-PIC  cascade
   5:          1          XT-PIC  soundblaster
   8:          1          XT-PIC  rtc
  12:       4231          XT-PIC  PS/2 Mouse
  14:     109373          XT-PIC  ide0
 NMI:          0
 ERR:          0


bash$ cat /proc/partitions
major minor  #blocks  name     rio rmerge rsect ruse wio wmerge wsect wuse running use aveq

    3     0    3007872 hda 4472 22260 114520 94240 3551 18703 50384 549710 0 111550 644030
    3     1      52416 hda1 27 395 844 960 4 2 14 180 0 800 1140
    3     2          1 hda2 0 0 0 0 0 0 0 0 0 0 0
    3     4     165280 hda4 10 0 20 210 0 0 0 0 0 210 210
    ...
```

```
bash$ cat /proc/loadavg
0.13 0.42 0.27 2/44 1119



bash$ cat /proc/apm
1.16 1.2 0x03 0x01 0xff 0x80 -1% -1 ?



bash$ cat /proc/acpi/battery/BAT0/info
present:                 yes
 design capacity:         43200 mWh
 last full capacity:      36640 mWh
 battery technology:      rechargeable
 design voltage:          10800 mV
 design capacity warning: 1832 mWh
 design capacity low:     200 mWh
 capacity granularity 1:  1 mWh
 capacity granularity 2:  1 mWh
 model number:            IBM-02K6897
 serial number:            1133
 battery type:            LION
 OEM info:                Panasonic



bash$ fgrep Mem /proc/meminfo
MemTotal:       515216 kB
 MemFree:        266248 kB
```

Shell scripts may extract data from certain of the files in `/proc`. [121]

```
FS=iso                       # ISO filesystem support in kernel?

grep $FS /proc/filesystems   # iso9660
```

```
kernel_version=$( awk '{ print $3 }' /proc/version )
```

```
CPU=$( awk '/model name/ {print $5}' < /proc/cpuinfo )

if [ "$CPU" = "Pentium(R)" ]
then
  run_some_commands
  ...
else
  run_other_commands
  ...
fi



cpu_speed=$( fgrep "cpu MHz" /proc/cpuinfo | awk '{print $4}' )
#  Current operating speed (in MHz) of the cpu on your machine.
#  On a laptop this may vary, depending on use of battery
#+ or AC power.
```

```
#!/bin/bash
# get-commandline.sh
# Get the command-line parameters of a process.

OPTION=cmdline

# Identify PID.
pid=$( echo $(pidof "$1") | awk '{ print $1 }' )
# Get only first             ^^^^^^^^^^^^^^^^^^ of multiple instances.

echo
echo "Process ID of (first instance of) "$1" = $pid"
echo -n "Command-line arguments: "
cat /proc/"$pid"/"$OPTION" | xargs -0 echo
#   Formats output:        ^^^^^^^^^^^^^^^
#   (Thanks, Han Holl, for the fixup!)

echo; echo


# For example:
# sh get-commandline.sh xterm
```

+

```
devfile="/proc/bus/usb/devices"
text="Spd"
USB1="Spd=12"
USB2="Spd=480"


bus_speed=$(fgrep -m 1 "$text" $devfile | awk '{print $9}')
#                 ^^^^ Stop after first match.

if [ "$bus_speed" = "$USB1" ]
then
  echo "USB 1.1 port found."
  # Do something appropriate for USB 1.1.
fi
```

It is even possible to control certain peripherals with commands sent to the `/proc` directory.

```
root# echo on > /proc/acpi/ibm/light
```

This turns on the *Thinklight* in certain models of IBM/Lenovo Thinkpads. (May not work on all Linux distros.)

Of course, caution is advised when writing to `/proc`.

The `/proc` directory contains subdirectories with unusual numerical names. Every one of these names maps to the process ID of a currently running process. Within each of these subdirectories, there are a number of files that hold useful information about the corresponding process. The `stat` and `status` files keep running statistics on the process, the `cmdline` file holds the command-line arguments the process was invoked with, and the `exe` file is a symbolic link to the complete path name of the invoking process. There are a few more such files, but these seem to be the most interesting from a scripting standpoint.

Example 29-3. Finding the process associated with a PID

```
#!/bin/bash
# pid-identifier.sh:
# Gives complete path name to process associated with pid.

ARGNO=1  # Number of arguments the script expects.
E_WRONGARGS=65
E_BADPID=66
E_NOSUCHPROCESS=67
E_NOPERMISSION=68
PROCFILE=exe

if [ $# -ne $ARGNO ]
then
  echo "Usage: `basename $0` PID-number" >&2  # Error message >stderr.
  exit $E_WRONGARGS
fi

pidno=$( ps ax | grep $1 | awk '{ print $1 }' | grep $1 )
# Checks for pid in "ps" listing, field #1.
# Then makes sure it is the actual process, not the process invoked by this script.
# The last "grep $1" filters out this possibility.
#
#    pidno=$( ps ax | awk '{ print $1 }' | grep $1 )
#    also works, as Teemu Huovila, points out.

if [ -z "$pidno" ]  #  If, after all the filtering, the result is a zero-length string,
then                #+ no running process corresponds to the pid given.
  echo "No such process running."
  exit $E_NOSUCHPROCESS
fi

# Alternatively:
#   if ! ps $1 > /dev/null 2>&1
#   then                # no running process corresponds to the pid given.
#     echo "No such process running."
#     exit $E_NOSUCHPROCESS
#    fi

# To simplify the entire process, use "pidof".


if [ ! -r "/proc/$1/$PROCFILE" ]  # Check for read permission.
then
  echo "Process $1 running, but..."
  echo "Can't get read permission on /proc/$1/$PROCFILE."
  exit $E_NOPERMISSION  # Ordinary user can't access some files in /proc.
fi

# The last two tests may be replaced by:
#    if ! kill -0 $1 > /dev/null 2>&1 # '0' is not a signal, but
                                      # this will test whether it is possible
                                      # to send a signal to the process.
#    then echo "PID doesn't exist or you're not its owner" >&2
#    exit $E_BADPID
#    fi



exe_file=$( ls -l /proc/$1 | grep "exe" | awk '{ print $11 }' )
# Or       exe_file=$( ls -l /proc/$1/exe | awk '{print $11}' )
#
#  /proc/pid-number/exe is a symbolic link
```

```
#+ to the complete path name of the invoking process.

if [ -e "$exe_file" ]  #  If /proc/pid-number/exe exists,
then                   #+ then the corresponding process exists.
  echo "Process #$1 invoked by $exe_file."
else
  echo "No such process running."
fi


#  This elaborate script can *almost* be replaced by
#       ps ax | grep $1 | awk '{ print $5 }'
#  However, this will not work...
#+ because the fifth field of 'ps' is argv[0] of the process,
#+ not the executable file path.
#
# However, either of the following would work.
#       find /proc/$1/exe -printf '%l\n'
#       lsof -aFn -p $1 -d txt | sed -ne 's/^n//p'

# Additional commentary by Stephane Chazelas.

exit 0
```

Example 29-4. On-line connect status

```
#!/bin/bash
# connect-stat.sh
#  Note that this script may need modification
#+ to work with a wireless connection.

PROCNAME=pppd        # ppp daemon
PROCFILENAME=status  # Where to look.
NOTCONNECTED=85
INTERVAL=2           # Update every 2 seconds.

pidno=$( ps ax | grep -v "ps ax" | grep -v grep | grep $PROCNAME |
awk '{ print $1 }' )

# Finding the process number of 'pppd', the 'ppp daemon'.
# Have to filter out the process lines generated by the search itself.
#
#  However, as Oleg Philon points out,
#+ this could have been considerably simplified by using "pidof".
#  pidno=$( pidof $PROCNAME )
#
#  Moral of the story:
#+ When a command sequence gets too complex, look for a shortcut.


if [ -z "$pidno" ]   # If no pid, then process is not running.
then
  echo "Not connected."
# exit $NOTCONNECTED
else
  echo "Connected."; echo
fi

while [ true ]       # Endless loop, script can be improved here.
do
```

```
  if [ ! -e "/proc/$pidno/$PROCFILENAME" ]
  # While process running, then "status" file exists.
  then
    echo "Disconnected."
#   exit $NOTCONNECTED
  fi

netstat -s | grep "packets received"  # Get some connect statistics.
netstat -s | grep "packets delivered"


  sleep $INTERVAL
  echo; echo

done

exit 0

# As it stands, this script must be terminated with a Control-C.

#    Exercises:
#    ---------
#    Improve the script so it exits on a "q" keystroke.
#    Make the script more user-friendly in other ways.
#    Fix the script to work with wireless/DSL connections.
```

In general, it is dangerous to *write* to the files in `/proc`, as this can corrupt the filesystem or crash the machine.

Chapter 30. Network Programming

The Net's a cross between an elephant and a white elephant sale: it never forgets, and it's always crap.

--Nemo

A Linux system has quite a number of tools for accessing, manipulating, and troubleshooting network connections. We can incorporate some of these tools into scripts -- scripts that expand our knowledge of networking, useful scripts that can facilitate the administration of a network.

Here is a simple CGI script that demonstrates connecting to a remote server.

Example 30-1. Print the server environment

```
#!/bin/bash
# test-cgi.sh
# by Michael Zick
# Used with permission

# May have to change the location for your site.
# (At the ISP's servers, Bash may not be in the usual place.)
# Other places: /usr/bin or /usr/local/bin
# Might even try it without any path in sha-bang.

# Disable filename globbing.
set -f

# Header tells browser what to expect.
echo Content-type: text/plain
echo

echo CGI/1.0 test script report:
echo

echo environment settings:
set
echo

echo whereis bash?
whereis bash
echo


echo who are we?
echo ${BASH_VERSINFO[*]}
echo

echo argc is $#. argv is "$*".
echo

# CGI/1.0 expected environment variables.

echo SERVER_SOFTWARE = $SERVER_SOFTWARE
echo SERVER_NAME = $SERVER_NAME
echo GATEWAY_INTERFACE = $GATEWAY_INTERFACE
```

```
echo SERVER_PROTOCOL = $SERVER_PROTOCOL
echo SERVER_PORT = $SERVER_PORT
echo REQUEST_METHOD = $REQUEST_METHOD
echo HTTP_ACCEPT = "$HTTP_ACCEPT"
echo PATH_INFO = "$PATH_INFO"
echo PATH_TRANSLATED = "$PATH_TRANSLATED"
echo SCRIPT_NAME = "$SCRIPT_NAME"
echo QUERY_STRING = "$QUERY_STRING"
echo REMOTE_HOST = $REMOTE_HOST
echo REMOTE_ADDR = $REMOTE_ADDR
echo REMOTE_USER = $REMOTE_USER
echo AUTH_TYPE = $AUTH_TYPE
echo CONTENT_TYPE = $CONTENT_TYPE
echo CONTENT_LENGTH = $CONTENT_LENGTH

exit 0

# Here document to give short instructions.
:<<-'_test_CGI_'

1) Drop this in your http://domain.name/cgi-bin directory.
2) Then, open http://domain.name/cgi-bin/test-cgi.sh.

_test_CGI_
```

For security purposes, it may be helpful to identify the IP addresses a computer is accessing.

Example 30-2. IP addresses

```
#!/bin/bash
# ip-addresses.sh
# List the IP addresses your computer is connected to.

#  Inspired by Greg Bledsoe's ddos.sh script,
#  Linux Journal, 09 March 2011.
#    URL:
#  http://www.linuxjournal.com/content/back-dead-simple-bash-complex-ddos
#  Greg licensed his script under the GPL2,
#+ and as a derivative, this script is likewise GPL2.

connection_type=TCP      # Also try UDP.
field=2           # Which field of the output we're interested in.
no_match=LISTEN   # Filter out records containing this. Why?
lsof_args=-ni     # -i lists Internet-associated files.
                  # -n preserves numerical IP addresses.
                  # What happens without the -n option? Try it.
router="[0-9][0-9][0-9][0-9][0-9]->"
#       Delete the router info.

lsof "$lsof_args" | grep $connection_type | grep -v "$no_match" |
      awk '{print $9}' | cut -d : -f $field | sort | uniq |
      sed s/"^$router"//

#  Bledsoe's script assigns the output of a filtered IP list,
#  (similar to lines 19-22, above) to a variable.
#  He checks for multiple connections to a single IP address,
#  then uses:
#
#    iptables -I INPUT -s $ip -p tcp -j REJECT --reject-with tcp-reset
```

```
#
#  ... within a 60-second delay loop to bounce packets from DDOS attacks.


#  Exercise:
#  --------
#  Use the 'iptables' command to extend this script
#+ to reject connection attempts from well-known spammer IP domains.
```

More examples of network programming:

1. Getting the time from *nist.gov*
2. Downloading a URL
3. A GRE tunnel
4. Checking if an Internet server is up
5. Example 16-41
6. Example A-28
7. Example A-29
8. Example 29-1

See also the networking commands in the System and Administrative Commands chapter and the communications commands in the External Filters, Programs and Commands chapter.

Chapter 31. Of Zeros and Nulls

Faultily faultless, icily regular, splendidly null

Dead perfection; no more.

--Alfred Lord Tennyson

`/dev/zero` ... `/dev/null`

Uses of `/dev/null`

Think of `/dev/null` as a *black hole*. It is essentially the equivalent of a write-only file. Everything written to it disappears. Attempts to read or output from it result in nothing. All the same, `/dev/null` can be quite useful from both the command-line and in scripts.

Suppressing `stdout`.

```
cat $filename >/dev/null
# Contents of the file will not list to stdout.
```

Suppressing `stderr` (from Example 16-3).

```
rm $badname 2>/dev/null
#           So error messages [stderr] deep-sixed.
```

Suppressing output from *both* `stdout` and `stderr`.

```
cat $filename 2>/dev/null >/dev/null
# If "$filename" does not exist, there will be no error message output.
# If "$filename" does exist, the contents of the file will not list to stdout.
# Therefore, no output at all will result from the above line of code.
#
#  This can be useful in situations where the return code from a command
#+ needs to be tested, but no output is desired.
#
# cat $filename &>/dev/null
#     also works, as Baris Cicek points out.
```

Deleting contents of a file, but preserving the file itself, with all attendant permissions (from Example 2-1 and Example 2-3):

```
cat /dev/null > /var/log/messages
#  : > /var/log/messages   has same effect, but does not spawn a new process.

cat /dev/null > /var/log/wtmp
```

Automatically emptying the contents of a logfile (especially good for dealing with those nasty "cookies" sent by commercial Web sites):

Example 31-1. Hiding the cookie jar

```
# Obsolete Netscape browser.
# Same principle applies to newer browsers.

if [ -f ~/.netscape/cookies ]  # Remove, if exists.
then
```

```
  rm -f ~/.netscape/cookies
fi

ln -s /dev/null ~/.netscape/cookies
# All cookies now get sent to a black hole, rather than saved to disk.
```

Uses of `/dev/zero`

Like `/dev/null`, `/dev/zero` is a pseudo-device file, but it actually produces a stream of nulls (*binary* zeros, not the ASCII kind). Output written to `/dev/zero` disappears, and it is fairly difficult to actually read the nulls emitted there, though it can be done with od or a hex editor. The chief use of `/dev/zero` is creating an initialized dummy file of predetermined length intended as a temporary swap file.

Example 31-2. Setting up a swapfile using `/dev/zero`

```
#!/bin/bash
# Creating a swap file.

#  A swap file provides a temporary storage cache
#+ which helps speed up certain filesystem operations.

ROOT_UID=0          # Root has $UID 0.
E_WRONG_USER=85     # Not root?

FILE=/swap
BLOCKSIZE=1024
MINBLOCKS=40
SUCCESS=0


# This script must be run as root.
if [ "$UID" -ne "$ROOT_UID" ]
then
  echo; echo "You must be root to run this script."; echo
  exit $E_WRONG_USER
fi


blocks=${1:-$MINBLOCKS}          #  Set to default of 40 blocks,
                                 #+ if nothing specified on command-line.
# This is the equivalent of the command block below.
# --------------------------------------------------
# if [ -n "$1" ]
# then
#   blocks=$1
# else
#   blocks=$MINBLOCKS
# fi
# --------------------------------------------------


if [ "$blocks" -lt $MINBLOCKS ]
then
  blocks=$MINBLOCKS              # Must be at least 40 blocks long.
fi


######################################################################
echo "Creating swap file of size $blocks blocks (KB)."
dd if=/dev/zero of=$FILE bs=$BLOCKSIZE count=$blocks  # Zero out file.
```

```
mkswap $FILE $blocks             # Designate it a swap file.
swapon $FILE                     # Activate swap file.
retcode=$?                       # Everything worked?
#  Note that if one or more of these commands fails,
#+ then it could cause nasty problems.
######################################################################

#  Exercise:
#  Rewrite the above block of code so that if it does not execute
#+ successfully, then:
#    1) an error message is echoed to stderr,
#    2) all temporary files are cleaned up, and
#    3) the script exits in an orderly fashion with an
#+      appropriate error code.

echo "Swap file created and activated."

exit $retcode
```

Another application of `/dev/zero` is to "zero out" a file of a designated size for a special purpose, such as mounting a filesystem on a loopback device (see Example 17-8) or "securely" deleting a file (see Example 16-61).

Example 31-3. Creating a ramdisk

```
#!/bin/bash
# ramdisk.sh

#  A "ramdisk" is a segment of system RAM memory
#+ which acts as if it were a filesystem.
#  Its advantage is very fast access (read/write time).
#  Disadvantages: volatility, loss of data on reboot or powerdown,
#+                less RAM available to system.
#
#  Of what use is a ramdisk?
#  Keeping a large dataset, such as a table or dictionary on ramdisk,
#+ speeds up data lookup, since memory access is much faster than disk access.


E_NON_ROOT_USER=70             # Must run as root.
ROOTUSER_NAME=root

MOUNTPT=/mnt/ramdisk           # Create with mkdir /mnt/ramdisk.
SIZE=2000                      # 2K blocks (change as appropriate)
BLOCKSIZE=1024                 # 1K (1024 byte) block size
DEVICE=/dev/ram0               # First ram device

username=`id -nu`
if [ "$username" != "$ROOTUSER_NAME" ]
then
  echo "Must be root to run \"`basename $0`\"."
  exit $E_NON_ROOT_USER
fi

if [ ! -d "$MOUNTPT" ]         #  Test whether mount point already there,
then                           #+ so no error if this script is run
  mkdir $MOUNTPT               #+ multiple times.
fi

##############################################################################
```

```
dd if=/dev/zero of=$DEVICE count=$SIZE bs=$BLOCKSIZE  # Zero out RAM device.
                                                      # Why is this necessary?
mke2fs $DEVICE                 # Create an ext2 filesystem on it.
mount $DEVICE $MOUNTPT         # Mount it.
chmod 777 $MOUNTPT             # Enables ordinary user to access ramdisk.
                               # However, must be root to unmount it.
##############################################################################
# Need to test whether above commands succeed. Could cause problems otherwise.
# Exercise: modify this script to make it safer.

echo "\"$MOUNTPT\" now available for use."
# The ramdisk is now accessible for storing files, even by an ordinary user.

#  Caution, the ramdisk is volatile, and its contents will disappear
#+ on reboot or power loss.
#  Copy anything you want saved to a regular directory.

# After reboot, run this script to again set up ramdisk.
# Remounting /mnt/ramdisk without the other steps will not work.

#  Suitably modified, this script can by invoked in /etc/rc.d/rc.local,
#+ to set up ramdisk automatically at bootup.
#  That may be appropriate on, for example, a database server.

exit 0
```

In addition to all the above, `/dev/zero` is needed by ELF (*Executable and Linking Format*) UNIX/Linux binaries.

Chapter 32. Debugging

Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.

--Brian Kernighan

The Bash shell contains no built-in debugger, and only bare-bones debugging-specific commands and constructs. Syntax errors or outright typos in the script generate cryptic error messages that are often of no help in debugging a non-functional script.

Example 32-1. A buggy script

```
#!/bin/bash
# ex74.sh

# This is a buggy script.
# Where, oh where is the error?

a=37

if [$a -gt 27 ]
then
  echo $a
fi

exit $?   # 0! Why?
```

Output from script:

```
./ex74.sh: [37: command not found
```

What's wrong with the above script? Hint: after the *if*.

Example 32-2. Missing keyword

```
#!/bin/bash
# missing-keyword.sh
# What error message will this script generate? And why?

for a in 1 2 3
do
  echo "$a"
# done     # Required keyword 'done' commented out in line 8.

exit 0     # Will not exit here!

# === #

# From command line, after script terminates:
  echo $?    # 2
```

Output from script:

```
missing-keyword.sh: line 10: syntax error: unexpected end of file
```

Note that the error message does *not* necessarily reference the line in which the error occurs, but the line where the Bash interpreter finally becomes aware of the error.

Error messages may disregard comment lines in a script when reporting the line number of a syntax error.

What if the script executes, but does not work as expected? This is the all too familiar logic error.

Example 32-3. *test24*: another buggy script

```
#!/bin/bash

#  This script is supposed to delete all filenames in current directory
#+ containing embedded spaces.
#  It doesn't work.
#  Why not?


badname=`ls | grep ' '`

# Try this:
# echo "$badname"

rm "$badname"

exit 0
```

Try to find out what's wrong with Example 32-3 by uncommenting the **`echo "$badname"`** line. Echo statements are useful for seeing whether what you expect is actually what you get.

In this particular case, **`rm "$badname"`** will not give the desired results because `$badname` should not be quoted. Placing it in quotes ensures that **rm** has only one argument (it will match only one filename). A partial fix is to remove to quotes from `$badname` and to reset `$IFS` to contain only a newline, **`IFS=$'\n'`**. However, there are simpler ways of going about it.

```
# Correct methods of deleting filenames containing spaces.
rm *\ *
rm *" "*
rm *' '*
# Thank you. S.C.
```

Summarizing the symptoms of a buggy script,

1. It bombs with a "syntax error" message, or
2. It runs, but does not work as expected (logic error).
3. It runs, works as expected, but has nasty side effects (logic bomb).

Tools for debugging non-working scripts include

1. Inserting echo statements at critical points in the script to trace the variables, and otherwise give a snapshot of what is going on.

   Even better is an **echo** that echoes only when *debug* is on.

   ```
   ### debecho (debug-echo), by Stefano Falsetto ###
   ```

```
### Will echo passed parameters only if DEBUG is set to a value. ###
debecho () {
  if [ ! -z "$DEBUG" ]; then
     echo "$1" >&2
     #         ^^^ to stderr
  fi
}

DEBUG=on
Whatever=whatnot
debecho $Whatever   # whatnot

DEBUG=
Whatever=notwhat
debecho $Whatever   # (Will not echo.)
```

2. Using the tee filter to check processes or data flows at critical points.
3. Setting option flags `-n -v -x`

   `sh -n scriptname` checks for syntax errors without actually running the script. This is the equivalent of inserting **`set -n`** or **`set -o noexec`** into the script. Note that certain types of syntax errors can slip past this check.

   `sh -v scriptname` echoes each command before executing it. This is the equivalent of inserting **`set -v`** or **`set -o verbose`** in the script.

   The `-n` and `-v` flags work well together. **`sh -nv scriptname`** gives a verbose syntax check.

   `sh -x scriptname` echoes the result each command, but in an abbreviated manner. This is the equivalent of inserting **`set -x`** or **`set -o xtrace`** in the script.

   Inserting **`set -u`** or **`set -o nounset`** in the script runs it, but gives an unbound variable error message and aborts the script.

```
set -u   # Or   set -o nounset

# Setting a variable to null will not trigger the error/abort.
# unset_var=

echo $unset_var   # Unset (and undeclared) variable.

echo "Should not echo!"

# sh t2.sh
# t2.sh: line 6: unset_var: unbound variable
```

4. Using an "assert" function to test a variable or condition at critical points in a script. (This is an idea borrowed from C.)

   Example 32-4. Testing a condition with an *assert*

```
#!/bin/bash
# assert.sh

#######################################################################
assert ()                 #  If condition false,
{                         #+ exit from script
```

```
                          #+ with appropriate error message.
  E_PARAM_ERR=98
  E_ASSERT_FAILED=99


  if [ -z "$2" ]          #  Not enough parameters passed
  then                    #+ to assert() function.
    return $E_PARAM_ERR   #  No damage done.
  fi

  lineno=$2

  if [ ! $1 ]
  then
    echo "Assertion failed:  \"$1\""
    echo "File \"$0\", line $lineno"    # Give name of file and line number.
    exit $E_ASSERT_FAILED
  # else
  #   return
  #   and continue executing the script.
  fi
} # Insert a similar assert() function into a script you need to debug.
#######################################################################


a=5
b=4
condition="$a -lt $b"     #  Error message and exit from script.
                          #  Try setting "condition" to something else
                          #+ and see what happens.

assert "$condition" $LINENO
# The remainder of the script executes only if the "assert" does not fail.


# Some commands.
# Some more commands . . .
echo "This statement echoes only if the \"assert\" does not fail."
# . . .
# More commands . . .

exit $?
```

5. Using the $LINENO variable and the caller builtin.
6. Trapping at exit.

   The exit command in a script triggers a signal 0, terminating the process, that is, the script itself. [122] It is often useful to trap the *exit*, forcing a "printout" of variables, for example. The *trap* must be the first command in the script.

Trapping signals

trap

Specifies an action on receipt of a signal; also useful for debugging.

A *signal* is a message sent to a process, either by the kernel or another process, telling it to take some specified action (usually to terminate). For example, hitting a Control-C sends a user interrupt, an INT signal, to a running program.

A simple instance:

```
trap '' 2
# Ignore interrupt 2 (Control-C), with no action specified.

trap 'echo "Control-C disabled."' 2
# Message when Control-C pressed.
```

Example 32-5. Trapping at exit

```
#!/bin/bash
# Hunting variables with a trap.

trap 'echo Variable Listing --- a = $a  b = $b' EXIT
#  EXIT is the name of the signal generated upon exit from a script.
#
#  The command specified by the "trap" doesn't execute until
#+ the appropriate signal is sent.

echo "This prints before the \"trap\" --"
echo "even though the script sees the \"trap\" first."
echo

a=39

b=36

exit 0
#  Note that commenting out the 'exit' command makes no difference,
#+ since the script exits in any case after running out of commands.
```

Example 32-6. Cleaning up after Control-C

```
#!/bin/bash
# logon.sh: A quick 'n dirty script to check whether you are on-line yet.

umask 177  # Make sure temp files are not world readable.


TRUE=1
LOGFILE=/var/log/messages
#  Note that $LOGFILE must be readable
#+ (as root, chmod 644 /var/log/messages).
TEMPFILE=temp.$$
#  Create a "unique" temp file name, using process id of the script.
#     Using 'mktemp' is an alternative.
#     For example:
#     TEMPFILE=`mktemp temp.XXXXXX`
KEYWORD=address
#  At logon, the line "remote IP address xxx.xxx.xxx.xxx"
#                      appended to /var/log/messages.
ONLINE=22
USER_INTERRUPT=13
CHECK_LINES=100
#  How many lines in log file to check.
```

```
trap 'rm -f $TEMPFILE; exit $USER_INTERRUPT' TERM INT
#  Cleans up the temp file if script interrupted by control-c.

echo

while [ $TRUE ]  #Endless loop.
do
  tail -n $CHECK_LINES $LOGFILE> $TEMPFILE
  #  Saves last 100 lines of system log file as temp file.
  #  Necessary, since newer kernels generate many log messages at log on.
  search=`grep $KEYWORD $TEMPFILE`
  #  Checks for presence of the "IP address" phrase,
  #+ indicating a successful logon.

  if [ ! -z "$search" ] #  Quotes necessary because of possible spaces.
  then
     echo "On-line"
     rm -f $TEMPFILE    #  Clean up temp file.
     exit $ONLINE
  else
     echo -n "."        #  The -n option to echo suppresses newline,
                        #+ so you get continuous rows of dots.
  fi

  sleep 1
done


#  Note: if you change the KEYWORD variable to "Exit",
#+ this script can be used while on-line
#+ to check for an unexpected logoff.

# Exercise: Change the script, per the above note,
#           and prettify it.

exit 0


# Nick Drage suggests an alternate method:

while true
  do ifconfig ppp0 | grep UP 1> /dev/null && echo "connected" && exit 0
  echo -n "."   # Prints dots (.....) until connected.
  sleep 2
done

# Problem: Hitting Control-C to terminate this process may be insufficient.
#+         (Dots may keep on echoing.)
# Exercise: Fix this.



# Stephane Chazelas has yet another alternative:

CHECK_INTERVAL=1

while ! tail -n 1 "$LOGFILE" | grep -q "$KEYWORD"
do echo -n .
   sleep $CHECK_INTERVAL
done
echo "On-line"
```

```
# Exercise: Discuss the relative strengths and weaknesses
#           of each of these various approaches.
```

Example 32-7. A Simple Implementation of a Progress Bar

```
#! /bin/bash
# progress-bar2.sh
# Author: Graham Ewart (with reformatting by ABS Guide author).
# Used in ABS Guide with permission (thanks!).

# Invoke this script with bash. It doesn't work with sh.

interval=1
long_interval=10

{
     trap "exit" SIGUSR1
     sleep $interval; sleep $interval
     while true
     do
       echo -n '.'     # Use dots.
       sleep $interval
     done; } &         # Start a progress bar as a background process.

pid=$!
trap "echo !; kill -USR1 $pid; wait $pid"  EXIT        # To handle ^C.

echo -n 'Long-running process '
sleep $long_interval
echo ' Finished!'

kill -USR1 $pid
wait $pid              # Stop the progress bar.
trap EXIT

exit $?
```

The DEBUG argument to **trap** causes a specified action to execute after every command in a script. This permits tracing variables, for example.

Example 32-8. Tracing a variable

```
#!/bin/bash

trap 'echo "VARIABLE-TRACE> \$variable = \"$variable\""' DEBUG
# Echoes the value of $variable after every command.

variable=29; line=$LINENO

echo "  Just initialized \$variable to $variable in line number $line."

let "variable *= 3"; line=$LINENO
echo "  Just multiplied \$variable by 3 in line number $line."

exit 0

#  The "trap 'command1 . . . command2 . . .' DEBUG" construct is
#+ more appropriate in the context of a complex script,
```

```
#+ where inserting multiple "echo $variable" statements might be
#+ awkward and time-consuming.

# Thanks, Stephane Chazelas for the pointer.


Output of script:

VARIABLE-TRACE> $variable = ""
VARIABLE-TRACE> $variable = "29"
  Just initialized $variable to 29.
VARIABLE-TRACE> $variable = "29"
VARIABLE-TRACE> $variable = "87"
  Just multiplied $variable by 3.
VARIABLE-TRACE> $variable = "87"
```

Of course, the **trap** command has other uses aside from debugging, such as disabling certain keystrokes within a script (see Example A-43).

Example 32-9. Running multiple processes (on an SMP box)

```
#!/bin/bash
# parent.sh
# Running multiple processes on an SMP box.
# Author: Tedman Eng

#  This is the first of two scripts,
#+ both of which must be present in the current working directory.




LIMIT=$1         # Total number of process to start
NUMPROC=4        # Number of concurrent threads (forks?)
PROCID=1         # Starting Process ID
echo "My PID is $$"

function start_thread() {
        if [ $PROCID -le $LIMIT ] ; then
                ./child.sh $PROCID&
                let "PROCID++"
        else
           echo "Limit reached."
           wait
           exit
        fi
}

while [ "$NUMPROC" -gt 0 ]; do
        start_thread;
        let "NUMPROC--"
done


while true
do

trap "start_thread" SIGRTMIN

done
```

```
exit 0



# ======== Second script follows ========


#!/bin/bash
# child.sh
# Running multiple processes on an SMP box.
# This script is called by parent.sh.
# Author: Tedman Eng

temp=$RANDOM
index=$1
shift
let "temp %= 5"
let "temp += 4"
echo "Starting $index  Time:$temp" "$@"
sleep ${temp}
echo "Ending $index"
kill -s SIGRTMIN $PPID

exit 0


# ====================== SCRIPT AUTHOR'S NOTES ====================== #
#  It's not completely bug free.
#  I ran it with limit = 500 and after the first few hundred iterations,
#+ one of the concurrent threads disappeared!
#  Not sure if this is collisions from trap signals or something else.
#  Once the trap is received, there's a brief moment while executing the
#+ trap handler but before the next trap is set.  During this time, it may
#+ be possible to miss a trap signal, thus miss spawning a child process.

#  No doubt someone may spot the bug and will be writing
#+ . . . in the future.



# ===================================================================== #



# ----------------------------------------------------------------------#



#################################################################
# The following is the original script written by Vernia Damiano.
# Unfortunately, it doesn't work properly.
#################################################################

#!/bin/bash

#  Must call script with at least one integer parameter
#+ (number of concurrent processes).
#  All other parameters are passed through to the processes started.


INDICE=8        # Total number of process to start
```

```
TEMPO=5          # Maximum sleep time per process
E_BADARGS=65     # No arg(s) passed to script.

if [ $# -eq 0 ] # Check for at least one argument passed to script.
then
  echo "Usage: `basename $0` number_of_processes [passed params]"
  exit $E_BADARGS
fi

NUMPROC=$1              # Number of concurrent process
shift
PARAMETRI=( "$@" )      # Parameters of each process

function avvia() {
         local temp
         local index
         temp=$RANDOM
         index=$1
         shift
         let "temp %= $TEMPO"
         let "temp += 1"
         echo "Starting $index Time:$temp" "$@"
         sleep ${temp}
         echo "Ending $index"
         kill -s SIGRTMIN $$
}

function parti() {
         if [ $INDICE -gt 0 ] ; then
              avvia $INDICE "${PARAMETRI[@]}" &
                let "INDICE--"
         else
                trap : SIGRTMIN
         fi
}

trap parti SIGRTMIN

while [ "$NUMPROC" -gt 0 ]; do
         parti;
         let "NUMPROC--"
done

wait
trap - SIGRTMIN

exit $?

: <<SCRIPT_AUTHOR_COMMENTS
I had the need to run a program, with specified options, on a number of
different files, using a SMP machine. So I thought [I'd] keep running
a specified number of processes and start a new one each time . . . one
of these terminates.

The "wait" instruction does not help, since it waits for a given process
or *all* process started in background. So I wrote [this] bash script
that can do the job, using the "trap" instruction.
  --Vernia Damiano
SCRIPT_AUTHOR_COMMENTS
```

`trap '' SIGNAL` (two adjacent apostrophes) disables SIGNAL for the remainder of the script. **`trap SIGNAL`** restores the functioning of SIGNAL once more. This is useful to protect a critical portion of a script from an undesirable interrupt.

```
trap '' 2  # Signal 2 is Control-C, now disabled.
command
command
command
trap 2     # Reenables Control-C
```

Version 3 of Bash adds the following internal variables for use by the debugger.

1. `$BASH_ARGC`

   Number of command-line arguments passed to script, similar to `$#`.
2. `$BASH_ARGV`

   Final command-line parameter passed to script, equivalent `${!#}`.
3. `$BASH_COMMAND`

   Command currently executing.
4. `$BASH_EXECUTION_STRING`

   The *option string* following the `-c` option to Bash.
5. `$BASH_LINENO`

   In a function, indicates the line number of the function call.
6. `$BASH_REMATCH`

   Array variable associated with **=~** conditional regex matching.
7. `$BASH_SOURCE`

   This is the name of the script, usually the same as $0.
8. `$BASH_SUBSHELL`

Chapter 33. Options

Options are settings that change shell and/or script behavior.

The set command enables options within a script. At the point in the script where you want the options to take effect, use **set -o option-name** or, in short form, **set -option-abbrev**. These two forms are equivalent.

```
#!/bin/bash

set -o verbose
# Echoes all commands before executing.
```

```
#!/bin/bash

set -v
# Exact same effect as above.
```

To *disable* an option within a script, use **set +o option-name** or **set +option-abbrev**.

```
#!/bin/bash

set -o verbose
# Command echoing on.
command
...
command

set +o verbose
# Command echoing off.
command
# Not echoed.


set -v
# Command echoing on.
command
...
command

set +v
# Command echoing off.
command

exit 0
```

An alternate method of enabling options in a script is to specify them immediately following the `#!` script header.

```
#!/bin/bash -x
#
# Body of script follows.
```

It is also possible to enable script options from the command line. Some options that will not work with **set** are available this way. Among these are `-i`, force script to run interactive.

`bash -v script-name`

`bash -o verbose script-name`

The following is a listing of some useful options. They may be specified in either abbreviated form (preceded by a single dash) or by complete name (preceded by a *double* dash or by `-o`).

Table 33-1. Bash options

Abbreviation	Name	Effect
`-B`	brace expansion	*Enable* brace expansion (default setting = *on*)
`+B`	brace expansion	*Disable* brace expansion
`-C`	noclobber	Prevent overwriting of files by redirection (may be overridden by >|)
`-D`	(none)	List double-quoted strings prefixed by $, but do not execute commands in script
`-a`	allexport	Export all defined variables
`-b`	notify	Notify when jobs running in background terminate (not of much use in a script)
`-c ...`	(none)	Read commands from **...**
`checkjobs`		Informs user of any open jobs upon shell exit. Introduced in version 4 of Bash, and still "experimental." *Usage:* shopt -s checkjobs (*Caution:* may hang!)
`-e`	errexit	Abort script at first error, when a command exits with non-zero status (except in until or while loops, if-tests, list constructs)
`-f`	noglob	Filename expansion (globbing) disabled
`globstar`	*globbing* star-match	Enables the ** globbing operator (version 4+ of Bash). *Usage:* shopt -s globstar
`-i`	interactive	Script runs in *interactive* mode
`-n`	noexec	Read commands in script, but do not execute them (syntax check)
`-o Option-Name`	(none)	Invoke the *Option-Name* option
`-o posix`	POSIX	Change the behavior of Bash, or invoked script, to conform to POSIX standard.
`-o pipefail`	pipe failure	Causes a pipeline to return the exit status of the last command in the pipe that returned a non-zero return value.
`-p`	privileged	Script runs as "suid" (caution!)
`-r`	restricted	Script runs in *restricted* mode (see Chapter 22).
`-s`	stdin	Read commands from `stdin`
`-t`	(none)	Exit after first command

`-u`	nounset	Attempt to use undefined variable outputs error message, and forces an exit
`-v`	verbose	Print each command to `stdout` before executing it
`-x`	xtrace	Similar to `-v`, but expands commands
`-`	(none)	End of options flag. All other arguments are positional parameters.
`--`	(none)	Unset positional parameters. If arguments given (`-- arg1 arg2`), positional parameters set to arguments.

Chapter 34. Gotchas

Turandot: Gli enigmi sono tre, la morte una!

Caleph: No, no! Gli enigmi sono tre, una la vita!

--Puccini

Here are some (non-recommended!) scripting practices that will bring excitement into an otherwise dull life.

- Assigning reserved words or characters to variable names.

```
case=value0       # Causes problems.
23skidoo=value1   # Also problems.
# Variable names starting with a digit are reserved by the shell.
# Try _23skidoo=value1. Starting variables with an underscore is okay.

# However . . .   using just an underscore will not work.
_=25
echo $_           # $_ is a special variable set to last arg of last command.
# But . . .       _ is a valid function name!

xyz((!*=value2    # Causes severe problems.
# As of version 3 of Bash, periods are not allowed within variable names.
```

- Using a hyphen or other reserved characters in a variable name (or function name).

```
var-1=23
# Use 'var_1' instead.

function-whatever ()   # Error
# Use 'function_whatever ()' instead.


# As of version 3 of Bash, periods are not allowed within function names.
function.whatever ()   # Error
# Use 'functionWhatever ()' instead.
```

- Using the same name for a variable and a function. This can make a script difficult to understand.

```
do_something ()
{
  echo "This function does something with \"$1\"."
}

do_something=do_something

do_something do_something

# All this is legal, but highly confusing.
```

- Using whitespace inappropriately. In contrast to other programming languages, Bash can be quite finicky about whitespace.

```
var1 = 23   # 'var1=23' is correct.
# On line above, Bash attempts to execute command "var1"
# with the arguments "=" and "23".
```

```
let c = $a - $b   # Instead:   let c=$a-$b   or   let "c = $a - $b"

if [ $a -le 5]    # if [ $a -le 5 ]   is correct.
#           ^^      if [ "$a" -le 5 ]   is even better.
                  # [[ $a -le 5 ]] also works.
```

- Not terminating with a semicolon the final command in a code block within curly brackets.

```
{ ls -l; df; echo "Done." }
# bash: syntax error: unexpected end of file

{ ls -l; df; echo "Done."; }
#                        ^     ### Final command needs semicolon.
```

- Assuming uninitialized variables (variables before a value is assigned to them) are "zeroed out". An uninitialized variable has a value of *null*, *not* zero.

```
#!/bin/bash

echo "uninitialized_var = $uninitialized_var"
# uninitialized_var =

# However . . .
# if $BASH_VERSION ≥ 4.2; then

if [[ ! -v uninitialized_var ]]
then
  uninitialized_var=0   # Initialize it to zero!
fi
```

- Mixing up = and *-eq* in a test. Remember, = is for comparing literal variables and *-eq* for integers.

```
if [ "$a" = 273 ]      # Is $a an integer or string?
if [ "$a" -eq 273 ]    # If $a is an integer.

# Sometimes you can interchange -eq and = without adverse consequences.
# However . . .


a=273.0   # Not an integer.

if [ "$a" = 273 ]
then
  echo "Comparison works."
else
  echo "Comparison does not work."
fi    # Comparison does not work.

# Same with   a=" 273"  and a="0273".


# Likewise, problems trying to use "-eq" with non-integer values.

if [ "$a" -eq 273.0 ]
then
  echo "a = $a"
fi  # Aborts with an error message.
# test.sh: [: 273.0: integer expression expected
```

- Misusing string comparison operators.

Example 34-1. Numerical and string comparison are not equivalent

```
#!/bin/bash
# bad-op.sh: Trying to use a string comparison on integers.

echo
number=1

#  The following while-loop has two errors:
#+ one blatant, and the other subtle.

while [ "$number" < 5 ]    # Wrong! Should be:  while [ "$number" -lt 5 ]
do
  echo -n "$number "
  let "number += 1"
done
#  Attempt to run this bombs with the error message:
#+ bad-op.sh: line 10: 5: No such file or directory
#  Within single brackets, "<" must be escaped,
#+ and even then, it's still wrong for comparing integers.

echo "---------------------"

while [ "$number" \< 5 ]    #  1 2 3 4
do                          #
  echo -n "$number "        #  It *seems* to work, but . . .
  let "number += 1"         #+ it actually does an ASCII comparison,
done                        #+ rather than a numerical one.

echo; echo "---------------------"

# This can cause problems. For example:

lesser=5
greater=105

if [ "$greater" \< "$lesser" ]
then
  echo "$greater is less than $lesser"
fi                          # 105 is less than 5
#  In fact, "105" actually is less than "5"
#+ in a string comparison (ASCII sort order).

echo

exit 0
```

- Attempting to use let to set string variables.

```
let "a = hello, you"
echo "$a"   # 0
```

- Sometimes variables within "test" brackets ([ ]) need to be quoted (double quotes). Failure to do so may cause unexpected behavior. See Example 7-6, Example 20-5, and Example 9-6.

- Quoting a variable containing whitespace prevents splitting. Sometimes this produces unintended consequences.
- Commands issued from a script may fail to execute because the script owner lacks execute permission for them. If a user cannot invoke a command from the command-line, then putting it into a script will likewise fail. Try changing the attributes of the command in question, perhaps even setting the suid bit (as *root*, of course).
- Attempting to use **-** as a redirection operator (which it is not) will usually result in an unpleasant surprise.

```
command1 2> - | command2
# Trying to redirect error output of command1 into a pipe . . .
# . . . will not work.

command1 2>& - | command2  # Also futile.

Thanks, S.C.
```

- Using Bash version 2+ functionality may cause a bailout with error messages. Older Linux machines may have version 1.XX of Bash as the default installation.

```
#!/bin/bash

minimum_version=2
# Since Chet Ramey is constantly adding features to Bash,
# you may set $minimum_version to 2.XX, 3.XX, or whatever is appropriate.
E_BAD_VERSION=80

if [ "$BASH_VERSION" \< "$minimum_version" ]
then
  echo "This script works only with Bash, version $minimum or greater."
  echo "Upgrade strongly recommended."
  exit $E_BAD_VERSION
fi

...
```

- Using Bash-specific functionality in a Bourne shell script (`#!/bin/sh`) on a non-Linux machine may cause unexpected behavior. A Linux system usually aliases **sh** to **bash**, but this does not necessarily hold true for a generic UNIX machine.
- Using undocumented features in Bash turns out to be a dangerous practice. In previous releases of this book there were several scripts that depended on the "feature" that, although the maximum value of an exit or return value was 255, that limit did not apply to *negative* integers. Unfortunately, in version 2.05b and later, that loophole disappeared. See Example 24-9.
- In certain contexts, a misleading exit status may be returned. This may occur when setting a local variable within a function or when assigning an arithmetic value to a variable.
- The exit status of an arithmetic expression is *not* equivalent to an *error code*.

```
var=1 && ((--var)) && echo $var
#        ^^^^^^^^^ Here the and-list terminates with exit status 1.
#                     $var doesn't echo!
echo $?   # 1
```

* A script with DOS-type newlines (`\r\n`) will fail to execute, since **`#!/bin/bash\r\n`** is *not* recognized, *not* the same as the expected **`#!/bin/bash\n`**. The fix is to convert the script to UNIX-style newlines.

```
#!/bin/bash

echo "Here"

unix2dos $0    # Script changes itself to DOS format.
chmod 755 $0   # Change back to execute permission.
               # The 'unix2dos' command removes execute permission.

./$0           # Script tries to run itself again.
               # But it won't work as a DOS file.

echo "There"

exit 0
```

* A shell script headed by **`#!/bin/sh`** will not run in full Bash-compatibility mode. Some Bash-specific functions might be disabled. Scripts that need complete access to all the Bash-specific extensions should start with **`#!/bin/bash`**.
* Putting whitespace in front of the terminating limit string of a here document will cause unexpected behavior in a script.
* Putting more than one *echo* statement in a function whose output is captured.

```
add2 ()
{
  echo "Whatever ... "   # Delete this line!
  let "retval = $1 + $2"
    echo $retval
    }

    num1=12
    num2=43
    echo "Sum of $num1 and $num2 = $(add2 $num1 $num2)"

#   Sum of 12 and 43 = Whatever ...
#   55

#        The "echoes" concatenate.
```

This will not work.

* A script may not **export** variables back to its parent process, the shell, or to the environment. Just as we learned in biology, a child process can inherit from a parent, but not vice versa.

```
WHATEVER=/home/bozo
export WHATEVER
exit 0
```

```
bash$ echo $WHATEVER

bash$
```

Sure enough, back at the command prompt, $WHATEVER remains unset.

* Setting and manipulating variables in a subshell, then attempting to use those same variables outside the scope of the subshell will result an unpleasant surprise.

Example 34-2. Subshell Pitfalls

```
#!/bin/bash
# Pitfalls of variables in a subshell.

outer_variable=outer
echo
echo "outer_variable = $outer_variable"
echo

(
# Begin subshell

echo "outer_variable inside subshell = $outer_variable"
inner_variable=inner  # Set
echo "inner_variable inside subshell = $inner_variable"
outer_variable=inner  # Will value change globally?
echo "outer_variable inside subshell = $outer_variable"

# Will 'exporting' make a difference?
#    export inner_variable
#    export outer_variable
# Try it and see.

# End subshell
)

echo
echo "inner_variable outside subshell = $inner_variable"  # Unset.
echo "outer_variable outside subshell = $outer_variable"  # Unchanged.
echo

exit 0

# What happens if you uncomment lines 19 and 20?
# Does it make a difference?
```

- Piping **echo** output to a read may produce unexpected results. In this scenario, the **read** acts as if it were running in a subshell. Instead, use the set command (as in Example 15-18).

Example 34-3. Piping the output of *echo* to a *read*

```
#!/bin/bash
#  badread.sh:
#  Attempting to use 'echo and 'read'
#+ to assign variables non-interactively.

#   shopt -s lastpipe

a=aaa
b=bbb
c=ccc

echo "one two three" | read a b c
# Try to reassign a, b, and c.

echo
echo "a = $a"  # a = aaa
echo "b = $b"  # b = bbb
```

```
echo "c = $c"  # c = ccc
# Reassignment failed.

### However . . .
##  Uncommenting line 6:
#   shopt -s lastpipe
##+ fixes the problem!
### This is a new feature in Bash, version 4.2.

# ------------------------------

# Try the following alternative.

var=`echo "one two three"`
set -- $var
a=$1; b=$2; c=$3

echo "-------"
echo "a = $a"  # a = one
echo "b = $b"  # b = two
echo "c = $c"  # c = three
# Reassignment succeeded.

# ------------------------------

#  Note also that an echo to a 'read' works within a subshell.
#  However, the value of the variable changes *only* within the subshell.

a=aaa          # Starting all over again.
b=bbb
c=ccc

echo; echo
echo "one two three" | ( read a b c;
echo "Inside subshell: "; echo "a = $a"; echo "b = $b"; echo "c = $c" )
# a = one
# b = two
# c = three
echo "-----------------"
echo "Outside subshell: "
echo "a = $a"  # a = aaa
echo "b = $b"  # b = bbb
echo "c = $c"  # c = ccc
echo

exit 0
```

In fact, as Anthony Richardson points out, piping to *any* loop can cause a similar problem.

```
# Loop piping troubles.
#  This example by Anthony Richardson,
#+ with addendum by Wilbert Berendsen.


foundone=false
find $HOME -type f -atime +30 -size 100k |
while true
do
   read f
   echo "$f is over 100KB and has not been accessed in over 30 days"
   echo "Consider moving the file to archives."
```

```
   foundone=true
   # ------------------------------------
     echo "Subshell level = $BASH_SUBSHELL"
   # Subshell level = 1
   # Yes, we're inside a subshell.
   # ------------------------------------
done

#  foundone will always be false here since it is
#+ set to true inside a subshell
if [ $foundone = false ]
then
   echo "No files need archiving."
fi

# =====================Now, here is the correct way:=================

foundone=false
for f in $(find $HOME -type f -atime +30 -size 100k)  # No pipe here.
do
   echo "$f is over 100KB and has not been accessed in over 30 days"
   echo "Consider moving the file to archives."
   foundone=true
done

if [ $foundone = false ]
then
   echo "No files need archiving."
fi

# ==================And here is another alternative==================

#  Places the part of the script that reads the variables
#+ within a code block, so they share the same subshell.
#  Thank you, W.B.

find $HOME -type f -atime +30 -size 100k | {
     foundone=false
     while read f
     do
       echo "$f is over 100KB and has not been accessed in over 30 days"
       echo "Consider moving the file to archives."
       foundone=true
     done

     if ! $foundone
     then
       echo "No files need archiving."
     fi
}
```

A lookalike problem occurs when trying to write the `stdout` of a **tail -f** piped to grep.

```
tail -f /var/log/messages | grep "$ERROR_MSG" >> error.log
#  The "error.log" file will not have anything written to it.
#  As Samuli Kaipiainen points out, this results from grep
#+ buffering its output.
#  The fix is to add the "--line-buffered" parameter to grep.
```

- Using "suid" commands within scripts is risky, as it may compromise system security. [123]

- Using shell scripts for CGI programming may be problematic. Shell script variables are not "typesafe," and this can cause undesirable behavior as far as CGI is concerned. Moreover, it is difficult to "cracker-proof" shell scripts.
- Bash does not handle the double slash (//) string correctly.
- Bash scripts written for Linux or BSD systems may need fixups to run on a commercial UNIX machine. Such scripts often employ the GNU set of commands and filters, which have greater functionality than their generic UNIX counterparts. This is particularly true of such text processing utilites as tr.
- Sadly, updates to Bash itself have broken older scripts that used to work perfectly fine. Let us recall how risky it is to use undocumented Bash features.

Danger is near thee --

Beware, beware, beware, beware.

Many brave hearts are asleep in the deep.

So beware --

Beware.

--A.J. Lamb and H.W. Petrie

Chapter 35. Scripting With Style

Get into the habit of writing shell scripts in a structured and systematic manner. Even on-the-fly and "written on the back of an envelope" scripts will benefit if you take a few minutes to plan and organize your thoughts before sitting down and coding.

Herewith are a few stylistic guidelines. This is not (necessarily) intended as an *Official Shell Scripting Stylesheet*.

35.1. Unofficial Shell Scripting Stylesheet

- Comment your code. This makes it easier for others to understand (and appreciate), and easier for you to maintain.

```
PASS="$PASS${MATRIX:$(($RANDOM%${#MATRIX})):1}"
#  It made perfect sense when you wrote it last year,
#+ but now it's a complete mystery.
#  (From Antek Sawicki's "pw.sh" script.)
```

  Add descriptive headers to your scripts and functions.

```
#!/bin/bash

#************************************************#
#                   xyz.sh                       #
#           written by Bozo Bozeman              #
#                July 05, 2001                   #
#                                                #
#           Clean up project files.              #
#************************************************#

E_BADDIR=85                       # No such directory.
projectdir=/home/bozo/projects    # Directory to clean up.

# --------------------------------------------------------- #
# cleanup_pfiles ()                                         #
# Removes all files in designated directory.                #
# Parameter: $target_directory                              #
# Returns: 0 on success, $E_BADDIR if something went wrong. #
# --------------------------------------------------------- #
cleanup_pfiles ()
{
  if [ ! -d "$1" ]  # Test if target directory exists.
  then
    echo "$1 is not a directory."
    return $E_BADDIR
  fi

  rm -f "$1"/*
  return 0   # Success.
}

cleanup_pfiles $projectdir

exit $?
```

- Avoid using "magic numbers," [124] that is, "hard-wired" literal constants. Use meaningful variable names instead. This makes the script easier to understand and permits making changes and updates without breaking the application.

```
if [ -f /var/log/messages ]
then
  ...
fi
#  A year later, you decide to change the script to check /var/log/syslog.
#  It is now necessary to manually change the script, instance by instance,
#+ and hope nothing breaks.

# A better way:
LOGFILE=/var/log/messages  # Only line that needs to be changed.
if [ -f "$LOGFILE" ]
then
  ...
fi
```

- Choose descriptive names for variables and functions.

```
fl=`ls -al $dirname`                 # Cryptic.
file_listing=`ls -al $dirname`       # Better.


MAXVAL=10   # All caps used for a script constant.
while [ "$index" -le "$MAXVAL" ]
...


E_NOTFOUND=95                        #  Uppercase for an errorcode,
                                     #+ and name prefixed with E_.
if [ ! -e "$filename" ]
then
  echo "File $filename not found."
  exit $E_NOTFOUND
fi


MAIL_DIRECTORY=/var/spool/mail/bozo  #  Uppercase for an environmental
export MAIL_DIRECTORY                #+ variable.


GetAnswer ()                         #  Mixed case works well for a
{                                    #+ function name, especially
  prompt=$1                          #+ when it improves legibility.
  echo -n $prompt
  read answer
  return $answer
}

GetAnswer "What is your favorite number? "
favorite_number=$?
echo $favorite_number


_uservariable=23                     # Permissible, but not recommended.
# It's better for user-defined variables not to start with an underscore.
# Leave that for system variables.
```

- Use exit codes in a systematic and meaningful way.

```
E_WRONG_ARGS=95
...
...
exit $E_WRONG_ARGS
```

See also Appendix E.

Ender suggests using the exit codes in `/usr/include/sysexits.h` in shell scripts, though these are primarily intended for C and C++ programming.

- Use standardized parameter flags for script invocation. *Ender* proposes the following set of flags.

```
-a      All: Return all information (including hidden file info).
-b      Brief: Short version, usually for other scripts.
-c      Copy, concatenate, etc.
-d      Daily: Use information from the whole day, and not merely
        information for a specific instance/user.
-e      Extended/Elaborate: (often does not include hidden file info).
-h      Help: Verbose usage w/descs, aux info, discussion, help.
        See also -V.
-l      Log output of script.
-m      Manual: Launch man-page for base command.
-n      Numbers: Numerical data only.
-r      Recursive: All files in a directory (and/or all sub-dirs).
-s      Setup & File Maintenance: Config files for this script.
-u      Usage: List of invocation flags for the script.
-v      Verbose: Human readable output, more or less formatted.
-V      Version / License / Copy(right|left) / Contribs (email too).
```

See also Section G.1.

- Break complex scripts into simpler modules. Use functions where appropriate. See Example 37-4.
- Don't use a complex construct where a simpler one will do.

```
COMMAND
if [ $? -eq 0 ]
...
# Redundant and non-intuitive.

if COMMAND
...
# More concise (if perhaps not quite as legible).
```

... reading the UNIX source code to the Bourne shell (/bin/sh). I was shocked at how much simple algorithms could be made cryptic, and therefore useless, by a poor choice of code style. I asked myself, "Could someone be proud of this code?"

--Landon Noll

Chapter 36. Miscellany

Nobody really knows what the Bourne shell's grammar is. Even examination of the source code is little help.

--Tom Duff

36.1. Interactive and non-interactive shells and scripts

An *interactive* shell reads commands from user input on a `tty`. Among other things, such a shell reads startup files on activation, displays a prompt, and enables job control by default. The user can *interact* with the shell.

A shell running a script is always a non-interactive shell. All the same, the script can still access its `tty`. It is even possible to emulate an interactive shell in a script.

```
#!/bin/bash
MY_PROMPT='$ '
while :
do
  echo -n "$MY_PROMPT"
  read line
  eval "$line"
  done

exit 0

# This example script, and much of the above explanation supplied by
# Stéphane Chazelas (thanks again).
```

Let us consider an *interactive* script to be one that requires input from the user, usually with read statements (see Example 15-3). "Real life" is actually a bit messier than that. For now, assume an interactive script is bound to a tty, a script that a user has invoked from the console or an *xterm*.

Init and startup scripts are necessarily non-interactive, since they must run without human intervention. Many administrative and system maintenance scripts are likewise non-interactive. Unvarying repetitive tasks cry out for automation by non-interactive scripts.

Non-interactive scripts can run in the background, but interactive ones hang, waiting for input that never comes. Handle that difficulty by having an **expect** script or embedded here document feed input to an interactive script running as a background job. In the simplest case, redirect a file to supply input to a **read** statement (**read variable <file**). These particular workarounds make possible general purpose scripts that run in either interactive or non-interactive modes.

If a script needs to test whether it is running in an interactive shell, it is simply a matter of finding whether the *prompt* variable, $PS1 is set. (If the user is being prompted for input, then the script needs to display a prompt.)

```
if [ -z $PS1 ] # no prompt?
### if [ -v PS1 ]   # On Bash 4.2+ ...
then
  # non-interactive
  ...
```

```
else
  # interactive
  ...
fi
```

Alternatively, the script can test for the presence of option "i" in the $- flag.

```
case $- in
*i*)    # interactive shell
;;
*)      # non-interactive shell
;;
# (Courtesy of "UNIX F.A.Q.," 1993)
```

However, John Lange describes an alternative method, using the -t *test* operator.

```
# Test for a terminal!

fd=0   # stdin

#  As we recall, the -t test option checks whether the stdin, [ -t 0 ],
#+ or stdout, [ -t 1 ], in a given script is running in a terminal.
if [ -t "$fd" ]
then
  echo interactive
else
  echo non-interactive
fi


#  But, as John points out:
#    if [ -t 0 ] works ... when you're logged in locally
#    but fails when you invoke the command remotely via ssh.
#    So for a true test you also have to test for a socket.

if [[ -t "$fd" || -p /dev/stdin ]]
then
  echo interactive
else
  echo non-interactive
fi
```

Scripts may be forced to run in interactive mode with the -i option or with a **`#!/bin/bash -i`** header. Be aware that this can cause erratic script behavior or show error messages even when no error is present.

36.2. Shell Wrappers

A *wrapper* is a shell script that embeds a system command or utility, that accepts and passes a set of parameters to that command. [125] Wrapping a script around a complex command-line simplifies invoking it. This is expecially useful with sed and awk.

A **sed** or **awk** script would normally be invoked from the command-line by a **`sed -e`** ***`'commands'`*** or **`awk`** ***`'commands'`***. Embedding such a script in a Bash script permits calling it more simply, and makes it *reusable*. This also enables combining the functionality of *sed* and *awk*, for example piping the output of a set of *sed* commands to *awk*. As a saved executable file, you can then repeatedly invoke it in its original form or modified, without the inconvenience of retyping it on the command-line.

Example 36-1. *shell wrapper*

```
#!/bin/bash

# This simple script removes blank lines from a file.
# No argument checking.
#
# You might wish to add something like:
#
# E_NOARGS=85
# if [ -z "$1" ]
# then
#  echo "Usage: `basename $0` target-file"
#  exit $E_NOARGS
# fi



sed -e /^$/d "$1"
# Same as
#    sed -e '/^$/d' filename
# invoked from the command-line.

#  The '-e' means an "editing" command follows (optional here).
#  '^' indicates the beginning of line, '$' the end.
#  This matches lines with nothing between the beginning and the end --
#+ blank lines.
#  The 'd' is the delete command.

#  Quoting the command-line arg permits
#+ whitespace and special characters in the filename.

#  Note that this script doesn't actually change the target file.
#  If you need to do that, redirect its output.

exit
```

Example 36-2. A slightly more complex *shell wrapper*

```
#!/bin/bash

#  subst.sh: a script that substitutes one pattern for
#+ another in a file,
#+ i.e., "sh subst.sh Smith Jones letter.txt".
#                     Jones replaces Smith.

ARGS=3         # Script requires 3 arguments.
E_BADARGS=85   # Wrong number of arguments passed to script.

if [ $# -ne "$ARGS" ]
then
  echo "Usage: `basename $0` old-pattern new-pattern filename"
  exit $E_BADARGS
fi

old_pattern=$1
new_pattern=$2

if [ -f "$3" ]
then
    file_name=$3
```

```
else
    echo "File \"$3\" does not exist."
    exit $E_BADARGS
fi


# -----------------------------------------------
#  Here is where the heavy work gets done.
sed -e "s/$old_pattern/$new_pattern/g" $file_name
# -----------------------------------------------

#  's' is, of course, the substitute command in sed,
#+ and /pattern/ invokes address matching.
#  The 'g,' or global flag causes substitution for EVERY
#+ occurence of $old_pattern on each line, not just the first.
#  Read the 'sed' docs for an in-depth explanation.

exit $?  # Redirect the output of this script to write to a file.
```

Example 36-3. A generic *shell wrapper* that writes to a logfile

```
#!/bin/bash
#  logging-wrapper.sh
#  Generic shell wrapper that performs an operation
#+ and logs it.

DEFAULT_LOGFILE=logfile.txt

# Set the following two variables.
OPERATION=
#         Can be a complex chain of commands,
#+        for example an awk script or a pipe . . .

LOGFILE=
if [ -z "$LOGFILE" ]
then     # If not set, default to ...
  LOGFILE="$DEFAULT_LOGFILE"
fi

#         Command-line arguments, if any, for the operation.
OPTIONS="$@"


# Log it.
echo "`date` + `whoami` + $OPERATION "$@"" >> $LOGFILE
# Now, do it.
exec $OPERATION "$@"

# It's necessary to do the logging before the operation.
# Why?
```

Example 36-4. A *shell wrapper* around an awk script

```
#!/bin/bash
# pr-ascii.sh: Prints a table of ASCII characters.

START=33   # Range of printable ASCII characters (decimal).
END=127    # Will not work for unprintable characters (> 127).

echo " Decimal   Hex     Character"   # Header.
```

```
echo " -------   ---     ---------"

for ((i=START; i<=END; i++))
do
  echo $i | awk '{printf("  %3d       %2x         %c\n", $1, $1, $1)}'
# The Bash printf builtin will not work in this context:
#     printf "%c" "$i"
done

exit 0


#  Decimal   Hex     Character
#  -------   ---     ---------
#    33       21         !
#    34       22         "
#    35       23         #
#    36       24         $
#
#    . . .
#
#   122       7a         z
#   123       7b         {
#   124       7c         |
#   125       7d         }


#  Redirect the output of this script to a file
#+ or pipe it to "more":  sh pr-asc.sh | more
```

Example 36-5. A *shell wrapper* around another awk script

```
#!/bin/bash

# Adds up a specified column (of numbers) in the target file.
# Floating-point (decimal) numbers okay, because awk can handle them.

ARGS=2
E_WRONGARGS=85

if [ $# -ne "$ARGS" ] # Check for proper number of command-line args.
then
   echo "Usage: `basename $0` filename column-number"
   exit $E_WRONGARGS
fi

filename=$1
column_number=$2

#  Passing shell variables to the awk part of the script is a bit tricky.
#  One method is to strong-quote the Bash-script variable
#+ within the awk script.
#     $'$BASH_SCRIPT_VAR'
#      ^                 ^
#  This is done in the embedded awk script below.
#  See the awk documentation for more details.

# A multi-line awk script is here invoked by
#   awk '
#   ...
#   ...
```

```
#   ...
#   '


# Begin awk script.
# -----------------------------
awk '

{ total += $'"${column_number}"'
}
END {
     print total
}

' "$filename"
# -----------------------------
# End awk script.


#   It may not be safe to pass shell variables to an embedded awk script,
#+  so Stephane Chazelas proposes the following alternative:
#   ---------------------------------------
#   awk -v column_number="$column_number" '
#   { total += $column_number
#   }
#   END {
#       print total
#   }' "$filename"
#   ---------------------------------------


exit 0
```

For those scripts needing a single do-it-all tool, a Swiss army knife, there is *Perl*. Perl combines the capabilities of sed and awk, and throws in a large subset of **C**, to boot. It is modular and contains support for everything ranging from object-oriented programming up to and including the kitchen sink. Short Perl scripts lend themselves to embedding within shell scripts, and there may be some substance to the claim that Perl can totally replace shell scripting (though the author of the *ABS Guide* remains skeptical).

Example 36-6. Perl embedded in a *Bash* script

```
#!/bin/bash

# Shell commands may precede the Perl script.
echo "This precedes the embedded Perl script within \"$0\"."
echo "==============================================================="

perl -e 'print "This line prints from an embedded Perl script.\n";'
# Like sed, Perl also uses the "-e" option.

echo "==============================================================="
echo "However, the script may also contain shell and system commands."

exit 0
```

It is even possible to combine a Bash script and Perl script within the same file. Depending on how the script is invoked, either the Bash part or the Perl part will execute.

Example 36-7. Bash and Perl scripts combined

```
#!/bin/bash
# bashandperl.sh

echo "Greetings from the Bash part of the script, $0."
# More Bash commands may follow here.

exit
# End of Bash part of the script.

# =======================================================

#!/usr/bin/perl
# This part of the script must be invoked with
#    perl -x bashandperl.sh

print "Greetings from the Perl part of the script, $0.\n";
#      Perl doesn't seem to like "echo" ...
# More Perl commands may follow here.

# End of Perl part of the script.
```

```
bash$ bash bashandperl.sh
Greetings from the Bash part of the script.


bash$ perl -x bashandperl.sh
Greetings from the Perl part of the script.
```

It is, of course, possible to embed even more exotic scripting languages within shell wrappers. *Python*, for example ...

Example 36-8. Python embedded in a *Bash* script

```
#!/bin/bash
# ex56py.sh

# Shell commands may precede the Python script.
echo "This precedes the embedded Python script within \"$0.\""
echo "============================================================"

python -c 'print "This line prints from an embedded Python script.\n";'
# Unlike sed and perl, Python uses the "-c" option.
python -c 'k = raw_input( "Hit a key to exit to outer script. " )'

echo "============================================================"
echo "However, the script may also contain shell and system commands."

exit 0
```

Wrapping a script around *mplayer* and the Google's translation server, you can create something that talks back to you.

Example 36-9. A script that speaks

```
#!/bin/bash
#   Courtesy of:
#   http://elinux.org/RPi_Text_to_Speech_(Speech_Synthesis)

#  You must be on-line for this script to work,
#+ so you can access the Google translation server.
#  Of course, mplayer must be present on your computer.

speak()
  {
  local IFS=+
  # Invoke mplayer, then connect to Google translation server.
  /usr/bin/mplayer -ao alsa -really-quiet -noconsolecontrols \
 "http://translate.google.com/translate_tts?tl=en&q="$*""
  # Google translates, but can also speak.
  }

LINES=4

spk=$(tail -$LINES $0) # Tail end of same script!
speak "$spk"
exit
# Browns. Nice talking to you.
```

One interesting example of a complex shell wrapper is Martin Matusiak's *undvd* script, which provides an easy-to-use command-line interface to the complex mencoder utility. Another example is Itzchak Rehberg's Ext3Undel, a set of scripts to recover deleted file on an *ext3* filesystem.

36.3. Tests and Comparisons: Alternatives

For tests, the [[ ]] construct may be more appropriate than **[ ]**. Likewise, arithmetic comparisons might benefit from the (( )) construct.

```
a=8

# All of the comparisons below are equivalent.
test "$a" -lt 16 && echo "yes, $a < 16"         # "and list"
/bin/test "$a" -lt 16 && echo "yes, $a < 16"
[ "$a" -lt 16 ] && echo "yes, $a < 16"
[[ $a -lt 16 ]] && echo "yes, $a < 16"          # Quoting variables within
(( a < 16 )) && echo "yes, $a < 16"             # [[ ]] and (( )) not necessary.

city="New York"
# Again, all of the comparisons below are equivalent.
test "$city" \< Paris && echo "Yes, Paris is greater than $city"
                                  # Greater ASCII order.
/bin/test "$city" \< Paris && echo "Yes, Paris is greater than $city"
[ "$city" \< Paris ] && echo "Yes, Paris is greater than $city"
[[ $city < Paris ]] && echo "Yes, Paris is greater than $city"
                                  # Need not quote $city.

# Thank you, S.C.
```

36.4. Recursion: a script calling itself

Can a script recursively call itself? Indeed.

Example 36-10. A (useless) script that recursively calls itself

```
#!/bin/bash
# recurse.sh

#  Can a script recursively call itself?
#  Yes, but is this of any practical use?
#  (See the following.)

RANGE=10
MAXVAL=9

i=$RANDOM
let "i %= $RANGE"  # Generate a random number between 0 and $RANGE - 1.

if [ "$i" -lt "$MAXVAL" ]
then
  echo "i = $i"
  ./$0             #  Script recursively spawns a new instance of itself.
fi                 #  Each child script does the same, until
                   #+ a generated $i equals $MAXVAL.

#  Using a "while" loop instead of an "if/then" test causes problems.
#  Explain why.

exit 0

# Note:
# ----
# This script must have execute permission for it to work properly.
# This is the case even if it is invoked by an "sh" command.
# Explain why.
```

Example 36-11. A (useful) script that recursively calls itself

```
#!/bin/bash
# pb.sh: phone book

# Written by Rick Boivie, and used with permission.
# Modifications by ABS Guide author.

MINARGS=1     #  Script needs at least one argument.
DATAFILE=./phonebook
              #  A data file in current working directory
              #+ named "phonebook" must exist.
PROGNAME=$0
E_NOARGS=70   #  No arguments error.

if [ $# -lt $MINARGS ]; then
      echo "Usage: "$PROGNAME" data-to-look-up"
      exit $E_NOARGS
fi


if [ $# -eq $MINARGS ]; then
      grep $1 "$DATAFILE"
      # 'grep' prints an error message if $DATAFILE not present.
else
      ( shift; "$PROGNAME" $* ) | grep $1
      # Script recursively calls itself.
fi
```

```
exit 0          #  Script exits here.
                #  Therefore, it's o.k. to put
                #+ non-hashmarked comments and data after this point.

# -------------------------------------------------------------------------
Sample "phonebook" datafile:

John Doe        1555 Main St., Baltimore, MD 21228          (410) 222-3333
Mary Moe        9899 Jones Blvd., Warren, NH 03787          (603) 898-3232
Richard Roe     856 E. 7th St., New York, NY 10009          (212) 333-4567
Sam Roe         956 E. 8th St., New York, NY 10009          (212) 444-5678
Zoe Zenobia     4481 N. Baker St., San Francisco, SF 94338  (415) 501-1631
# -------------------------------------------------------------------------

$bash pb.sh Roe
Richard Roe     856 E. 7th St., New York, NY 10009          (212) 333-4567
Sam Roe         956 E. 8th St., New York, NY 10009          (212) 444-5678

$bash pb.sh Roe Sam
Sam Roe         956 E. 8th St., New York, NY 10009          (212) 444-5678

#  When more than one argument is passed to this script,
#+ it prints *only* the line(s) containing all the arguments.
```

Example 36-12. Another (useful) script that recursively calls itself

```
#!/bin/bash
# usrmnt.sh, written by Anthony Richardson
# Used in ABS Guide with permission.

# usage:       usrmnt.sh
# description: mount device, invoking user must be listed in the
#              MNTUSERS group in the /etc/sudoers file.

# ----------------------------------------------------------
#  This is a usermount script that reruns itself using sudo.
#  A user with the proper permissions only has to type

#   usermount /dev/fd0 /mnt/floppy

# instead of

#   sudo usermount /dev/fd0 /mnt/floppy

#  I use this same technique for all of my
#+ sudo scripts, because I find it convenient.
# ----------------------------------------------------------

#  If SUDO_COMMAND variable is not set we are not being run through
#+ sudo, so rerun ourselves. Pass the user's real and group id . . .

if [ -z "$SUDO_COMMAND" ]
then
   mntusr=$(id -u) grpusr=$(id -g) sudo $0 $*
   exit 0
fi

# We will only get here if we are being run by sudo.
/bin/mount $* -o uid=$mntusr,gid=$grpusr
```

```
exit 0

# Additional notes (from the author of this script):
# -------------------------------------------------

# 1) Linux allows the "users" option in the /etc/fstab
#    file so that any user can mount removable media.
#    But, on a server, I like to allow only a few
#    individuals access to removable media.
#    I find using sudo gives me more control.

# 2) I also find sudo to be more convenient than
#    accomplishing this task through groups.

# 3) This method gives anyone with proper permissions
#    root access to the mount command, so be careful
#    about who you allow access.
#    You can get finer control over which access can be mounted
#    by using this same technique in separate mntfloppy, mntcdrom,
#    and mntsamba scripts.
```

Too many levels of recursion can exhaust the script's stack space, causing a segfault.

36.5. "Colorizing" Scripts

The ANSI [126] escape sequences set screen attributes, such as bold text, and color of foreground and background. DOS batch files commonly used ANSI escape codes for *color* output, and so can Bash scripts.

Example 36-13. A "colorized" address database

```
#!/bin/bash
# ex30a.sh: "Colorized" version of ex30.sh.
#             Crude address database


clear                                   # Clear the screen.

echo -n "          "
echo -e '\E[37;44m'"\033[1mContact List\033[0m"
                                        # White on blue background
echo; echo
echo -e "\033[1mChoose one of the following persons:\033[0m"
                                        # Bold
tput sgr0                               # Reset attributes.
echo "(Enter only the first letter of name.)"
echo
echo -en '\E[47;34m'"\033[1mE\033[0m"   # Blue
tput sgr0                               # Reset colors to "normal."
echo "vans, Roland"                     # "[E]vans, Roland"
echo -en '\E[47;35m'"\033[1mJ\033[0m"   # Magenta
tput sgr0
echo "ambalaya, Mildred"
echo -en '\E[47;32m'"\033[1mS\033[0m"   # Green
tput sgr0
echo "mith, Julie"
echo -en '\E[47;31m'"\033[1mZ\033[0m"   # Red
```

```
tput sgr0
echo "ane, Morris"
echo

read person

case "$person" in
# Note variable is quoted.

  "E" | "e" )
  # Accept upper or lowercase input.
  echo
  echo "Roland Evans"
  echo "4321 Flash Dr."
  echo "Hardscrabble, CO 80753"
  echo "(303) 734-9874"
  echo "(303) 734-9892 fax"
  echo "revans@zzy.net"
  echo "Business partner & old friend"
  ;;

  "J" | "j" )
  echo
  echo "Mildred Jambalaya"
  echo "249 E. 7th St., Apt. 19"
  echo "New York, NY 10009"
  echo "(212) 533-2814"
  echo "(212) 533-9972 fax"
  echo "milliej@loisaida.com"
  echo "Girlfriend"
  echo "Birthday: Feb. 11"
  ;;

# Add info for Smith & Zane later.

          * )
   # Default option.
   # Empty input (hitting RETURN) fits here, too.
   echo
   echo "Not yet in database."
  ;;

esac

tput sgr0                                # Reset colors to "normal."

echo

exit 0
```

Example 36-14. Drawing a box

```
#!/bin/bash
# Draw-box.sh: Drawing a box using ASCII characters.

# Script by Stefano Palmeri, with minor editing by document author.
# Minor edits suggested by Jim Angstadt.
# Used in the ABS Guide with permission.


######################################################################
```

```
###  draw_box function doc  ###

#  The "draw_box" function lets the user
#+ draw a box in a terminal.
#
#  Usage: draw_box ROW COLUMN HEIGHT WIDTH [COLOR]
#  ROW and COLUMN represent the position
#+ of the upper left angle of the box you're going to draw.
#  ROW and COLUMN must be greater than 0
#+ and less than current terminal dimension.
#  HEIGHT is the number of rows of the box, and must be > 0.
#  HEIGHT + ROW must be <= than current terminal height.
#  WIDTH is the number of columns of the box and must be > 0.
#  WIDTH + COLUMN must be <= than current terminal width.
#
# E.g.: If your terminal dimension is 20x80,
#  draw_box 2 3 10 45 is good
#  draw_box 2 3 19 45 has bad HEIGHT value (19+2 > 20)
#  draw_box 2 3 18 78 has bad WIDTH value (78+3 > 80)
#
#  COLOR is the color of the box frame.
#  This is the 5th argument and is optional.
#  0=black 1=red 2=green 3=tan 4=blue 5=purple 6=cyan 7=white.
#  If you pass the function bad arguments,
#+ it will just exit with code 65,
#+ and no messages will be printed on stderr.
#
#  Clear the terminal before you start to draw a box.
#  The clear command is not contained within the function.
#  This allows the user to draw multiple boxes, even overlapping ones.

###  end of draw_box function doc  ###
######################################################################

draw_box(){

#=============#
HORZ="-"
VERT="|"
CORNER_CHAR="+"

MINARGS=4
E_BADARGS=65
#=============#


if [ $# -lt "$MINARGS" ]; then          # If args are less than 4, exit.
    exit $E_BADARGS
fi

# Looking for non digit chars in arguments.
# Probably it could be done better (exercise for the reader?).
if echo $@ | tr -d [:blank:] | tr -d [:digit:] | grep . &> /dev/null; then
   exit $E_BADARGS
fi

BOX_HEIGHT=`expr $3 - 1`   #  -1 correction needed because angle char "+"
BOX_WIDTH=`expr $4 - 1`    #+ is a part of both box height and width.
T_ROWS=`tput lines`        #  Define current terminal dimension
T_COLS=`tput cols`         #+ in rows and columns.

if [ $1 -lt 1 ] || [ $1 -gt $T_ROWS ]; then    #  Start checking if arguments
```

```
   exit $E_BADARGS                              #+ are correct.
fi
if [ $2 -lt 1 ] || [ $2 -gt $T_COLS ]; then
   exit $E_BADARGS
fi
if [ `expr $1 + $BOX_HEIGHT + 1` -gt $T_ROWS ]; then
   exit $E_BADARGS
fi
if [ `expr $2 + $BOX_WIDTH + 1` -gt $T_COLS ]; then
   exit $E_BADARGS
fi
if [ $3 -lt 1 ] || [ $4 -lt 1 ]; then
   exit $E_BADARGS
fi                                 # End checking arguments.

plot_char(){                       # Function within a function.
   echo -e "\E[${1};${2}H"$3
}

echo -ne "\E[3${5}m"               # Set box frame color, if defined.

# start drawing the box

count=1                                         #  Draw vertical lines using
for (( r=$1; count<=$BOX_HEIGHT; r++)); do      #+ plot_char function.
  plot_char $r $2 $VERT
  let count=count+1
done

count=1
c=`expr $2 + $BOX_WIDTH`
for (( r=$1; count<=$BOX_HEIGHT; r++)); do
  plot_char $r $c $VERT
  let count=count+1
done

count=1                                        #  Draw horizontal lines using
for (( c=$2; count<=$BOX_WIDTH; c++)); do      #+ plot_char function.
  plot_char $1 $c $HORZ
  let count=count+1
done

count=1
r=`expr $1 + $BOX_HEIGHT`
for (( c=$2; count<=$BOX_WIDTH; c++)); do
  plot_char $r $c $HORZ
  let count=count+1
done

plot_char $1 $2 $CORNER_CHAR                   # Draw box angles.
plot_char $1 `expr $2 + $BOX_WIDTH` $CORNER_CHAR
plot_char `expr $1 + $BOX_HEIGHT` $2 $CORNER_CHAR
plot_char `expr $1 + $BOX_HEIGHT` `expr $2 + $BOX_WIDTH` $CORNER_CHAR

echo -ne "\E[0m"             #  Restore old colors.

P_ROWS=`expr $T_ROWS - 1`    #  Put the prompt at bottom of the terminal.

echo -e "\E[${P_ROWS};1H"
}
```

```
# Now, let's try drawing a box.
clear                       # Clear the terminal.
R=2      # Row
C=3      # Column
H=10     # Height
W=45     # Width
col=1    # Color (red)
draw_box $R $C $H $W $col   # Draw the box.

exit 0

# Exercise:
# --------
# Add the option of printing text within the drawn box.
```

The simplest, and perhaps most useful ANSI escape sequence is bold text, **\033[1m ... \033[0m**. The \033 represents an escape, the "[1" turns on the bold attribute, while the "[0" switches it off. The "m" terminates each term of the escape sequence.

```
bash$ echo -e "\033[1mThis is bold text.\033[0m"
```

A similar escape sequence switches on the underline attribute (on an *rxvt* and an *aterm*).

```
bash$ echo -e "\033[4mThis is underlined text.\033[0m"
```

With an **echo**, the `-e` option enables the escape sequences.

Other escape sequences change the text and/or background color.

```
bash$ echo -e '\E[34;47mThis prints in blue.'; tput sgr0


bash$ echo -e '\E[33;44m'"yellow text on blue background"; tput sgr0


bash$ echo -e '\E[1;33;44m'"BOLD yellow text on blue background"; tput sgr0
```

It's usually advisable to set the *bold* attribute for light-colored foreground text.

The **tput sgr0** restores the terminal settings to normal. Omitting this lets all subsequent output from that particular terminal remain blue.

Since **tput sgr0** fails to restore terminal settings under certain circumstances, **echo -ne \E[0m** may be a better choice.

Use the following template for writing colored text on a colored background.

`echo -e '\E[COLOR1;COLOR2mSome text goes here.'`

The "\E[" begins the escape sequence. The semicolon-separated numbers "COLOR1" and "COLOR2" specify a foreground and a background color, according to the table below. (The order of the numbers does not matter, since the foreground and background numbers fall in non-overlapping ranges.) The "m" terminates the escape sequence, and the text begins immediately after that.

Note also that single quotes enclose the remainder of the command sequence following the **echo -e**.

The numbers in the following table work for an *rxvt* terminal. Results may vary for other terminal emulators.

Table 36-1. Numbers representing colors in Escape Sequences

Color	Foreground	Background
`black`	30	40
`red`	31	41
`green`	32	42
`yellow`	33	43
`blue`	34	44
`magenta`	35	45
`cyan`	36	46
`white`	37	47

Example 36-15. Echoing colored text

```
#!/bin/bash
# color-echo.sh: Echoing text messages in color.

# Modify this script for your own purposes.
# It's easier than hand-coding color.

black='\E[30;47m'
red='\E[31;47m'
green='\E[32;47m'
yellow='\E[33;47m'
blue='\E[34;47m'
magenta='\E[35;47m'
cyan='\E[36;47m'
white='\E[37;47m'


alias Reset="tput sgr0"      #  Reset text attributes to normal
                             #+ without clearing screen.


cecho ()                     # Color-echo.
                             # Argument $1 = message
                             # Argument $2 = color
{
local default_msg="No message passed."
                             # Doesn't really need to be a local variable.

message=${1:-$default_msg}   # Defaults to default message.
color=${2:-$black}           # Defaults to black, if not specified.

  echo -e "$color"
  echo "$message"
  Reset                      # Reset to normal.

  return
}
```

```
# Now, let's try it out.
# ----------------------------------------------------
cecho "Feeling blue..." $blue
cecho "Magenta looks more like purple." $magenta
cecho "Green with envy." $green
cecho "Seeing red?" $red
cecho "Cyan, more familiarly known as aqua." $cyan
cecho "No color passed (defaults to black)."
       # Missing $color argument.
cecho "\"Empty\" color passed (defaults to black)." ""
       # Empty $color argument.
cecho
       # Missing $message and $color arguments.
cecho "" ""
       # Empty $message and $color arguments.
# ----------------------------------------------------

echo

exit 0

# Exercises:
# ---------
# 1) Add the "bold" attribute to the 'cecho ()' function.
# 2) Add options for colored backgrounds.
```

Example 36-16. A "horserace" game

```
#!/bin/bash
# horserace.sh: Very simple horserace simulation.
# Author: Stefano Palmeri
# Used with permission.

################################################################
#  Goals of the script:
#  playing with escape sequences and terminal colors.
#
#  Exercise:
#  Edit the script to make it run less randomly,
#+ set up a fake betting shop . . .
#  Um . . . um . . . it's starting to remind me of a movie . . .
#
#  The script gives each horse a random handicap.
#  The odds are calculated upon horse handicap
#+ and are expressed in European(?) style.
#  E.g., odds=3.75 means that if you bet $1 and win,
#+ you receive $3.75.
#
#  The script has been tested with a GNU/Linux OS,
#+ using xterm and rxvt, and konsole.
#  On a machine with an AMD 900 MHz processor,
#+ the average race time is 75 seconds.
#  On faster computers the race time would be lower.
#  So, if you want more suspense, reset the USLEEP_ARG variable.
#
#  Script by Stefano Palmeri.
################################################################
```

```
E_RUNERR=65

# Check if md5sum and bc are installed.
if ! which bc &> /dev/null; then
   echo bc is not installed.
   echo "Can\'t run . . . "
   exit $E_RUNERR
fi
if ! which md5sum &> /dev/null; then
   echo md5sum is not installed.
   echo "Can\'t run . . . "
   exit $E_RUNERR
fi

#  Set the following variable to slow down script execution.
#  It will be passed as the argument for usleep (man usleep)
#+ and is expressed in microseconds (500000 = half a second).
USLEEP_ARG=0

#  Clean up the temp directory, restore terminal cursor and
#+ terminal colors -- if script interrupted by Ctl-C.
trap 'echo -en "\E[?25h"; echo -en "\E[0m"; stty echo;\
tput cup 20 0; rm -fr  $HORSE_RACE_TMP_DIR'  TERM EXIT
#  See the chapter on debugging for an explanation of 'trap.'

# Set a unique (paranoid) name for the temp directory the script needs.
HORSE_RACE_TMP_DIR=$HOME/.horserace-`date +%s`-`head -c10 /dev/urandom \
| md5sum | head -c30`

# Create the temp directory and move right in.
mkdir $HORSE_RACE_TMP_DIR
cd $HORSE_RACE_TMP_DIR


#  This function moves the cursor to line $1 column $2 and then prints $3.
#  E.g.: "move_and_echo 5 10 linux" is equivalent to
#+ "tput cup 4 9; echo linux", but with one command instead of two.
#  Note: "tput cup" defines 0 0 the upper left angle of the terminal,
#+ echo defines 1 1 the upper left angle of the terminal.
move_and_echo() {
          echo -ne "\E[${1};${2}H""$3"
}

# Function to generate a pseudo-random number between 1 and 9.
random_1_9 ()
{
    head -c10 /dev/urandom | md5sum | tr -d [a-z] | tr -d 0 | cut -c1
}

#  Two functions that simulate "movement," when drawing the horses.
draw_horse_one() {
               echo -n " "//$MOVE_HORSE//
}
draw_horse_two(){
              echo -n " "\\\\$MOVE_HORSE\\\\
}


# Define current terminal dimension.
N_COLS=`tput cols`
N_LINES=`tput lines`
```

```
# Need at least a 20-LINES X 80-COLUMNS terminal. Check it.
if [ $N_COLS -lt 80 ] || [ $N_LINES -lt 20 ]; then
   echo "`basename $0` needs a 80-cols X 20-lines terminal."
   echo "Your terminal is ${N_COLS}-cols X ${N_LINES}-lines."
   exit $E_RUNERR
fi


# Start drawing the race field.

# Need a string of 80 chars. See below.
BLANK80=`seq -s "" 100 | head -c80`

clear

# Set foreground and background colors to white.
echo -ne '\E[37;47m'

# Move the cursor on the upper left angle of the terminal.
tput cup 0 0

# Draw six white lines.
for n in `seq 5`; do
      echo $BLANK80   # Use the 80 chars string to colorize the terminal.
done

# Sets foreground color to black.
echo -ne '\E[30m'

move_and_echo 3 1 "START  1"
move_and_echo 3 75 FINISH
move_and_echo 1 5 "|"
move_and_echo 1 80 "|"
move_and_echo 2 5 "|"
move_and_echo 2 80 "|"
move_and_echo 4 5 "|  2"
move_and_echo 4 80 "|"
move_and_echo 5 5 "V  3"
move_and_echo 5 80 "V"

# Set foreground color to red.
echo -ne '\E[31m'

# Some ASCII art.
move_and_echo 1 8 "..@@@..@@@@@...@@@@@.@...@..@@@@..."
move_and_echo 2 8 ".@...@...@.......@...@...@.@......."
move_and_echo 3 8 ".@@@@@...@.......@...@@@@@.@@@@...."
move_and_echo 4 8 ".@...@...@.......@...@...@.@......."
move_and_echo 5 8 ".@...@...@.......@...@...@..@@@@..."
move_and_echo 1 43 "@@@@...@@@...@@@@..@@@@..@@@@."
move_and_echo 2 43 "@...@.@...@.@.....@.....@....."
move_and_echo 3 43 "@@@@..@@@@@.@.....@@@@...@@@.."
move_and_echo 4 43 "@..@..@...@.@.....@.........@."
move_and_echo 5 43 "@...@.@...@..@@@@..@@@@.@@@@.."


# Set foreground and background colors to green.
echo -ne '\E[32;42m'

# Draw  eleven green lines.
tput cup 5 0
for n in `seq 11`; do
```

```
      echo $BLANK80
done

# Set foreground color to black.
echo -ne '\E[30m'
tput cup 5 0

# Draw the fences.
echo "++++++++++++++++++++++++++++++++++++++\
++++++++++++++++++++++++++++++++++++++++++"

tput cup 15 0
echo "++++++++++++++++++++++++++++++++++++++\
++++++++++++++++++++++++++++++++++++++++++"

# Set foreground and background colors to white.
echo -ne '\E[37;47m'

# Draw three white lines.
for n in `seq 3`; do
      echo $BLANK80
done

# Set foreground color to black.
echo -ne '\E[30m'

# Create 9 files to stores handicaps.
for n in `seq 10 7 68`; do
      touch $n
done

# Set the first type of "horse" the script will draw.
HORSE_TYPE=2

#  Create position-file and odds-file for every "horse".
#+ In these files, store the current position of the horse,
#+ the type and the odds.
for HN in `seq 9`; do
      touch horse_${HN}_position
      touch odds_${HN}
      echo \-1 > horse_${HN}_position
      echo $HORSE_TYPE >>  horse_${HN}_position
      # Define a random handicap for horse.
       HANDICAP=`random_1_9`
      # Check if the random_1_9 function returned a good value.
      while ! echo $HANDICAP | grep [1-9] &> /dev/null; do
                HANDICAP=`random_1_9`
      done
      # Define last handicap position for horse.
      LHP=`expr $HANDICAP \* 7 + 3`
      for FILE in `seq 10 7 $LHP`; do
            echo $HN >> $FILE
      done

      # Calculate odds.
      case $HANDICAP in
              1) ODDS=`echo $HANDICAP \* 0.25 + 1.25 | bc`
                                 echo $ODDS > odds_${HN}
              ;;
              2 | 3) ODDS=`echo $HANDICAP \* 0.40 + 1.25 | bc`
                                       echo $ODDS > odds_${HN}
              ;;
```

```
              4 | 5 | 6) ODDS=`echo $HANDICAP \* 0.55 + 1.25 | bc`
                                           echo $ODDS > odds_${HN}
              ;;
              7 | 8) ODDS=`echo $HANDICAP \* 0.75 + 1.25 | bc`
                                       echo $ODDS > odds_${HN}
              ;;
              9) ODDS=`echo $HANDICAP \* 0.90 + 1.25 | bc`
                                  echo $ODDS > odds_${HN}
      esac


done


# Print odds.
print_odds() {
tput cup 6 0
echo -ne '\E[30;42m'
for HN in `seq 9`; do
      echo "#$HN odds->" `cat odds_${HN}`
done
}

# Draw the horses at starting line.
draw_horses() {
tput cup 6 0
echo -ne '\E[30;42m'
for HN in `seq 9`; do
      echo /\\$HN/\\"                               "
done
}

print_odds

echo -ne '\E[47m'
# Wait for a enter key press to start the race.
# The escape sequence '\E[?25l' disables the cursor.
tput cup 17 0
echo -e '\E[?25l'Press [enter] key to start the race...
read -s

#  Disable normal echoing in the terminal.
#  This avoids key presses that might "contaminate" the screen
#+ during the race.
stty -echo

# ----------------------------------------------------------
# Start the race.

draw_horses
echo -ne '\E[37;47m'
move_and_echo 18 1 $BLANK80
echo -ne '\E[30m'
move_and_echo 18 1 Starting...
sleep 1

# Set the column of the finish line.
WINNING_POS=74

# Define the time the race started.
START_TIME=`date +%s`
```

```
# COL variable needed by following "while" construct.
COL=0

while [ $COL -lt $WINNING_POS ]; do

          MOVE_HORSE=0

          # Check if the random_1_9 function has returned a good value.
          while ! echo $MOVE_HORSE | grep [1-9] &> /dev/null; do
                MOVE_HORSE=`random_1_9`
          done

          # Define old type and position of the "randomized horse".
          HORSE_TYPE=`cat  horse_${MOVE_HORSE}_position | tail -n 1`
          COL=$(expr `cat  horse_${MOVE_HORSE}_position | head -n 1`)

          ADD_POS=1
          # Check if the current position is an handicap position.
          if seq 10 7 68 | grep -w $COL &> /dev/null; then
                if grep -w $MOVE_HORSE $COL &> /dev/null; then
                      ADD_POS=0
                      grep -v -w  $MOVE_HORSE $COL > ${COL}_new
                      rm -f $COL
                      mv -f ${COL}_new $COL
                      else ADD_POS=1
                fi
          else ADD_POS=1
          fi
          COL=`expr $COL + $ADD_POS`
          echo $COL >  horse_${MOVE_HORSE}_position  # Store new position.

         # Choose the type of horse to draw.
          case $HORSE_TYPE in
                1) HORSE_TYPE=2; DRAW_HORSE=draw_horse_two
                ;;
                2) HORSE_TYPE=1; DRAW_HORSE=draw_horse_one
          esac
          echo $HORSE_TYPE >>  horse_${MOVE_HORSE}_position
          # Store current type.

          # Set foreground color to black and background to green.
          echo -ne '\E[30;42m'

          # Move the cursor to new horse position.
          tput cup `expr $MOVE_HORSE + 5` \
          `cat  horse_${MOVE_HORSE}_position | head -n 1`

          # Draw the horse.
          $DRAW_HORSE
           usleep $USLEEP_ARG

           # When all horses have gone beyond field line 15, reprint odds.
           touch fieldline15
           if [ $COL = 15 ]; then
             echo $MOVE_HORSE >> fieldline15
           fi
           if [ `wc -l fieldline15 | cut -f1 -d " "` = 9 ]; then
               print_odds
               : > fieldline15
           fi

          # Define the leading horse.
```

```
          HIGHEST_POS=`cat *position | sort -n | tail -1`

          # Set background color to white.
          echo -ne '\E[47m'
          tput cup 17 0
          echo -n Current leader: `grep -w $HIGHEST_POS *position | cut -c7`\
          "                              "

done

# Define the time the race finished.
FINISH_TIME=`date +%s`

# Set background color to green and enable blinking text.
echo -ne '\E[30;42m'
echo -en '\E[5m'

# Make the winning horse blink.
tput cup `expr $MOVE_HORSE + 5` \
`cat  horse_${MOVE_HORSE}_position | head -n 1`
$DRAW_HORSE

# Disable blinking text.
echo -en '\E[25m'

# Set foreground and background color to white.
echo -ne '\E[37;47m'
move_and_echo 18 1 $BLANK80

# Set foreground color to black.
echo -ne '\E[30m'

# Make winner blink.
tput cup 17 0
echo -e "\E[5mWINNER: $MOVE_HORSE\E[25m""  Odds: `cat odds_${MOVE_HORSE}`"\
"  Race time: `expr $FINISH_TIME - $START_TIME` secs"

# Restore cursor and old colors.
echo -en "\E[?25h"
echo -en "\E[0m"

# Restore echoing.
stty echo

# Remove race temp directory.
rm -rf $HORSE_RACE_TMP_DIR

tput cup 19 0

exit 0
```

See also Example A-21, Example A-44, Example A-52, and Example A-40.

There is, however, a major problem with all this. *ANSI escape sequences are emphatically non-portable*. What works fine on some terminal emulators (or the console) may work differently, or not at all, on others. A "colorized" script that looks stunning on the script author's machine may produce unreadable output on someone else's. This somewhat compromises the usefulness of colorizing scripts, and possibly relegates this technique to the status of a gimmick. Colorized scripts are probably inappropriate in a commercial setting, i.e., your supervisor might disapprove.

Alister's ansi-color utility (based on Moshe Jacobson's color utility considerably simplifies using ANSI escape sequences. It substitutes a clean and logical syntax for the clumsy constructs just discussed.

Henry/teikedvl has likewise created a utility (http://scriptechocolor.sourceforge.net/) to simplify creation of colorized scripts.

36.6. Optimizations

Most shell scripts are quick 'n dirty solutions to non-complex problems. As such, optimizing them for speed is not much of an issue. Consider the case, though, where a script carries out an important task, does it well, but runs too slowly. Rewriting it in a compiled language may not be a palatable option. The simplest fix would be to rewrite the parts of the script that slow it down. Is it possible to apply principles of code optimization even to a lowly shell script?

Check the loops in the script. Time consumed by repetitive operations adds up quickly. If at all possible, remove time-consuming operations from within loops.

Use builtin commands in preference to system commands. Builtins execute faster and usually do not launch a subshell when invoked.

Avoid unnecessary commands, particularly in a pipe.

```
cat "$file" | grep "$word"

grep "$word" "$file"

#  The above command-lines have an identical effect,
#+ but the second runs faster since it launches one fewer subprocess.
```

The cat command seems especially prone to overuse in scripts.

Disabling certain Bash options can speed up scripts.

As Erik Brandsberg points out:

If you don't need Unicode support, you can get potentially a 2x or more improvement in speed by simply setting the **LC_ALL** variable.

```
   export LC_ALL=C

   [specifies the locale as ANSI C,
   thereby disabling Unicode support]

[In an example script ...]

Without [Unicode support]:
erik@erik-desktop:~/capture$ time ./cap-ngrep.sh
live2.pcap > out.txt

  real        0m20.483s
  user        1m34.470s
  sys         0m12.869s
```

```
With [Unicode support]:
erik@erik-desktop:~/capture$ time ./cap-ngrep.sh
live2.pcap > out.txt

  real        0m50.232s
  user        3m51.118s
  sys         0m11.221s

A large part of the overhead that is optimized is, I believe,
regex match using [[ string =~ REGEX ]],
but it may help with other portions of the code as well.
I hadn't [seen it] mentioned that this optimization helped
with Bash, but I had seen it helped with "grep,"
so why not try?
```

Certain operators, notably expr, are very inefficient and might be replaced by double parentheses arithmetic expansion. See Example A-59.

```
Math tests

math via $(( ))
real          0m0.294s
user          0m0.288s
sys           0m0.008s

math via expr:
real          1m17.879s   # Much slower!
user          0m3.600s
sys           0m8.765s

math via let:
real          0m0.364s
user          0m0.372s
sys           0m0.000s
```

Condition testing constructs in scripts deserve close scrutiny. Substitute case for if-then constructs and combine tests when possible, to minimize script execution time. Again, refer to Example A-59.

```
Test using "case" construct:
real          0m0.329s
user          0m0.320s
sys           0m0.000s


Test with if [], no quotes:
real          0m0.438s
user          0m0.432s
sys           0m0.008s


Test with if [], quotes:
real          0m0.476s
user          0m0.452s
sys           0m0.024s


Test with if [], using -eq:
real          0m0.457s
user          0m0.456s
```

```
sys             0m0.000s
```

Erik Brandsberg recommends using associative arrays in preference to conventional numeric-indexed arrays in most cases. When overwriting values in a numeric array, there is a significant performance penalty vs. associative arrays. Running a test script confirms this. See Example A-60.

```
Assignment tests

Assigning a simple variable
real            0m0.418s
user            0m0.416s
sys             0m0.004s

Assigning a numeric index array entry
real            0m0.582s
user            0m0.564s
sys             0m0.016s

Overwriting a numeric index array entry
real            0m21.931s
user            0m21.913s
sys             0m0.016s

Linear reading of numeric index array
real            0m0.422s
user            0m0.416s
sys             0m0.004s

Assigning an associative array entry
real            0m1.800s
user            0m1.796s
sys             0m0.004s

Overwriting an associative array entry
real            0m1.798s
user            0m1.784s
sys             0m0.012s

Linear reading an associative array entry
real            0m0.420s
user            0m0.420s
sys             0m0.000s

Assigning a random number to a simple variable
real            0m0.402s
user            0m0.388s
sys             0m0.016s

Assigning a sparse numeric index array entry randomly into 64k cells
real            0m12.678s
user            0m12.649s
sys             0m0.028s

Reading sparse numeric index array entry
real            0m0.087s
user            0m0.084s
sys             0m0.000s

Assigning a sparse associative array entry randomly into 64k cells
real            0m0.698s
```

```
user          0m0.696s
sys           0m0.004s

Reading sparse associative index array entry
real          0m0.083s
user          0m0.084s
sys           0m0.000s
```

Use the time and times tools to profile computation-intensive commands. Consider rewriting time-critical code sections in C, or even in assembler.

Try to minimize file I/O. Bash is not particularly efficient at handling files, so consider using more appropriate tools for this within the script, such as awk or Perl.

Write your scripts in a modular and coherent form, [127] so they can be reorganized and tightened up as necessary. Some of the optimization techniques applicable to high-level languages may work for scripts, but others, such as *loop unrolling*, are mostly irrelevant. Above all, use common sense.

For an excellent demonstration of how optimization can dramatically reduce the execution time of a script, see Example 16-47.

36.7. Assorted Tips

36.7.1. Ideas for more powerful scripts

- You have a problem that you want to solve by writing a Bash script. Unfortunately, you don't know quite where to start. One method is to plunge right in and code those parts of the script that come easily, and write the hard parts as *pseudo-code*.

```
#!/bin/bash

ARGCOUNT=1                     # Need name as argument.
E_WRONGARGS=65

if [ number-of-arguments is-not-equal-to "$ARGCOUNT" ]
#    ^^^^^^^^^^^^^^^^^^^ ^^^^^^^^^^^^^^^
#  Can't figure out how to code this . . .
#+ . . . so write it in pseudo-code.

then
  echo "Usage: name-of-script name"
  #            ^^^^^^^^^^^^^^     More pseudo-code.
  exit $E_WRONGARGS
fi

. . .

exit 0


# Later on, substitute working code for the pseudo-code.

# Line 6 becomes:
if [ $# -ne "$ARGCOUNT" ]
```

```
# Line 12 becomes:
  echo "Usage: `basename $0` name"
```

For an example of using pseudo-code, see the Square Root exercise.

- To keep a record of which user scripts have run during a particular session or over a number of sessions, add the following lines to each script you want to keep track of. This will keep a continuing file record of the script names and invocation times.

```
# Append (>>) following to end of each script tracked.

whoami>> $SAVE_FILE    # User invoking the script.
echo $0>> $SAVE_FILE   # Script name.
date>> $SAVE_FILE      # Date and time.
echo>> $SAVE_FILE      # Blank line as separator.

#  Of course, SAVE_FILE defined and exported as environmental variable in ~/.bashrc
#+ (something like ~/.scripts-run)
```

- The >> operator *appends* lines to a file. What if you wish to *prepend* a line to an existing file, that is, to paste it in at the beginning?

```
file=data.txt
title="***This is the title line of data text file***"

echo $title | cat - $file >$file.new
# "cat -" concatenates stdout to $file.
#  End result is
#+ to write a new file with $title appended at *beginning*.
```

This is a simplified variant of the Example 19-13 script given earlier. And, of course, sed can also do this.

- A shell script may act as an embedded command inside another shell script, a *Tcl* or *wish* script, or even a Makefile. It can be invoked as an external shell command in a C program using the `system()` call, i.e., `system("script_name");`.
- Setting a variable to the contents of an embedded *sed* or *awk* script increases the readability of the surrounding shell wrapper. See Example A-1 and Example 15-20.
- Put together files containing your favorite and most useful definitions and functions. As necessary, "include" one or more of these "library files" in scripts with either the dot (**.**) or source command.

```
# SCRIPT LIBRARY
# ------ -------

# Note:
# No "#!" here.
# No "live code" either.


# Useful variable definitions

ROOT_UID=0             # Root has $UID 0.
E_NOTROOT=101          # Not root user error.
MAXRETVAL=255          # Maximum (positive) return value of a function.
SUCCESS=0
FAILURE=-1
```

```
# Functions

Usage ()                # "Usage:" message.
{
  if [ -z "$1" ]        # No arg passed.
  then
    msg=filename
  else
    msg=$@
  fi

  echo "Usage: `basename $0` "$msg""
}


Check_if_root ()        # Check if root running script.
{                       # From "ex39.sh" example.
  if [ "$UID" -ne "$ROOT_UID" ]
  then
    echo "Must be root to run this script."
    exit $E_NOTROOT
  fi
}


CreateTempfileName ()   # Creates a "unique" temp filename.
{                       # From "ex51.sh" example.
  prefix=temp
  suffix=`eval date +%s`
  Tempfilename=$prefix.$suffix
}


isalpha2 ()             # Tests whether *entire string* is alphabetic.
{                       # From "isalpha.sh" example.
  [ $# -eq 1 ] || return $FAILURE

  case $1 in
  *[!a-zA-Z]*|"") return $FAILURE;;
  *) return $SUCCESS;;
  esac                  # Thanks, S.C.
}


abs ()                            # Absolute value.
{                                 # Caution: Max return value = 255.
  E_ARGERR=-999999

  if [ -z "$1" ]                  # Need arg passed.
  then
    return $E_ARGERR              # Obvious error value returned.
  fi

  if [ "$1" -ge 0 ]               # If non-negative,
  then                            #
    absval=$1                     # stays as-is.
  else                            # Otherwise,
    let "absval = (( 0 - $1 ))"  # change sign.
  fi
```

```
  return $absval
}


tolower ()             #  Converts string(s) passed as argument(s)
{                      #+ to lowercase.

  if [ -z "$1" ]       #  If no argument(s) passed,
  then                 #+ send error message
    echo "(null)"      #+ (C-style void-pointer error message)
    return             #+ and return from function.
  fi

  echo "$@" | tr A-Z a-z
  # Translate all passed arguments ($@).

  return

# Use command substitution to set a variable to function output.
# For example:
#    oldvar="A seT of miXed-caSe LEtTerS"
#    newvar=`tolower "$oldvar"`
#    echo "$newvar"    # a set of mixed-case letters
#
# Exercise: Rewrite this function to change lowercase passed argument(s)
#           to uppercase ... toupper()  [easy].
}
```

- Use special-purpose comment headers to increase clarity and legibility in scripts.

```
## Caution.
rm -rf *.zzy   ##  The "-rf" options to "rm" are very dangerous,
               ##+ especially with wild cards.

#+ Line continuation.
#  This is line 1
#+ of a multi-line comment,
#+ and this is the final line.

#* Note.

#o List item.

#> Another point of view.
while [ "$var1" != "end" ]    #> while test "$var1" != "end"
```

- Dotan Barak contributes template code for a *progress bar* in a script.

Example 36-17. A Progress Bar

```
#!/bin/bash
# progress-bar.sh

# Author: Dotan Barak (very minor revisions by ABS Guide author).
# Used in ABS Guide with permission (thanks!).
```

```
BAR_WIDTH=50
BAR_CHAR_START="["
BAR_CHAR_END="]"
BAR_CHAR_EMPTY="."
BAR_CHAR_FULL="="
BRACKET_CHARS=2
LIMIT=100

print_progress_bar()
{
        # Calculate how many characters will be full.
        let "full_limit = ((($1 - $BRACKET_CHARS) * $2) / $LIMIT)"

        # Calculate how many characters will be empty.
        let "empty_limit = ($1 - $BRACKET_CHARS) - ${full_limit}"

        # Prepare the bar.
        bar_line="${BAR_CHAR_START}"
        for ((j=0; j<full_limit; j++)); do
                bar_line="${bar_line}${BAR_CHAR_FULL}"
        done

        for ((j=0; j<empty_limit; j++)); do
                bar_line="${bar_line}${BAR_CHAR_EMPTY}"
        done

        bar_line="${bar_line}${BAR_CHAR_END}"

        printf "%3d%% %s" $2 ${bar_line}
}

# Here is a sample of code that uses it.
MAX_PERCENT=100
for ((i=0; i<=MAX_PERCENT; i++)); do
        #
        usleep 10000
        # ... Or run some other commands ...
        #
        print_progress_bar ${BAR_WIDTH} ${i}
        echo -en "\r"
done

echo ""

exit
```

- A particularly clever use of if-test constructs is for comment blocks.

```
#!/bin/bash

COMMENT_BLOCK=
#  Try setting the above variable to some value
#+ for an unpleasant surprise.

if [ $COMMENT_BLOCK ]; then

Comment block --
=================================
This is a comment line.
This is another comment line.
This is yet another comment line.
```

```
=================================

echo "This will not echo."

Comment blocks are error-free! Whee!

fi

echo "No more comments, please."

exit 0
```

Compare this with using here documents to comment out code blocks.

- Using the $? exit status variable, a script may test if a parameter contains only digits, so it can be treated as an integer.

```
#!/bin/bash

SUCCESS=0
E_BADINPUT=85

test "$1" -ne 0 -o "$1" -eq 0 2>/dev/null
# An integer is either equal to 0 or not equal to 0.
# 2>/dev/null suppresses error message.

if [ $? -ne "$SUCCESS" ]
then
  echo "Usage: `basename $0` integer-input"
  exit $E_BADINPUT
fi

let "sum = $1 + 25"             # Would give error if $1 not integer.
echo "Sum = $sum"

# Any variable, not just a command-line parameter, can be tested this way.

exit 0
```

- The 0 - 255 range for function return values is a severe limitation. Global variables and other workarounds are often problematic. An alternative method for a function to communicate a value back to the main body of the script is to have the function write to `stdout` (usually with echo) the "return value," and assign this to a variable. This is actually a variant of command substitution.

Example 36-18. Return value trickery

```
#!/bin/bash
# multiplication.sh

multiply ()                     # Multiplies params passed.
{                               # Will accept a variable number of args.

  local product=1

  until [ -z "$1" ]             # Until uses up arguments passed...
  do
    let "product *= $1"
    shift
  done

  echo $product                 #  Will not echo to stdout,
```

```
}                               #+ since this will be assigned to a variable.

mult1=15383; mult2=25211
val1=`multiply $mult1 $mult2`
# Assigns stdout (echo) of function to the variable val1.
echo "$mult1 X $mult2 = $val1"                   # 387820813

mult1=25; mult2=5; mult3=20
val2=`multiply $mult1 $mult2 $mult3`
echo "$mult1 X $mult2 X $mult3 = $val2"          # 2500

mult1=188; mult2=37; mult3=25; mult4=47
val3=`multiply $mult1 $mult2 $mult3 $mult4`
echo "$mult1 X $mult2 X $mult3 X $mult4 = $val3" # 8173300

exit 0
```

The same technique also works for alphanumeric strings. This means that a function can "return" a non-numeric value.

```
capitalize_ichar ()          #  Capitalizes initial character
{                            #+ of argument string(s) passed.

  string0="$@"               # Accepts multiple arguments.

  firstchar=${string0:0:1}   # First character.
  string1=${string0:1}       # Rest of string(s).

  FirstChar=`echo "$firstchar" | tr a-z A-Z`
                             # Capitalize first character.

  echo "$FirstChar$string1"  # Output to stdout.

}

newstring=`capitalize_ichar "every sentence should start with a capital letter."`
echo "$newstring"          # Every sentence should start with a capital letter.
```

It is even possible for a function to "return" multiple values with this method.

Example 36-19. Even more return value trickery

```
#!/bin/bash
# sum-product.sh
# A function may "return" more than one value.

sum_and_product ()   # Calculates both sum and product of passed args.
{
  echo $(( $1 + $2 )) $(( $1 * $2 ))
# Echoes to stdout each calculated value, separated by space.
}

echo
echo "Enter first number "
read first

echo
echo "Enter second number "
read second
echo
```

```
retval=`sum_and_product $first $second`      # Assigns output of function.
sum=`echo "$retval" | awk '{print $1}'`      # Assigns first field.
product=`echo "$retval" | awk '{print $2}'`  # Assigns second field.

echo "$first + $second = $sum"
echo "$first * $second = $product"
echo

exit 0
```

There can be only **one** *echo* statement in the function for this to work. If you alter the previous example:

```
sum_and_product ()
{
  echo "This is the sum_and_product function." # This messes things up!
  echo $(( $1 + $2 )) $(( $1 * $2 ))
}
...
retval=`sum_and_product $first $second`      # Assigns output of function.
# Now, this will not work correctly.
```

- Next in our bag of tricks are techniques for passing an array to a function, then "returning" an array back to the main body of the script.

  Passing an array involves loading the space-separated elements of the array into a variable with command substitution. Getting an array back as the "return value" from a function uses the previously mentioned strategem of echoing the array in the function, then invoking command substitution and the **(...)** operator to assign it to an array.

Example 36-20. Passing and returning arrays

```
#!/bin/bash
# array-function.sh: Passing an array to a function and ...
#                    "returning" an array from a function


Pass_Array ()
{
  local passed_array   # Local variable!
  passed_array=( `echo "$1"` )
  echo "${passed_array[@]}"
  #  List all the elements of the new array
  #+ declared and set within the function.
}


original_array=( element1 element2 element3 element4 element5 )

echo
echo "original_array = ${original_array[@]}"
#                      List all elements of original array.


# This is the trick that permits passing an array to a function.
# **********************************
argument=`echo ${original_array[@]}`
```

```
# **********************************
#  Pack a variable
#+ with all the space-separated elements of the original array.
#
# Attempting to just pass the array itself will not work.


# This is the trick that allows grabbing an array as a "return value".
# *****************************************
returned_array=( `Pass_Array "$argument"` )
# *****************************************
# Assign 'echoed' output of function to array variable.

echo "returned_array = ${returned_array[@]}"

echo "============================================================="

#  Now, try it again,
#+ attempting to access (list) the array from outside the function.
Pass_Array "$argument"

# The function itself lists the array, but ...
#+ accessing the array from outside the function is forbidden.
echo "Passed array (within function) = ${passed_array[@]}"
# NULL VALUE since the array is a variable local to the function.

echo

############################################

# And here is an even more explicit example:

ret_array ()
{
  for element in {11..20}
  do
    echo "$element "   #  Echo individual elements
  done                 #+ of what will be assembled into an array.
}

arr=( $(ret_array) )   #  Assemble into array.

echo "Capturing array \"arr\" from function ret_array () ..."
echo "Third element of array \"arr\" is ${arr[2]}."   # 13  (zero-indexed)
echo -n "Entire array is: "
echo ${arr[@]}                # 11 12 13 14 15 16 17 18 19 20

echo

exit 0

#  Nathan Coulter points out that passing arrays with elements containing
#+ whitespace breaks this example.
```

For a more elaborate example of passing arrays to functions, see Example A-10.

- Using the double-parentheses construct, it is possible to use C-style syntax for setting and incrementing/decrementing variables and in for and while loops. See Example 11-13 and Example 11-18.

* Setting the path and umask at the beginning of a script makes it more portable -- more likely to run on a "foreign" machine whose user may have bollixed up the `$PATH` and **umask**.

```
#!/bin/bash
PATH=/bin:/usr/bin:/usr/local/bin ; export PATH
umask 022   # Files that the script creates will have 755 permission.

# Thanks to Ian D. Allen, for this tip.
```

* A useful scripting technique is to *repeatedly* feed the output of a filter (by piping) back to the *same filter*, but with a different set of arguments and/or options. Especially suitable for this are tr and grep.

```
# From "wstrings.sh" example.

wlist=`strings "$1" | tr A-Z a-z | tr '[:space:]' Z | \
tr -cs '[:alpha:]' Z | tr -s '\173-\377' Z | tr Z ' '`
```

Example 36-21. Fun with anagrams

```
#!/bin/bash
# agram.sh: Playing games with anagrams.

# Find anagrams of...
LETTERSET=etaoinshrdlu
FILTER='.......'       # How many letters minimum?
#       1234567

anagram "$LETTERSET" | # Find all anagrams of the letterset...
grep "$FILTER" |       # With at least 7 letters,
grep '^is' |           # starting with 'is'
grep -v 's$' |         # no plurals
grep -v 'ed$'          # no past tense verbs
# Possible to add many combinations of conditions and filters.

#  Uses "anagram" utility
#+ that is part of the author's "yawl" word list package.
#  http://ibiblio.org/pub/Linux/libs/yawl-0.3.2.tar.gz
#  http://bash.deta.in/yawl-0.3.2.tar.gz

exit 0                 # End of code.


bash$ sh agram.sh
islander
isolate
isolead
isotheral



#  Exercises:
#  ---------
#  Modify this script to take the LETTERSET as a command-line parameter.
#  Parameterize the filters in lines 11 - 13 (as with $FILTER),
#+ so that they can be specified by passing arguments to a function.

#  For a slightly different approach to anagramming,
#+ see the agram2.sh script.
```

See also Example 29-4, Example 16-25, and Example A-9.

- Use "anonymous here documents" to comment out blocks of code, to save having to individually comment out each line with a #. See Example 19-11.
- Running a script on a machine that relies on a command that might not be installed is dangerous. Use whatis to avoid potential problems with this.

```
CMD=command1                 # First choice.
PlanB=command2               # Fallback option.

command_test=$(whatis "$CMD" | grep 'nothing appropriate')
#  If 'command1' not found on system , 'whatis' will return
#+ "command1: nothing appropriate."
#
#  A safer alternative is:
#     command_test=$(whereis "$CMD" | grep \/)
#  But then the sense of the following test would have to be reversed,
#+ since the $command_test variable holds content only if
#+ the $CMD exists on the system.
#     (Thanks, bojster.)


if [[ -z "$command_test" ]]  # Check whether command present.
then
  $CMD option1 option2       #  Run command1 with options.
else                         #  Otherwise,
  $PlanB                     #+ run command2.
fi
```

- An if-grep test may not return expected results in an error case, when text is output to `stderr`, rather that `stdout`.

```
if ls -l nonexistent_filename | grep -q 'No such file or directory'
  then echo "File \"nonexistent_filename\" does not exist."
fi
```

Redirecting `stderr` to `stdout` fixes this.

```
if ls -l nonexistent_filename 2>&1 | grep -q 'No such file or directory'
#                             ^^^^
  then echo "File \"nonexistent_filename\" does not exist."
fi

# Thanks, Chris Martin, for pointing this out.
```

- If you absolutely must access a subshell variable outside the subshell, here's a way to do it.

```
TMPFILE=tmpfile                  # Create a temp file to store the variable.

(   # Inside the subshell ...
inner_variable=Inner
echo $inner_variable
echo $inner_variable >>$TMPFILE  # Append to temp file.
)

    # Outside the subshell ...

echo; echo "-----"; echo
echo $inner_variable             # Null, as expected.
echo "-----"; echo

# Now ...
```

```
read inner_variable <$TMPFILE    # Read back shell variable.
rm -f "$TMPFILE"                 # Get rid of temp file.
echo "$inner_variable"           # It's an ugly kludge, but it works.
```

- The run-parts command is handy for running a set of command scripts in a particular sequence, especially in combination with cron or at.
- For doing multiple revisions on a complex script, use the *rcs* Revision Control System package.

  Among other benefits of this is automatically updated ID header tags. The **co** command in *rcs* does a parameter replacement of certain reserved key words, for example, replacing `# $Id$` in a script with something like:

  ```
  # $Id: hello-world.sh,v 1.1 2004/10/16 02:43:05 bozo Exp $
  ```

36.7.2. Widgets

It would be nice to be able to invoke X-Windows widgets from a shell script. There happen to exist several packages that purport to do so, namely *Xscript*, *Xmenu*, and *widtools*. The first two of these no longer seem to be maintained. Fortunately, it is still possible to obtain *widtools* here.

The *widtools* (widget tools) package requires the *XForms* library to be installed. Additionally, the Makefile needs some judicious editing before the package will build on a typical Linux system. Finally, three of the six widgets offered do not work (and, in fact, segfault).

The *dialog* family of tools offers a method of calling "dialog" widgets from a shell script. The original *dialog* utility works in a text console, but its successors, *gdialog*, *Xdialog*, and *kdialog* use X-Windows-based widget sets.

Example 36-22. Widgets invoked from a shell script

```
#!/bin/bash
# dialog.sh: Using 'gdialog' widgets.

# Must have 'gdialog' installed on your system to run this script.
# Or, you can replace all instance of 'gdialog' below with 'kdialog' ...
# Version 1.1 (corrected 04/05/05)

# This script was inspired by the following article.
#     "Scripting for X Productivity," by Marco Fioretti,
#      LINUX JOURNAL, Issue 113, September 2003, pp. 86-9.
# Thank you, all you good people at LJ.


# Input error in dialog box.
E_INPUT=85
# Dimensions of display, input widgets.
HEIGHT=50
WIDTH=60

# Output file name (constructed out of script name).
OUTFILE=$0.output
```

```
# Display this script in a text widget.
gdialog --title "Displaying: $0" --textbox $0 $HEIGHT $WIDTH



# Now, we'll try saving input in a file.
echo -n "VARIABLE=" > $OUTFILE
gdialog --title "User Input" --inputbox "Enter variable, please:" \
$HEIGHT $WIDTH 2>> $OUTFILE


if [ "$?" -eq 0 ]
# It's good practice to check exit status.
then
  echo "Executed \"dialog box\" without errors."
else
  echo "Error(s) in \"dialog box\" execution."
        # Or, clicked on "Cancel", instead of "OK" button.
  rm $OUTFILE
  exit $E_INPUT
fi



# Now, we'll retrieve and display the saved variable.
. $OUTFILE   # 'Source' the saved file.
echo "The variable input in the \"input box\" was: "$VARIABLE""


rm $OUTFILE  # Clean up by removing the temp file.
             # Some applications may need to retain this file.

exit $?

# Exercise: Rewrite this script using the 'zenity' widget set.
```

The xmessage command is a simple method of popping up a message/query window. For example:

```
xmessage Fatal error in script! -button exit
```

The latest entry in the widget sweepstakes is zenity. This utility pops up *GTK+* dialog widgets-and-windows, and it works very nicely within a script.

```
get_info ()
{
  zenity --entry       #  Pops up query window . . .
                       #+ and prints user entry to stdout.

                       #  Also try the --calendar and --scale options.
}

answer=$( get_info )   #  Capture stdout in $answer variable.

echo "User entered: "$answer""
```

For other methods of scripting with widgets, try *Tk* or *wish* (*Tcl* derivatives), *PerlTk* (*Perl* with *Tk* extensions), *tksh* (*ksh* with *Tk* extensions), *XForms4Perl* (*Perl* with *XForms* extensions), *Gtk-Perl* (*Perl* with *Gtk* extensions), or *PyQt* (*Python* with *Qt* extensions).

36.8. Security Issues

36.8.1. Infected Shell Scripts

A brief warning about script security is indicated. A shell script may contain a *worm*, *trojan*, or even a *virus*. For that reason, never run as *root* a script (or permit it to be inserted into the system startup scripts in `/etc/rc.d`) unless you have obtained said script from a trusted source or you have carefully analyzed it to make certain it does nothing harmful.

Various researchers at Bell Labs and other sites, including M. Douglas McIlroy, Tom Duff, and Fred Cohen have investigated the implications of shell script viruses. They conclude that it is all too easy for even a novice, a "script kiddie," to write one. [128]

Here is yet another reason to learn scripting. Being able to look at and understand scripts may protect your system from being compromised by a rogue script.

36.8.2. Hiding Shell Script Source

For security purposes, it may be necessary to render a script unreadable. If only there were a utility to create a stripped binary executable from a script. Francisco Rosales' shc -- generic shell script compiler does exactly that.

Unfortunately, according to an article in the October, 2005 *Linux Journal*, the binary can, in at least some cases, be decrypted to recover the original script source. Still, this could be a useful method of keeping scripts secure from all but the most skilled hackers.

36.8.3. Writing Secure Shell Scripts

Dan Stromberg suggests the following guidelines for writing (relatively) secure shell scripts.

- Don't put secret data in environment variables.
- Don't pass secret data in an external command's arguments (pass them in via a pipe or redirection instead).
- Set your $PATH carefully. Don't just trust whatever path you inherit from the caller if your script is running as *root*. In fact, whenever you use an environment variable inherited from the caller, think about what could happen if the caller put something misleading in the variable, e.g., if the caller set $HOME to `/etc`.

36.9. Portability Issues

It is easier to port a shell than a shell script.

--Larry Wall

This book deals specifically with Bash scripting on a GNU/Linux system. All the same, users of **sh** and **ksh** will find much of value here.

As it happens, many of the various shells and scripting languages seem to be converging toward the POSIX 1003.2 standard. Invoking Bash with the `--posix` option or inserting a **set -o posix** at the head of a script causes Bash to conform very closely to this standard. Another alternative is to use a *#!/bin/sh* sha-bang header in the script, rather than *#!/bin/bash*. [129] Note that `/bin/sh` is a link to `/bin/bash` in Linux and certain other flavors of UNIX, and a script invoked this way disables extended Bash functionality.

Most Bash scripts will run as-is under **ksh**, and vice-versa, since Chet Ramey has been busily porting **ksh** features to the latest versions of Bash.

On a commercial UNIX machine, scripts using GNU-specific features of standard commands may not work. This has become less of a problem in the last few years, as the GNU utilities have pretty much displaced their proprietary counterparts even on "big-iron" UNIX. Caldera's release of the source to many of the original UNIX utilities has accelerated the trend.

Bash has certain features that the traditional Bourne shell lacks. Among these are:

- Certain extended invocation options
- Command substitution using **$( )** notation
- Brace expansion
- Certain array operations, and associative arrays
- The double brackets extended test construct
- The double-parentheses arithmetic-evaluation construct
- Certain string manipulation operations
- Process substitution
- A Regular Expression matching operator
- Bash-specific builtins
- Coprocesses

See the Bash F.A.Q. for a complete listing.

36.9.1. A Test Suite

Let us illustrate some of the incompatibilities between Bash and the classic Bourne shell. Download and install the "Heirloom Bourne Shell" and run the following script, first using Bash, then the classic *sh*.

Example 36-23. Test Suite

```
#!/bin/bash
# test-suite.sh
# A partial Bash compatibility test suite.
# Run this on your version of Bash, or some other shell.

default_option=FAIL         # Tests below will fail unless . . .

echo
echo -n "Testing "
sleep 1; echo -n ". "
sleep 1; echo -n ". "
sleep 1; echo ". "
echo
```

```
# Double brackets
String="Double brackets supported?"
echo -n "Double brackets test: "
if [[ "$String" = "Double brackets supported?" ]]
then
  echo "PASS"
else
  echo "FAIL"
fi


# Double brackets and regex matching
String="Regex matching supported?"
echo -n "Regex matching: "
if [[ "$String" =~ R.....matching* ]]
then
  echo "PASS"
else
  echo "FAIL"
fi


# Arrays
test_arr=$default_option     # FAIL
Array=( If supports arrays will print PASS )
test_arr=${Array[5]}
echo "Array test: $test_arr"


# Command Substitution
csub_test ()
{
  echo "PASS"
}

test_csub=$default_option    # FAIL
test_csub=$(csub_test)
echo "Command substitution test: $test_csub"

echo

#  Completing this script is an exercise for the reader.
#  Add to the above similar tests for double parentheses,
#+ brace expansion, process substitution, etc.

exit $?
```

36.10. Shell Scripting Under Windows

Even users running *that other* OS can run UNIX-like shell scripts, and therefore benefit from many of the lessons of this book. The Cygwin package from Cygnus and the MKS utilities from Mortice Kern Associates add shell scripting capabilities to Windows.

Another alternative is UWIN, written by David Korn of AT&T, of Korn Shell fame.

In 2006, Microsoft released the Windows Powershell®, which contains limited Bash-like command-line scripting capabilities.

Chapter 37. Bash, versions 2, 3, and 4

37.1. Bash, version 2

The current version of *Bash*, the one you have running on your machine, is most likely version 2.xx.yy, 3.xx.yy, or 4.xx.yy.

```
bash$ echo $BASH_VERSION
3.2.25(1)-release
```

The version 2 update of the classic Bash scripting language added array variables, string and parameter expansion, and a better method of indirect variable references, among other features.

Example 37-1. String expansion

```
#!/bin/bash

# String expansion.
# Introduced with version 2 of Bash.

#  Strings of the form $'xxx'
#+ have the standard escaped characters interpreted.

echo $'Ringing bell 3 times \a \a \a'
     # May only ring once with certain terminals.
     # Or ...
     # May not ring at all, depending on terminal settings.
echo $'Three form feeds \f \f \f'
echo $'10 newlines \n\n\n\n\n\n\n\n\n\n'
echo $'\102\141\163\150'
     #   B   a   s   h
     # Octal equivalent of characters.

exit
```

Example 37-2. Indirect variable references - the new way

```
#!/bin/bash

# Indirect variable referencing.
# This has a few of the attributes of references in C++.


a=letter_of_alphabet
letter_of_alphabet=z

echo "a = $a"           # Direct reference.

echo "Now a = ${!a}"    # Indirect reference.
#  The ${!variable} notation is more intuitive than the old
#+ eval var1=\$$var2

echo
```

```
t=table_cell_3
table_cell_3=24
echo "t = ${!t}"                       # t = 24
table_cell_3=387
echo "Value of t changed to ${!t}"    # 387
# No 'eval' necessary.

#  This is useful for referencing members of an array or table,
#+ or for simulating a multi-dimensional array.
#  An indexing option (analogous to pointer arithmetic)
#+ would have been nice. Sigh.

exit 0

# See also, ind-ref.sh example.
```

Example 37-3. Simple database application, using indirect variable referencing

```
#!/bin/bash
# resistor-inventory.sh
# Simple database / table-lookup application.

# ============================================================== #
# Data

B1723_value=470                                   # Ohms
B1723_powerdissip=.25                             # Watts
B1723_colorcode="yellow-violet-brown"             # Color bands
B1723_loc=173                                     # Where they are
B1723_inventory=78                                # How many

B1724_value=1000
B1724_powerdissip=.25
B1724_colorcode="brown-black-red"
B1724_loc=24N
B1724_inventory=243

B1725_value=10000
B1725_powerdissip=.125
B1725_colorcode="brown-black-orange"
B1725_loc=24N
B1725_inventory=89

# ============================================================== #


echo

PS3='Enter catalog number: '

echo

select catalog_number in "B1723" "B1724" "B1725"
do
  Inv=${catalog_number}_inventory
  Val=${catalog_number}_value
  Pdissip=${catalog_number}_powerdissip
  Loc=${catalog_number}_loc
  Ccode=${catalog_number}_colorcode
```

```
  echo
  echo "Catalog number $catalog_number:"
  # Now, retrieve value, using indirect referencing.
  echo "There are ${!Inv} of  [${!Val} ohm / ${!Pdissip} watt]\
  resistors in stock."  #        ^             ^
  # As of Bash 4.2, you can replace "ohm" with \u2126 (using echo -e).
  echo "These are located in bin # ${!Loc}."
  echo "Their color code is \"${!Ccode}\"."

  break
done

echo; echo

# Exercises:
# ---------
# 1) Rewrite this script to read its data from an external file.
# 2) Rewrite this script to use arrays,
#+   rather than indirect variable referencing.
#    Which method is more straightforward and intuitive?
#    Which method is easier to code?


# Notes:
# -----
#  Shell scripts are inappropriate for anything except the most simple
#+ database applications, and even then it involves workarounds and kludges.
#  Much better is to use a language with native support for data structures,
#+ such as C++ or Java (or even Perl).

exit 0
```

Example 37-4. Using arrays and other miscellaneous trickery to deal four random hands from a deck of cards

```
#!/bin/bash
# cards.sh

# Deals four random hands from a deck of cards.

UNPICKED=0
PICKED=1

DUPE_CARD=99

LOWER_LIMIT=0
UPPER_LIMIT=51
CARDS_IN_SUIT=13
CARDS=52

declare -a Deck
declare -a Suits
declare -a Cards
#  It would have been easier to implement and more intuitive
#+ with a single, 3-dimensional array.
#  Perhaps a future version of Bash will support multidimensional arrays.


initialize_Deck ()
{
i=$LOWER_LIMIT
```

```
until [ "$i" -gt $UPPER_LIMIT ]
do
  Deck[i]=$UNPICKED   # Set each card of "Deck" as unpicked.
  let "i += 1"
done
echo
}

initialize_Suits ()
{
Suits[0]=C #Clubs
Suits[1]=D #Diamonds
Suits[2]=H #Hearts
Suits[3]=S #Spades
}

initialize_Cards ()
{
Cards=(2 3 4 5 6 7 8 9 10 J Q K A)
# Alternate method of initializing an array.
}

pick_a_card ()
{
card_number=$RANDOM
let "card_number %= $CARDS" # Restrict range to 0 - 51, i.e., 52 cards.
if [ "${Deck[card_number]}" -eq $UNPICKED ]
then
  Deck[card_number]=$PICKED
  return $card_number
else
  return $DUPE_CARD
fi
}

parse_card ()
{
number=$1
let "suit_number = number / CARDS_IN_SUIT"
suit=${Suits[suit_number]}
echo -n "$suit-"
let "card_no = number % CARDS_IN_SUIT"
Card=${Cards[card_no]}
printf %-4s $Card
# Print cards in neat columns.
}

seed_random ()  # Seed random number generator.
{               # What happens if you don't do this?
seed=`eval date +%s`
let "seed %= 32766"
RANDOM=$seed
} # Consider other methods of seeding the random number generator.

deal_cards ()
{
echo

cards_picked=0
while [ "$cards_picked" -le $UPPER_LIMIT ]
do
  pick_a_card
```

```
  t=$?

  if [ "$t" -ne $DUPE_CARD ]
  then
    parse_card $t

    u=$cards_picked+1
    # Change back to 1-based indexing, temporarily. Why?
    let "u %= $CARDS_IN_SUIT"
    if [ "$u" -eq 0 ]   # Nested if/then condition test.
    then
     echo
     echo
    fi                  # Each hand set apart with a blank line.

    let "cards_picked += 1"
  fi
done

echo

return 0
}


# Structured programming:
# Entire program logic modularized in functions.

#================
seed_random
initialize_Deck
initialize_Suits
initialize_Cards
deal_cards
#================

exit



# Exercise 1:
# Add comments to thoroughly document this script.

# Exercise 2:
# Add a routine (function) to print out each hand sorted in suits.
# You may add other bells and whistles if you like.

# Exercise 3:
# Simplify and streamline the logic of the script.
```

37.2. Bash, version 3

On July 27, 2004, Chet Ramey released version 3 of Bash. This update fixed quite a number of bugs and added new features.

Some of the more important added features:

A new, more generalized **{a..z}** brace expansion operator.

```
#!/bin/bash

for i in {1..10}
#  Simpler and more straightforward than
#+ for i in $(seq 10)
do
  echo -n "$i "
done

echo

# 1 2 3 4 5 6 7 8 9 10



# Or just . . .

echo {a..z}    #  a b c d e f g h i j k l m n o p q r s t u v w x y z
echo {e..m}    #  e f g h i j k l m
echo {z..a}    #  z y x w v u t s r q p o n m l k j i h g f e d c b a
               #  Works backwards, too.
echo {25..30}  #  25 26 27 28 29 30
echo {3..-2}   #  3 2 1 0 -1 -2
echo {X..d}    #  X Y Z [  ] ^ _ ` a b c d
               #  Shows (some of) the ASCII characters between Z and a,
               #+ but don't rely on this type of behavior because . . .
echo {]..a}    #  {]..a}
               #  Why?


# You can tack on prefixes and suffixes.
echo "Number #"{1..4}, "..."
     # Number #1, Number #2, Number #3, Number #4, ...


# You can concatenate brace-expansion sets.
echo {1..3}{x..z}" +" "..."
     # 1x + 1y + 1z + 2x + 2y + 2z + 3x + 3y + 3z + ...
     # Generates an algebraic expression.
     # This could be used to find permutations.

# You can nest brace-expansion sets.
echo {{a..c},{1..3}}
     # a b c 1 2 3
     # The "comma operator" splices together strings.

# ########## ######### ############ ########### ######### ###############
# Unfortunately, brace expansion does not lend itself to parameterization.
var1=1
var2=5
echo {$var1..$var2}   # {1..5}


# Yet, as Emiliano G. points out, using "eval" overcomes this limitation.

start=0
end=10
for index in $(eval echo {$start..$end})
do
  echo -n "$index "   # 0 1 2 3 4 5 6 7 8 9 10
```

- ```
  done

  echo
  ```
- The **${!array[@]}** operator, which expands to all the indices of a given array.

  ```
  #!/bin/bash

  Array=(element-zero element-one element-two element-three)

  echo ${Array[0]}   # element-zero
                     # First element of array.

  echo ${!Array[@]}  # 0 1 2 3
                     # All the indices of Array.

  for i in ${!Array[@]}
  do
    echo ${Array[i]} # element-zero
                     # element-one
                     # element-two
                     # element-three
                     #
                     # All the elements in Array.
  done
  ```
- The **=~** Regular Expression matching operator within a double brackets test expression. (Perl has a similar operator.)

  ```
  #!/bin/bash

  variable="This is a fine mess."

  echo "$variable"

  # Regex matching with =~ operator within [[ double brackets ]].
  if [[ "$variable" =~ T.........fin*es* ]]
  # NOTE: As of version 3.2 of Bash, expression to match no longer quoted.
  then
    echo "match found"
        # match found
  fi
  ```

  Or, more usefully:

  ```
  #!/bin/bash

  input=$1


  if [[ "$input" =~ "[0-9][0-9][0-9]-[0-9][0-9]-[0-9][0-9][0-9][0-9]" ]]
  #                 ^ NOTE: Quoting not necessary, as of version 3.2 of Bash.
  # NNN-NN-NNNN (where each N is a digit).
  then
    echo "Social Security number."
    # Process SSN.
  else
    echo "Not a Social Security number!"
    # Or, ask for corrected input.
  fi
  ```
  ```

For additional examples of using the **=~** operator, see Example A-29, Example 19-14, Example A-35, and Example A-24.

- The new `set -o pipefail` option is useful for debugging pipes. If this option is set, then the exit status of a pipe is the exit status of the last command in the pipe to *fail* (return a non-zero value), rather than the actual final command in the pipe.

  See Example 16-43.

The update to version 3 of Bash breaks a few scripts that worked under earlier versions. *Test critical legacy scripts to make sure they still work!*

As it happens, a couple of the scripts in the *Advanced Bash Scripting Guide* had to be fixed up (see Example 9-4, for instance).

37.2.1. Bash, version 3.1

The version 3.1 update of Bash introduces a number of bugfixes and a few minor changes.

- The += operator is now permitted in in places where previously only the = assignment operator was recognized.

```
a=1
echo $a        # 1

a+=5           # Won't work under versions of Bash earlier than 3.1.
echo $a        # 15

a+=Hello
echo $a        # 15Hello
```

Here, += functions as a *string concatenation* operator. Note that its behavior in this particular context is different than within a let construct.

```
a=1
echo $a        # 1

let a+=5       # Integer arithmetic, rather than string concatenation.
echo $a        # 6

let a+=Hello   # Doesn't "add" anything to a.
echo $a        # 6
```

Jeffrey Haemer points out that this concatenation operator can be quite useful. In this instance, we append a directory to the `$PATH`.

```
bash$ echo $PATH
/usr/bin:/bin:/usr/local/bin:/usr/X11R6/bin/:/usr/games


bash$ PATH+=:/opt/bin

bash$ echo $PATH
/usr/bin:/bin:/usr/local/bin:/usr/X11R6/bin/:/usr/games:/opt/bin
```

37.2.2. Bash, version 3.2

This is pretty much a bugfix update.

- In *global* parameter substitutions, the pattern no longer anchors at the start of the string.
- The `--wordexp` option disables process substitution.
- The **=~** Regular Expression match operator no longer requires quoting of the *pattern* within [[...]].

In fact, quoting in this context is *not* advisable as it may cause *regex* evaluation to fail. Chet Ramey states in the Bash FAQ that quoting explicitly disables regex evaluation. See also the Ubuntu Bug List and Wikinerds on Bash syntax.

Setting *shopt -s compat31* in a script causes reversion to the original behavior.

37.3. Bash, version 4

Chet Ramey announced Version 4 of Bash on the 20th of February, 2009. This release has a number of significant new features, as well as some important bugfixes.

Among the new goodies:

- Associative arrays. [130]

An *associative* array can be thought of as a set of two linked arrays -- one holding the *data*, and the other the *keys* that index the individual elements of the *data* array.

Example 37-5. A simple address database

```
#!/bin/bash4
# fetch_address.sh

declare -A address
#       -A option declares associative array.

address[Charles]="414 W. 10th Ave., Baltimore, MD 21236"
address[John]="202 E. 3rd St., New York, NY 10009"
address[Wilma]="1854 Vermont Ave, Los Angeles, CA 90023"


echo "Charles's address is ${address[Charles]}."
# Charles's address is 414 W. 10th Ave., Baltimore, MD 21236.
echo "Wilma's address is ${address[Wilma]}."
# Wilma's address is 1854 Vermont Ave, Los Angeles, CA 90023.
echo "John's address is ${address[John]}."
# John's address is 202 E. 3rd St., New York, NY 10009.

echo

echo "${!address[*]}"   # The array indices ...
# Charles John Wilma
```

Example 37-6. A somewhat more elaborate address database

```
#!/bin/bash4
# fetch_address-2.sh
# A more elaborate version of fetch_address.sh.

SUCCESS=0
E_DB=99    # Error code for missing entry.

declare -A address
#       -A option declares associative array.


store_address ()
{
  address[$1]="$2"
  return $?
}


fetch_address ()
{
  if [[ -z "${address[$1]}" ]]
  then
    echo "$1's address is not in database."
    return $E_DB
  fi

  echo "$1's address is ${address[$1]}."
  return $?
}


store_address "Lucas Fayne" "414 W. 13th Ave., Baltimore, MD 21236"
store_address "Arvid Boyce" "202 E. 3rd St., New York, NY 10009"
store_address "Velma Winston" "1854 Vermont Ave, Los Angeles, CA 90023"
#  Exercise:
#  Rewrite the above store_address calls to read data from a file,
#+ then assign field 1 to name, field 2 to address in the array.
#  Each line in the file would have a format corresponding to the above.
#  Use a while-read loop to read from file, sed or awk to parse the fields.

fetch_address "Lucas Fayne"
# Lucas Fayne's address is 414 W. 13th Ave., Baltimore, MD 21236.
fetch_address "Velma Winston"
# Velma Winston's address is 1854 Vermont Ave, Los Angeles, CA 90023.
fetch_address "Arvid Boyce"
# Arvid Boyce's address is 202 E. 3rd St., New York, NY 10009.
fetch_address "Bozo Bozeman"
# Bozo Bozeman's address is not in database.

exit $?   # In this case, exit code = 99, since that is function return.
```

See Example A-53 for an interesting usage of an *associative array*.

Elements of the *index* array may include embedded space characters, or even leading and/or trailing space characters. However, index array elements containing *only whitespace* are *not* permitted.

```
address[   ]="Blank"   # Error!
```

- Enhancements to the case construct: the `;;&` and `;&` terminators.

Example 37-7. Testing characters

```
#!/bin/bash4

test_char ()
{
  case "$1" in
    [[:print:]] )  echo "$1 is a printable character.";;&       # |
    # The ;;& terminator continues to the next pattern test.      |
    [[:alnum:]] )  echo "$1 is an alpha/numeric character.";;&  # v
    [[:alpha:]] )  echo "$1 is an alphabetic character.";;&     # v
    [[:lower:]] )  echo "$1 is a lowercase alphabetic character.";;&
    [[:digit:]] )  echo "$1 is an numeric character.";&         # |
    # The ;& terminator executes the next statement ...         # |
    %%%@@@@@    )  echo "********************************";;    # v
#   ^^^^^^^^  ... even with a dummy pattern.
  esac
}

echo

test_char 3
# 3 is a printable character.
# 3 is an alpha/numeric character.
# 3 is an numeric character.
# ********************************
echo

test_char m
# m is a printable character.
# m is an alpha/numeric character.
# m is an alphabetic character.
# m is a lowercase alphabetic character.
echo

test_char /
# / is a printable character.

echo

# The ;;& terminator can save complex if/then conditions.
# The ;& is somewhat less useful.
```

- The new **coproc** builtin enables two parallel processes to communicate and interact. As Chet Ramey states in the Bash FAQ [131] , ver. 4.01:

```
There is a new 'coproc' reserved word that specifies a coprocess
an asynchronous command run with two pipes connected to the creat
shell. Coprocs can be named. The input and output file descripto
and the PID of the coprocess are available to the calling shell i
variables with coproc-specific names.

George Dimitriu explains,
"... coproc ... is a feature used in Bash process substitution,
which now is made publicly available."
This means it can be explicitly invoked in a script, rather than
just being a behind-the-scenes mechanism used by Bash.
```

Coprocesses use *file descriptors*. File descriptors enable processes and pipes to communicate.

```
#!/bin/bash4
# A coprocess communicates with a while-read loop.


coproc { cat mx_data.txt; sleep 2; }
#                          ^^^^^^^
# Try running this without "sleep 2" and see what happens.

while read -u ${COPROC[0]} line    #  ${COPROC[0]} is the
do                                 #+ file descriptor of the coprocess.
  echo "$line" | sed -e 's/line/NOT-ORIGINAL-TEXT/'
done

kill $COPROC_PID                   #  No longer need the coprocess,
                                   #+ so kill its PID.
```

But, be careful!

```
#!/bin/bash4

echo; echo
a=aaa
b=bbb
c=ccc

coproc echo "one two three"
while read -u ${COPROC[0]} a b c;  #  Note that this loop
do                                 #+ runs in a subshell.
  echo "Inside while-read loop: ";
  echo "a = $a"; echo "b = $b"; echo "c = $c"
  echo "coproc file descriptor: ${COPROC[0]}"
done

# a = one
# b = two
# c = three
# So far, so good, but ...

echo "-----------------"
echo "Outside while-read loop: "
echo "a = $a"  # a =
echo "b = $b"  # b =
echo "c = $c"  # c =
echo "coproc file descriptor: ${COPROC[0]}"
echo
#  The coproc is still running, but ...
#+ it still doesn't enable the parent process
#+ to "inherit" variables from the child process, the while-read loop.

#  Compare this to the "badread.sh" script.
```

The coprocess is *asynchronous*, and this might cause a problem. It may terminate before another process has finished communicating with it.

```
#!/bin/bash4
```

```
coproc cpname { for i in {0..10}; do echo "index = $i"; done; }
#      ^^^^^^ This is a *named* coprocess.
read -u ${cpname[0]}
echo $REPLY         #  index = 0
echo ${COPROC[0]}   #+ No output ... the coprocess timed out
#  after the first loop iteration.



# However, George Dimitriu has a partial fix.

coproc cpname { for i in {0..10}; do echo "index = $i"; done; sleep 1;
echo hi > myo; cat - >> myo; }
#       ^^^^^ This is a *named* coprocess.

echo "I am main"$'\04' >&${cpname[1]}
myfd=${cpname[0]}
echo myfd=$myfd

### while read -u $myfd
### do
###   echo $REPLY;
### done

echo $cpname_PID

#  Run this with and without the commented-out while-loop, and it is
#+ apparent that each process, the executing shell and the coprocess,
#+ waits for the other to finish writing in its own write-enabled pipe.
```

- The new **mapfile** builtin makes it possible to load an array with the contents of a text file without using a loop or command substitution.

```
#!/bin/bash4

mapfile Arr1 < $0
# Same result as     Arr1=( $(cat $0) )
echo "${Arr1[@]}"  # Copies this entire script out to stdout.

echo "--"; echo

# But, not the same as   read -a   !!!
read -a Arr2 < $0
echo "${Arr2[@]}"  # Reads only first line of script into the array.

exit
```

- The read builtin got a minor facelift. The `-t` timeout option now accepts (decimal) fractional values [132] and the `-i` option permits preloading the edit buffer. [133] Unfortunately, these enhancements are still a work in progress and not (yet) usable in scripts.
- Parameter substitution gets *case-modification* operators.

```
#!/bin/bash4

var=veryMixedUpVariable
echo ${var}            # veryMixedUpVariable
echo ${var^}           # VeryMixedUpVariable
#         *              First char --> uppercase.
echo ${var^^}          # VERYMIXEDUPVARIABLE
#         **             All chars  --> uppercase.
echo ${var,}           # veryMixedUpVariable
```

```
#         *              First char --> lowercase.
echo ${var,,}          # verymixedupvariable
#         **             All chars  --> lowercase.
```

- The declare builtin now accepts the `-l` *lowercase* and `-c` *capitalize* options.

```
#!/bin/bash4

declare -l var1            # Will change to lowercase
var1=MixedCaseVARIABLE
echo "$var1"               # mixedcasevariable
# Same effect as             echo $var1 | tr A-Z a-z

declare -c var2            # Changes only initial char to uppercase.
var2=originally_lowercase
echo "$var2"               # Originally_lowercase
# NOT the same effect as     echo $var2 | tr a-z A-Z
```

- Brace expansion has more options.

  Increment/decrement, specified in the final term within braces.

```
#!/bin/bash4

echo {40..60..2}
# 40 42 44 46 48 50 52 54 56 58 60
# All the even numbers, between 40 and 60.

echo {60..40..2}
# 60 58 56 54 52 50 48 46 44 42 40
# All the even numbers, between 40 and 60, counting backwards.
# In effect, a decrement.
echo {60..40..-2}
# The same output. The minus sign is not necessary.

# But, what about letters and symbols?
echo {X..d}
# X Y Z [  ] ^ _ ` a b c d
# Does not echo the \ which escapes a space.
```

  Zero-padding, specified in the first term within braces, prefixes each term in the output with the *same number* of zeroes.

```
bash4$ echo {010..15}
010 011 012 013 014 015


bash4$ echo {000..10}
000 001 002 003 004 005 006 007 008 009 010
```

- *Substring extraction* on *positional parameters* now starts with $0 as the *zero-index*. (This corrects an inconsistency in the treatment of positional parameters.)

```
#!/bin/bash
# show-params.bash
# Requires version 4+ of Bash.

# Invoke this scripts with at least one positional parameter.
```

```
E_BADPARAMS=99

if [ -z "$1" ]
then
  echo "Usage $0 param1 ..."
  exit $E_BADPARAMS
fi

echo ${@:0}

# bash3 show-params.bash4 one two three
# one two three

# bash4 show-params.bash4 one two three
# show-params.bash4 one two three

# $0                $1  $2  $3
```

* The new ** globbing operator matches filenames and directories recursively.

```
#!/bin/bash4
# filelist.bash4

shopt -s globstar  # Must enable globstar, otherwise ** doesn't work.
                   # The globstar shell option is new to version 4 of Bash.

echo "Using *"; echo
for filename in *
do
  echo "$filename"
done   # Lists only files in current directory ($PWD).

echo; echo "--------------"; echo

echo "Using **"
for filename in **
do
  echo "$filename"
done   # Lists complete file tree, recursively.

exit

Using *

allmyfiles
filelist.bash4

--------------

Using **

allmyfiles
allmyfiles/file.index.txt
allmyfiles/my_music
allmyfiles/my_music/me-singing-60s-folksongs.ogg
allmyfiles/my_music/me-singing-opera.ogg
allmyfiles/my_music/piano-lesson.1.ogg
allmyfiles/my_pictures
allmyfiles/my_pictures/at-beach-with-Jade.png
allmyfiles/my_pictures/picnic-with-Melissa.png
filelist.bash4
```

* The new $BASHPID internal variable.

* There is a new builtin error-handling function named **command_not_found_handle**.

```
#!/bin/bash4

command_not_found_handle ()
{ # Accepts implicit parameters.
  echo "The following command is not valid: \""$1\"""
  echo "With the following argument(s): \""$2\"" \""$3\"""   # $4, $5 ...
} # $1, $2, etc. are not explicitly passed to the function.

bad_command arg1 arg2

# The following command is not valid: "bad_command"
# With the following argument(s): "arg1" "arg2"
```

Editorial comment

Associative arrays? Coprocesses? Whatever happened to the lean and mean Bash we have come to know and love? Could it be suffering from (horrors!) "feature creep"? Or perhaps even Korn shell envy?

Note to Chet Ramey: Please add only *essential* features in future Bash releases -- perhaps *for-each* loops and support for multi-dimensional arrays. [134] Most Bash users won't need, won't use, and likely won't greatly appreciate complex "features" like built-in debuggers, Perl interfaces, and bolt-on rocket boosters.

37.3.1. Bash, version 4.1

Version 4.1 of Bash, released in May, 2010, was primarily a bugfix update.

* The printf command now accepts a `-v` option for setting array indices.
* Within double brackets, the **>** and **<** string comparison operators now conform to the locale. Since the locale setting may affect the sorting order of string expressions, this has side-effects on comparison tests within *[[...]]* expressions.
* The read builtin now takes a `-N` option (*read -N chars*), which causes the *read* to terminate after *chars* characters.

Example 37-8. Reading N characters

```
#!/bin/bash
# Requires Bash version -ge 4.1 ...

num_chars=61

read -N $num_chars var < $0   # Read first 61 characters of script!
echo "$var"
exit

####### Output of Script #######

#!/bin/bash
# Requires Bash version -ge 4.1 ...

num_chars=61
```

* Here documents embedded in **$(...)** command substitution constructs may terminate with a simple).

Example 37-9. Using a *here document* to set a variable

```
#!/bin/bash
# here-commsub.sh
# Requires Bash version -ge 4.1 ...

multi_line_var=$( cat <<ENDxxx
------------------------------
This is line 1 of the variable
This is line 2 of the variable
This is line 3 of the variable
------------------------------
ENDxxx)

#  Rather than what Bash 4.0 requires:
#+ that the terminating limit string and
#+ the terminating close-parenthesis be on separate lines.

# ENDxxx
# )


echo "$multi_line_var"

#  Bash still emits a warning, though.
#  warning: here-document at line 10 delimited
#+ by end-of-file (wanted `ENDxxx')
```

37.3.2. Bash, version 4.2

Version 4.2 of Bash, released in February, 2011, contains a number of new features and enhancements, in addition to bugfixes.

* Bash now supports the the *\u* and *\U Unicode* escape.

> Unicode is a cross-platform standard for encoding into numerical values letters and graphic symbols. This permits representing and displaying characters in foreign alphabets and unusual fonts.

```
echo -e '\u2630'   # Horizontal triple bar character.
# Equivalent to the more roundabout:
echo -e "\xE2\x98\xB0"
                   # Recognized by earlier Bash versions.

echo -e '\u220F'   # PI (Greek letter and mathematical symbol)
echo -e '\u0416'   # Capital "ZHE" (Cyrillic letter)
echo -e '\u2708'   # Airplane (Dingbat font) symbol
echo -e '\u2622'   # Radioactivity trefoil

echo -e "The amplifier circuit requires a 100 \u2126 pull-up resistor."
```

```
unicode_var='\u2640'
echo -e $unicode_var      # Female symbol
printf "$unicode_var \n"  # Female symbol, with newline


#  And for something a bit more elaborate . . .

#  We can store Unicode symbols in an associative array,
#+ then retrieve them by name.
#  Run this in a gnome-terminal or a terminal with a large, bold font
#+ for better legibility.

declare -A symbol  # Associative array.

symbol[script_E]='\u2130'
symbol[script_F]='\u2131'
symbol[script_J]='\u2110'
symbol[script_M]='\u2133'
symbol[Rx]='\u211E'
symbol[TEL]='\u2121'
symbol[FAX]='\u213B'
symbol[care_of]='\u2105'
symbol[account]='\u2100'
symbol[trademark]='\u2122'


echo -ne "${symbol[script_E]}   "
echo -ne "${symbol[script_F]}   "
echo -ne "${symbol[script_J]}   "
echo -ne "${symbol[script_M]}   "
echo -ne "${symbol[Rx]}   "
echo -ne "${symbol[TEL]}   "
echo -ne "${symbol[FAX]}   "
echo -ne "${symbol[care_of]}   "
echo -ne "${symbol[account]}   "
echo -ne "${symbol[trademark]}   "
echo
```

The above example uses the **$' ... '** *string-expansion* construct.

- When the `lastpipe` shell option is set, the last command in a pipe *doesn't run in a subshell*.

Example 37-10. Piping input to a read

```
#!/bin/bash
# lastpipe-option.sh

line=''                   # Null value.
echo "\$line = "$line""   # $line =

echo

shopt -s lastpipe         # Error on Bash version -lt 4.2.
echo "Exit status of attempting to set \"lastpipe\" option is $?"
#     1 if Bash version -lt 4.2, 0 otherwise.

echo
```

```
head -1 $0 | read line    # Pipe the first line of the script to read.
#            ^^^^^^^^^      Not in a subshell!!!

echo "\$line = "$line""
# Older Bash releases       $line =
# Bash version 4.2          $line = #!/bin/bash
```

This option offers possible "fixups" for these example scripts: Example 34-3 and Example 15-8.

- Negative array indices permit counting backwards from the end of an array.

Example 37-11. Negative array indices

```
#!/bin/bash
# neg-array.sh
# Requires Bash, version -ge 4.2.

array=( zero one two three four five )   # Six-element array.
#         0    1   2    3    4    5
#        -6   -5  -4   -3   -2   -1

# Negative array indices now permitted.
echo ${array[-1]}   # five
echo ${array[-2]}   # four
# ...
echo ${array[-6]}   # zero
# Negative array indices count backward from the last element+1.

# But, you cannot index past the beginning of the array.
echo ${array[-7]}   # array: bad array subscript


# So, what is this new feature good for?

echo "The last element in the array is "${array[-1]}""
# Which is quite a bit more straightforward than:
echo "The last element in the array is "${array[${#array[*]}-1]}""
echo

# And ...

index=0
let "neg_element_count = 0 - ${#array[*]}"
# Number of elements, converted to a negative number.

while [ $index -gt $neg_element_count ]; do
  ((index--)); echo -n "${array[index]} "
done  # Lists the elements in the array, backwards.
      # We have just simulated the "tac" command on this array.

echo

# See also neg-offset.sh.
```

- Substring extraction uses a negative *length* parameter to specify an offset from the *end* of the target string.

Example 37-12. Negative parameter in string-extraction construct

```
#!/bin/bash
# Bash, version -ge 4.2
# Negative length-index in substring extraction.
```

```
# Important: It changes the interpretation of this construct!

stringZ=abcABC123ABCabc

echo ${stringZ}                              # abcABC123ABCabc
#                   Position within string:    0123456789.....
echo ${stringZ:2:3}                          #   cAB
#  Count 2 chars forward from string beginning, and extract 3 chars.
#  ${string:position:length}

#  So far, nothing new, but now ...

                                             # abcABC123ABCabc
#                   Position within string:    0123....6543210
echo ${stringZ:3:-6}                         #    ABC123
#                ^
#  Index 3 chars forward from beginning and 6 chars backward from end,
#+ and extract everything in between.
#  ${string:offset-from-front:offset-from-end}
#  When the "length" parameter is negative,
#+ it serves as an offset-from-end parameter.

#  See also neg-array.sh.
```

Chapter 38. Endnotes

38.1. Author's Note

doce ut discas

(Teach, that you yourself may learn.)

How did I come to write a scripting book? It's a strange tale. It seems that a few years back I needed to learn shell scripting -- and what better way to do that than to read a good book on the subject? I was looking to buy a tutorial and reference covering all aspects of the subject. I was looking for a book that would take difficult concepts, turn them inside out, and explain them in excruciating detail, with well-commented examples. [135] In fact, I was looking for *this very book*, or something very much like it. Unfortunately, it didn't exist, and if I wanted it, I'd have to write it. And so, here we are, folks.

That reminds me of the apocryphal story about a mad professor. Crazy as a loon, the fellow was. At the sight of a book, any book -- at the library, at a bookstore, anywhere -- he would become totally obsessed with the idea that he could have written it, should have written it -- and done a better job of it to boot. He would thereupon rush home and proceed to do just that, write a book with the very same title. When he died some years later, he allegedly had several thousand books to his credit, probably putting even Asimov to shame. The books might not have been any good, who knows, but does that really matter? Here's a fellow who lived his dream, even if he was obsessed by it, driven by it . . . and somehow I can't help admiring the old coot.

38.2. About the Author

Who is this guy anyhow?

The author claims no credentials or special qualifications, [136] other than a compulsion to write. [137]

This book is somewhat of a departure from his other major work, HOW-2 Meet Women: The Shy Man's Guide to Relationships. He has also written the Software-Building HOWTO. Of late, he has been trying his (heavy) hand at fiction: Dave Dawson Over Berlin (First Installment) Dave Dawson Over Berlin (Second Installment) and Dave Dawson Over Berlin (Third Installment) . He also has a few *Instructables* (here, here, here, here, here, here, and here to his (dis)credit.

A Linux user since 1995 (Slackware 2.2, kernel 1.2.1), the author has emitted a few software truffles, including the cruft one-time pad encryption utility, the mcalc mortgage calculator, the judge Scrabble® adjudicator, the yawl word gaming list package, and the Quacky anagramming gaming package. He got off to a rather shaky start in the computer game -- programming FORTRAN IV on a CDC 3800 (on paper coding pads, with occasional forays on a keypunch machine and a Friden Flexowriter) -- and is not the least bit nostalgic for those days.

Living in an out-of-the-way community with wife and orange tabby, he cherishes human frailty, especially his own. [138]

38.3. Where to Go For Help

The author is no longer supporting or updating this document. He will not answer questions about this book or about general scripting topics.

> If you need assistance with a schoolwork assignment, read the pertinent sections of this and other reference works. Do your best to solve the problem using your own wits and resources. *Please do not waste the author's time.* You will get neither help nor sympathy. [139]
>
> Likewise, kindly refrain from annoying the author with solicitations, offers of employment, or "business opportunities." He is doing just fine, and requires neither help nor sympathy, thank you.
>
> Please note that the author will *not* answer scripting questions for Sun/Solaris/Oracle or Apple systems. The endarkened execs and the arachnoid corporate attorneys of those particular outfits have been using litigation in a predatory manner and/or as a weapon against the Open Source Community. Any Solaris or Apple users needing scripting help will therefore kindly direct their concerns to corporate customer service.

... sophisticated in mechanism but possibly agile operating under noises being extremely suppressed ...

--CI-300 printer manual

38.4. Tools Used to Produce This Book

38.4.1. Hardware

A used IBM Thinkpad, model 760XL laptop (P166, 104 meg RAM) running Red Hat 7.1/7.3. Sure, it's slow and has a funky keyboard, but it beats the heck out of a No. 2 pencil and a Big Chief tablet.

Update: upgraded to a 770Z Thinkpad (P2-366, 192 meg RAM) running FC3. Anyone feel like donating a later-model laptop to a starving writer <g>?

Update: upgraded to a T61 Thinkpad running Mandriva 2011. No longer starving <g>, but not too proud to accept donations.

38.4.2. Software and Printware

i. Bram Moolenaar's powerful SGML-aware vim text editor.
ii. OpenJade, a DSSSL rendering engine for converting SGML documents into other formats.
iii. Norman Walsh's DSSSL stylesheets.
iv. *DocBook, The Definitive Guide*, by Norman Walsh and Leonard Muellner (O'Reilly, ISBN 1-56592-580-7). This is still the standard reference for anyone attempting to write a document in Docbook SGML format.

38.5. Credits

Community participation made this project possible. The author gratefully acknowledges that writing this book would have been unthinkable without help and feedback from all you people out there.

Philippe Martin translated the first version (0.1) of this document into DocBook/SGML. While not on the job at a small French company as a software developer, he enjoys working on GNU/Linux documentation and software, reading literature, playing music, and, for his peace of mind, making merry with friends. You may run across him somewhere in France or in the Basque Country, or you can email him at feloy@free.fr.

Philippe Martin also pointed out that positional parameters past $9 are possible using {bracket} notation. (See Example 4-5).

Stéphane Chazelas sent a long list of corrections, additions, and example scripts. More than a contributor, he had, in effect, for a while taken on the role of *co-editor* for this document. *Merci beaucoup!*

Paulo Marcel Coelho Aragao offered many corrections, both major and minor, and contributed quite a number of helpful suggestions.

I would like to especially thank *Patrick Callahan*, *Mike Novak*, and *Pal Domokos* for catching bugs, pointing out ambiguities, and for suggesting clarifications and changes in the preliminary version (0.1) of this document. Their lively discussion of shell scripting and general documentation issues inspired me to try to make this document more readable.

I'm grateful to Jim Van Zandt for pointing out errors and omissions in version 0.2 of this document. He also contributed an instructive example script.

Many thanks to Jordi Sanfeliu for giving permission to use his fine tree script (Example A-16), and to Rick Boivie for revising it.

Likewise, thanks to Michel Charpentier for permission to use his dc factoring script (Example 16-52).

Kudos to Noah Friedman for permission to use his string function script (Example A-18).

Emmanuel Rouat suggested corrections and additions on command substitution, aliases, and path management. He also contributed a very nice sample `.bashrc` file (Appendix M).

Heiner Steven kindly gave permission to use his base conversion script, Example 16-48. He also made a number of corrections and many helpful suggestions. Special thanks.

Rick Boivie contributed the delightfully recursive *pb.sh* script (Example 36-11), revised the *tree.sh* script (Example A-16), and suggested performance improvements for the *monthlypmt.sh* script (Example 16-47).

Florian Wisser enlightened me on some of the fine points of testing strings (see Example 7-6), and on other matters.

Oleg Philon sent suggestions concerning cut and pidof.

Michael Zick extended the empty array example to demonstrate some surprising array properties. He also contributed the *isspammer* scripts (Example 16-41 and Example A-28).

Marc-Jano Knopp sent corrections and clarifications on DOS batch files.

Hyun Jin Cha found several typos in the document in the process of doing a Korean translation. Thanks for pointing these out.

Andreas Abraham sent in a long list of typographical errors and other corrections. Special thanks!

Others contributing scripts, making helpful suggestions, and pointing out errors were Gabor Kiss, Leopold Toetsch, Peter Tillier, Marcus Berglof, Tony Richardson, Nick Drage (script ideas!), Rich Bartell, Jess Thrysoee, Adam Lazur, Bram Moolenaar, Baris Cicek, Greg Keraunen, Keith Matthews, Sandro Magi, Albert Reiner, Dim Segebart, Rory Winston, Lee Bigelow, Wayne Pollock, "jipe," "bojster," "nyal," "Hobbit," "Ender," "Little Monster" (Alexis), "Mark," "Patsie," "vladz," Peggy Russell, Emilio Conti, Ian. D. Allen, Hans-Joerg Diers, Arun Giridhar, Dennis Leeuw, Dan Jacobson, Aurelio Marinho Jargas, Edward Scholtz, Jean Helou, Chris Martin, Lee Maschmeyer, Bruno Haible, Wilbert Berendsen, Sebastien Godard, Bjön Eriksson, John MacDonald, John Lange, Joshua Tschida, Troy Engel, Manfred Schwarb, Amit Singh, Bill Gradwohl, E. Choroba, David Lombard, Jason Parker, Steve Parker, Bruce W. Clare, William Park, Vernia Damiano, Mihai Maties, Mark Alexander, Jeremy Impson, Ken Fuchs, Jared Martin, Frank Wang, Sylvain Fourmanoit, Matthew Sage, Matthew Walker, Kenny Stauffer, Filip Moritz, Andrzej Stefanski, Daniel Albers, Jeffrey Haemer, Stefano Palmeri, Nils Radtke, Sigurd Solaas, Serghey Rodin, Jeroen Domburg, Alfredo Pironti, Phil Braham, Bruno de Oliveira Schneider, Stefano Falsetto, Chris Morgan, Walter Dnes, Linc Fessenden, Michael Iatrou, Pharis Monalo, Jesse Gough, Fabian Kreutz, Mark Norman, Harald Koenig, Dan Stromberg, Peter Knowles, Francisco Lobo, Mariusz Gniazdowski, Sebastian Arming, Chetankumar Phulpagare, Benno Schulenberg, Tedman Eng, Jochen DeSmet, Juan Nicolas Ruiz, Oliver Beckstein, Achmed Darwish, Dotan Barak, Richard Neill, Albert Siersema, Omair Eshkenazi, Geoff Lee, Graham Ewart, JuanJo Ciarlante, Cliff Bamford, Nathan Coulter, Ramses Rodriguez Martinez, Evgeniy Ivanov, Craig Barnes, George Dimitriu, Kevin LeBlanc, Antonio Macchi, Tomas Pospisek, David Wheeler, Erik Brandsberg, YongYe, Andreas Kühne, Pádraig Brady, Joseph Steinhauser, and David Lawyer (himself an author of four HOWTOs).

My gratitude to Chet Ramey and Brian Fox for writing *Bash*, and building into it elegant and powerful scripting capabilities rivaling those of *ksh*.

Very special thanks to the hard-working volunteers at the Linux Documentation Project. The LDP hosts a repository of Linux knowledge and lore, and has, to a great extent, enabled the publication of this book.

Thanks and appreciation to IBM, Red Hat, Google, the Free Software Foundation, and all the good people fighting the good fight to keep Open Source software free and open.

Belated thanks to my fourth grade teacher, Miss Spencer, for emotional support and for convincing me that maybe, just maybe I wasn't a total loss.

Thanks most of all to my wife, Anita, for her encouragement, inspiration, and emotional support.

38.6. Disclaimer

(This is a variant of the standard LDP disclaimer.)

No liability for the contents of this document can be accepted. Use the concepts, examples and information at your own risk. There may be errors, omissions, and inaccuracies that could cause you to lose data, harm your system, or induce involuntary electrocution, so *proceed with appropriate caution*. The author takes no responsibility for any damages, incidental or otherwise.

As it happens, it is highly unlikely that either you or your system will suffer ill effects, aside from uncontrollable hiccups. In fact, the *raison d'etre* of this book is to enable its readers to analyze shell scripts and determine whether they have unanticipated consequences.

Bibliography

Those who do not understand UNIX are condemned to reinvent it, poorly.

--Henry Spencer

Edited by Peter Denning, *Computers Under Attack: Intruders, Worms, and Viruses*, ACM Press, 1990, 0-201-53067-8.

This compendium contains a couple of articles on shell script viruses.

*

Ken Burtch, *Linux Shell Scripting with Bash*, 1st edition, Sams Publishing (Pearson), 2004, 0672326426.

Covers much of the same material as the *ABS Guide*, though in a different style.

*

Daniel Goldman, *Definitive Guide to Sed*, 1st edition, 2013.

This ebook is an excellent introduction to *sed*. Rather than being a conversion from a printed volume, it was specifically designed and formatted for viewing on an ebook reader. Well-written, informative, and useful as a reference as well as a tutorial. Highly recommended.

*

Dale Dougherty and Arnold Robbins, *Sed and Awk*, 2nd edition, O'Reilly and Associates, 1997, 1-156592-225-5.

Unfolding the full power of shell scripting requires at least a passing familiarity with *sed* and *awk*. This is the classic tutorial. It includes an excellent introduction to *Regular Expressions*. Recommended.

*

Jeffrey Friedl, *Mastering Regular Expressions*, O'Reilly and Associates, 2002, 0-596-00289-0.

Still the best all-around reference on Regular Expressions.

*

Aeleen Frisch, *Essential System Administration*, 3rd edition, O'Reilly and Associates, 2002, 0-596-00343-9.

This excellent manual provides a decent introduction to shell scripting from a sys admin point of view. It includes comprehensive explanations of the startup and initialization scripts in a UNIX system.

*

Stephen Kochan and Patrick Wood, *Unix Shell Programming*, Hayden, 1990, 067248448X.

Still considered a standard reference, though somewhat dated, and a bit "wooden" stylistically speaking. [140] In fact, this book was the *ABS Guide* author's first exposure to UNIX shell scripting, lo these many years ago.

*

Neil Matthew and Richard Stones, *Beginning Linux Programming*, Wrox Press, 1996, 1874416680.

Surprisingly good in-depth coverage of various programming languages available for Linux, including a fairly strong chapter on shell scripting.

*

Herbert Mayer, *Advanced C Programming on the IBM PC*, Windcrest Books, 1989, 0830693637.

Excellent coverage of algorithms and general programming practices. Highly recommended, but unfortunately out of print.

*

David Medinets, *Unix Shell Programming Tools*, McGraw-Hill, 1999, 0070397333.

Pretty good treatment of shell scripting, with examples, and a short intro to Tcl and Perl.

*

Cameron Newham and Bill Rosenblatt, *Learning the Bash Shell*, 2nd edition, O'Reilly and Associates, 1998, 1-56592-347-2.

This is a valiant effort at a decent shell primer, but sadly deficient in its coverage of writing scripts and lacking sufficient examples.

*

Anatole Olczak, *Bourne Shell Quick Reference Guide*, ASP, Inc., 1991, 093573922X.

A very handy pocket reference, despite lacking coverage of Bash-specific features.

*

Jerry Peek, Tim O'Reilly, and Mike Loukides, *Unix Power Tools*, 3rd edition, O'Reilly and Associates, Random House, 2002, 0-596-00330-7.

Contains a couple of sections of very informative in-depth articles on shell programming, but falls short of being a self-teaching manual. It reproduces much of the Regular Expressions tutorial from the Dougherty and Robbins book, above. The comprehensive coverage of UNIX commands makes this book worthy of a place on your bookshelf.

*

Clifford Pickover, *Computers, Pattern, Chaos, and Beauty*, St. Martin's Press, 1990, 0-312-04123-3.

A treasure trove of ideas and recipes for computer-based exploration of mathematical oddities.

*

George Polya, *How To Solve It*, Princeton University Press, 1973, 0-691-02356-5.

The classic tutorial on problem-solving methods (algorithms), with special emphasis on how to teach them.

*

Chet Ramey and Brian Fox, *The GNU Bash Reference Manual*, Network Theory Ltd, 2003, 0-9541617-7-7.

This manual is the definitive reference for GNU Bash. The authors of this manual, Chet Ramey and Brian Fox, are the original developers of GNU Bash. For each copy sold, the publisher donates $1 to the Free Software Foundation.

*

Arnold Robbins, *Bash Reference Card*, SSC, 1998, 1-58731-010-5.

Excellent Bash pocket reference (don't leave home without it, especially if you're a sysadmin). A bargain at $4.95, but unfortunately no longer available for free download.

*

Arnold Robbins, *Effective Awk Programming*, Free Software Foundation / O'Reilly and Associates, 2000, 1-882114-26-4.

The absolute best awk tutorial and reference. The free electronic version of this book is part of the *awk* documentation, and printed copies of the latest version are available from O'Reilly and Associates.

This book served as an inspiration for the author of the *ABS Guide*.

*

Bill Rosenblatt, *Learning the Korn Shell*, O'Reilly and Associates, 1993, 1-56592-054-6.

This well-written book contains some excellent pointers on shell scripting in general.

*

Paul Sheer, *LINUX: Rute User's Tutorial and Exposition*, 1st edition, , 2002, 0-13-033351-4.

Very detailed and readable introduction to Linux system administration.

The book is available in print, or on-line.

*

Ellen Siever and the staff of O'Reilly and Associates, *Linux in a Nutshell*, 2nd edition, O'Reilly and Associates, 1999, 1-56592-585-8.

The all-around best Linux command reference. It even has a Bash section.

*

Dave Taylor, *Wicked Cool Shell Scripts: 101 Scripts for Linux, Mac OS X, and Unix Systems*, 1st edition, No Starch Press, 2004, 1-59327-012-7.

Pretty much what the title promises . . .

*

The UNIX CD Bookshelf, 3rd edition, O'Reilly and Associates, 2003, 0-596-00392-7.

An array of seven UNIX books on CD ROM, including *UNIX Power Tools*, *Sed and Awk*, and *Learning the Korn Shell*. A complete set of all the UNIX references and tutorials you would ever need at about $130. Buy this one, even if it means going into debt and not paying the rent.

Update: Seems to have somehow fallen out of print. Ah, well. You can still buy the dead-tree editions of these books.

*

The O'Reilly books on Perl. (Actually, *any* O'Reilly books.)

* * *

Other Resources

Fioretti, Marco, "Scripting for X Productivity," *Linux Journal*, Issue 113, September, 2003, pp. 86-9.

Ben Okopnik's well-written *introductory Bash scripting* articles in issues 53, 54, 55, 57, and 59 of the *Linux Gazette*, and his explanation of "The Deep, Dark Secrets of Bash" in issue 56.

Chet Ramey's *Bash - The GNU Shell*, a two-part series published in issues 3 and 4 of the *Linux Journal*, July-August 1994.

Mike G's Bash-Programming-Intro HOWTO.

Richard's Unix Scripting Universe.

Chet Ramey's Bash FAQ.

Greg's WIKI: Bash FAQ.

Example shell scripts at Lucc's Shell Scripts .

Example shell scripts at SHELLdorado .

Example shell scripts at Noah Friedman's script site.

Examples from the *The Bash Scripting Cookbook*, by Albing, Vossen, and Newham.

Example shell scripts at zazzybob.

Steve Parker's Shell Programming Stuff. In fact, all of his shell scripting books are highly recommended. See also Steve's Arcade Games written in a shell script.

An excellent collection of Bash scripting tips, tricks, and resources at the Bash Hackers Wiki.

Giles Orr's Bash-Prompt HOWTO.

The *Pixelbeat* command-line reference.

Very nice **sed**, **awk**, and regular expression tutorials at The UNIX Grymoire.

The GNU sed and gawk manuals. As you recall, gawk is the enhanced GNU version of **awk**.

Many interesting sed scripts at the seder's grab bag.

Tips and tricks at Linux Reviews.

Trent Fisher's groff tutorial.

David Wheeler's Filenames in Shell essay.

"Shelltris" and "shellitaire" at Shell Script Games.

YongYe's wonderfully complex Tetris game script.

Mark Komarinski's Printing-Usage HOWTO.

The Linux USB subsystem (helpful in writing scripts affecting USB peripherals).

There is some nice material on I/O redirection in chapter 10 of the textutils documentation at the University of Alberta site.

Rick Hohensee has written the *osimpa* i386 assembler entirely as Bash scripts.

dgatwood has a very nice shell script games site, featuring a Tetris® clone and solitaire.

Aurelio Marinho Jargas has written a Regular expression wizard. He has also written an informative book on Regular Expressions, in Portuguese.

Ben Tomkins has created the Bash Navigator directory management tool.

William Park has been working on a project to incorporate certain *Awk* and *Python* features into Bash. Among these is a *gdbm* interface. He has released *bashdiff* on Freshmeat.net. He has an article in the November, 2004 issue of the *Linux Gazette* on adding string functions to Bash, with a followup article in the December issue, and yet another in the January, 2005 issue.

Peter Knowles has written an elaborate Bash script that generates a book list on the Sony Librie e-book reader. This useful tool facilitates loading non-DRM user content on the *Librie* (and the newer *PRS-xxx-series* devices).

Tim Waugh's xmlto is an elaborate Bash script for converting Docbook XML documents to other formats.

Philip Patterson's logforbash logging/debugging script.

AuctionGallery, an application for eBay "power sellers" coded in Bash.

Of historical interest are Colin Needham's *original International Movie Database (IMDB) reader polling scripts*, which nicely illustrate the use of awk for string parsing. Unfortunately, the URL link is broken.

Fritz Mehner has written a bash-support plugin for the *vim* text editor. He has also also come up with his own stylesheet for Bash. Compare it with the ABS Guide Unofficial Stylesheet.

Penguin Pete has quite a number of shell scripting tips and hints on his superb site. Highly recommended.

The excellent *Bash Reference Manual*, by Chet Ramey and Brian Fox, distributed as part of the *bash-2-doc* package (available as an rpm). See especially the instructive example scripts in this package.

John Lion's classic, *A Commentary on the Sixth Edition UNIX Operating System*.

The comp.os.unix.shell newsgroup.

The *dd* thread on Linux Questions.

The comp.os.unix.shell FAQ.

Assorted comp.os.unix FAQs.

The *Wikipedia* article covering dc.

The manpages for **bash** and **bash2**, **date**, **expect**, **expr**, **find**, **grep**, **gzip**, **ln**, **patch**, **tar**, **tr**, **bc**, **xargs**. The *texinfo* documentation on **bash**, **dd**, **m4**, **gawk**, and **sed**.

Appendix A. Contributed Scripts

These scripts, while not fitting into the text of this document, do illustrate some interesting shell programming techniques. Some are useful, too. Have fun analyzing and running them.

Example A-1. *mailformat*: Formatting an e-mail message

```
#!/bin/bash
# mail-format.sh (ver. 1.1): Format e-mail messages.

# Gets rid of carets, tabs, and also folds excessively long lines.

# =================================================================
#                 Standard Check for Script Argument(s)
ARGS=1
E_BADARGS=85
E_NOFILE=86

if [ $# -ne $ARGS ]  # Correct number of arguments passed to script?
then
  echo "Usage: `basename $0` filename"
  exit $E_BADARGS
fi

if [ -f "$1" ]       # Check if file exists.
then
    file_name=$1
else
    echo "File \"$1\" does not exist."
    exit $E_NOFILE
fi
# -----------------------------------------------------------------

MAXWIDTH=70          # Width to fold excessively long lines to.

# =================================
# A variable can hold a sed script.
# It's a useful technique.
sedscript='s/^>//
s/^  *>//
s/^  *//
s/              *//'
# =================================

#  Delete carets and tabs at beginning of lines,
#+ then fold lines to $MAXWIDTH characters.
sed "$sedscript" $1 | fold -s --width=$MAXWIDTH
                        #  -s option to "fold"
                        #+ breaks lines at whitespace, if possible.


#  This script was inspired by an article in a well-known trade journal
#+ extolling a 164K MS Windows utility with similar functionality.
#
#  An nice set of text processing utilities and an efficient
#+ scripting language provide an alternative to the bloated executables
#+ of a clunky operating system.
```

```
exit $?
```

Example A-2. *rn*: A simple-minded file renaming utility

This script is a modification of Example 16-22.

```
#! /bin/bash
# rn.sh

# Very simpleminded filename "rename" utility (based on "lowercase.sh").
#
#  The "ren" utility, by Vladimir Lanin (lanin@csd2.nyu.edu),
#+ does a much better job of this.


ARGS=2
E_BADARGS=85
ONE=1                     # For getting singular/plural right (see below).

if [ $# -ne "$ARGS" ]
then
  echo "Usage: `basename $0` old-pattern new-pattern"
  # As in "rn gif jpg", which renames all gif files in working directory to jpg.
  exit $E_BADARGS
fi

number=0                  # Keeps track of how many files actually renamed.


for filename in *$1*      #Traverse all matching files in directory.
do
   if [ -f "$filename" ]  # If finds match...
   then
     fname=`basename $filename`            # Strip off path.
     n=`echo $fname | sed -e "s/$1/$2/"`   # Substitute new for old in filename.
     mv $fname $n                          # Rename.
     let "number += 1"
   fi
done

if [ "$number" -eq "$ONE" ]                # For correct grammar.
then
 echo "$number file renamed."
else
 echo "$number files renamed."
fi

exit $?


# Exercises:
# ---------
# What types of files will this not work on?
# How can this be fixed?
```

Example A-3. *blank-rename*: Renames filenames containing blanks

This is an even simpler-minded version of previous script.

```
#! /bin/bash
# blank-rename.sh
#
# Substitutes underscores for blanks in all the filenames in a directory.

ONE=1                      # For getting singular/plural right (see below).
number=0                   # Keeps track of how many files actually renamed.
FOUND=0                    # Successful return value.

for filename in *          #Traverse all files in directory.
do
     echo "$filename" | grep -q " "         #  Check whether filename
     if [ $? -eq $FOUND ]                   #+ contains space(s).
     then
       fname=$filename                      # Yes, this filename needs work.
       n=`echo $fname | sed -e "s/ /_/g"`   # Substitute underscore for blank.
       mv "$fname" "$n"                     # Do the actual renaming.
       let "number += 1"
     fi
done

if [ "$number" -eq "$ONE" ]                 # For correct grammar.
then
 echo "$number file renamed."
else
 echo "$number files renamed."
fi

exit 0
```

Example A-4. *encryptedpw*: Uploading to an ftp site, using a locally encrypted password

```
#!/bin/bash

# Example "ex72.sh" modified to use encrypted password.

#  Note that this is still rather insecure,
#+ since the decrypted password is sent in the clear.
#  Use something like "ssh" if this is a concern.

E_BADARGS=85

if [ -z "$1" ]
then
  echo "Usage: `basename $0` filename"
  exit $E_BADARGS
fi

Username=bozo           # Change to suit.
pword=/home/bozo/secret/password_encrypted.file
# File containing encrypted password.

Filename=`basename $1`  # Strips pathname out of file name.

Server="XXX"
Directory="YYY"         # Change above to actual server name & directory.


Password=`cruft <$pword`          # Decrypt password.
#  Uses the author's own "cruft" file encryption package,
#+ based on the classic "onetime pad" algorithm,
```

```
#+ and obtainable from:
#+ Primary-site:   ftp://ibiblio.org/pub/Linux/utils/file
#+                 cruft-0.2.tar.gz [16k]


ftp -n $Server <<End-Of-Session
user $Username $Password
binary
bell
cd $Directory
put $Filename
bye
End-Of-Session
# -n option to "ftp" disables auto-logon.
# Note that "bell" rings 'bell' after each file transfer.

exit 0
```

Example A-5. *copy-cd*: Copying a data CD

```
#!/bin/bash
# copy-cd.sh: copying a data CD

CDROM=/dev/cdrom                           # CD ROM device
OF=/home/bozo/projects/cdimage.iso         # output file
#       /xxxx/xxxxxxxx/                      Change to suit your system.
BLOCKSIZE=2048
# SPEED=10                                 # If unspecified, uses max spd.
# DEVICE=/dev/cdrom                          older version.
DEVICE="1,0,0"

echo; echo "Insert source CD, but do *not* mount it."
echo "Press ENTER when ready. "
read ready                                 # Wait for input, $ready not used.

echo; echo "Copying the source CD to $OF."
echo "This may take a while. Please be patient."

dd if=$CDROM of=$OF bs=$BLOCKSIZE          # Raw device copy.


echo; echo "Remove data CD."
echo "Insert blank CDR."
echo "Press ENTER when ready. "
read ready                                 # Wait for input, $ready not used.

echo "Copying $OF to CDR."

# cdrecord -v -isosize speed=$SPEED dev=$DEVICE $OF   # Old version.
wodim -v -isosize dev=$DEVICE $OF
# Uses Joerg Schilling's "cdrecord" package (see its docs).
# http://www.fokus.gmd.de/nthp/employees/schilling/cdrecord.html
# Newer Linux distros may use "wodim" rather than "cdrecord" ...


echo; echo "Done copying $OF to CDR on device $CDROM."

echo "Do you want to erase the image file (y/n)? "  # Probably a huge file.
read answer

case "$answer" in
```

```
[yY]) rm -f $OF
      echo "$OF erased."
      ;;
*)    echo "$OF not erased.";;
esac

echo

# Exercise:
# Change the above "case" statement to also accept "yes" and "Yes" as input.

exit 0
```

Example A-6. Collatz series

```
#!/bin/bash
# collatz.sh

#  The notorious "hailstone" or Collatz series.
#  -------------------------------------------
#  1) Get the integer "seed" from the command-line.
#  2) NUMBER <-- seed
#  3) Print NUMBER.
#  4)  If NUMBER is even, divide by 2, or
#  5)+ if odd, multiply by 3 and add 1.
#  6) NUMBER <-- result
#  7) Loop back to step 3 (for specified number of iterations).
#
#  The theory is that every such sequence,
#+ no matter how large the initial value,
#+ eventually settles down to repeating "4,2,1..." cycles,
#+ even after fluctuating through a wide range of values.
#
#  This is an instance of an "iterate,"
#+ an operation that feeds its output back into its input.
#  Sometimes the result is a "chaotic" series.


MAX_ITERATIONS=200
# For large seed numbers (>32000), try increasing MAX_ITERATIONS.

h=${1:-$$}                      #  Seed.
                                #  Use $PID as seed,
                                #+ if not specified as command-line arg.

echo
echo "C($h) -*- $MAX_ITERATIONS Iterations"
echo

for ((i=1; i<=MAX_ITERATIONS; i++))
do

# echo -n "$h   "
#             ^^^
#             tab
# printf does it better ...
COLWIDTH=%7d
printf $COLWIDTH $h

  let "remainder = h % 2"
  if [ "$remainder" -eq 0 ]   # Even?
```

```
  then
    let "h /= 2"              # Divide by 2.
  else
    let "h = h*3 + 1"         # Multiply by 3 and add 1.
  fi


COLUMNS=10                    # Output 10 values per line.
let "line_break = i % $COLUMNS"
if [ "$line_break" -eq 0 ]
then
  echo
fi

done

echo

#  For more information on this strange mathematical function,
#+ see _Computers, Pattern, Chaos, and Beauty_, by Pickover, p. 185 ff.,
#+ as listed in the bibliography.

exit 0
```

Example A-7. *days-between*: Days between two dates

```
#!/bin/bash
# days-between.sh:    Number of days between two dates.
# Usage: ./days-between.sh [M]M/[D]D/YYYY [M]M/[D]D/YYYY
#
# Note: Script modified to account for changes in Bash, v. 2.05b +,
#+      that closed the loophole permitting large negative
#+      integer return values.

ARGS=2                # Two command-line parameters expected.
E_PARAM_ERR=85        # Param error.

REFYR=1600            # Reference year.
CENTURY=100
DIY=365
ADJ_DIY=367           # Adjusted for leap year + fraction.
MIY=12
DIM=31
LEAPCYCLE=4

MAXRETVAL=255         #  Largest permissible
                      #+ positive return value from a function.

diff=                 # Declare global variable for date difference.
value=                # Declare global variable for absolute value.
day=                  # Declare globals for day, month, year.
month=
year=


Param_Error ()        # Command-line parameters wrong.
{
  echo "Usage: `basename $0` [M]M/[D]D/YYYY [M]M/[D]D/YYYY"
  echo "       (date must be after 1/3/1600)"
  exit $E_PARAM_ERR
```

```
}


Parse_Date ()                  # Parse date from command-line params.
{
  month=${1%%/**}
  dm=${1%/**}                  # Day and month.
  day=${dm#*/}
  let "year = `basename $1`"  # Not a filename, but works just the same.
}


check_date ()                  # Checks for invalid date(s) passed.
{
  [ "$day" -gt "$DIM" ] || [ "$month" -gt "$MIY" ] ||
  [ "$year" -lt "$REFYR" ] && Param_Error
  # Exit script on bad value(s).
  # Uses or-list / and-list.
  #
  # Exercise: Implement more rigorous date checking.
}


strip_leading_zero () #  Better to strip possible leading zero(s)
{                     #+ from day and/or month
  return ${1#0}       #+ since otherwise Bash will interpret them
}                     #+ as octal values (POSIX.2, sect 2.9.2.1).


day_index ()          # Gauss' Formula:
{                     # Days from March 1, 1600 to date passed as param.
                      #           ^^^^^^^^^^^^^
  day=$1
  month=$2
  year=$3

  let "month = $month - 2"
  if [ "$month" -le 0 ]
  then
    let "month += 12"
    let "year -= 1"
  fi

  let "year -= $REFYR"
  let "indexyr = $year / $CENTURY"


  let "Days = $DIY*$year + $year/$LEAPCYCLE - $indexyr \
              + $indexyr/$LEAPCYCLE + $ADJ_DIY*$month/$MIY + $day - $DIM"
  #  For an in-depth explanation of this algorithm, see
  #+   http://weblogs.asp.net/pgreborio/archive/2005/01/06/347968.aspx


  echo $Days

}


calculate_difference ()            # Difference between two day indices.
{
  let "diff = $1 - $2"             # Global variable.
}
```

```
abs ()                              #  Absolute value
{                                   #  Uses global "value" variable.
  if [ "$1" -lt 0 ]                 #  If negative
  then                              #+ then
    let "value = 0 - $1"            #+ change sign,
  else                              #+ else
    let "value = $1"                #+ leave it alone.
  fi
}



if [ $# -ne "$ARGS" ]               # Require two command-line params.
then
  Param_Error
fi

Parse_Date $1
check_date $day $month $year        #  See if valid date.

strip_leading_zero $day             #  Remove any leading zeroes
day=$?                              #+ on day and/or month.
strip_leading_zero $month
month=$?

let "date1 = `day_index $day $month $year`"


Parse_Date $2
check_date $day $month $year

strip_leading_zero $day
day=$?
strip_leading_zero $month
month=$?

date2=$(day_index $day $month $year) # Command substitution.


calculate_difference $date1 $date2

abs $diff                            # Make sure it's positive.
diff=$value

echo $diff

exit 0

#  Exercise:
#  --------
#  If given only one command-line parameter, have the script
#+ use today's date as the second.


#  Compare this script with
#+ the implementation of Gauss' Formula in a C program at
#+    http://buschencrew.hypermart.net/software/datedif
```

Example A-8. Making a *dictionary*

```
#!/bin/bash
# makedict.sh  [make dictionary]

# Modification of /usr/sbin/mkdict (/usr/sbin/cracklib-forman) script.
# Original script copyright 1993, by Alec Muffett.
#
#  This modified script included in this document in a manner
#+ consistent with the "LICENSE" document of the "Crack" package
#+ that the original script is a part of.

#  This script processes text files to produce a sorted list
#+ of words found in the files.
#  This may be useful for compiling dictionaries
#+ and for other lexicographic purposes.


E_BADARGS=85

if [ ! -r "$1" ]                    #  Need at least one
then                                #+ valid file argument.
  echo "Usage: $0 files-to-process"
  exit $E_BADARGS
fi


# SORT="sort"                       #  No longer necessary to define
                                    #+ options to sort. Changed from
                                    #+ original script.

cat $* |                            #  Dump specified files to stdout.
        tr A-Z a-z |                #  Convert to lowercase.
        tr ' ' '\012' |             #  New: change spaces to newlines.
#       tr -cd '\012[a-z][0-9]' |   #  Get rid of everything
                                    #+ non-alphanumeric (in orig. script).
        tr -c '\012a-z'  '\012' |   #  Rather than deleting non-alpha
                                    #+ chars, change them to newlines.
        sort |                      #  $SORT options unnecessary now.
        uniq |                      #  Remove duplicates.
        grep -v '^#' |              #  Delete lines starting with #.
        grep -v '^$'                #  Delete blank lines.

exit $?
```

Example A-9. Soundex conversion

```
#!/bin/bash
# soundex.sh: Calculate "soundex" code for names

# =======================================================
#        Soundex script
#              by
#         Mendel Cooper
#     thegrendel.abs@gmail.com
#     reldate: 23 January, 2002
#
#   Placed in the Public Domain.
#
# A slightly different version of this script appeared in
#+ Ed Schaefer's July, 2002 "Shell Corner" column
#+ in "Unix Review" on-line,
```

```
#+ http://www.unixreview.com/documents/uni1026336632258/
# =========================================================


ARGCOUNT=1                     # Need name as argument.
E_WRONGARGS=90

if [ $# -ne "$ARGCOUNT" ]
then
  echo "Usage: `basename $0` name"
  exit $E_WRONGARGS
fi


assign_value ()                #  Assigns numerical value
{                              #+ to letters of name.

  val1=bfpv                    # 'b,f,p,v' = 1
  val2=cgjkqsxz                # 'c,g,j,k,q,s,x,z' = 2
  val3=dt                      #  etc.
  val4=l
  val5=mn
  val6=r

# Exceptionally clever use of 'tr' follows.
# Try to figure out what is going on here.

value=$( echo "$1" \
| tr -d wh \
| tr $val1 1 | tr $val2 2 | tr $val3 3 \
| tr $val4 4 | tr $val5 5 | tr $val6 6 \
| tr -s 123456 \
| tr -d aeiouy )

# Assign letter values.
# Remove duplicate numbers, except when separated by vowels.
# Ignore vowels, except as separators, so delete them last.
# Ignore 'w' and 'h', even as separators, so delete them first.
#
# The above command substitution lays more pipe than a plumber <g>.

}


input_name="$1"
echo
echo "Name = $input_name"


# Change all characters of name input to lowercase.
# ------------------------------------------------
name=$( echo $input_name | tr A-Z a-z )
# ------------------------------------------------
# Just in case argument to script is mixed case.


# Prefix of soundex code: first letter of name.
# --------------------------------------------


char_pos=0                     # Initialize character position.
prefix0=${name:$char_pos:1}
```

```
prefix=`echo $prefix0 | tr a-z A-Z`
                               # Uppercase 1st letter of soundex.

let "char_pos += 1"            # Bump character position to 2nd letter of name.
name1=${name:$char_pos}


# ++++++++++++++++++++++++++ Exception Patch ++++++++++++++++++++++++++++++
#  Now, we run both the input name and the name shifted one char
#+ to the right through the value-assigning function.
#  If we get the same value out, that means that the first two characters
#+ of the name have the same value assigned, and that one should cancel.
#  However, we also need to test whether the first letter of the name
#+ is a vowel or 'w' or 'h', because otherwise this would bollix things up.

char1=`echo $prefix | tr A-Z a-z`    # First letter of name, lowercased.

assign_value $name
s1=$value
assign_value $name1
s2=$value
assign_value $char1
s3=$value
s3=9$s3                              #  If first letter of name is a vowel
                                     #+ or 'w' or 'h',
                                     #+ then its "value" will be null (unset).
                                     #+ Therefore, set it to 9, an otherwise
                                     #+ unused value, which can be tested for.


if [[ "$s1" -ne "$s2" || "$s3" -eq 9 ]]
then
  suffix=$s2
else
  suffix=${s2:$char_pos}
fi
# ++++++++++++++++++++++ end Exception Patch ++++++++++++++++++++++++++++++


padding=000                    # Use at most 3 zeroes to pad.


soun=$prefix$suffix$padding    # Pad with zeroes.

MAXLEN=4                       # Truncate to maximum of 4 chars.
soundex=${soun:0:$MAXLEN}

echo "Soundex = $soundex"

echo

#  The soundex code is a method of indexing and classifying names
#+ by grouping together the ones that sound alike.
#  The soundex code for a given name is the first letter of the name,
#+ followed by a calculated three-number code.
#  Similar sounding names should have almost the same soundex codes.

#   Examples:
#   Smith and Smythe both have a "S-530" soundex.
#   Harrison = H-625
#   Hargison = H-622
#   Harriman = H-655
```

```
#  This works out fairly well in practice, but there are numerous anomalies.
#
#
#  The U.S. Census and certain other governmental agencies use soundex,
#  as do genealogical researchers.
#
#  For more information,
#+ see the "National Archives and Records Administration home page",
#+ http://www.nara.gov/genealogy/soundex/soundex.html



# Exercise:
# --------
# Simplify the "Exception Patch" section of this script.

exit 0
```

Example A-10. *Game of Life*

```
#!/bin/bash
# life.sh: "Life in the Slow Lane"
# Author: Mendel Cooper
# License: GPL3

# Version 0.2:   Patched by Daniel Albers
#+               to allow non-square grids as input.
# Version 0.2.1: Added 2-second delay between generations.

# ###################################################################### #
# This is the Bash script version of John Conway's "Game of Life".      #
# "Life" is a simple implementation of cellular automata.               #
# ---------------------------------------------------------------------- #
# On a rectangular grid, let each "cell" be either "living" or "dead."  #
# Designate a living cell with a dot, and a dead one with a blank space.#
#      Begin with an arbitrarily drawn dot-and-blank grid,              #
#+     and let this be the starting generation: generation 0.           #
# Determine each successive generation by the following rules:          #
#   1) Each cell has 8 neighbors, the adjoining cells                   #
#+     left, right, top, bottom, and the 4 diagonals.                   #
#                                                                       #
#                       123                                             #
#                       4*5     The * is the cell under consideration.  #
#                       678                                             #
#                                                                       #
# 2) A living cell with either 2 or 3 living neighbors remains alive.   #
SURVIVE=2                                                               #
# 3) A dead cell with 3 living neighbors comes alive, a "birth."        #
BIRTH=3                                                                 #
# 4) All other cases result in a dead cell for the next generation.     #
# ###################################################################### #


startfile=gen0   # Read the starting generation from the file "gen0" ...
                 # Default, if no other file specified when invoking script.
                 #
if [ -n "$1" ]   # Specify another "generation 0" file.
then
    startfile="$1"
```

```
fi

############################################
#  Abort script if "startfile" not specified
#+ and
#+ default file "gen0" not present.

E_NOSTARTFILE=86

if [ ! -e "$startfile" ]
then
  echo "Startfile \""$startfile"\" missing!"
  exit $E_NOSTARTFILE
fi
############################################


ALIVE1=.
DEAD1=_
                 # Represent living and dead cells in the start-up file.

#  ---------------------------------------------------#
#  This script uses a 10 x 10 grid (may be increased,
#+ but a large grid will slow down execution).
ROWS=10
COLS=10
#  Change above two variables to match desired grid size.
#  ---------------------------------------------------#

GENERATIONS=10          #  How many generations to cycle through.
                        #  Adjust this upwards
                        #+ if you have time on your hands.

NONE_ALIVE=85           #  Exit status on premature bailout,
                        #+ if no cells left alive.
DELAY=2                 #  Pause between generations.
TRUE=0
FALSE=1
ALIVE=0
DEAD=1

avar=                   # Global; holds current generation.
generation=0            # Initialize generation count.

# =================================================================

let "cells = $ROWS * $COLS"   # How many cells.

# Arrays containing "cells."
declare -a initial
declare -a current

display ()
{

alive=0                 # How many cells alive at any given time.
                        # Initially zero.

declare -a arr
arr=( `echo "$1"` )     # Convert passed arg to array.

element_count=${#arr[*]}
```

```
local i
local rowcheck

for ((i=0; i<$element_count; i++))
do

  # Insert newline at end of each row.
  let "rowcheck = $i % COLS"
  if [ "$rowcheck" -eq 0 ]
  then
    echo                # Newline.
    echo -n "      "    # Indent.
  fi

  cell=${arr[i]}

  if [ "$cell" = . ]
  then
    let "alive += 1"
  fi

  echo -n "$cell" | sed -e 's/_/ /g'
  # Print out array, changing underscores to spaces.
done

return

}

IsValid ()                            # Test if cell coordinate valid.
{

  if [ -z "$1"  -o -z "$2" ]          # Mandatory arguments missing?
  then
    return $FALSE
  fi

local row
local lower_limit=0                   # Disallow negative coordinate.
local upper_limit
local left
local right

let "upper_limit = $ROWS * $COLS - 1" # Total number of cells.


if [ "$1" -lt "$lower_limit" -o "$1" -gt "$upper_limit" ]
then
  return $FALSE                       # Out of array bounds.
fi

row=$2
let "left = $row * $COLS"             # Left limit.
let "right = $left + $COLS - 1"       # Right limit.

if [ "$1" -lt "$left" -o "$1" -gt "$right" ]
then
  return $FALSE                       # Beyond row boundary.
fi

return $TRUE                          # Valid coordinate.
```

```
}


IsAlive ()              #  Test whether cell is alive.
                        #  Takes array, cell number, and
{                       #+ state of cell as arguments.
  GetCount "$1" $2      #  Get alive cell count in neighborhood.
  local nhbd=$?

  if [ "$nhbd" -eq "$BIRTH" ]  # Alive in any case.
  then
    return $ALIVE
  fi

  if [ "$3" = "." -a "$nhbd" -eq "$SURVIVE" ]
  then                  # Alive only if previously alive.
    return $ALIVE
  fi

  return $DEAD          # Defaults to dead.

}


GetCount ()             # Count live cells in passed cell's neighborhood.
                        # Two arguments needed:
                        # $1) variable holding array
                        # $2) cell number
{
  local cell_number=$2
  local array
  local top
  local center
  local bottom
  local r
  local row
  local i
  local t_top
  local t_cen
  local t_bot
  local count=0
  local ROW_NHBD=3

  array=( `echo "$1"` )

  let "top = $cell_number - $COLS - 1"    # Set up cell neighborhood.
  let "center = $cell_number - 1"
  let "bottom = $cell_number + $COLS - 1"
  let "r = $cell_number / $COLS"

  for ((i=0; i<$ROW_NHBD; i++))           # Traverse from left to right.
  do
    let "t_top = $top + $i"
    let "t_cen = $center + $i"
    let "t_bot = $bottom + $i"


    let "row = $r"                        # Count center row.
    IsValid $t_cen $row                   # Valid cell position?
    if [ $? -eq "$TRUE" ]
    then
```

```
      if [ ${array[$t_cen]} = "$ALIVE1" ] # Is it alive?
      then                                # If yes, then ...
        let "count += 1"                  # Increment count.
      fi
    fi

    let "row = $r - 1"                    # Count top row.
    IsValid $t_top $row
    if [ $? -eq "$TRUE" ]
    then
      if [ ${array[$t_top]} = "$ALIVE1" ] # Redundancy here.
      then                                # Can it be optimized?
        let "count += 1"
      fi
    fi

    let "row = $r + 1"                    # Count bottom row.
    IsValid $t_bot $row
    if [ $? -eq "$TRUE" ]
    then
      if [ ${array[$t_bot]} = "$ALIVE1" ]
      then
        let "count += 1"
      fi
    fi

  done


  if [ ${array[$cell_number]} = "$ALIVE1" ]
  then
    let "count -= 1"        #  Make sure value of tested cell itself
  fi                        #+ is not counted.


  return $count

}

next_gen ()               # Update generation array.
{

local array
local i=0

array=( `echo "$1"` )     # Convert passed arg to array.

while [ "$i" -lt "$cells" ]
do
  IsAlive "$1" $i ${array[$i]}   # Is the cell alive?
  if [ $? -eq "$ALIVE" ]
  then                           #  If alive, then
    array[$i]=.                  #+ represent the cell as a period.
  else
    array[$i]="_"                #  Otherwise underscore
   fi                            #+ (will later be converted to space).
  let "i += 1"
done


#    let "generation += 1"       # Increment generation count.
###  Why was the above line commented out?
```

```
# Set variable to pass as parameter to "display" function.
avar=`echo ${array[@]}`   # Convert array back to string variable.
display "$avar"           # Display it.
echo; echo
echo "Generation $generation  -  $alive alive"

if [ "$alive" -eq 0 ]
then
  echo
  echo "Premature exit: no more cells alive!"
  exit $NONE_ALIVE        #  No point in continuing
fi                        #+ if no live cells.

}


# =========================================================

# main ()
# {

# Load initial array with contents of startup file.
initial=( `cat "$startfile" | sed -e '/#/d' | tr -d '\n' |\
# Delete lines containing '#' comment character.
           sed -e 's/\./\. /g' -e 's/_/_ /g'` )
# Remove linefeeds and insert space between elements.

clear          # Clear screen.

echo #         Title
setterm -reverse on
echo "======================="
setterm -reverse off
echo "    $GENERATIONS generations"
echo "           of"
echo "\"Life in the Slow Lane\""
setterm -reverse on
echo "======================="
setterm -reverse off

sleep $DELAY   # Display "splash screen" for 2 seconds.


# -------- Display first generation. --------
Gen0=`echo ${initial[@]}`
display "$Gen0"           # Display only.
echo; echo
echo "Generation $generation  -  $alive alive"
sleep $DELAY
# -------------------------------------------


let "generation += 1"     # Bump generation count.
echo

# ------- Display second generation. -------
Cur=`echo ${initial[@]}`
next_gen "$Cur"          # Update & display.
sleep $DELAY
# ------------------------------------------
```

```
let "generation += 1"     # Increment generation count.

# ------ Main loop for displaying subsequent generations ------
while [ "$generation" -le "$GENERATIONS" ]
do
  Cur="$avar"
  next_gen "$Cur"
  let "generation += 1"
  sleep $DELAY
done
# ==============================================================

echo
# }

exit 0   # CEOF:EOF



# The grid in this script has a "boundary problem."
# The the top, bottom, and sides border on a void of dead cells.
# Exercise: Change the script to have the grid wrap around,
# +          so that the left and right sides will "touch,"
# +          as will the top and bottom.
#
# Exercise: Create a new "gen0" file to seed this script.
#           Use a 12 x 16 grid, instead of the original 10 x 10 one.
#           Make the necessary changes to the script,
#+          so it will run with the altered file.
#
# Exercise: Modify this script so that it can determine the grid size
#+          from the "gen0" file, and set any variables necessary
#+          for the script to run.
#           This would make unnecessary any changes to variables
#+          in the script for an altered grid size.
#
# Exercise: Optimize this script.
#           It has redundant code.
```

Example A-11. Data file for *Game of Life*

```
# gen0
#
# This is an example "generation 0" start-up file for "life.sh".
# --------------------------------------------------------------
#  The "gen0" file is a 10 x 10 grid using a period (.) for live cells,
#+ and an underscore (_) for dead ones. We cannot simply use spaces
#+ for dead cells in this file because of a peculiarity in Bash arrays.
#  [Exercise for the reader: explain this.]
#
# Lines beginning with a '#' are comments, and the script ignores them.
__.__..___
__.._.____
____.___..
_._______.
____._____
..__...___
____._____
___...____
__.._..___
```

```
_..___..__
```

+++

The following script is by Mark Moraes of the University of Toronto. See the file `Moraes-COPYRIGHT` for permissions and restrictions. This file is included in the combined HTML/source tarball of the *ABS Guide*.

Example A-12. *behead*: Removing mail and news message headers

```
#! /bin/sh
#  Strips off the header from a mail/News message i.e. till the first
#+ empty line.
#  Author: Mark Moraes, University of Toronto

# ==> These comments added by author of this document.

if [ $# -eq 0 ]; then
# ==> If no command-line args present, then works on file redirected to stdin.
        sed -e '1,/^$/d' -e '/^[        ]*$/d'
        # --> Delete empty lines and all lines until
        # --> first one beginning with white space.
else
# ==> If command-line args present, then work on files named.
        for i do
                sed -e '1,/^$/d' -e '/^[        ]*$/d' $i
                # --> Ditto, as above.
        done
fi

exit

# ==> Exercise: Add error checking and other options.
# ==>
# ==> Note that the small sed script repeats, except for the arg passed.
# ==> Does it make sense to embed it in a function? Why or why not?


/*
 * Copyright University of Toronto 1988, 1989.
 * Written by Mark Moraes
 *
 * Permission is granted to anyone to use this software for any purpose on
 * any computer system, and to alter it and redistribute it freely, subject
 * to the following restrictions:
 *
 * 1. The author and the University of Toronto are not responsible
 *    for the consequences of use of this software, no matter how awful,
 *    even if they arise from flaws in it.
 *
 * 2. The origin of this software must not be misrepresented, either by
 *    explicit claim or by omission.  Since few users ever read sources,
 *    credits must appear in the documentation.
 *
 * 3. Altered versions must be plainly marked as such, and must not be
 *    misrepresented as being the original software.  Since few users
 *    ever read sources, credits must appear in the documentation.
 *
 * 4. This notice may not be removed or altered.
 */
```

+

Antek Sawicki contributed the following script, which makes very clever use of the parameter substitution operators discussed in Section 10.2.

Example A-13. *password*: Generating random 8-character passwords

```
#!/bin/bash
#
#
#  Random password generator for Bash 2.x +
#+ by Antek Sawicki <tenox@tenox.tc>,
#+ who generously gave usage permission to the ABS Guide author.
#
# ==> Comments added by document author ==>


MATRIX="0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz"
# ==> Password will consist of alphanumeric characters.
LENGTH="8"
# ==> May change 'LENGTH' for longer password.


while [ "${n:=1}" -le "$LENGTH" ]
# ==> Recall that := is "default substitution" operator.
# ==> So, if 'n' has not been initialized, set it to 1.
do
        PASS="$PASS${MATRIX:$(($RANDOM%${#MATRIX})):1}"
        # ==> Very clever, but tricky.

        # ==> Starting from the innermost nesting...
        # ==> ${#MATRIX} returns length of array MATRIX.

        # ==> $RANDOM%${#MATRIX} returns random number between 1
        # ==> and [length of MATRIX] - 1.

        # ==> ${MATRIX:$(($RANDOM%${#MATRIX})):1}
        # ==> returns expansion of MATRIX at random position, by length 1.
        # ==> See {var:pos:len} parameter substitution in Chapter 9.
        # ==> and the associated examples.

        # ==> PASS=... simply pastes this result onto previous PASS (concatenation).

        # ==> To visualize this more clearly, uncomment the following line
        #                 echo "$PASS"
        # ==> to see PASS being built up,
        # ==> one character at a time, each iteration of the loop.

        let n+=1
        # ==> Increment 'n' for next pass.
done

echo "$PASS"      # ==> Or, redirect to a file, as desired.

exit 0
```

+

James R. Van Zandt contributed this script which uses named pipes and, in his words, "really exercises quoting and escaping."

Example A-14. *fifo*: Making daily backups, using named pipes

```
#!/bin/bash
# ==> Script by James R. Van Zandt, and used here with his permission.

# ==> Comments added by author of this document.


  HERE=`uname -n`    # ==> hostname
  THERE=bilbo
  echo "starting remote backup to $THERE at `date +%r`"
  # ==> `date +%r` returns time in 12-hour format, i.e. "08:08:34 PM".

  # make sure /pipe really is a pipe and not a plain file
  rm -rf /pipe
  mkfifo /pipe       # ==> Create a "named pipe", named "/pipe" ...

  # ==> 'su xyz' runs commands as user "xyz".
  # ==> 'ssh' invokes secure shell (remote login client).
  su xyz -c "ssh $THERE \"cat > /home/xyz/backup/${HERE}-daily.tar.gz\" < /pipe"&
  cd /
  tar -czf - bin boot dev etc home info lib man root sbin share usr var > /pipe
  # ==> Uses named pipe, /pipe, to communicate between processes:
  # ==> 'tar/gzip' writes to /pipe and 'ssh' reads from /pipe.

  # ==> The end result is this backs up the main directories, from / on down.

  # ==>  What are the advantages of a "named pipe" in this situation,
  # ==>+ as opposed to an "anonymous pipe", with |?
  # ==>  Will an anonymous pipe even work here?

  # ==>  Is it necessary to delete the pipe before exiting the script?
  # ==>  How could that be done?


  exit 0
```

+

Stéphane Chazelas used the following script to demonstrate generating prime numbers without arrays.

Example A-15. Generating prime numbers using the modulo operator

```
#!/bin/bash
# primes.sh: Generate prime numbers, without using arrays.
# Script contributed by Stephane Chazelas.

#  This does *not* use the classic "Sieve of Eratosthenes" algorithm,
#+ but instead the more intuitive method of testing each candidate number
#+ for factors (divisors), using the "%" modulo operator.


LIMIT=1000                    # Primes, 2 ... 1000.

Primes()
{
 (( n = $1 + 1 ))             # Bump to next integer.
```

```
 shift                         # Next parameter in list.
#  echo "_n=$n i=$i_"

 if (( n == LIMIT ))
 then echo $*
 return
 fi

 for i; do                     # "i" set to "@", previous values of $n.
#   echo "-n=$n i=$i-"
   (( i * i > n )) && break   # Optimization.
   (( n % i )) && continue    # Sift out non-primes using modulo operator.
   Primes $n $@               # Recursion inside loop.
   return
   done

   Primes $n $@ $n            #  Recursion outside loop.
                              #  Successively accumulate
                              #+ positional parameters.
                              #  "$@" is the accumulating list of primes.
}

Primes 1

exit $?

# Pipe output of the script to 'fmt' for prettier printing.

#  Uncomment lines 16 and 24 to help figure out what is going on.

#  Compare the speed of this algorithm for generating primes
#+ with the Sieve of Eratosthenes (ex68.sh).


#  Exercise: Rewrite this script without recursion.
```

+

Rick Boivie's revision of Jordi Sanfeliu's *tree* script.

Example A-16. *tree*: Displaying a directory tree

```
#!/bin/bash
# tree.sh

#  Written by Rick Boivie.
#  Used with permission.
#  This is a revised and simplified version of a script
#+ by Jordi Sanfeliu (the original author), and patched by Ian Kjos.
#  This script replaces the earlier version used in
#+ previous releases of the Advanced Bash Scripting Guide.
#  Copyright (c) 2002, by Jordi Sanfeliu, Rick Boivie, and Ian Kjos.

# ==> Comments added by the author of this document.


search () {
for dir in `echo *`
#  ==> `echo *` lists all the files in current working directory,
#+ ==> without line breaks.
```

```
#  ==> Similar effect to for dir in *
#  ==> but "dir in `echo *`" will not handle filenames with blanks.
do
  if [ -d "$dir" ] ; then # ==> If it is a directory (-d)...
  zz=0                    # ==> Temp variable, keeping track of
                          #     directory level.
  while [ $zz != $1 ]     # Keep track of inner nested loop.
    do
      echo -n "| "        # ==> Display vertical connector symbol,
                          # ==> with 2 spaces & no line feed
                          #     in order to indent.
      zz=`expr $zz + 1`   # ==> Increment zz.
    done

    if [ -L "$dir" ] ; then # ==> If directory is a symbolic link...
      echo "+---$dir" `ls -l $dir | sed 's/^.*'$dir' //'`
      # ==> Display horiz. connector and list directory name, but...
      # ==> delete date/time part of long listing.
    else
      echo "+---$dir"       # ==> Display horizontal connector symbol...
      # ==> and print directory name.
      numdirs=`expr $numdirs + 1` # ==> Increment directory count.
      if cd "$dir" ; then         # ==> If can move to subdirectory...
        search `expr $1 + 1`      # with recursion ;-)
        # ==> Function calls itself.
        cd ..
      fi
    fi
  fi
done
}

if [ $# != 0 ] ; then
  cd $1   # Move to indicated directory.
  #else   # stay in current directory
fi

echo "Initial directory = `pwd`"
numdirs=0

search 0
echo "Total directories = $numdirs"

exit 0
```

Patsie's version of a directory *tree* script.

Example A-17. *tree2*: Alternate directory tree script

```
#!/bin/bash
# tree2.sh

# Lightly modified/reformatted by ABS Guide author.
# Included in ABS Guide with permission of script author (thanks!).

## Recursive file/dirsize checking script, by Patsie
##
## This script builds a list of files/directories and their size (du -akx)
## and processes this list to a human readable tree shape
## The 'du -akx' is only as good as the permissions the owner has.
## So preferably run as root* to get the best results, or use only on
```

```
## directories for which you have read permissions. Anything you can't
## read is not in the list.

#* ABS Guide author advises caution when running scripts as root!


##########  THIS IS CONFIGURABLE  ##########

TOP=5                   # Top 5 biggest (sub)directories.
MAXRECURS=5             # Max 5 subdirectories/recursions deep.
E_BL=80                 # Blank line already returned.
E_DIR=81                # Directory not specified.


##########  DON'T CHANGE ANYTHING BELOW THIS LINE  ##########

PID=$$                            # Our own process ID.
SELF=`basename $0`                # Our own program name.
TMP="/tmp/${SELF}.${PID}.tmp"     # Temporary 'du' result.

# Convert number to dotted thousand.
function dot { echo "            $*" |
               sed -e :a -e 's/\(.*[0-9]\)\([0-9]\{3\}\)/\1,\2/;ta' |
               tail -c 12; }

# Usage: tree <recursion> <indent prefix> <min size> <directory>
function tree {
  recurs="$1"           # How deep nested are we?
  prefix="$2"           # What do we display before file/dirname?
  minsize="$3"          # What is the minumum file/dirsize?
  dirname="$4"          # Which directory are we checking?

# Get ($TOP) biggest subdirs/subfiles from TMP file.
  LIST=`egrep "[[:space:]]${dirname}/[^/]*$" "$TMP" |
        awk '{if($1>'$minsize') print;}' | sort -nr | head -$TOP`
  [ -z "$LIST" ] && return        # Empty list, then go back.

  cnt=0
  num=`echo "$LIST" | wc -l`      # How many entries in the list.

  ## Main loop
  echo "$LIST" | while read size name; do
    ((cnt+=1))                    # Count entry number.
    bname=`basename "$name"`      # We only need a basename of the entry.
    [ -d "$name" ] && bname="$bname/"
                                  # If it's a directory, append a slash.
    echo "`dot $size`$prefix +-$bname"
                                  # Display the result.
    #  Call ourself recursively if it's a directory
    #+ and we're not nested too deep ($MAXRECURS).
    #  The recursion goes up: $((recurs+1))
    #  The prefix gets a space if it's the last entry,
    #+ or a pipe if there are more entries.
    #  The minimum file/dirsize becomes
    #+ a tenth of his parent: $((size/10)).
    # Last argument is the full directory name to check.
    if [ -d "$name" -a $recurs -lt $MAXRECURS ]; then
      [ $cnt -lt $num ] \
        || (tree $((recurs+1)) "$prefix  " $((size/10)) "$name") \
        && (tree $((recurs+1)) "$prefix |" $((size/10)) "$name")
    fi
  done
```

```
  [ $? -eq 0 ] && echo "           $prefix"
  # Every time we jump back add a 'blank' line.
  return $E_BL
  # We return 80 to tell we added a blank line already.
}

###                ###
###  main program  ###
###                ###

rootdir="$@"
[ -d "$rootdir" ] ||
  { echo "$SELF: Usage: $SELF <directory>" >&2; exit $E_DIR; }
  # We should be called with a directory name.

echo "Building inventory list, please wait ..."
     # Show "please wait" message.
du -akx "$rootdir" 1>"$TMP" 2>/dev/null
     # Build a temporary list of all files/dirs and their size.
size=`tail -1 "$TMP" | awk '{print $1}'`
     # What is our rootdirectory's size?
echo "`dot $size` $rootdir"
     # Display rootdirectory's entry.
tree 0 "" 0 "$rootdir"
     # Display the tree below our rootdirectory.

rm "$TMP" 2>/dev/null
     # Clean up TMP file.

exit $?
```

Noah Friedman permitted use of his *string function* script. It essentially reproduces some of the *C*-library string manipulation functions.

Example A-18. *string functions*: C-style string functions

```
#!/bin/bash

# string.bash --- bash emulation of string(3) library routines
# Author: Noah Friedman <friedman@prep.ai.mit.edu>
# ==>     Used with his kind permission in this document.
# Created: 1992-07-01
# Last modified: 1993-09-29
# Public domain

# Conversion to bash v2 syntax done by Chet Ramey

# Commentary:
# Code:

#:docstring strcat:
# Usage: strcat s1 s2
#
# Strcat appends the value of variable s2 to variable s1.
#
# Example:
#    a="foo"
#    b="bar"
#    strcat a b
```

```
#    echo $a
#    => foobar
#
#:end docstring:

###;;;autoload   ==> Autoloading of function commented out.
function strcat ()
{
    local s1_val s2_val

    s1_val=${!1}                        # indirect variable expansion
    s2_val=${!2}
    eval "$1"=\'"${s1_val}${s2_val}"\'
    # ==> eval $1='${s1_val}${s2_val}' avoids problems,
    # ==> if one of the variables contains a single quote.
}

#:docstring strncat:
# Usage: strncat s1 s2 $n
#
# Line strcat, but strncat appends a maximum of n characters from the value
# of variable s2.  It copies fewer if the value of variabl s2 is shorter
# than n characters.  Echoes result on stdout.
#
# Example:
#    a=foo
#    b=barbaz
#    strncat a b 3
#    echo $a
#    => foobar
#
#:end docstring:

###;;;autoload
function strncat ()
{
    local s1="$1"
    local s2="$2"
    local -i n="$3"
    local s1_val s2_val

    s1_val=${!s1}                       # ==> indirect variable expansion
    s2_val=${!s2}

    if [ ${#s2_val} -gt ${n} ]; then
       s2_val=${s2_val:0:$n}            # ==> substring extraction
    fi

    eval "$s1"=\'"${s1_val}${s2_val}"\'
    # ==> eval $1='${s1_val}${s2_val}' avoids problems,
    # ==> if one of the variables contains a single quote.
}

#:docstring strcmp:
# Usage: strcmp $s1 $s2
#
# Strcmp compares its arguments and returns an integer less than, equal to,
# or greater than zero, depending on whether string s1 is lexicographically
# less than, equal to, or greater than string s2.
#:end docstring:

###;;;autoload
```

```
function strcmp ()
{
    [ "$1" = "$2" ] && return 0

    [ "${1}" '<' "${2}" ] > /dev/null && return -1

    return 1
}

#:docstring strncmp:
# Usage: strncmp $s1 $s2 $n
#
# Like strcmp, but makes the comparison by examining a maximum of n
# characters (n less than or equal to zero yields equality).
#:end docstring:

###;;;autoload
function strncmp ()
{
    if [ -z "${3}" -o "${3}" -le "0" ]; then
       return 0
    fi

    if [ ${3} -ge ${#1} -a ${3} -ge ${#2} ]; then
       strcmp "$1" "$2"
       return $?
    else
       s1=${1:0:$3}
       s2=${2:0:$3}
       strcmp $s1 $s2
       return $?
    fi
}

#:docstring strlen:
# Usage: strlen s
#
# Strlen returns the number of characters in string literal s.
#:end docstring:

###;;;autoload
function strlen ()
{
    eval echo "\${#${1}}"
    # ==> Returns the length of the value of the variable
    # ==> whose name is passed as an argument.
}

#:docstring strspn:
# Usage: strspn $s1 $s2
#
# Strspn returns the length of the maximum initial segment of string s1,
# which consists entirely of characters from string s2.
#:end docstring:

###;;;autoload
function strspn ()
{
    # Unsetting IFS allows whitespace to be handled as normal chars.
    local IFS=
    local result="${1%%[!${2}]*}"
```

```
    echo ${#result}
}

#:docstring strcspn:
# Usage: strcspn $s1 $s2
#
# Strcspn returns the length of the maximum initial segment of string s1,
# which consists entirely of characters not from string s2.
#:end docstring:

###;;;autoload
function strcspn ()
{
    # Unsetting IFS allows whitspace to be handled as normal chars.
    local IFS=
    local result="${1%%[${2}]*}"

    echo ${#result}
}

#:docstring strstr:
# Usage: strstr s1 s2
#
# Strstr echoes a substring starting at the first occurrence of string s2 in
# string s1, or nothing if s2 does not occur in the string.  If s2 points to
# a string of zero length, strstr echoes s1.
#:end docstring:

###;;;autoload
function strstr ()
{
    # if s2 points to a string of zero length, strstr echoes s1
    [ ${#2} -eq 0 ] && { echo "$1" ; return 0; }

    # strstr echoes nothing if s2 does not occur in s1
    case "$1" in
    *$2*) ;;
    *) return 1;;
    esac

    # use the pattern matching code to strip off the match and everything
    # following it
    first=${1/$2*/}

    # then strip off the first unmatched portion of the string
    echo "${1##$first}"
}

#:docstring strtok:
# Usage: strtok s1 s2
#
# Strtok considers the string s1 to consist of a sequence of zero or more
# text tokens separated by spans of one or more characters from the
# separator string s2.  The first call (with a non-empty string s1
# specified) echoes a string consisting of the first token on stdout. The
# function keeps track of its position in the string s1 between separate
# calls, so that subsequent calls made with the first argument an empty
# string will work through the string immediately following that token.  In
# this way subsequent calls will work through the string s1 until no tokens
# remain.  The separator string s2 may be different from call to call.
# When no token remains in s1, an empty value is echoed on stdout.
#:end docstring:
```

```
###;;;autoload
function strtok ()
{
 :
}

#:docstring strtrunc:
# Usage: strtrunc $n $s1 {$s2} {$...}
#
# Used by many functions like strncmp to truncate arguments for comparison.
# Echoes the first n characters of each string s1 s2 ... on stdout.
#:end docstring:

###;;;autoload
function strtrunc ()
{
    n=$1 ; shift
    for z; do
        echo "${z:0:$n}"
    done
}

# provide string

# string.bash ends here


# ========================================================================== #
# ==> Everything below here added by the document author.

# ==> Suggested use of this script is to delete everything below here,
# ==> and "source" this file into your own scripts.

# strcat
string0=one
string1=two
echo
echo "Testing \"strcat\" function:"
echo "Original \"string0\" = $string0"
echo "\"string1\" = $string1"
strcat string0 string1
echo "New \"string0\" = $string0"
echo

# strlen
echo
echo "Testing \"strlen\" function:"
str=123456789
echo "\"str\" = $str"
echo -n "Length of \"str\" = "
strlen str
echo



# Exercise:
# --------
# Add code to test all the other string functions above.


exit 0
```

Michael Zick's complex array example uses the md5sum check sum command to encode directory information.

Example A-19. Directory information

```
#! /bin/bash
# directory-info.sh
# Parses and lists directory information.

# NOTE: Change lines 273 and 353 per "README" file.

# Michael Zick is the author of this script.
# Used here with his permission.

# Controls
# If overridden by command arguments, they must be in the order:
#   Arg1: "Descriptor Directory"
#   Arg2: "Exclude Paths"
#   Arg3: "Exclude Directories"
#
# Environment Settings override Defaults.
# Command arguments override Environment Settings.

# Default location for content addressed file descriptors.
MD5UCFS=${1:-${MD5UCFS:-'/tmpfs/ucfs'}}

# Directory paths never to list or enter
declare -a \
  EXCLUDE_PATHS=${2:-${EXCLUDE_PATHS:-'(/proc /dev /devfs /tmpfs)'}}

# Directories never to list or enter
declare -a \
  EXCLUDE_DIRS=${3:-${EXCLUDE_DIRS:-'(ucfs lost+found tmp wtmp)'}}

# Files never to list or enter
declare -a \
  EXCLUDE_FILES=${3:-${EXCLUDE_FILES:-'(core "Name with Spaces")'}}


# Here document used as a comment block.
: <<LSfieldsDoc
# # # # # List Filesystem Directory Information # # # # #
#
#       ListDirectory "FileGlob" "Field-Array-Name"
# or
#       ListDirectory -of "FileGlob" "Field-Array-Filename"
#       '-of' meaning 'output to filename'
# # # # #

String format description based on: ls (GNU fileutils) version 4.0.36

Produces a line (or more) formatted:
inode permissions hard-links owner group ...
32736 -rw-------    1 mszick   mszick

size    day month date hh:mm:ss year path
2756608 Sun Apr 20 08:53:06 2003 /home/mszick/core

Unless it is formatted:
inode permissions hard-links owner group ...
```

```
266705 crw-rw----    1    root  uucp

major minor day month date hh:mm:ss year path
4,  68 Sun Apr 20 09:27:33 2003 /dev/ttyS4
NOTE: that pesky comma after the major number

NOTE: the 'path' may be multiple fields:
/home/mszick/core
/proc/982/fd/0 -> /dev/null
/proc/982/fd/1 -> /home/mszick/.xsession-errors
/proc/982/fd/13 -> /tmp/tmpfZVVOCs (deleted)
/proc/982/fd/7 -> /tmp/kde-mszick/ksycoca
/proc/982/fd/8 -> socket:[11586]
/proc/982/fd/9 -> pipe:[11588]

If that isn't enough to keep your parser guessing,
either or both of the path components may be relative:
../Built-Shared -> Built-Static
../linux-2.4.20.tar.bz2 -> ../../../SRCS/linux-2.4.20.tar.bz2

The first character of the 11 (10?) character permissions field:
's' Socket
'd' Directory
'b' Block device
'c' Character device
'l' Symbolic link
NOTE: Hard links not marked - test for identical inode numbers
on identical filesystems.
All information about hard linked files are shared, except
for the names and the name's location in the directory system.
NOTE: A "Hard link" is known as a "File Alias" on some systems.
'-' An undistingushed file

Followed by three groups of letters for: User, Group, Others
Character 1: '-' Not readable; 'r' Readable
Character 2: '-' Not writable; 'w' Writable
Character 3, User and Group: Combined execute and special
'-' Not Executable, Not Special
'x' Executable, Not Special
's' Executable, Special
'S' Not Executable, Special
Character 3, Others: Combined execute and sticky (tacky?)
'-' Not Executable, Not Tacky
'x' Executable, Not Tacky
't' Executable, Tacky
'T' Not Executable, Tacky

Followed by an access indicator
Haven't tested this one, it may be the eleventh character
or it may generate another field
' ' No alternate access
'+' Alternate access
LSfieldsDoc


ListDirectory()
{
        local -a T
        local -i of=0           # Default return in variable
#       OLD_IFS=$IFS            # Using BASH default ' \t\n'

        case "$#" in
```

```
        3)      case "$1" in
                -of)    of=1 ; shift ;;
                 * )    return 1 ;;
                esac ;;
        2)      : ;;            # Poor man's "continue"
        *)      return 1 ;;
        esac

        # NOTE: the (ls) command is NOT quoted (")
        T=( $(ls --inode --ignore-backups --almost-all --directory \
        --full-time --color=none --time=status --sort=none \
        --format=long $1) )

        case $of in
        # Assign T back to the array whose name was passed as $2
                0) eval $2=\( \"\$\{T\[@\]\}\" \) ;;
        # Write T into filename passed as $2
                1) echo "${T[@]}" > "$2" ;;
        esac
        return 0
   }

# # # # # Is that string a legal number? # # # # #
#
#       IsNumber "Var"
# # # # # There has to be a better way, sigh...

IsNumber()
{
        local -i int
        if [ $# -eq 0 ]
        then
                return 1
        else
                (let int=$1)  2>/dev/null
                return $?       # Exit status of the let thread
        fi
}

# # # # # Index Filesystem Directory Information # # # # #
#
#       IndexList "Field-Array-Name" "Index-Array-Name"
# or
#       IndexList -if Field-Array-Filename Index-Array-Name
#       IndexList -of Field-Array-Name Index-Array-Filename
#       IndexList -if -of Field-Array-Filename Index-Array-Filename
# # # # #

: <<IndexListDoc
Walk an array of directory fields produced by ListDirectory

Having suppressed the line breaks in an otherwise line oriented
report, build an index to the array element which starts each line.

Each line gets two index entries, the first element of each line
(inode) and the element that holds the pathname of the file.

The first index entry pair (Line-Number==0) are informational:
Index-Array-Name[0] : Number of "Lines" indexed
Index-Array-Name[1] : "Current Line" pointer into Index-Array-Name

The following index pairs (if any) hold element indexes into
```

```
the Field-Array-Name per:
Index-Array-Name[Line-Number * 2] : The "inode" field element.
NOTE: This distance may be either +11 or +12 elements.
Index-Array-Name[(Line-Number * 2) + 1] : The "pathname" element.
NOTE: This distance may be a variable number of elements.
Next line index pair for Line-Number+1.
IndexListDoc



IndexList()
{
        local -a LIST                   # Local of listname passed
        local -a -i INDEX=( 0 0 )       # Local of index to return
        local -i Lidx Lcnt
        local -i if=0 of=0              # Default to variable names

        case "$#" in                    # Simplistic option testing
                0) return 1 ;;
                1) return 1 ;;
                2) : ;;                 # Poor man's continue
                3) case "$1" in
                        -if) if=1 ;;
                        -of) of=1 ;;
                         * ) return 1 ;;
                   esac ; shift ;;
                4) if=1 ; of=1 ; shift ; shift ;;
                *) return 1
        esac

        # Make local copy of list
        case "$if" in
                0) eval LIST=\( \"\$\{$1\[@\]\}\" \) ;;
                1) LIST=( $(cat $1) ) ;;
        esac

        # Grok (grope?) the array
        Lcnt=${#LIST[@]}
        Lidx=0
        until (( Lidx >= Lcnt ))
        do
        if IsNumber ${LIST[$Lidx]}
        then
                local -i inode name
                local ft
                inode=Lidx
                local m=${LIST[$Lidx+2]}        # Hard Links field
                ft=${LIST[$Lidx+1]:0:1}         # Fast-Stat
                case $ft in
                b)      ((Lidx+=12)) ;;         # Block device
                c)      ((Lidx+=12)) ;;         # Character device
                *)      ((Lidx+=11)) ;;         # Anything else
                esac
                name=Lidx
                case $ft in
                -)      ((Lidx+=1)) ;;          # The easy one
                b)      ((Lidx+=1)) ;;          # Block device
                c)      ((Lidx+=1)) ;;          # Character device
                d)      ((Lidx+=1)) ;;          # The other easy one
                l)      ((Lidx+=3)) ;;          # At LEAST two more fields
#  A little more elegance here would handle pipes,
#+ sockets, deleted files - later.
```

```
                *)      until IsNumber ${LIST[$Lidx]} || ((Lidx >= Lcnt))
                        do
                                ((Lidx+=1))
                        done
                        ;;                      # Not required
                esac
                INDEX[${#INDEX[*]}]=$inode
                INDEX[${#INDEX[*]}]=$name
                INDEX[0]=${INDEX[0]}+1          # One more "line" found
# echo "Line: ${INDEX[0]} Type: $ft Links: $m Inode: \
# ${LIST[$inode]} Name: ${LIST[$name]}"

        else
                ((Lidx+=1))
        fi
        done
        case "$of" in
                0) eval $2=\( \"\$\{INDEX\[@\]\}\" \) ;;
                1) echo "${INDEX[@]}" > "$2" ;;
        esac
        return 0                                # What could go wrong?
}

# # # # # Content Identify File # # # # #
#
#       DigestFile Input-Array-Name Digest-Array-Name
# or
#       DigestFile -if Input-FileName Digest-Array-Name
# # # # #

# Here document used as a comment block.
: <<DigestFilesDoc

The key (no pun intended) to a Unified Content File System (UCFS)
is to distinguish the files in the system based on their content.
Distinguishing files by their name is just so 20th Century.

The content is distinguished by computing a checksum of that content.
This version uses the md5sum program to generate a 128 bit checksum
representative of the file's contents.
There is a chance that two files having different content might
generate the same checksum using md5sum (or any checksum).  Should
that become a problem, then the use of md5sum can be replace by a
cyrptographic signature.  But until then...

The md5sum program is documented as outputting three fields (and it
does), but when read it appears as two fields (array elements).  This
is caused by the lack of whitespace between the second and third field.
So this function gropes the md5sum output and returns:
        [0]     32 character checksum in hexidecimal (UCFS filename)
        [1]     Single character: ' ' text file, '*' binary file
        [2]     Filesystem (20th Century Style) name
        Note: That name may be the character '-' indicating STDIN read.

DigestFilesDoc



DigestFile()
{
        local if=0              # Default, variable name
        local -a T1 T2
```

```
        case "$#" in
        3)      case "$1" in
                -if)    if=1 ; shift ;;
                 * )    return 1 ;;
                esac ;;
        2)      : ;;            # Poor man's "continue"
        *)      return 1 ;;
        esac

        case $if in
        0) eval T1=\( \"\$\{$1\[@\]\}\" \)
           T2=( $(echo ${T1[@]} | md5sum -) )
           ;;
        1) T2=( $(md5sum $1) )
           ;;
        esac

        case ${#T2[@]} in
        0) return 1 ;;
        1) return 1 ;;
        2) case ${T2[1]:0:1} in         # SanScrit-2.0.5
           \*) T2[${#T2[@]}]=${T2[1]:1}
               T2[1]=\*
               ;;
            *) T2[${#T2[@]}]=${T2[1]}
               T2[1]=" "
               ;;
           esac
           ;;
        3) : ;; # Assume it worked
        *) return 1 ;;
        esac

        local -i len=${#T2[0]}
        if [ $len -ne 32 ] ; then return 1 ; fi
        eval $2=\( \"\$\{T2\[@\]\}\" \)
}

# # # # # Locate File # # # # #
#
#       LocateFile [-l] FileName Location-Array-Name
# or
#       LocateFile [-l] -of FileName Location-Array-FileName
# # # # #

# A file location is Filesystem-id and inode-number

# Here document used as a comment block.
: <<StatFieldsDoc
        Based on stat, version 2.2
        stat -t and stat -lt fields
        [0]     name
        [1]     Total size
                File - number of bytes
                Symbolic link - string length of pathname
        [2]     Number of (512 byte) blocks allocated
        [3]     File type and Access rights (hex)
        [4]     User ID of owner
        [5]     Group ID of owner
        [6]     Device number
        [7]     Inode number
```

```
        [8]     Number of hard links
        [9]     Device type (if inode device) Major
        [10]    Device type (if inode device) Minor
        [11]    Time of last access
                May be disabled in 'mount' with noatime
                atime of files changed by exec, read, pipe, utime, mknod (mmap?)
                atime of directories changed by addition/deletion of files
        [12]    Time of last modification
                mtime of files changed by write, truncate, utime, mknod
                mtime of directories changed by addtition/deletion of files
        [13]    Time of last change
                ctime reflects time of changed inode information (owner, group
                permissions, link count
-*-*- Per:
        Return code: 0
        Size of array: 14
        Contents of array
        Element 0: /home/mszick
        Element 1: 4096
        Element 2: 8
        Element 3: 41e8
        Element 4: 500
        Element 5: 500
        Element 6: 303
        Element 7: 32385
        Element 8: 22
        Element 9: 0
        Element 10: 0
        Element 11: 1051221030
        Element 12: 1051214068
        Element 13: 1051214068

        For a link in the form of linkname -> realname
        stat -t  linkname returns the linkname (link) information
        stat -lt linkname returns the realname information

        stat -tf and stat -ltf fields
        [0]     name
        [1]     ID-0?           # Maybe someday, but Linux stat structure
        [2]     ID-0?           # does not have either LABEL nor UUID
                                # fields, currently information must come
                                # from file-system specific utilities
        These will be munged into:
        [1]     UUID if possible
        [2]     Volume Label if possible
        Note: 'mount -l' does return the label and could return the UUID

        [3]     Maximum length of filenames
        [4]     Filesystem type
        [5]     Total blocks in the filesystem
        [6]     Free blocks
        [7]     Free blocks for non-root user(s)
        [8]     Block size of the filesystem
        [9]     Total inodes
        [10]    Free inodes

-*-*- Per:
        Return code: 0
        Size of array: 11
        Contents of array
        Element 0: /home/mszick
        Element 1: 0
```

```
        Element 2: 0
        Element 3: 255
        Element 4: ef53
        Element 5: 2581445
        Element 6: 2277180
        Element 7: 2146050
        Element 8: 4096
        Element 9: 1311552
        Element 10: 1276425

StatFieldsDoc


#       LocateFile [-l] FileName Location-Array-Name
#       LocateFile [-l] -of FileName Location-Array-FileName

LocateFile()
{
        local -a LOC LOC1 LOC2
        local lk="" of=0

        case "$#" in
        0) return 1 ;;
        1) return 1 ;;
        2) : ;;
        *) while (( "$#" > 2 ))
           do
              case "$1" in
               -l) lk=-1 ;;
              -of) of=1 ;;
                *) return 1 ;;
              esac
           shift
           done ;;
        esac

# More Sanscrit-2.0.5
      # LOC1=( $(stat -t $lk $1) )
      # LOC2=( $(stat -tf $lk $1) )
      # Uncomment above two lines if system has "stat" command installed.
        LOC=( ${LOC1[@]:0:1} ${LOC1[@]:3:11}
              ${LOC2[@]:1:2} ${LOC2[@]:4:1} )

        case "$of" in
                0) eval $2=\( \"\$\{LOC\[@\]\}\" \) ;;
                1) echo "${LOC[@]}" > "$2" ;;
        esac
        return 0
# Which yields (if you are lucky, and have "stat" installed)
# -*-*- Location Discriptor -*-*-
#       Return code: 0
#       Size of array: 15
#       Contents of array
#       Element 0: /home/mszick         20th Century name
#       Element 1: 41e8                 Type and Permissions
#       Element 2: 500                  User
#       Element 3: 500                  Group
#       Element 4: 303                  Device
#       Element 5: 32385                inode
#       Element 6: 22                   Link count
#       Element 7: 0                    Device Major
#       Element 8: 0                    Device Minor
```

```
#       Element 9: 1051224608           Last Access
#       Element 10: 1051214068          Last Modify
#       Element 11: 1051214068          Last Status
#       Element 12: 0                   UUID (to be)
#       Element 13: 0                   Volume Label (to be)
#       Element 14: ef53                Filesystem type
}



# And then there was some test code

ListArray() # ListArray Name
{
        local -a Ta

        eval Ta=\( \"\$\{$1\[@\]\}\" \)
        echo
        echo "-*-*- List of Array -*-*-"
        echo "Size of array $1: ${#Ta[*]}"
        echo "Contents of array $1:"
        for (( i=0 ; i<${#Ta[*]} ; i++ ))
        do
            echo -e "\tElement $i: ${Ta[$i]}"
        done
        return 0
}

declare -a CUR_DIR
# For small arrays
ListDirectory "${PWD}" CUR_DIR
ListArray CUR_DIR

declare -a DIR_DIG
DigestFile CUR_DIR DIR_DIG
echo "The new \"name\" (checksum) for ${CUR_DIR[9]} is ${DIR_DIG[0]}"

declare -a DIR_ENT
# BIG_DIR # For really big arrays - use a temporary file in ramdisk
# BIG-DIR # ListDirectory -of "${CUR_DIR[11]}/*" "/tmpfs/junk2"
ListDirectory "${CUR_DIR[11]}/*" DIR_ENT

declare -a DIR_IDX
# BIG-DIR # IndexList -if "/tmpfs/junk2" DIR_IDX
IndexList DIR_ENT DIR_IDX

declare -a IDX_DIG
# BIG-DIR # DIR_ENT=( $(cat /tmpfs/junk2) )
# BIG-DIR # DigestFile -if /tmpfs/junk2 IDX_DIG
DigestFile DIR_ENT IDX_DIG
# Small (should) be able to parallize IndexList & DigestFile
# Large (should) be able to parallize IndexList & DigestFile & the assignment
echo "The \"name\" (checksum) for the contents of ${PWD} is ${IDX_DIG[0]}"

declare -a FILE_LOC
LocateFile ${PWD} FILE_LOC
ListArray FILE_LOC

exit 0
```

Stéphane Chazelas demonstrates object-oriented programming in a Bash script.

Mariusz Gniazdowski contributed a hash library for use in scripts.

Example A-20. Library of hash functions

```
# Hash:
# Hash function library
# Author: Mariusz Gniazdowski <mariusz.gn-at-gmail.com>
# Date: 2005-04-07

# Functions making emulating hashes in Bash a little less painful.


#    Limitations:
#  * Only global variables are supported.
#  * Each hash instance generates one global variable per value.
#  * Variable names collisions are possible
#+   if you define variable like __hash__hashname_key
#  * Keys must use chars that can be part of a Bash variable name
#+   (no dashes, periods, etc.).
#  * The hash is created as a variable:
#    ... hashname_keyname
#    So if somone will create hashes like:
#      myhash_ + mykey = myhash__mykey
#      myhash + _mykey = myhash__mykey
#    Then there will be a collision.
#    (This should not pose a major problem.)


Hash_config_varname_prefix=__hash__


# Emulates:  hash[key]=value
#
# Params:
# 1 - hash
# 2 - key
# 3 - value
function hash_set {
        eval "${Hash_config_varname_prefix}${1}_${2}=\"${3}\""
}


# Emulates:  value=hash[key]
#
# Params:
# 1 - hash
# 2 - key
# 3 - value (name of global variable to set)
function hash_get_into {
        eval "$3=\"\$${Hash_config_varname_prefix}${1}_${2}\""
}


# Emulates:  echo hash[key]
#
# Params:
# 1 - hash
# 2 - key
# 3 - echo params (like -n, for example)
function hash_echo {
```

```
        eval "echo $3 \"\$${Hash_config_varname_prefix}${1}_${2}\""
}


# Emulates:  hash1[key1]=hash2[key2]
#
# Params:
# 1 - hash1
# 2 - key1
# 3 - hash2
# 4 - key2
function hash_copy {
eval "${Hash_config_varname_prefix}${1}_${2}\
=\"\$${Hash_config_varname_prefix}${3}_${4}\""
}


# Emulates:  hash[keyN-1]=hash[key2]=...hash[key1]
#
# Copies first key to rest of keys.
#
# Params:
# 1 - hash1
# 2 - key1
# 3 - key2
# . . .
# N - keyN
function hash_dup {
  local hashName="$1" keyName="$2"
  shift 2
  until [ ${#} -le 0 ]; do
    eval "${Hash_config_varname_prefix}${hashName}_${1}\
=\"\$${Hash_config_varname_prefix}${hashName}_${keyName}\""
  shift;
  done;
}


# Emulates:  unset hash[key]
#
# Params:
# 1 - hash
# 2 - key
function hash_unset {
        eval "unset ${Hash_config_varname_prefix}${1}_${2}"
}


# Emulates something similar to:  ref=&hash[key]
#
# The reference is name of the variable in which value is held.
#
# Params:
# 1 - hash
# 2 - key
# 3 - ref - Name of global variable to set.
function hash_get_ref_into {
        eval "$3=\"${Hash_config_varname_prefix}${1}_${2}\""
}


# Emulates something similar to:  echo &hash[key]
```

```
#
# That reference is name of variable in which value is held.
#
# Params:
# 1 - hash
# 2 - key
# 3 - echo params (like -n for example)
function hash_echo_ref {
        eval "echo $3 \"${Hash_config_varname_prefix}${1}_${2}\""
}



# Emulates something similar to:  $$hash[key](param1, param2, ...)
#
# Params:
# 1 - hash
# 2 - key
# 3,4, ... - Function parameters
function hash_call {
  local hash key
  hash=$1
  key=$2
  shift 2
  eval "eval \"\$${Hash_config_varname_prefix}${hash}_${key} \\\"\\\$@\\\"\""
}


# Emulates something similar to:  isset(hash[key]) or hash[key]==NULL
#
# Params:
# 1 - hash
# 2 - key
# Returns:
# 0 - there is such key
# 1 - there is no such key
function hash_is_set {
  eval "if [[ \"\${${Hash_config_varname_prefix}${1}_${2}-a}\" = \"a\" &&
  \"\${${Hash_config_varname_prefix}${1}_${2}-b}\" = \"b\" ]]
    then return 1; else return 0; fi"
}


# Emulates something similar to:
#   foreach($hash as $key => $value) { fun($key,$value); }
#
# It is possible to write different variations of this function.
# Here we use a function call to make it as "generic" as possible.
#
# Params:
# 1 - hash
# 2 - function name
function hash_foreach {
  local keyname oldIFS="$IFS"
  IFS=' '
  for i in $(eval "echo \${!${Hash_config_varname_prefix}${1}_*}"); do
    keyname=$(eval "echo \${i##${Hash_config_varname_prefix}${1}_}")
    eval "$2 $keyname \"\$$i\""
  done
IFS="$oldIFS"
}
```

```
#  NOTE: In lines 103 and 116, ampersand changed.
#  But, it doesn't matter, because these are comment lines anyhow.
```

Here is an example script using the foregoing hash library.

Example A-21. Colorizing text using hash functions

```
#!/bin/bash
# hash-example.sh: Colorizing text.
# Author: Mariusz Gniazdowski <mariusz.gn-at-gmail.com>

. Hash.lib      # Load the library of functions.

hash_set colors red           "\033[0;31m"
hash_set colors blue          "\033[0;34m"
hash_set colors light_blue    "\033[1;34m"
hash_set colors light_red     "\033[1;31m"
hash_set colors cyan          "\033[0;36m"
hash_set colors light_green   "\033[1;32m"
hash_set colors light_gray    "\033[0;37m"
hash_set colors green         "\033[0;32m"
hash_set colors yellow        "\033[1;33m"
hash_set colors light_purple "\033[1;35m"
hash_set colors purple        "\033[0;35m"
hash_set colors reset_color   "\033[0;00m"


# $1 - keyname
# $2 - value
try_colors() {
        echo -en "$2"
        echo "This line is $1."
}
hash_foreach colors try_colors
hash_echo colors reset_color -en

echo -e '\nLet us overwrite some colors with yellow.\n'
# It's hard to read yellow text on some terminals.
hash_dup colors yellow   red light_green blue green light_gray cyan
hash_foreach colors try_colors
hash_echo colors reset_color -en

echo -e '\nLet us delete them and try colors once more . . .\n'

for i in red light_green blue green light_gray cyan; do
        hash_unset colors $i
done
hash_foreach colors try_colors
hash_echo colors reset_color -en

hash_set other txt "Other examples . . ."
hash_echo other txt
hash_get_into other txt text
echo $text

hash_set other my_fun try_colors
hash_call other my_fun   purple "`hash_echo colors purple`"
hash_echo colors reset_color -en

echo; echo "Back to normal?"; echo
```

```
exit $?

#  On some terminals, the "light" colors print in bold,
#  and end up looking darker than the normal ones.
#  Why is this?
```

An example illustrating the mechanics of hashing, but from a different point of view.

Example A-22. More on hash functions

```
#!/bin/bash
# $Id: ha.sh,v 1.2 2005/04/21 23:24:26 oliver Exp $
# Copyright 2005 Oliver Beckstein
# Released under the GNU Public License
# Author of script granted permission for inclusion in ABS Guide.
# (Thank you!)

#----------------------------------------------------------------
# pseudo hash based on indirect parameter expansion
# API: access through functions:
#
# create the hash:
#
#      newhash Lovers
#
# add entries (note single quotes for spaces)
#
#      addhash Lovers Tristan Isolde
#      addhash Lovers 'Romeo Montague' 'Juliet Capulet'
#
# access value by key
#
#      gethash Lovers Tristan   ---->  Isolde
#
# show all keys
#
#      keyshash Lovers         ----> 'Tristan'  'Romeo Montague'
#
#
# Convention: instead of perls' foo{bar} = boing' syntax,
# use
#       '_foo_bar=boing' (two underscores, no spaces)
#
# 1) store key   in _NAME_keys[]
# 2) store value in _NAME_values[] using the same integer index
# The integer index for the last entry is _NAME_ptr
#
# NOTE: No error or sanity checks, just bare bones.


function _inihash () {
    # private function
    # call at the beginning of each procedure
    # defines: _keys _values _ptr
    #
    # Usage: _inihash NAME
    local name=$1
    _keys=_${name}_keys
    _values=_${name}_values
    _ptr=_${name}_ptr
}
```

```
function newhash () {
    # Usage: newhash NAME
    #        NAME should not contain spaces or dots.
    #        Actually: it must be a legal name for a Bash variable.
    # We rely on Bash automatically recognising arrays.
    local name=$1
    local _keys _values _ptr
    _inihash ${name}
    eval ${_ptr}=0
}


function addhash () {
    # Usage: addhash NAME KEY 'VALUE with spaces'
    #        arguments with spaces need to be quoted with single quotes ''
    local name=$1 k="$2" v="$3"
    local _keys _values _ptr
    _inihash ${name}

    #echo "DEBUG(addhash): ${_ptr}=${!_ptr}"

    eval let ${_ptr}=${_ptr}+1
    eval "$_keys[${!_ptr}]=\"${k}\""
    eval "$_values[${!_ptr}]=\"${v}\""
}

function gethash () {
    #  Usage: gethash NAME KEY
    #         Returns boing
    #         ERR=0 if entry found, 1 otherwise
    #  That's not a proper hash --
    #+ we simply linearly search through the keys.
    local name=$1 key="$2"
    local _keys _values _ptr
    local k v i found h
    _inihash ${name}

    # _ptr holds the highest index in the hash
    found=0

    for i in $(seq 1 ${!_ptr}); do
        h="\${${_keys}[${i}]}"  #  Safer to do it in two steps,
        eval k=${h}             #+ especially when quoting for spaces.
        if [ "${k}" = "${key}" ]; then found=1; break; fi
    done;

    [ ${found} = 0 ] && return 1;
    # else: i is the index that matches the key
    h="\${${_values}[${i}]}"
    eval echo "${h}"
    return 0;
}

function keyshash () {
    # Usage: keyshash NAME
    # Returns list of all keys defined for hash name.
    local name=$1 key="$2"
    local _keys _values _ptr
    local k i h
    _inihash ${name}
```

```
    # _ptr holds the highest index in the hash
    for i in $(seq 1 ${!_ptr}); do
        h="\${${_keys}[${i}]}"   #  Safer to do it in two steps,
        eval k=${h}              #+ especially when quoting for spaces.
        echo -n "'${k}' "
    done;
}


# -------------------------------------------------------------------------

# Now, let's test it.
# (Per comments at the beginning of the script.)
newhash Lovers
addhash Lovers Tristan Isolde
addhash Lovers 'Romeo Montague' 'Juliet Capulet'

# Output results.
echo
gethash Lovers Tristan      # Isolde
echo
keyshash Lovers             # 'Tristan' 'Romeo Montague'
echo; echo


exit 0

# Exercise:
# --------

# Add error checks to the functions.
```

Now for a script that installs and mounts those cute USB keychain solid-state "hard drives."

Example A-23. Mounting USB keychain storage devices

```
#!/bin/bash
# ==> usb.sh
# ==> Script for mounting and installing pen/keychain USB storage devices.
# ==> Runs as root at system startup (see below).
# ==>
# ==> Newer Linux distros (2004 or later) autodetect
# ==> and install USB pen drives, and therefore don't need this script.
# ==> But, it's still instructive.

#  This code is free software covered by GNU GPL license version 2 or above.
#  Please refer to http://www.gnu.org/ for the full license text.
#
#  Some code lifted from usb-mount by Michael Hamilton's usb-mount (LGPL)
#+ see http://users.actrix.co.nz/michael/usbmount.html
#
#  INSTALL
#  -------
#  Put this in /etc/hotplug/usb/diskonkey.
#  Then look in /etc/hotplug/usb.distmap, and copy all usb-storage entries
#+ into /etc/hotplug/usb.usermap, substituting "usb-storage" for "diskonkey".
#  Otherwise this code is only run during the kernel module invocation/removal
#+ (at least in my tests), which defeats the purpose.
#
#  TODO
#  ----
```

```
#  Handle more than one diskonkey device at one time (e.g. /dev/diskonkey1
#+ and /mnt/diskonkey1), etc. The biggest problem here is the handling in
#+ devlabel, which I haven't yet tried.
#
#  AUTHOR and SUPPORT
#  ------------------
#  Konstantin Riabitsev, <icon linux duke edu>.
#  Send any problem reports to my email address at the moment.
#
# ==> Comments added by ABS Guide author.



SYMLINKDEV=/dev/diskonkey
MOUNTPOINT=/mnt/diskonkey
DEVLABEL=/sbin/devlabel
DEVLABELCONFIG=/etc/sysconfig/devlabel
IAM=$0

##
# Functions lifted near-verbatim from usb-mount code.
#
function allAttachedScsiUsb {
  find /proc/scsi/ -path '/proc/scsi/usb-storage*' -type f |
  xargs grep -l 'Attached: Yes'
}
function scsiDevFromScsiUsb {
  echo $1 | awk -F"[-/]" '{ n=$(NF-1);
  print "/dev/sd" substr("abcdefghijklmnopqrstuvwxyz", n+1, 1) }'
}

if [ "${ACTION}" = "add" ] && [ -f "${DEVICE}" ]; then
    ##
    # lifted from usbcam code.
    #
    if [ -f /var/run/console.lock ]; then
        CONSOLEOWNER=`cat /var/run/console.lock`
    elif [ -f /var/lock/console.lock ]; then
        CONSOLEOWNER=`cat /var/lock/console.lock`
    else
        CONSOLEOWNER=
    fi
    for procEntry in $(allAttachedScsiUsb); do
        scsiDev=$(scsiDevFromScsiUsb $procEntry)
        #  Some bug with usb-storage?
        #  Partitions are not in /proc/partitions until they are accessed
        #+ somehow.
        /sbin/fdisk -l $scsiDev >/dev/null
        ##
        #  Most devices have partitioning info, so the data would be on
        #+ /dev/sd?1. However, some stupider ones don't have any partitioning
        #+ and use the entire device for data storage. This tries to
        #+ guess semi-intelligently if we have a /dev/sd?1 and if not, then
        #+ it uses the entire device and hopes for the better.
        #
        if grep -q `basename $scsiDev`1 /proc/partitions; then
            part="$scsiDev""1"
        else
            part=$scsiDev
        fi
        ##
        #  Change ownership of the partition to the console user so they can
```

```
        #+ mount it.
        #
        if [ ! -z "$CONSOLEOWNER" ]; then
            chown $CONSOLEOWNER:disk $part
        fi
        ##
        # This checks if we already have this UUID defined with devlabel.
        # If not, it then adds the device to the list.
        #
        prodid=`$DEVLABEL printid -d $part`
        if ! grep -q $prodid $DEVLABELCONFIG; then
            # cross our fingers and hope it works
            $DEVLABEL add -d $part -s $SYMLINKDEV 2>/dev/null
        fi
        ##
        # Check if the mount point exists and create if it doesn't.
        #
        if [ ! -e $MOUNTPOINT ]; then
            mkdir -p $MOUNTPOINT
        fi
        ##
        # Take care of /etc/fstab so mounting is easy.
        #
        if ! grep -q "^$SYMLINKDEV" /etc/fstab; then
            # Add an fstab entry
            echo -e \
                "$SYMLINKDEV\t\t$MOUNTPOINT\t\tauto\tnoauto,owner,kudzu 0 0" \
                >> /etc/fstab
        fi
    done
    if [ ! -z "$REMOVER" ]; then
        ##
        # Make sure this script is triggered on device removal.
        #
        mkdir -p `dirname $REMOVER`
        ln -s $IAM $REMOVER
    fi
elif [ "${ACTION}" = "remove" ]; then
    ##
    # If the device is mounted, unmount it cleanly.
    #
    if grep -q "$MOUNTPOINT" /etc/mtab; then
        # unmount cleanly
        umount -l $MOUNTPOINT
    fi
    ##
    # Remove it from /etc/fstab if it's there.
    #
    if grep -q "^$SYMLINKDEV" /etc/fstab; then
        grep -v "^$SYMLINKDEV" /etc/fstab > /etc/.fstab.new
        mv -f /etc/.fstab.new /etc/fstab
    fi
fi

exit 0
```

Converting a text file to HTML format.

Example A-24. Converting to HTML

```
#!/bin/bash
# tohtml.sh [v. 0.2.01, reldate: 04/13/12, a teeny bit less buggy]

# Convert a text file to HTML format.
# Author: Mendel Cooper
# License: GPL3
# Usage: sh tohtml.sh < textfile > htmlfile
# Script can easily be modified to accept source and target filenames.

#    Assumptions:
# 1) Paragraphs in (target) text file are separated by a blank line.
# 2) Jpeg images (*.jpg) are located in "images" subdirectory.
#    In the target file, the image names are enclosed in square brackets,
#    for example, [image01.jpg].
# 3) Emphasized (italic) phrases begin with a space+underscore
#+   or the first character on the line is an underscore,
#+   and end with an underscore+space or underscore+end-of-line.


# Settings
FNTSIZE=2        # Small-medium font size
IMGDIR="images"  # Image directory
# Headers
HDR01='<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">'
HDR02='<!-- Converted to HTML by ***tohtml.sh*** script -->'
HDR03='<!-- script author: M. Leo Cooper <thegrendel.abs@gmail.com> -->'
HDR10='<html>'
HDR11='<head>'
HDR11a='</head>'
HDR12a='<title>'
HDR12b='</title>'
HDR121='<META NAME="GENERATOR" CONTENT="tohtml.sh script">'
HDR13='<body bgcolor="#dddddd">'   # Change background color to suit.
HDR14a='<font size='
HDR14b='>'
# Footers
FTR10='</body>'
FTR11='</html>'
# Tags
BOLD="<b>"
CENTER="<center>"
END_CENTER="</center>"
LF="<br>"


write_headers ()
  {
  echo "$HDR01"
  echo
  echo "$HDR02"
  echo "$HDR03"
  echo
  echo
  echo "$HDR10"
  echo "$HDR11"
  echo "$HDR121"
  echo "$HDR11a"
  echo "$HDR13"
  echo
  echo -n "$HDR14a"
  echo -n "$FNTSIZE"
  echo "$HDR14b"
```

```
  echo
  echo "$BOLD"        # Everything in bold (more easily readable).
  }


process_text ()
  {
  while read line     # Read one line at a time.
  do
    {
    if [ ! "$line" ]  # Blank line?
    then              # Then new paragraph must follow.
      echo
      echo "$LF"      # Insert two <br> tags.
      echo "$LF"
      echo
      continue        # Skip the underscore test.
    else              # Otherwise . . .

      if [[ "$line" =~ \[*jpg\] ]]    # Is a graphic?
      then                            # Strip away brackets.
        temp=$( echo "$line" | sed -e 's/\[//' -e 's/\]//' )
        line=""$CENTER" <img src="\"$IMGDIR"/$temp\"> "$END_CENTER" "
                                      # Add image tag.
                                      # And, center it.
      fi

    fi


    echo "$line" | grep -q _
    if [ "$?" -eq 0 ]    # If line contains underscore ...
    then
      # ===================================================
      # Convert underscored phrase to italics.
      temp=$( echo "$line" |
              sed -e 's/ _/ <i>/' -e 's/_/<\/i> /' |
              sed -e 's/^_/<i>/'  -e 's/_/<\/i>/' )
      #  Process only underscores prefixed by space,
      #+ or at beginning or end of line.
      #  Do not convert underscores embedded within a word!
      line="$temp"
      # Slows script execution. Can be optimized?
      # ===================================================
    fi



#   echo
    echo "$line"
#   echo
#   Don't want extra blank lines in generated text!
    } # End while
  done
  }   # End process_text ()


write_footers ()  # Termination tags.
  {
  echo "$FTR10"
  echo "$FTR11"
  }
```

```
# main () {
# =========
write_headers
process_text
write_footers
# =========
#         }

exit $?

#  Exercises:
#  ---------
#  1) Fixup: Check for closing underscore before a comma or period.
#  2) Add a test for the presence of a closing underscore
#+    in phrases to be italicized.
```

Here is something to warm the hearts of webmasters and mistresses: a script that saves weblogs.

Example A-25. Preserving weblogs

```
#!/bin/bash
# archiveweblogs.sh v1.0

# Troy Engel <tengel@fluid.com>
# Slightly modified by document author.
# Used with permission.
#
#  This script will preserve the normally rotated and
#+ thrown away weblogs from a default RedHat/Apache installation.
#  It will save the files with a date/time stamp in the filename,
#+ bzipped, to a given directory.
#
#  Run this from crontab nightly at an off hour,
#+ as bzip2 can suck up some serious CPU on huge logs:
#  0 2 * * * /opt/sbin/archiveweblogs.sh


PROBLEM=66

# Set this to your backup dir.
BKP_DIR=/opt/backups/weblogs

# Default Apache/RedHat stuff
LOG_DAYS="4 3 2 1"
LOG_DIR=/var/log/httpd
LOG_FILES="access_log error_log"

# Default RedHat program locations
LS=/bin/ls
MV=/bin/mv
ID=/usr/bin/id
CUT=/bin/cut
COL=/usr/bin/column
BZ2=/usr/bin/bzip2

# Are we root?
USER=`$ID -u`
if [ "X$USER" != "X0" ]; then
  echo "PANIC: Only root can run this script!"
```

```
  exit $PROBLEM
fi

# Backup dir exists/writable?
if [ ! -x $BKP_DIR ]; then
  echo "PANIC: $BKP_DIR doesn't exist or isn't writable!"
  exit $PROBLEM
fi

# Move, rename and bzip2 the logs
for logday in $LOG_DAYS; do
  for logfile in $LOG_FILES; do
    MYFILE="$LOG_DIR/$logfile.$logday"
    if [ -w $MYFILE ]; then
      DTS=`$LS -lgo --time-style=+%Y%m%d $MYFILE | $COL -t | $CUT -d ' ' -f7`
      $MV $MYFILE $BKP_DIR/$logfile.$DTS
      $BZ2 $BKP_DIR/$logfile.$DTS
    else
      # Only spew an error if the file exits (ergo non-writable).
      if [ -f $MYFILE ]; then
        echo "ERROR: $MYFILE not writable. Skipping."
      fi
    fi
  done
done

exit 0
```

How to keep the shell from expanding and reinterpreting text strings.

Example A-26. Protecting literal strings

```
#! /bin/bash
# protect_literal.sh

# set -vx

:<<-'_Protect_Literal_String_Doc'

    Copyright (c) Michael S. Zick, 2003; All Rights Reserved
    License: Unrestricted reuse in any form, for any purpose.
    Warranty: None
    Revision: $ID$

    Documentation redirected to the Bash no-operation.
    Bash will '/dev/null' this block when the script is first read.
    (Uncomment the above set command to see this action.)

    Remove the first (Sha-Bang) line when sourcing this as a library
    procedure.  Also comment out the example use code in the two
    places where shown.


    Usage:
        _protect_literal_str 'Whatever string meets your ${fancy}'
        Just echos the argument to standard out, hard quotes
        restored.

        $(_protect_literal_str 'Whatever string meets your ${fancy}')
        as the right-hand-side of an assignment statement.
```

```
    Does:
        As the right-hand-side of an assignment, preserves the
        hard quotes protecting the contents of the literal during
        assignment.

    Notes:
        The strange names (_*) are used to avoid trampling on
        the user's chosen names when this is sourced as a
        library.

_Protect_Literal_String_Doc

# The 'for illustration' function form

_protect_literal_str() {

# Pick an un-used, non-printing character as local IFS.
# Not required, but shows that we are ignoring it.
    local IFS=$'\x1B'               # \ESC character

# Enclose the All-Elements-Of in hard quotes during assignment.
    local tmp=$'\x27'$@$'\x27'
#    local tmp=$'\''$@$'\''         # Even uglier.

    local len=${#tmp}               # Info only.
    echo $tmp is $len long.         # Output AND information.
}

# This is the short-named version.
_pls() {
    local IFS=$'x1B'                # \ESC character (not required)
    echo $'\x27'$@$'\x27'           # Hard quoted parameter glob
}

# :<<-'_Protect_Literal_String_Test'
# # # Remove the above "# " to disable this code. # # #

# See how that looks when printed.
echo
echo "- - Test One - -"
_protect_literal_str 'Hello $user'
_protect_literal_str 'Hello "${username}"'
echo

# Which yields:
# - - Test One - -
# 'Hello $user' is 13 long.
# 'Hello "${username}"' is 21 long.

#  Looks as expected, but why all of the trouble?
#  The difference is hidden inside the Bash internal order
#+ of operations.
#  Which shows when you use it on the RHS of an assignment.

# Declare an array for test values.
declare -a arrayZ

# Assign elements with various types of quotes and escapes.
arrayZ=( zero "$(_pls 'Hello ${Me}')" 'Hello ${You}' "\'Pass: ${pw}\'" )

# Now list that array and see what is there.
echo "- - Test Two - -"
```

```
for (( i=0 ; i<${#arrayZ[*]} ; i++ ))
do
    echo  Element $i: ${arrayZ[$i]} is: ${#arrayZ[$i]} long.
done
echo

# Which yields:
# - - Test Two - -
# Element 0: zero is: 4 long.           # Our marker element
# Element 1: 'Hello ${Me}' is: 13 long. # Our "$(_pls '...' )"
# Element 2: Hello ${You} is: 12 long.  # Quotes are missing
# Element 3: \'Pass: \' is: 10 long.    # ${pw} expanded to nothing

# Now make an assignment with that result.
declare -a array2=( ${arrayZ[@]} )

# And print what happened.
echo "- - Test Three - -"
for (( i=0 ; i<${#array2[*]} ; i++ ))
do
    echo  Element $i: ${array2[$i]} is: ${#array2[$i]} long.
done
echo

# Which yields:
# - - Test Three - -
# Element 0: zero is: 4 long.           # Our marker element.
# Element 1: Hello ${Me} is: 11 long.   # Intended result.
# Element 2: Hello is: 5 long.          # ${You} expanded to nothing.
# Element 3: 'Pass: is: 6 long.         # Split on the whitespace.
# Element 4: ' is: 1 long.              # The end quote is here now.

#  Our Element 1 has had its leading and trailing hard quotes stripped.
#  Although not shown, leading and trailing whitespace is also stripped.
#  Now that the string contents are set, Bash will always, internally,
#+ hard quote the contents as required during its operations.

#  Why?
#  Considering our "$(_pls 'Hello ${Me}')" construction:
#  " ... " -> Expansion required, strip the quotes.
#  $( ... ) -> Replace with the result of..., strip this.
#  _pls ' ... ' -> called with literal arguments, strip the quotes.
#  The result returned includes hard quotes; BUT the above processing
#+ has already been done, so they become part of the value assigned.
#
#  Similarly, during further usage of the string variable, the ${Me}
#+ is part of the contents (result) and survives any operations
#  (Until explicitly told to evaluate the string).

#  Hint: See what happens when the hard quotes ($'\x27') are replaced
#+ with soft quotes ($'\x22') in the above procedures.
#  Interesting also is to remove the addition of any quoting.

# _Protect_Literal_String_Test
# # # Remove the above "# " to disable this code. # # #

exit 0
```

But, what if you *want* the shell to expand and reinterpret strings?

Example A-27. Unprotecting literal strings

```
#! /bin/bash
# unprotect_literal.sh

# set -vx

:<<-'_UnProtect_Literal_String_Doc'

    Copyright (c) Michael S. Zick, 2003; All Rights Reserved
    License: Unrestricted reuse in any form, for any purpose.
    Warranty: None
    Revision: $ID$

    Documentation redirected to the Bash no-operation. Bash will
    '/dev/null' this block when the script is first read.
    (Uncomment the above set command to see this action.)

    Remove the first (Sha-Bang) line when sourcing this as a library
    procedure.  Also comment out the example use code in the two
    places where shown.


    Usage:
        Complement of the "$(_pls 'Literal String')" function.
        (See the protect_literal.sh example.)

        StringVar=$(_upls ProtectedSringVariable)

    Does:
        When used on the right-hand-side of an assignment statement;
        makes the substitions embedded in the protected string.

    Notes:
        The strange names (_*) are used to avoid trampling on
        the user's chosen names when this is sourced as a
        library.


_UnProtect_Literal_String_Doc

_upls() {
    local IFS=$'x1B'                # \ESC character (not required)
    eval echo $@                    # Substitution on the glob.
}

# :<<-'_UnProtect_Literal_String_Test'
# # # Remove the above "# " to disable this code. # # #


_pls() {
    local IFS=$'x1B'                # \ESC character (not required)
    echo $'\x27'$@$'\x27'           # Hard quoted parameter glob
}

# Declare an array for test values.
declare -a arrayZ

# Assign elements with various types of quotes and escapes.
arrayZ=( zero "$(_pls 'Hello ${Me}')" 'Hello ${You}' "\'Pass: ${pw}\'" )

# Now make an assignment with that result.
declare -a array2=( ${arrayZ[@]} )
```

```
# Which yielded:
# - - Test Three - -
# Element 0: zero is: 4 long            # Our marker element.
# Element 1: Hello ${Me} is: 11 long    # Intended result.
# Element 2: Hello is: 5 long           # ${You} expanded to nothing.
# Element 3: 'Pass: is: 6 long          # Split on the whitespace.
# Element 4: ' is: 1 long               # The end quote is here now.

# set -vx

#  Initialize 'Me' to something for the embedded ${Me} substitution.
#  This needs to be done ONLY just prior to evaluating the
#+ protected string.
#  (This is why it was protected to begin with.)

Me="to the array guy."

# Set a string variable destination to the result.
newVar=$(_upls ${array2[1]})

# Show what the contents are.
echo $newVar

# Do we really need a function to do this?
newerVar=$(eval echo ${array2[1]})
echo $newerVar

#  I guess not, but the _upls function gives us a place to hang
#+ the documentation on.
#  This helps when we forget what a # construction like:
#+ $(eval echo ... ) means.

# What if Me isn't set when the protected string is evaluated?
unset Me
newestVar=$(_upls ${array2[1]})
echo $newestVar

# Just gone, no hints, no runs, no errors.

#  Why in the world?
#  Setting the contents of a string variable containing character
#+ sequences that have a meaning in Bash is a general problem in
#+ script programming.
#
#  This problem is now solved in eight lines of code
#+ (and four pages of description).

#  Where is all this going?
#  Dynamic content Web pages as an array of Bash strings.
#  Content set per request by a Bash 'eval' command
#+ on the stored page template.
#  Not intended to replace PHP, just an interesting thing to do.
###
#  Don't have a webserver application?
#  No problem, check the example directory of the Bash source;
#+ there is a Bash script for that also.

# _UnProtect_Literal_String_Test
# # # Remove the above "# " to disable this code. # # #

exit 0
```

This interesting script helps hunt down spammers.

Example A-28. Spammer Identification

```
#!/bin/bash

# $Id: is_spammer.bash,v 1.12.2.11 2004/10/01 21:42:33 mszick Exp $
# Above line is RCS info.

# The latest version of this script is available from http://www.morethan.org.
#
# Spammer-identification
# by Michael S. Zick
# Used in the ABS Guide with permission.



#######################################################
# Documentation
# See also "Quickstart" at end of script.
#######################################################

:<<-'__is_spammer_Doc_'

    Copyright (c) Michael S. Zick, 2004
    License: Unrestricted reuse in any form, for any purpose.
    Warranty: None -{Its a script; the user is on their own.}-

Impatient?
    Application code: goto "# # # Hunt the Spammer' program code # # #"
    Example output: ":<<-'_is_spammer_outputs_'"
    How to use: Enter script name without arguments.
                Or goto "Quickstart" at end of script.

Provides
    Given a domain name or IP(v4) address as input:

    Does an exhaustive set of queries to find the associated
    network resources (short of recursing into TLDs).

    Checks the IP(v4) addresses found against Blacklist
    nameservers.

    If found to be a blacklisted IP(v4) address,
    reports the blacklist text records.
    (Usually hyper-links to the specific report.)

Requires
    A working Internet connection.
    (Exercise: Add check and/or abort if not on-line when running script.)
    Bash with arrays (2.05b+).

    The external program 'dig' --
    a utility program provided with the 'bind' set of programs.
    Specifically, the version which is part of Bind series 9.x
    See: http://www.isc.org

    All usages of 'dig' are limited to wrapper functions,
    which may be rewritten as required.
    See: dig_wrappers.bash for details.
         ("Additional documentation" -- below)

Usage
    Script requires a single argument, which may be:
```

```
    1) A domain name;
    2) An IP(v4) address;
    3) A filename, with one name or address per line.

    Script accepts an optional second argument, which may be:
    1) A Blacklist server name;
    2) A filename, with one Blacklist server name per line.

    If the second argument is not provided, the script uses
    a built-in set of (free) Blacklist servers.

    See also, the Quickstart at the end of this script (after 'exit').

Return Codes
    0 - All OK
    1 - Script failure
    2 - Something is Blacklisted

Optional environment variables
    SPAMMER_TRACE
        If set to a writable file,
        script will log an execution flow trace.

    SPAMMER_DATA
        If set to a writable file, script will dump its
        discovered data in the form of GraphViz file.
        See: http://www.research.att.com/sw/tools/graphviz

    SPAMMER_LIMIT
        Limits the depth of resource tracing.

        Default is 2 levels.

        A setting of 0 (zero) means 'unlimited' . . .
          Caution: script might recurse the whole Internet!

        A limit of 1 or 2 is most useful when processing
        a file of domain names and addresses.
        A higher limit can be useful when hunting spam gangs.


Additional documentation
    Download the archived set of scripts
    explaining and illustrating the function contained within this script.
    http://bash.deta.in/mszick_clf.tar.bz2


Study notes
    This script uses a large number of functions.
    Nearly all general functions have their own example script.
    Each of the example scripts have tutorial level comments.

Scripting project
    Add support for IP(v6) addresses.
    IP(v6) addresses are recognized but not processed.

Advanced project
    Add the reverse lookup detail to the discovered information.

    Report the delegation chain and abuse contacts.

    Modify the GraphViz file output to include the
```

```
    newly discovered information.

__is_spammer_Doc_

#######################################################




#### Special IFS settings used for string parsing. ####

# Whitespace == :Space:Tab:Line Feed:Carriage Return:
WSP_IFS=$'\x20'$'\x09'$'\x0A'$'\x0D'

# No Whitespace == Line Feed:Carriage Return
NO_WSP=$'\x0A'$'\x0D'

# Field separator for dotted decimal IP addresses
ADR_IFS=${NO_WSP}'.'

# Array to dotted string conversions
DOT_IFS='.'${WSP_IFS}

# # # Pending operations stack machine # # #
# This set of functions described in func_stack.bash.
# (See "Additional documentation" above.)
# # #

# Global stack of pending operations.
declare -f -a _pending_
# Global sentinel for stack runners
declare -i _p_ctrl_
# Global holder for currently executing function
declare -f _pend_current_

# # # Debug version only - remove for regular use # # #
#
# The function stored in _pend_hook_ is called
# immediately before each pending function is
# evaluated.  Stack clean, _pend_current_ set.
#
# This thingy demonstrated in pend_hook.bash.
declare -f _pend_hook_
# # #

# The do nothing function
pend_dummy() { : ; }

# Clear and initialize the function stack.
pend_init() {
    unset _pending_[@]
    pend_func pend_stop_mark
    _pend_hook_='pend_dummy'  # Debug only.
}

# Discard the top function on the stack.
pend_pop() {
    if [ ${#_pending_[@]} -gt 0 ]
    then
        local -i _top_
        _top_=${#_pending_[@]}-1
        unset _pending_[$_top_]
```

```
    fi
}

# pend_func function_name [$(printf '%q\n' arguments)]
pend_func() {
    local IFS=${NO_WSP}
    set -f
    _pending_[${#_pending_[@]}]=$@
    set +f
}

# The function which stops the release:
pend_stop_mark() {
    _p_ctrl_=0
}

pend_mark() {
    pend_func pend_stop_mark
}

# Execute functions until 'pend_stop_mark' . . .
pend_release() {
    local -i _top_             # Declare _top_ as integer.
    _p_ctrl_=${#_pending_[@]}
    while [ ${_p_ctrl_} -gt 0 ]
    do
       _top_=${#_pending_[@]}-1
       _pend_current_=${_pending_[$_top_]}
       unset _pending_[$_top_]
       $_pend_hook_            # Debug only.
       eval $_pend_current_
    done
}

# Drop functions until 'pend_stop_mark' . . .
pend_drop() {
    local -i _top_
    local _pd_ctrl_=${#_pending_[@]}
    while [ ${_pd_ctrl_} -gt 0 ]
    do
       _top_=$_pd_ctrl_-1
       if [ "${_pending_[$_top_]}" == 'pend_stop_mark' ]
       then
           unset _pending_[$_top_]
           break
       else
           unset _pending_[$_top_]
           _pd_ctrl_=$_top_
       fi
    done
    if [ ${#_pending_[@]} -eq 0 ]
    then
        pend_func pend_stop_mark
    fi
}

#### Array editors ####

# This function described in edit_exact.bash.
# (See "Additional documentation," above.)
# edit_exact <excludes_array_name> <target_array_name>
edit_exact() {
```

```
    [ $# -eq 2 ] ||
    [ $# -eq 3 ] || return 1
    local -a _ee_Excludes
    local -a _ee_Target
    local _ee_x
    local _ee_t
    local IFS=${NO_WSP}
    set -f
    eval _ee_Excludes=\( \$\{$1\[@\]\} \)
    eval _ee_Target=\( \$\{$2\[@\]\} \)
    local _ee_len=${#_ee_Target[@]}     # Original length.
    local _ee_cnt=${#_ee_Excludes[@]}   # Exclude list length.
    [ ${_ee_len} -ne 0 ] || return 0    # Can't edit zero length.
    [ ${_ee_cnt} -ne 0 ] || return 0    # Can't edit zero length.
    for (( x = 0; x < ${_ee_cnt} ; x++ ))
    do
        _ee_x=${_ee_Excludes[$x]}
        for (( n = 0 ; n < ${_ee_len} ; n++ ))
        do
            _ee_t=${_ee_Target[$n]}
            if [ x"${_ee_t}" == x"${_ee_x}" ]
            then
                unset _ee_Target[$n]     # Discard match.
                [ $# -eq 2 ] && break    # If 2 arguments, then done.
            fi
        done
    done
    eval $2=\( \$\{_ee_Target\[@\]\} \)
    set +f
    return 0
}

# This function described in edit_by_glob.bash.
# edit_by_glob <excludes_array_name> <target_array_name>
edit_by_glob() {
    [ $# -eq 2 ] ||
    [ $# -eq 3 ] || return 1
    local -a _ebg_Excludes
    local -a _ebg_Target
    local _ebg_x
    local _ebg_t
    local IFS=${NO_WSP}
    set -f
    eval _ebg_Excludes=\( \$\{$1\[@\]\} \)
    eval _ebg_Target=\( \$\{$2\[@\]\} \)
    local _ebg_len=${#_ebg_Target[@]}
    local _ebg_cnt=${#_ebg_Excludes[@]}
    [ ${_ebg_len} -ne 0 ] || return 0
    [ ${_ebg_cnt} -ne 0 ] || return 0
    for (( x = 0; x < ${_ebg_cnt} ; x++ ))
    do
        _ebg_x=${_ebg_Excludes[$x]}
        for (( n = 0 ; n < ${_ebg_len} ; n++ ))
        do
            [ $# -eq 3 ] && _ebg_x=${_ebg_x}'*'  #  Do prefix edit
            if [ ${_ebg_Target[$n]:=} ]          #+ if defined & set.
            then
                _ebg_t=${_ebg_Target[$n]/#${_ebg_x}/}
                [ ${#_ebg_t} -eq 0 ] && unset _ebg_Target[$n]
            fi
        done
    done
```

```
    eval $2=\( \$\{_ebg_Target\[@\]\} \)
    set +f
    return 0
}

# This function described in unique_lines.bash.
# unique_lines <in_name> <out_name>
unique_lines() {
    [ $# -eq 2 ] || return 1
    local -a _ul_in
    local -a _ul_out
    local -i _ul_cnt
    local -i _ul_pos
    local _ul_tmp
    local IFS=${NO_WSP}
    set -f
    eval _ul_in=\( \$\{$1\[@\]\} \)
    _ul_cnt=${#_ul_in[@]}
    for (( _ul_pos = 0 ; _ul_pos < ${_ul_cnt} ; _ul_pos++ ))
    do
        if [ ${_ul_in[${_ul_pos}]:=} ]      # If defined & not empty
        then
            _ul_tmp=${_ul_in[${_ul_pos}]}
            _ul_out[${#_ul_out[@]}]=${_ul_tmp}
            for (( zap = _ul_pos ; zap < ${_ul_cnt} ; zap++ ))
            do
                [ ${_ul_in[${zap}]:=} ] &&
                [ 'x'${_ul_in[${zap}]} == 'x'${_ul_tmp} ] &&
                    unset _ul_in[${zap}]
            done
        fi
    done
    eval $2=\( \$\{_ul_out\[@\]\} \)
    set +f
    return 0
}

# This function described in char_convert.bash.
# to_lower <string>
to_lower() {
    [ $# -eq 1 ] || return 1
    local _tl_out
    _tl_out=${1//A/a}
    _tl_out=${_tl_out//B/b}
    _tl_out=${_tl_out//C/c}
    _tl_out=${_tl_out//D/d}
    _tl_out=${_tl_out//E/e}
    _tl_out=${_tl_out//F/f}
    _tl_out=${_tl_out//G/g}
    _tl_out=${_tl_out//H/h}
    _tl_out=${_tl_out//I/i}
    _tl_out=${_tl_out//J/j}
    _tl_out=${_tl_out//K/k}
    _tl_out=${_tl_out//L/l}
    _tl_out=${_tl_out//M/m}
    _tl_out=${_tl_out//N/n}
    _tl_out=${_tl_out//O/o}
    _tl_out=${_tl_out//P/p}
    _tl_out=${_tl_out//Q/q}
    _tl_out=${_tl_out//R/r}
    _tl_out=${_tl_out//S/s}
    _tl_out=${_tl_out//T/t}
```

```
    _tl_out=${_tl_out//U/u}
    _tl_out=${_tl_out//V/v}
    _tl_out=${_tl_out//W/w}
    _tl_out=${_tl_out//X/x}
    _tl_out=${_tl_out//Y/y}
    _tl_out=${_tl_out//Z/z}
    echo ${_tl_out}
    return 0
}

#### Application helper functions ####

# Not everybody uses dots as separators (APNIC, for example).
# This function described in to_dot.bash
# to_dot <string>
to_dot() {
    [ $# -eq 1 ] || return 1
    echo ${1//[#|@|%]/.}
    return 0
}

# This function described in is_number.bash.
# is_number <input>
is_number() {
    [ "$#" -eq 1 ]    || return 1  # is blank?
    [ x"$1" == 'x0' ] && return 0  # is zero?
    local -i tst
    let tst=$1 2>/dev/null         # else is numeric!
    return $?
}

# This function described in is_address.bash.
# is_address <input>
is_address() {
    [ $# -eq 1 ] || return 1    # Blank ==> false
    local -a _ia_input
    local IFS=${ADR_IFS}
    _ia_input=( $1 )
    if  [ ${#_ia_input[@]} -eq 4 ]  &&
        is_number ${_ia_input[0]}   &&
        is_number ${_ia_input[1]}   &&
        is_number ${_ia_input[2]}   &&
        is_number ${_ia_input[3]}   &&
        [ ${_ia_input[0]} -lt 256 ] &&
        [ ${_ia_input[1]} -lt 256 ] &&
        [ ${_ia_input[2]} -lt 256 ] &&
        [ ${_ia_input[3]} -lt 256 ]
    then
        return 0
    else
        return 1
    fi
}

#  This function described in split_ip.bash.
#  split_ip <IP_address>
#+ <array_name_norm> [<array_name_rev>]
split_ip() {
    [ $# -eq 3 ] ||              #  Either three
    [ $# -eq 2 ] || return 1     #+ or two arguments
    local -a _si_input
    local IFS=${ADR_IFS}
```

```
    _si_input=( $1 )
    IFS=${WSP_IFS}
    eval $2=\(\ \$\{_si_input\[@\]\}\ \)
    if [ $# -eq 3 ]
    then
        # Build query order array.
        local -a _dns_ip
        _dns_ip[0]=${_si_input[3]}
        _dns_ip[1]=${_si_input[2]}
        _dns_ip[2]=${_si_input[1]}
        _dns_ip[3]=${_si_input[0]}
        eval $3=\(\ \$\{_dns_ip\[@\]\}\ \)
    fi
    return 0
}

# This function described in dot_array.bash.
# dot_array <array_name>
dot_array() {
    [ $# -eq 1 ] || return 1     # Single argument required.
    local -a _da_input
    eval _da_input=\(\ \$\{$1\[@\]\}\ \)
    local IFS=${DOT_IFS}
    local _da_output=${_da_input[@]}
    IFS=${WSP_IFS}
    echo ${_da_output}
    return 0
}

# This function described in file_to_array.bash
# file_to_array <file_name> <line_array_name>
file_to_array() {
    [ $# -eq 2 ] || return 1  # Two arguments required.
    local IFS=${NO_WSP}
    local -a _fta_tmp_
    _fta_tmp_=( $(cat $1) )
    eval $2=\( \$\{_fta_tmp_\[@\]\} \)
    return 0
}

#  Columnized print of an array of multi-field strings.
#  col_print <array_name> <min_space> <
#+ tab_stop [tab_stops]>
col_print() {
    [ $# -gt 2 ] || return 0
    local -a _cp_inp
    local -a _cp_spc
    local -a _cp_line
    local _cp_min
    local _cp_mcnt
    local _cp_pos
    local _cp_cnt
    local _cp_tab
    local -i _cp
    local -i _cpf
    local _cp_fld
    # WARNING: FOLLOWING LINE NOT BLANK -- IT IS QUOTED SPACES.
    local _cp_max='                                                            '
    set -f
    local IFS=${NO_WSP}
    eval _cp_inp=\(\ \$\{$1\[@\]\}\ \)
    [ ${#_cp_inp[@]} -gt 0 ] || return 0 # Empty is easy.
```

```
    _cp_mcnt=$2
    _cp_min=${_cp_max:1:${_cp_mcnt}}
    shift
    shift
    _cp_cnt=$#
    for (( _cp = 0 ; _cp < _cp_cnt ; _cp++ ))
    do
        _cp_spc[${#_cp_spc[@]}]="${_cp_max:2:$1}" #"
        shift
    done
    _cp_cnt=${#_cp_inp[@]}
    for (( _cp = 0 ; _cp < _cp_cnt ; _cp++ ))
    do
        _cp_pos=1
        IFS=${NO_WSP}$'\x20'
        _cp_line=( ${_cp_inp[${_cp}]} )
        IFS=${NO_WSP}
        for (( _cpf = 0 ; _cpf < ${#_cp_line[@]} ; _cpf++ ))
        do
            _cp_tab=${_cp_spc[${_cpf}]:${_cp_pos}}
            if [ ${#_cp_tab} -lt ${_cp_mcnt} ]
            then
                _cp_tab="${_cp_min}"
            fi
            echo -n "${_cp_tab}"
            (( _cp_pos = ${_cp_pos} + ${#_cp_tab} ))
            _cp_fld="${_cp_line[${_cpf}]}"
            echo -n ${_cp_fld}
            (( _cp_pos = ${_cp_pos} + ${#_cp_fld} ))
        done
        echo
    done
    set +f
    return 0
}

# # # # 'Hunt the Spammer' data flow # # # #

# Application return code
declare -i _hs_RC

# Original input, from which IP addresses are removed
# After which, domain names to check
declare -a uc_name

# Original input IP addresses are moved here
# After which, IP addresses to check
declare -a uc_address

# Names against which address expansion run
# Ready for name detail lookup
declare -a chk_name

# Addresses against which name expansion run
# Ready for address detail lookup
declare -a chk_address

#  Recursion is depth-first-by-name.
#  The expand_input_address maintains this list
#+ to prohibit looking up addresses twice during
#+ domain name recursion.
declare -a been_there_addr
```

```
been_there_addr=( '127.0.0.1' ) # Whitelist localhost

# Names which we have checked (or given up on)
declare -a known_name

# Addresses which we have checked (or given up on)
declare -a known_address

#  List of zero or more Blacklist servers to check.
#  Each 'known_address' will be checked against each server,
#+ with negative replies and failures suppressed.
declare -a list_server

# Indirection limit - set to zero == no limit
indirect=${SPAMMER_LIMIT:=2}

# # # # 'Hunt the Spammer' information output data # # # #

# Any domain name may have multiple IP addresses.
# Any IP address may have multiple domain names.
# Therefore, track unique address-name pairs.
declare -a known_pair
declare -a reverse_pair

#  In addition to the data flow variables; known_address
#+ known_name and list_server, the following are output to the
#+ external graphics interface file.

# Authority chain, parent -> SOA fields.
declare -a auth_chain

# Reference chain, parent name -> child name
declare -a ref_chain

# DNS chain - domain name -> address
declare -a name_address

# Name and service pairs - domain name -> service
declare -a name_srvc

# Name and resource pairs - domain name -> Resource Record
declare -a name_resource

# Parent and Child pairs - parent name -> child name
# This MAY NOT be the same as the ref_chain followed!
declare -a parent_child

# Address and Blacklist hit pairs - address->server
declare -a address_hits

# Dump interface file data
declare -f _dot_dump
_dot_dump=pend_dummy   # Initially a no-op

#  Data dump is enabled by setting the environment variable SPAMMER_DATA
#+ to the name of a writable file.
declare _dot_file

# Helper function for the dump-to-dot-file function
# dump_to_dot <array_name> <prefix>
dump_to_dot() {
    local -a _dda_tmp
```

```
    local -i _dda_cnt
    local _dda_form='    '${2}'%04u %s\n'
    local IFS=${NO_WSP}
    eval _dda_tmp=\(\ \$\{$1\[@\]\}\ \)
    _dda_cnt=${#_dda_tmp[@]}
    if [ ${_dda_cnt} -gt 0 ]
    then
        for (( _dda = 0 ; _dda < _dda_cnt ; _dda++ ))
        do
            printf "${_dda_form}" \
                   "${_dda}" "${_dda_tmp[${_dda}]}" >>${_dot_file}
        done
    fi
}

# Which will also set _dot_dump to this function . . .
dump_dot() {
    local -i _dd_cnt
    echo '# Data vintage: '$(date -R) >${_dot_file}
    echo '# ABS Guide: is_spammer.bash; v2, 2004-msz' >>${_dot_file}
    echo >>${_dot_file}
    echo 'digraph G {' >>${_dot_file}

    if [ ${#known_name[@]} -gt 0 ]
    then
        echo >>${_dot_file}
        echo '# Known domain name nodes' >>${_dot_file}
        _dd_cnt=${#known_name[@]}
        for (( _dd = 0 ; _dd < _dd_cnt ; _dd++ ))
        do
            printf '    N%04u [label="%s"] ;\n' \
                   "${_dd}" "${known_name[${_dd}]}" >>${_dot_file}
        done
    fi

    if [ ${#known_address[@]} -gt 0 ]
    then
        echo >>${_dot_file}
        echo '# Known address nodes' >>${_dot_file}
        _dd_cnt=${#known_address[@]}
        for (( _dd = 0 ; _dd < _dd_cnt ; _dd++ ))
        do
            printf '    A%04u [label="%s"] ;\n' \
                   "${_dd}" "${known_address[${_dd}]}" >>${_dot_file}
        done
    fi

    echo                                   >>${_dot_file}
    echo '/*'                              >>${_dot_file}
    echo ' * Known relationships :: User conversion to'  >>${_dot_file}
    echo ' * graphic form by hand or program required.'  >>${_dot_file}
    echo ' *'                              >>${_dot_file}

    if [ ${#auth_chain[@]} -gt 0 ]
    then
      echo >>${_dot_file}
      echo '# Authority ref. edges followed & field source.' >>${_dot_file}
        dump_to_dot auth_chain AC
    fi

    if [ ${#ref_chain[@]} -gt 0 ]
    then
```

```
        echo >>${_dot_file}
        echo '# Name ref. edges followed and field source.' >>${_dot_file}
        dump_to_dot ref_chain RC
    fi

    if [ ${#name_address[@]} -gt 0 ]
    then
        echo >>${_dot_file}
        echo '# Known name->address edges' >>${_dot_file}
        dump_to_dot name_address NA
    fi

    if [ ${#name_srvc[@]} -gt 0 ]
    then
        echo >>${_dot_file}
        echo '# Known name->service edges' >>${_dot_file}
        dump_to_dot name_srvc NS
    fi

    if [ ${#name_resource[@]} -gt 0 ]
    then
        echo >>${_dot_file}
        echo '# Known name->resource edges' >>${_dot_file}
        dump_to_dot name_resource NR
    fi

    if [ ${#parent_child[@]} -gt 0 ]
    then
        echo >>${_dot_file}
        echo '# Known parent->child edges' >>${_dot_file}
        dump_to_dot parent_child PC
    fi

    if [ ${#list_server[@]} -gt 0 ]
    then
        echo >>${_dot_file}
        echo '# Known Blacklist nodes' >>${_dot_file}
        _dd_cnt=${#list_server[@]}
        for (( _dd = 0 ; _dd < _dd_cnt ; _dd++ ))
        do
            printf '    LS%04u [label="%s"] ;\n' \
                   "${_dd}" "${list_server[${_dd}]}" >>${_dot_file}
        done
    fi

    unique_lines address_hits address_hits
    if [ ${#address_hits[@]} -gt 0 ]
    then
      echo >>${_dot_file}
      echo '# Known address->Blacklist_hit edges' >>${_dot_file}
      echo '# CAUTION: dig warnings can trigger false hits.' >>${_dot_file}
       dump_to_dot address_hits AH
    fi
    echo          >>${_dot_file}
    echo ' *'     >>${_dot_file}
    echo ' * That is a lot of relationships. Happy graphing.' >>${_dot_file}
    echo ' */'    >>${_dot_file}
    echo '}'      >>${_dot_file}
    return 0
}

# # # # 'Hunt the Spammer' execution flow # # # #
```

```
#  Execution trace is enabled by setting the
#+ environment variable SPAMMER_TRACE to the name of a writable file.
declare -a _trace_log
declare _log_file

# Function to fill the trace log
trace_logger() {
    _trace_log[${#_trace_log[@]}]=${_pend_current_}
}

# Dump trace log to file function variable.
declare -f _log_dump
_log_dump=pend_dummy   # Initially a no-op.

# Dump the trace log to a file.
dump_log() {
    local -i _dl_cnt
    _dl_cnt=${#_trace_log[@]}
    for (( _dl = 0 ; _dl < _dl_cnt ; _dl++ ))
    do
        echo ${_trace_log[${_dl}]} >> ${_log_file}
    done
    _dl_cnt=${#_pending_[@]}
    if [ ${_dl_cnt} -gt 0 ]
    then
        _dl_cnt=${_dl_cnt}-1
        echo '# # # Operations stack not empty # # #' >> ${_log_file}
        for (( _dl = ${_dl_cnt} ; _dl >= 0 ; _dl-- ))
        do
            echo ${_pending_[${_dl}]} >> ${_log_file}
        done
    fi
}

# # # Utility program 'dig' wrappers # # #
#
#  These wrappers are derived from the
#+ examples shown in dig_wrappers.bash.
#
#  The major difference is these return
#+ their results as a list in an array.
#
#  See dig_wrappers.bash for details and
#+ use that script to develop any changes.
#
# # #

# Short form answer: 'dig' parses answer.

# Forward lookup :: Name -> Address
# short_fwd <domain_name> <array_name>
short_fwd() {
    local -a _sf_reply
    local -i _sf_rc
    local -i _sf_cnt
    IFS=${NO_WSP}
echo -n '.'
# echo 'sfwd: '${1}
  _sf_reply=( $(dig +short ${1} -c in -t a 2>/dev/null) )
  _sf_rc=$?
  if [ ${_sf_rc} -ne 0 ]
```

```
  then
    _trace_log[${#_trace_log[@]}]='## Lookup error '${_sf_rc}' on '${1}' ##'
# [ ${_sf_rc} -ne 9 ] && pend_drop
        return ${_sf_rc}
    else
        # Some versions of 'dig' return warnings on stdout.
        _sf_cnt=${#_sf_reply[@]}
        for (( _sf = 0 ; _sf < ${_sf_cnt} ; _sf++ ))
        do
            [ 'x'${_sf_reply[${_sf}]:0:2} == 'x;;' ] &&
                unset _sf_reply[${_sf}]
        done
        eval $2=\( \$\{_sf_reply\[@\]\} \)
    fi
    return 0
}

# Reverse lookup :: Address -> Name
# short_rev <ip_address> <array_name>
short_rev() {
    local -a _sr_reply
    local -i _sr_rc
    local -i _sr_cnt
    IFS=${NO_WSP}
echo -n '.'
# echo 'srev: '${1}
  _sr_reply=( $(dig +short -x ${1} 2>/dev/null) )
  _sr_rc=$?
  if [ ${_sr_rc} -ne 0 ]
  then
    _trace_log[${#_trace_log[@]}]='## Lookup error '${_sr_rc}' on '${1}' ##'
# [ ${_sr_rc} -ne 9 ] && pend_drop
        return ${_sr_rc}
    else
        # Some versions of 'dig' return warnings on stdout.
        _sr_cnt=${#_sr_reply[@]}
        for (( _sr = 0 ; _sr < ${_sr_cnt} ; _sr++ ))
        do
            [ 'x'${_sr_reply[${_sr}]:0:2} == 'x;;' ] &&
                unset _sr_reply[${_sr}]
        done
        eval $2=\( \$\{_sr_reply\[@\]\} \)
    fi
    return 0
}

# Special format lookup used to query blacklist servers.
# short_text <ip_address> <array_name>
short_text() {
    local -a _st_reply
    local -i _st_rc
    local -i _st_cnt
    IFS=${NO_WSP}
# echo 'stxt: '${1}
  _st_reply=( $(dig +short ${1} -c in -t txt 2>/dev/null) )
  _st_rc=$?
  if [ ${_st_rc} -ne 0 ]
  then
    _trace_log[${#_trace_log[@]}]='##Text lookup error '${_st_rc}' on '${1}'##'
# [ ${_st_rc} -ne 9 ] && pend_drop
        return ${_st_rc}
    else
```

```
        # Some versions of 'dig' return warnings on stdout.
        _st_cnt=${#_st_reply[@]}
        for (( _st = 0 ; _st < ${#_st_cnt} ; _st++ ))
        do
            [ 'x'${_st_reply[${_st}]:0:2} == 'x;;' ] &&
                unset _st_reply[${_st}]
        done
        eval $2=\( \$\{_st_reply\[@\]\} \)
    fi
    return 0
}

# The long forms, a.k.a., the parse it yourself versions

# RFC 2782   Service lookups
# dig +noall +nofail +answer _ldap._tcp.openldap.org -t srv
# _<service>._<protocol>.<domain_name>
# _ldap._tcp.openldap.org. 3600   IN     SRV    0 0 389 ldap.openldap.org.
# domain TTL Class SRV Priority Weight Port Target

# Forward lookup :: Name -> poor man's zone transfer
# long_fwd <domain_name> <array_name>
long_fwd() {
    local -a _lf_reply
    local -i _lf_rc
    local -i _lf_cnt
    IFS=${NO_WSP}
echo -n ':'
# echo 'lfwd: '${1}
  _lf_reply=( $(
     dig +noall +nofail +answer +authority +additional \
         ${1} -t soa ${1} -t mx ${1} -t any 2>/dev/null) )
  _lf_rc=$?
  if [ ${_lf_rc} -ne 0 ]
  then
    _trace_log[${#_trace_log[@]}]='# Zone lookup err '${_lf_rc}' on '${1}' #'
# [ ${_lf_rc} -ne 9 ] && pend_drop
        return ${_lf_rc}
    else
        # Some versions of 'dig' return warnings on stdout.
        _lf_cnt=${#_lf_reply[@]}
        for (( _lf = 0 ; _lf < ${_lf_cnt} ; _lf++ ))
        do
            [ 'x'${_lf_reply[${_lf}]:0:2} == 'x;;' ] &&
                unset _lf_reply[${_lf}]
        done
        eval $2=\( \$\{_lf_reply\[@\]\} \)
    fi
    return 0
}
#  The reverse lookup domain name corresponding to the IPv6 address:
#      4321:0:1:2:3:4:567:89ab
#  would be (nibble, I.E: Hexdigit) reversed:
#  b.a.9.8.7.6.5.0.4.0.0.0.3.0.0.0.2.0.0.0.1.0.0.0.0.0.0.0.1.2.3.4.IP6.ARPA.

# Reverse lookup :: Address -> poor man's delegation chain
# long_rev <rev_ip_address> <array_name>
long_rev() {
    local -a _lr_reply
    local -i _lr_rc
    local -i _lr_cnt
    local _lr_dns
```

```
    _lr_dns=${1}'.in-addr.arpa.'
    IFS=${NO_WSP}
echo -n ':'
# echo 'lrev: '${1}
  _lr_reply=( $(
      dig +noall +nofail +answer +authority +additional \
          ${_lr_dns} -t soa ${_lr_dns} -t any 2>/dev/null) )
  _lr_rc=$?
  if [ ${_lr_rc} -ne 0 ]
  then
    _trace_log[${#_trace_log[@]}]='# Deleg lkp error '${_lr_rc}' on '${1}' #'
# [ ${_lr_rc} -ne 9 ] && pend_drop
        return ${_lr_rc}
    else
        # Some versions of 'dig' return warnings on stdout.
        _lr_cnt=${#_lr_reply[@]}
        for (( _lr = 0 ; _lr < ${_lr_cnt} ; _lr++ ))
        do
            [ 'x'${_lr_reply[${_lr}]:0:2} == 'x;;' ] &&
                unset _lr_reply[${_lr}]
        done
        eval $2=\( \$\{_lr_reply\[@\]\} \)
    fi
    return 0
}

# # # Application specific functions # # #

# Mung a possible name; suppresses root and TLDs.
# name_fixup <string>
name_fixup(){
    local -a _nf_tmp
    local -i _nf_end
    local _nf_str
    local IFS
    _nf_str=$(to_lower ${1})
    _nf_str=$(to_dot ${_nf_str})
    _nf_end=${#_nf_str}-1
    [ ${_nf_str:${_nf_end}} != '.' ] &&
        _nf_str=${_nf_str}'.'
    IFS=${ADR_IFS}
    _nf_tmp=( ${_nf_str} )
    IFS=${WSP_IFS}
    _nf_end=${#_nf_tmp[@]}
    case ${_nf_end} in
    0) # No dots, only dots.
        echo
        return 1
    ;;
    1) # Only a TLD.
        echo
        return 1
    ;;
    2) # Maybe okay.
       echo ${_nf_str}
       return 0
       # Needs a lookup table?
       if [ ${#_nf_tmp[1]} -eq 2 ]
       then # Country coded TLD.
           echo
           return 1
       else
```

```
            echo ${_nf_str}
            return 0
        fi
    ;;
    esac
    echo ${_nf_str}
    return 0
}

# Grope and mung original input(s).
split_input() {
    [ ${#uc_name[@]} -gt 0 ] || return 0
    local -i _si_cnt
    local -i _si_len
    local _si_str
    unique_lines uc_name uc_name
    _si_cnt=${#uc_name[@]}
    for (( _si = 0 ; _si < _si_cnt ; _si++ ))
    do
        _si_str=${uc_name[$_si]}
        if is_address ${_si_str}
        then
            uc_address[${#uc_address[@]}]=${_si_str}
            unset uc_name[$_si]
        else
            if ! uc_name[$_si]=$(name_fixup ${_si_str})
            then
                unset ucname[$_si]
            fi
        fi
    done
  uc_name=( ${uc_name[@]} )
  _si_cnt=${#uc_name[@]}
  _trace_log[${#_trace_log[@]}]='#Input '${_si_cnt}' unchkd name input(s).#'
  _si_cnt=${#uc_address[@]}
  _trace_log[${#_trace_log[@]}]='#Input '${_si_cnt}' unchkd addr input(s).#'
    return 0
}

# # # Discovery functions -- recursively interlocked by external data # # #
# # # The leading 'if list is empty; return 0' in each is required. # # #

# Recursion limiter
# limit_chk() <next_level>
limit_chk() {
    local -i _lc_lmt
    # Check indirection limit.
    if [ ${indirect} -eq 0 ] || [ $# -eq 0 ]
    then
        # The 'do-forever' choice
        echo 1                  # Any value will do.
        return 0                # OK to continue.
    else
        # Limiting is in effect.
        if [ ${indirect} -lt ${1} ]
        then
            echo ${1}           # Whatever.
            return 1            # Stop here.
        else
            _lc_lmt=${1}+1      # Bump the given limit.
            echo ${_lc_lmt}     # Echo it.
            return 0            # OK to continue.
```

```
        fi
    fi
}

# For each name in uc_name:
#     Move name to chk_name.
#     Add addresses to uc_address.
#     Pend expand_input_address.
#     Repeat until nothing new found.
# expand_input_name <indirection_limit>
expand_input_name() {
    [ ${#uc_name[@]} -gt 0 ] || return 0
    local -a _ein_addr
    local -a _ein_new
    local -i _ucn_cnt
    local -i _ein_cnt
    local _ein_tst
    _ucn_cnt=${#uc_name[@]}

    if  ! _ein_cnt=$(limit_chk ${1})
    then
        return 0
    fi

    for (( _ein = 0 ; _ein < _ucn_cnt ; _ein++ ))
    do
        if short_fwd ${uc_name[${_ein}]} _ein_new
        then
          for (( _ein_cnt = 0 ; _ein_cnt < ${#_ein_new[@]}; _ein_cnt++ ))
          do
              _ein_tst=${_ein_new[${_ein_cnt}]}
              if is_address ${_ein_tst}
              then
                  _ein_addr[${#_ein_addr[@]}]=${_ein_tst}
              fi
    done
        fi
    done
    unique_lines _ein_addr _ein_addr     # Scrub duplicates.
    edit_exact chk_address _ein_addr     # Scrub pending detail.
    edit_exact known_address _ein_addr   # Scrub already detailed.
 if [ ${#_ein_addr[@]} -gt 0 ]        # Anything new?
 then
   uc_address=( ${uc_address[@]} ${_ein_addr[@]} )
   pend_func expand_input_address ${1}
   _trace_log[${#_trace_log[@]}]='#Add '${#_ein_addr[@]}' unchkd addr inp.#'
    fi
    edit_exact chk_name uc_name          # Scrub pending detail.
    edit_exact known_name uc_name        # Scrub already detailed.
    if [ ${#uc_name[@]} -gt 0 ]
    then
        chk_name=( ${chk_name[@]} ${uc_name[@]}  )
        pend_func detail_each_name ${1}
    fi
    unset uc_name[@]
    return 0
}

# For each address in uc_address:
#     Move address to chk_address.
#     Add names to uc_name.
#     Pend expand_input_name.
```

```
#     Repeat until nothing new found.
# expand_input_address <indirection_limit>
expand_input_address() {
    [ ${#uc_address[@]} -gt 0 ] || return 0
    local -a _eia_addr
    local -a _eia_name
    local -a _eia_new
    local -i _uca_cnt
    local -i _eia_cnt
    local _eia_tst
    unique_lines uc_address _eia_addr
    unset uc_address[@]
    edit_exact been_there_addr _eia_addr
    _uca_cnt=${#_eia_addr[@]}
    [ ${_uca_cnt} -gt 0 ] &&
        been_there_addr=( ${been_there_addr[@]} ${_eia_addr[@]} )

    for (( _eia = 0 ; _eia < _uca_cnt ; _eia++ ))
     do
       if short_rev ${_eia_addr[${_eia}]} _eia_new
       then
         for (( _eia_cnt = 0 ; _eia_cnt < ${#_eia_new[@]} ; _eia_cnt++ ))
         do
           _eia_tst=${_eia_new[${_eia_cnt}]}
           if _eia_tst=$(name_fixup ${_eia_tst})
           then
             _eia_name[${#_eia_name[@]}]=${_eia_tst}
       fi
     done
           fi
    done
    unique_lines _eia_name _eia_name     # Scrub duplicates.
    edit_exact chk_name _eia_name        # Scrub pending detail.
    edit_exact known_name _eia_name      # Scrub already detailed.
 if [ ${#_eia_name[@]} -gt 0 ]        # Anything new?
 then
   uc_name=( ${uc_name[@]} ${_eia_name[@]} )
   pend_func expand_input_name ${1}
   _trace_log[${#_trace_log[@]}]='#Add '${#_eia_name[@]}' unchkd name inp.#'
    fi
    edit_exact chk_address _eia_addr     # Scrub pending detail.
    edit_exact known_address _eia_addr   # Scrub already detailed.
    if [ ${#_eia_addr[@]} -gt 0 ]        # Anything new?
    then
        chk_address=( ${chk_address[@]} ${_eia_addr[@]} )
        pend_func detail_each_address ${1}
    fi
    return 0
}

# The parse-it-yourself zone reply.
# The input is the chk_name list.
# detail_each_name <indirection_limit>
detail_each_name() {
    [ ${#chk_name[@]} -gt 0 ] || return 0
    local -a _den_chk       # Names to check
    local -a _den_name      # Names found here
    local -a _den_address   # Addresses found here
    local -a _den_pair      # Pairs found here
    local -a _den_rev       # Reverse pairs found here
    local -a _den_tmp       # Line being parsed
    local -a _den_auth      # SOA contact being parsed
```

```
    local -a _den_new       # The zone reply
    local -a _den_pc        # Parent-Child gets big fast
    local -a _den_ref       # So does reference chain
    local -a _den_nr        # Name-Resource can be big
    local -a _den_na        # Name-Address
    local -a _den_ns        # Name-Service
    local -a _den_achn      # Chain of Authority
    local -i _den_cnt       # Count of names to detail
    local -i _den_lmt       # Indirection limit
    local _den_who          # Named being processed
    local _den_rec          # Record type being processed
    local _den_cont         # Contact domain
    local _den_str          # Fixed up name string
    local _den_str2         # Fixed up reverse
    local IFS=${WSP_IFS}

    # Local, unique copy of names to check
    unique_lines chk_name _den_chk
    unset chk_name[@]       # Done with globals.

    # Less any names already known
    edit_exact known_name _den_chk
    _den_cnt=${#_den_chk[@]}

    # If anything left, add to known_name.
    [ ${_den_cnt} -gt 0 ] &&
        known_name=( ${known_name[@]} ${_den_chk[@]} )

    # for the list of (previously) unknown names . . .
    for (( _den = 0 ; _den < _den_cnt ; _den++ ))
    do
        _den_who=${_den_chk[${_den}]}
        if long_fwd ${_den_who} _den_new
        then
            unique_lines _den_new _den_new
            if [ ${#_den_new[@]} -eq 0 ]
            then
                _den_pair[${#_den_pair[@]}]='0.0.0.0 '${_den_who}
            fi

            # Parse each line in the reply.
            for (( _line = 0 ; _line < ${#_den_new[@]} ; _line++ ))
            do
                IFS=${NO_WSP}$'\x09'$'\x20'
                _den_tmp=( ${_den_new[${_line}]} )
                IFS=${WSP_IFS}
              # If usable record and not a warning message . . .
              if [ ${#_den_tmp[@]} -gt 4 ] && [ 'x'${_den_tmp[0]} != 'x;;' ]
              then
                    _den_rec=${_den_tmp[3]}
                    _den_nr[${#_den_nr[@]}]=${_den_who}' '${_den_rec}
                    # Begin at RFC1033 (+++)
                    case ${_den_rec} in

#<name> [<ttl>]  [<class>] SOA <origin> <person>
                    SOA) # Start Of Authority
    if _den_str=$(name_fixup ${_den_tmp[0]})
    then
      _den_name[${#_den_name[@]}]=${_den_str}
      _den_achn[${#_den_achn[@]}]=${_den_who}' '${_den_str}' SOA'
      # SOA origin -- domain name of master zone record
      if _den_str2=$(name_fixup ${_den_tmp[4]})
```

```
    then
      _den_name[${#_den_name[@]}]=${_den_str2}
      _den_achn[${#_den_achn[@]}]=${_den_who}' '${_den_str2}' SOA.O'
    fi
    # Responsible party e-mail address (possibly bogus).
    # Possibility of first.last@domain.name ignored.
    set -f
    if _den_str2=$(name_fixup ${_den_tmp[5]})
    then
      IFS=${ADR_IFS}
      _den_auth=( ${_den_str2} )
      IFS=${WSP_IFS}
      if [ ${#_den_auth[@]} -gt 2 ]
      then
        _den_cont=${_den_auth[1]}
        for (( _auth = 2 ; _auth < ${#_den_auth[@]} ; _auth++ ))
        do
          _den_cont=${_den_cont}'.'${_den_auth[${_auth}]}
        done
        _den_name[${#_den_name[@]}]=${_den_cont}'.'
        _den_achn[${#_den_achn[@]}]=${_den_who}' '${_den_cont}'. SOA.C'
                                fi
      fi
      set +f
                        fi
                    ;;


    A) # IP(v4) Address Record
    if _den_str=$(name_fixup ${_den_tmp[0]})
    then
      _den_name[${#_den_name[@]}]=${_den_str}
      _den_pair[${#_den_pair[@]}]=${_den_tmp[4]}' '${_den_str}
      _den_na[${#_den_na[@]}]=${_den_str}' '${_den_tmp[4]}
      _den_ref[${#_den_ref[@]}]=${_den_who}' '${_den_str}' A'
    else
      _den_pair[${#_den_pair[@]}]=${_den_tmp[4]}' unknown.domain'
      _den_na[${#_den_na[@]}]='unknown.domain '${_den_tmp[4]}
      _den_ref[${#_den_ref[@]}]=${_den_who}' unknown.domain A'
    fi
    _den_address[${#_den_address[@]}]=${_den_tmp[4]}
    _den_pc[${#_den_pc[@]}]=${_den_who}' '${_den_tmp[4]}
           ;;

           NS) # Name Server Record
           # Domain name being serviced (may be other than current)
             if _den_str=$(name_fixup ${_den_tmp[0]})
               then
                 _den_name[${#_den_name[@]}]=${_den_str}
                 _den_ref[${#_den_ref[@]}]=${_den_who}' '${_den_str}' NS'

           # Domain name of service provider
           if _den_str2=$(name_fixup ${_den_tmp[4]})
           then
             _den_name[${#_den_name[@]}]=${_den_str2}
             _den_ref[${#_den_ref[@]}]=${_den_who}' '${_den_str2}' NSH'
             _den_ns[${#_den_ns[@]}]=${_den_str2}' NS'
             _den_pc[${#_den_pc[@]}]=${_den_str}' '${_den_str2}
            fi
             fi
                    ;;
```

```
MX) # Mail Server Record
    # Domain name being serviced (wildcards not handled here)
if _den_str=$(name_fixup ${_den_tmp[0]})
then
  _den_name[${#_den_name[@]}]=${_den_str}
  _den_ref[${#_den_ref[@]}]=${_den_who}' '${_den_str}' MX'
fi
# Domain name of service provider
if _den_str=$(name_fixup ${_den_tmp[5]})
then
  _den_name[${#_den_name[@]}]=${_den_str}
  _den_ref[${#_den_ref[@]}]=${_den_who}' '${_den_str}' MXH'
  _den_ns[${#_den_ns[@]}]=${_den_str}' MX'
  _den_pc[${#_den_pc[@]}]=${_den_who}' '${_den_str}
fi
       ;;

PTR) # Reverse address record
     # Special name
if _den_str=$(name_fixup ${_den_tmp[0]})
then
  _den_ref[${#_den_ref[@]}]=${_den_who}' '${_den_str}' PTR'
  # Host name (not a CNAME)
  if _den_str2=$(name_fixup ${_den_tmp[4]})
  then
    _den_rev[${#_den_rev[@]}]=${_den_str}' '${_den_str2}
    _den_ref[${#_den_ref[@]}]=${_den_who}' '${_den_str2}' PTRH'
    _den_pc[${#_den_pc[@]}]=${_den_who}' '${_den_str}
  fi
fi
       ;;

AAAA) # IP(v6) Address Record
if _den_str=$(name_fixup ${_den_tmp[0]})
then
  _den_name[${#_den_name[@]}]=${_den_str}
  _den_pair[${#_den_pair[@]}]=${_den_tmp[4]}' '${_den_str}
  _den_na[${#_den_na[@]}]=${_den_str}' '${_den_tmp[4]}
  _den_ref[${#_den_ref[@]}]=${_den_who}' '${_den_str}' AAAA'
  else
    _den_pair[${#_den_pair[@]}]=${_den_tmp[4]}' unknown.domain'
    _den_na[${#_den_na[@]}]='unknown.domain '${_den_tmp[4]}
    _den_ref[${#_den_ref[@]}]=${_den_who}' unknown.domain'
  fi
  # No processing for IPv6 addresses
  _den_pc[${#_den_pc[@]}]=${_den_who}' '${_den_tmp[4]}
       ;;

CNAME) # Alias name record
       # Nickname
if _den_str=$(name_fixup ${_den_tmp[0]})
then
  _den_name[${#_den_name[@]}]=${_den_str}
  _den_ref[${#_den_ref[@]}]=${_den_who}' '${_den_str}' CNAME'
  _den_pc[${#_den_pc[@]}]=${_den_who}' '${_den_str}
fi
       # Hostname
if _den_str=$(name_fixup ${_den_tmp[4]})
then
  _den_name[${#_den_name[@]}]=${_den_str}
  _den_ref[${#_den_ref[@]}]=${_den_who}' '${_den_str}' CHOST'
  _den_pc[${#_den_pc[@]}]=${_den_who}' '${_den_str}
```

```
            fi
                   ;;
#           TXT)
#           ;;
                   esac
               fi
           done
       else # Lookup error == 'A' record 'unknown address'
           _den_pair[${#_den_pair[@]}]='0.0.0.0 '${_den_who}
       fi
   done

   # Control dot array growth.
   unique_lines _den_achn _den_achn      # Works best, all the same.
   edit_exact auth_chain _den_achn       # Works best, unique items.
   if [ ${#_den_achn[@]} -gt 0 ]
   then
       IFS=${NO_WSP}
       auth_chain=( ${auth_chain[@]} ${_den_achn[@]} )
       IFS=${WSP_IFS}
   fi

   unique_lines _den_ref _den_ref      # Works best, all the same.
   edit_exact ref_chain _den_ref       # Works best, unique items.
   if [ ${#_den_ref[@]} -gt 0 ]
   then
       IFS=${NO_WSP}
       ref_chain=( ${ref_chain[@]} ${_den_ref[@]} )
       IFS=${WSP_IFS}
   fi

   unique_lines _den_na _den_na
   edit_exact name_address _den_na
   if [ ${#_den_na[@]} -gt 0 ]
   then
       IFS=${NO_WSP}
       name_address=( ${name_address[@]} ${_den_na[@]} )
       IFS=${WSP_IFS}
   fi

   unique_lines _den_ns _den_ns
   edit_exact name_srvc _den_ns
   if [ ${#_den_ns[@]} -gt 0 ]
   then
       IFS=${NO_WSP}
       name_srvc=( ${name_srvc[@]} ${_den_ns[@]} )
       IFS=${WSP_IFS}
   fi

   unique_lines _den_nr _den_nr
   edit_exact name_resource _den_nr
   if [ ${#_den_nr[@]} -gt 0 ]
   then
       IFS=${NO_WSP}
       name_resource=( ${name_resource[@]} ${_den_nr[@]} )
       IFS=${WSP_IFS}
   fi

   unique_lines _den_pc _den_pc
   edit_exact parent_child _den_pc
   if [ ${#_den_pc[@]} -gt 0 ]
   then
```

```
        IFS=${NO_WSP}
        parent_child=( ${parent_child[@]} ${_den_pc[@]} )
        IFS=${WSP_IFS}
    fi

    # Update list known_pair (Address and Name).
    unique_lines _den_pair _den_pair
    edit_exact known_pair _den_pair
    if [ ${#_den_pair[@]} -gt 0 ]  # Anything new?
    then
        IFS=${NO_WSP}
        known_pair=( ${known_pair[@]} ${_den_pair[@]} )
        IFS=${WSP_IFS}
    fi

    # Update list of reverse pairs.
    unique_lines _den_rev _den_rev
    edit_exact reverse_pair _den_rev
    if [ ${#_den_rev[@]} -gt 0 ]   # Anything new?
    then
        IFS=${NO_WSP}
        reverse_pair=( ${reverse_pair[@]} ${_den_rev[@]} )
        IFS=${WSP_IFS}
    fi

    # Check indirection limit -- give up if reached.
    if ! _den_lmt=$(limit_chk ${1})
    then
        return 0
    fi

# Execution engine is LIFO. Order of pend operations is important.
# Did we define any new addresses?
unique_lines _den_address _den_address    # Scrub duplicates.
edit_exact known_address _den_address     # Scrub already processed.
edit_exact un_address _den_address        # Scrub already waiting.
if [ ${#_den_address[@]} -gt 0 ]          # Anything new?
then
  uc_address=( ${uc_address[@]} ${_den_address[@]} )
  pend_func expand_input_address ${_den_lmt}
  _trace_log[${#_trace_log[@]}]='# Add '${#_den_address[@]}' unchkd addr. #'
    fi

# Did we find any new names?
unique_lines _den_name _den_name          # Scrub duplicates.
edit_exact known_name _den_name           # Scrub already processed.
edit_exact uc_name _den_name              # Scrub already waiting.
if [ ${#_den_name[@]} -gt 0 ]             # Anything new?
then
  uc_name=( ${uc_name[@]} ${_den_name[@]} )
  pend_func expand_input_name ${_den_lmt}
  _trace_log[${#_trace_log[@]}]='#Added '${#_den_name[@]}' unchkd name#'
    fi
    return 0
}

# The parse-it-yourself delegation reply
# Input is the chk_address list.
# detail_each_address <indirection_limit>
detail_each_address() {
    [ ${#chk_address[@]} -gt 0 ] || return 0
    unique_lines chk_address chk_address
```

```
    edit_exact known_address chk_address
    if [ ${#chk_address[@]} -gt 0 ]
    then
        known_address=( ${known_address[@]} ${chk_address[@]} )
        unset chk_address[@]
    fi
    return 0
}

# # # Application specific output functions # # #

# Pretty print the known pairs.
report_pairs() {
    echo
    echo 'Known network pairs.'
    col_print known_pair 2 5 30

    if [ ${#auth_chain[@]} -gt 0 ]
    then
        echo
        echo 'Known chain of authority.'
        col_print auth_chain 2 5 30 55
    fi

    if [ ${#reverse_pair[@]} -gt 0 ]
    then
        echo
        echo 'Known reverse pairs.'
        col_print reverse_pair 2 5 55
    fi
    return 0
}

# Check an address against the list of blacklist servers.
# A good place to capture for GraphViz: address->status(server(reports))
# check_lists <ip_address>
check_lists() {
    [ $# -eq 1 ] || return 1
    local -a _cl_fwd_addr
    local -a _cl_rev_addr
    local -a _cl_reply
    local -i _cl_rc
    local -i _ls_cnt
    local _cl_dns_addr
    local _cl_lkup

    split_ip ${1} _cl_fwd_addr _cl_rev_addr
    _cl_dns_addr=$(dot_array _cl_rev_addr)'.'
    _ls_cnt=${#list_server[@]}
    echo '    Checking address '${1}
    for (( _cl = 0 ; _cl < _ls_cnt ; _cl++ ))
    do
      _cl_lkup=${_cl_dns_addr}${list_server[${_cl}]}
      if short_text ${_cl_lkup} _cl_reply
      then
        if [ ${#_cl_reply[@]} -gt 0 ]
        then
          echo '        Records from '${list_server[${_cl}]}
          address_hits[${#address_hits[@]}]=${1}' '${list_server[${_cl}]}
          _hs_RC=2
          for (( _clr = 0 ; _clr < ${#_cl_reply[@]} ; _clr++ ))
          do
```

```
            echo '            '${_cl_reply[${_clr}]}
          done
        fi
      fi
    done
    return 0
}

# # # The usual application glue # # #

# Who did it?
credits() {
   echo
   echo 'Advanced Bash Scripting Guide: is_spammer.bash, v2, 2004-msz'
}

# How to use it?
# (See also, "Quickstart" at end of script.)
usage() {
    cat <<-'_usage_statement_'
    The script is_spammer.bash requires either one or two arguments.

    arg 1) May be one of:
        a) A domain name
        b) An IPv4 address
        c) The name of a file with any mix of names
           and addresses, one per line.

    arg 2) May be one of:
        a) A Blacklist server domain name
        b) The name of a file with Blacklist server
           domain names, one per line.
        c) If not present, a default list of (free)
           Blacklist servers is used.
        d) If a filename of an empty, readable, file
           is given,
           Blacklist server lookup is disabled.

    All script output is written to stdout.

    Return codes: 0 -> All OK, 1 -> Script failure,
                  2 -> Something is Blacklisted.

    Requires the external program 'dig' from the 'bind-9'
    set of DNS programs.  See: http://www.isc.org

    The domain name lookup depth limit defaults to 2 levels.
    Set the environment variable SPAMMER_LIMIT to change.
    SPAMMER_LIMIT=0 means 'unlimited'

    Limit may also be set on the command-line.
    If arg#1 is an integer, the limit is set to that value
    and then the above argument rules are applied.

    Setting the environment variable 'SPAMMER_DATA' to a filename
    will cause the script to write a GraphViz graphic file.

    For the development version;
    Setting the environment variable 'SPAMMER_TRACE' to a filename
    will cause the execution engine to log a function call trace.

_usage_statement_
```

```
}

# The default list of Blacklist servers:
# Many choices, see: http://www.spews.org/lists.html

declare -a default_servers
# See: http://www.spamhaus.org (Conservative, well maintained)
default_servers[0]='sbl-xbl.spamhaus.org'
# See: http://ordb.org (Open mail relays)
default_servers[1]='relays.ordb.org'
# See: http://www.spamcop.net/ (You can report spammers here)
default_servers[2]='bl.spamcop.net'
# See: http://www.spews.org (An 'early detect' system)
default_servers[3]='l2.spews.dnsbl.sorbs.net'
# See: http://www.dnsbl.us.sorbs.net/using.shtml
default_servers[4]='dnsbl.sorbs.net'
# See: http://dsbl.org/usage (Various mail relay lists)
default_servers[5]='list.dsbl.org'
default_servers[6]='multihop.dsbl.org'
default_servers[7]='unconfirmed.dsbl.org'

# User input argument #1
setup_input() {
    if [ -e ${1} ] && [ -r ${1} ]  # Name of readable file
    then
        file_to_array ${1} uc_name
        echo 'Using filename >'${1}'< as input.'
    else
        if is_address ${1}          # IP address?
        then
            uc_address=( ${1} )
            echo 'Starting with address >'${1}'<'
        else                       # Must be a name.
            uc_name=( ${1} )
            echo 'Starting with domain name >'${1}'<'
        fi
    fi
    return 0
}

# User input argument #2
setup_servers() {
    if [ -e ${1} ] && [ -r ${1} ]  # Name of a readable file
    then
        file_to_array ${1} list_server
        echo 'Using filename >'${1}'< as blacklist server list.'
    else
        list_server=( ${1} )
        echo 'Using blacklist server >'${1}'<'
    fi
    return 0
}

# User environment variable SPAMMER_TRACE
live_log_die() {
    if [ ${SPAMMER_TRACE:=} ]    # Wants trace log?
    then
        if [ ! -e ${SPAMMER_TRACE} ]
        then
            if ! touch ${SPAMMER_TRACE} 2>/dev/null
            then
                pend_func echo $(printf '%q\n' \
```

```
                'Unable to create log file >'${SPAMMER_TRACE}'<')
                pend_release
                exit 1
            fi
            _log_file=${SPAMMER_TRACE}
            _pend_hook_=trace_logger
            _log_dump=dump_log
        else
            if [ ! -w ${SPAMMER_TRACE} ]
            then
                pend_func echo $(printf '%q\n' \
                'Unable to write log file >'${SPAMMER_TRACE}'<')
                pend_release
                exit 1
            fi
            _log_file=${SPAMMER_TRACE}
            echo '' > ${_log_file}
            _pend_hook_=trace_logger
            _log_dump=dump_log
        fi
    fi
    return 0
}

# User environment variable SPAMMER_DATA
data_capture() {
    if [ ${SPAMMER_DATA:=} ]    # Wants a data dump?
    then
        if [ ! -e ${SPAMMER_DATA} ]
        then
            if ! touch ${SPAMMER_DATA} 2>/dev/null
            then
                pend_func echo $(printf '%q]n' \
                'Unable to create data output file >'${SPAMMER_DATA}'<')
                pend_release
                exit 1
            fi
            _dot_file=${SPAMMER_DATA}
            _dot_dump=dump_dot
        else
            if [ ! -w ${SPAMMER_DATA} ]
            then
                pend_func echo $(printf '%q\n' \
                'Unable to write data output file >'${SPAMMER_DATA}'<')
                pend_release
                exit 1
            fi
            _dot_file=${SPAMMER_DATA}
            _dot_dump=dump_dot
        fi
    fi
    return 0
}

# Grope user specified arguments.
do_user_args() {
    if [ $# -gt 0 ] && is_number $1
    then
        indirect=$1
        shift
    fi
```

```
    case $# in                      # Did user treat us well?
        1)
            if ! setup_input $1    # Needs error checking.
            then
                pend_release
                $_log_dump
                exit 1
            fi
            list_server=( ${default_servers[@]} )
            _list_cnt=${#list_server[@]}
            echo 'Using default blacklist server list.'
            echo 'Search depth limit: '${indirect}
            ;;
        2)
            if ! setup_input $1    # Needs error checking.
            then
                pend_release
                $_log_dump
                exit 1
            fi
            if ! setup_servers $2  # Needs error checking.
            then
                pend_release
                $_log_dump
                exit 1
            fi
            echo 'Search depth limit: '${indirect}
            ;;
        *)
            pend_func usage
            pend_release
            $_log_dump
            exit 1
            ;;
    esac
    return 0
}

# A general purpose debug tool.
# list_array <array_name>
list_array() {
    [ $# -eq 1 ] || return 1  # One argument required.

    local -a _la_lines
    set -f
    local IFS=${NO_WSP}
    eval _la_lines=\(\ \$\{$1\[@\]\}\ \)
    echo
    echo "Element count "${#_la_lines[@]}" array "${1}
    local _ln_cnt=${#_la_lines[@]}

    for (( _i = 0; _i < ${_ln_cnt}; _i++ ))
    do
        echo 'Element '$_i' >'${_la_lines[$_i]}'<'
    done
    set +f
    return 0
}

# # # 'Hunt the Spammer' program code # # #
pend_init                               # Ready stack engine.
pend_func credits                       # Last thing to print.
```

```
# # # Deal with user # # #
live_log_die                            # Setup debug trace log.
data_capture                            # Setup data capture file.
echo
do_user_args $@

# # # Haven't exited yet - There is some hope # # #
# Discovery group - Execution engine is LIFO - pend
# in reverse order of execution.
_hs_RC=0                                # Hunt the Spammer return code
pend_mark
    pend_func report_pairs              # Report name-address pairs.

    # The two detail_* are mutually recursive functions.
    # They also pend expand_* functions as required.
    # These two (the last of ???) exit the recursion.
    pend_func detail_each_address       # Get all resources of addresses.
    pend_func detail_each_name          # Get all resources of names.

    #  The two expand_* are mutually recursive functions,
    #+ which pend additional detail_* functions as required.
    pend_func expand_input_address 1    # Expand input names by address.
    pend_func expand_input_name 1       # #xpand input addresses by name.

    # Start with a unique set of names and addresses.
    pend_func unique_lines uc_address uc_address
    pend_func unique_lines uc_name uc_name

    # Separate mixed input of names and addresses.
    pend_func split_input
pend_release

# # # Pairs reported -- Unique list of IP addresses found
echo
_ip_cnt=${#known_address[@]}
if [ ${#list_server[@]} -eq 0 ]
then
    echo 'Blacklist server list empty, none checked.'
else
    if [ ${_ip_cnt} -eq 0 ]
    then
        echo 'Known address list empty, none checked.'
    else
        _ip_cnt=${_ip_cnt}-1   # Start at top.
        echo 'Checking Blacklist servers.'
        for (( _ip = _ip_cnt ; _ip >= 0 ; _ip-- ))
        do
          pend_func check_lists $( printf '%q\n' ${known_address[$_ip]} )
        done
    fi
fi
pend_release
$_dot_dump                   # Graphics file dump
$_log_dump                   # Execution trace
echo


##############################
# Example output from script #
##############################
:<<-'_is_spammer_outputs_'
```

```
./is_spammer.bash 0 web4.alojamentos7.com

Starting with domain name >web4.alojamentos7.com<
Using default blacklist server list.
Search depth limit: 0
.:....::::...:::...:::.......::..::...:::.......::
Known network pairs.
    66.98.208.97             web4.alojamentos7.com.
    66.98.208.97             ns1.alojamentos7.com.
    69.56.202.147            ns2.alojamentos.ws.
    66.98.208.97             alojamentos7.com.
    66.98.208.97             web.alojamentos7.com.
    69.56.202.146            ns1.alojamentos.ws.
    69.56.202.146            alojamentos.ws.
    66.235.180.113           ns1.alojamentos.org.
    66.235.181.192           ns2.alojamentos.org.
    66.235.180.113           alojamentos.org.
    66.235.180.113           web6.alojamentos.org.
    216.234.234.30           ns1.theplanet.com.
    12.96.160.115            ns2.theplanet.com.
    216.185.111.52           mail1.theplanet.com.
    69.56.141.4              spooling.theplanet.com.
    216.185.111.40           theplanet.com.
    216.185.111.40           www.theplanet.com.
    216.185.111.52           mail.theplanet.com.

Checking Blacklist servers.
  Checking address 66.98.208.97
      Records from dnsbl.sorbs.net
  "Spam Received See: http://www.dnsbl.sorbs.net/lookup.shtml?66.98.208.97"
    Checking address 69.56.202.147
    Checking address 69.56.202.146
    Checking address 66.235.180.113
    Checking address 66.235.181.192
    Checking address 216.185.111.40
    Checking address 216.234.234.30
    Checking address 12.96.160.115
    Checking address 216.185.111.52
    Checking address 69.56.141.4

Advanced Bash Scripting Guide: is_spammer.bash, v2, 2004-msz

_is_spammer_outputs_

exit ${_hs_RC}

####################################################
#  The script ignores everything from here on down #
#+ because of the 'exit' command, just above.      #
####################################################



Quickstart
==========

 Prerequisites

  Bash version 2.05b or 3.00 (bash --version)
  A version of Bash which supports arrays. Array
  support is included by default Bash configurations.
```

```
  'dig,' version 9.x.x (dig $HOSTNAME, see first line of output)
  A version of dig which supports the +short options.
  See: dig_wrappers.bash for details.


 Optional Prerequisites

  'named,' a local DNS caching program. Any flavor will do.
  Do twice: dig $HOSTNAME
  Check near bottom of output for: SERVER: 127.0.0.1#53
  That means you have one running.


 Optional Graphics Support

  'date,' a standard *nix thing. (date -R)

  dot Program to convert graphic description file to a
  diagram. (dot -V)
  A part of the Graph-Viz set of programs.
  See: [http://www.research.att.com/sw/tools/graphviz||GraphViz]

  'dotty,' a visual editor for graphic description files.
  Also a part of the Graph-Viz set of programs.




 Quick Start

In the same directory as the is_spammer.bash script;
Do: ./is_spammer.bash

 Usage Details

1. Blacklist server choices.

  (a) To use default, built-in list: Do nothing.

  (b) To use your own list:

    i. Create a file with a single Blacklist server
       domain name per line.

    ii. Provide that filename as the last argument to
        the script.

  (c) To use a single Blacklist server: Last argument
      to the script.

  (d) To disable Blacklist lookups:

    i. Create an empty file (touch spammer.nul)
       Your choice of filename.

    ii. Provide the filename of that empty file as the
        last argument to the script.

2. Search depth limit.

  (a) To use the default value of 2: Do nothing.
```

```
  (b) To set a different limit:
      A limit of 0 means: no limit.

    i. export SPAMMER_LIMIT=1
       or whatever limit you want.

    ii. OR provide the desired limit as the first
       argument to the script.

3. Optional execution trace log.

  (a) To use the default setting of no log output: Do nothing.

  (b) To write an execution trace log:
      export SPAMMER_TRACE=spammer.log
      or whatever filename you want.

4. Optional graphic description file.

  (a) To use the default setting of no graphic file: Do nothing.

  (b) To write a Graph-Viz graphic description file:
      export SPAMMER_DATA=spammer.dot
      or whatever filename you want.

5. Where to start the search.

  (a) Starting with a single domain name:

    i. Without a command-line search limit: First
       argument to script.

    ii. With a command-line search limit: Second
        argument to script.

  (b) Starting with a single IP address:

    i. Without a command-line search limit: First
       argument to script.

    ii. With a command-line search limit: Second
        argument to script.

  (c) Starting with (mixed) multiple name(s) and/or address(es):
      Create a file with one name or address per line.
      Your choice of filename.

    i. Without a command-line search limit: Filename as
       first argument to script.

    ii. With a command-line search limit: Filename as
        second argument to script.

6. What to do with the display output.

  (a) To view display output on screen: Do nothing.

  (b) To save display output to a file: Redirect stdout to a filename.

  (c) To discard display output: Redirect stdout to /dev/null.
```

```
7. Temporary end of decision making.
   press RETURN
   wait (optionally, watch the dots and colons).

8. Optionally check the return code.

  (a) Return code 0: All OK

  (b) Return code 1: Script setup failure

  (c) Return code 2: Something was blacklisted.

9. Where is my graph (diagram)?

The script does not directly produce a graph (diagram).
It only produces a graphic description file. You can
process the graphic descriptor file that was output
with the 'dot' program.

Until you edit that descriptor file, to describe the
relationships you want shown, all that you will get is
a bunch of labeled name and address nodes.

All of the script's discovered relationships are within
a comment block in the graphic descriptor file, each
with a descriptive heading.

The editing required to draw a line between a pair of
nodes from the information in the descriptor file may
be done with a text editor.

Given these lines somewhere in the descriptor file:

# Known domain name nodes

N0000 [label="guardproof.info."] ;

N0002 [label="third.guardproof.info."] ;



# Known address nodes

A0000 [label="61.141.32.197"] ;



/*

# Known name->address edges

NA0000 third.guardproof.info. 61.141.32.197



# Known parent->child edges

PC0000 guardproof.info. third.guardproof.info.

 */

Turn that into the following lines by substituting node
```

```
identifiers into the relationships:

# Known domain name nodes

N0000 [label="guardproof.info."] ;

N0002 [label="third.guardproof.info."] ;



# Known address nodes

A0000 [label="61.141.32.197"] ;



# PC0000 guardproof.info. third.guardproof.info.

N0000->N0002 ;



# NA0000 third.guardproof.info. 61.141.32.197

N0002->A0000 ;



/*

# Known name->address edges

NA0000 third.guardproof.info. 61.141.32.197



# Known parent->child edges

PC0000 guardproof.info. third.guardproof.info.

 */

Process that with the 'dot' program, and you have your
first network diagram.

In addition to the conventional graphic edges, the
descriptor file includes similar format pair-data that
describes services, zone records (sub-graphs?),
blacklisted addresses, and other things which might be
interesting to include in your graph. This additional
information could be displayed as different node
shapes, colors, line sizes, etc.

The descriptor file can also be read and edited by a
Bash script (of course). You should be able to find
most of the functions required within the
"is_spammer.bash" script.

# End Quickstart.
```

```
Additional Note
========== ====

Michael Zick points out that there is a "makeviz.bash" interactive
Web site at rediris.es. Can't give the full URL, since this is not
a publically accessible site.
```

Another anti-spam script.

Example A-29. Spammer Hunt

```
#!/bin/bash
# whx.sh: "whois" spammer lookup
# Author: Walter Dnes
# Slight revisions (first section) by ABS Guide author.
# Used in ABS Guide with permission.

# Needs version 3.x or greater of Bash to run (because of =~ operator).
# Commented by script author and ABS Guide author.



E_BADARGS=85        # Missing command-line arg.
E_NOHOST=86         # Host not found.
E_TIMEOUT=87        # Host lookup timed out.
E_UNDEF=88          # Some other (undefined) error.

HOSTWAIT=10         # Specify up to 10 seconds for host query reply.
                    # The actual wait may be a bit longer.
OUTFILE=whois.txt   # Output file.
PORT=4321


if [ -z "$1" ]      # Check for (required) command-line arg.
then
  echo "Usage: $0 domain name or IP address"
  exit $E_BADARGS
fi


if [[ "$1" =~ [a-zA-Z][a-zA-Z]$ ]]  #  Ends in two alpha chars?
then                                  #  It's a domain name &&
                                      #+ must do host lookup.
  IPADDR=$(host -W $HOSTWAIT $1 | awk '{print $4}')
                                      #  Doing host lookup
                                      #+ to get IP address.
                                      #  Extract final field.
else
  IPADDR="$1"                         #  Command-line arg was IP address.
fi

echo; echo "IP Address is: "$IPADDR""; echo

if [ -e "$OUTFILE" ]
then
  rm -f "$OUTFILE"
  echo "Stale output file \"$OUTFILE\" removed."; echo
fi
```

```
#  Sanity checks.
#  (This section needs more work.)
#  ===============================
if [ -z "$IPADDR" ]
# No response.
then
  echo "Host not found!"
  exit $E_NOHOST    # Bail out.
fi

if [[ "$IPADDR" =~ ^[;;] ]]
#  ;; Connection timed out; no servers could be reached.
then
  echo "Host lookup timed out!"
  exit $E_TIMEOUT   # Bail out.
fi

if [[ "$IPADDR" =~ [(NXDOMAIN)]$ ]]
#  Host xxxxxxxxx.xxx not found: 3(NXDOMAIN)
then
  echo "Host not found!"
  exit $E_NOHOST    # Bail out.
fi

if [[ "$IPADDR" =~ [(SERVFAIL)]$ ]]
#  Host xxxxxxxxx.xxx not found: 2(SERVFAIL)
then
  echo "Host not found!"
  exit $E_NOHOST    # Bail out.
fi




# ======================== Main body of script ========================

AFRINICquery() {
#  Define the function that queries AFRINIC. Echo a notification to the
#+ screen, and then run the actual query, redirecting output to $OUTFILE.

  echo "Searching for $IPADDR in whois.afrinic.net"
  whois -h whois.afrinic.net "$IPADDR" > $OUTFILE

#  Check for presence of reference to an rwhois.
#  Warn about non-functional rwhois.infosat.net server
#+ and attempt rwhois query.
  if grep -e "^remarks: .*rwhois\.[^ ]\+" "$OUTFILE"
  then
    echo " " >> $OUTFILE
    echo "***" >> $OUTFILE
    echo "***" >> $OUTFILE
    echo "Warning: rwhois.infosat.net was not working \
      as of 2005/02/02" >> $OUTFILE
    echo "         when this script was written." >> $OUTFILE
    echo "***" >> $OUTFILE
    echo "***" >> $OUTFILE
    echo " " >> $OUTFILE
    RWHOIS=`grep "^remarks: .*rwhois\.[^ ]\+" "$OUTFILE" | tail -n 1 |\
    sed "s/\(^.*\)\(rwhois\..*\)\(:4.*\)/\2/"`
    whois -h ${RWHOIS}:${PORT} "$IPADDR" >> $OUTFILE
  fi
```

```
}

APNICquery() {
  echo "Searching for $IPADDR in whois.apnic.net"
  whois -h whois.apnic.net "$IPADDR" > $OUTFILE

#  Just  about  every  country has its own internet registrar.
#  I don't normally bother consulting them, because the regional registry
#+ usually supplies sufficient information.
#  There are a few exceptions, where the regional registry simply
#+ refers to the national registry for direct data.
#  These are Japan and South Korea in APNIC, and Brasil in LACNIC.
#  The following if statement checks $OUTFILE (whois.txt) for the presence
#+ of "KR" (South Korea) or "JP" (Japan) in the country field.
#  If either is found, the query is re-run against the appropriate
#+ national registry.

  if grep -E "^country:[ ]+KR$" "$OUTFILE"
  then
    echo "Searching for $IPADDR in whois.krnic.net"
    whois -h whois.krnic.net "$IPADDR" >> $OUTFILE
  elif grep -E "^country:[ ]+JP$" "$OUTFILE"
  then
    echo "Searching for $IPADDR in whois.nic.ad.jp"
    whois -h whois.nic.ad.jp "$IPADDR"/e >> $OUTFILE
  fi
}

ARINquery() {
  echo "Searching for $IPADDR in whois.arin.net"
  whois -h whois.arin.net "$IPADDR" > $OUTFILE

#  Several large internet providers listed by ARIN have their own
#+ internal whois service, referred to as "rwhois".
#  A large block of IP addresses is listed with the provider
#+ under the ARIN registry.
#  To get the IP addresses of 2nd-level ISPs or other large customers,
#+ one has to refer to the rwhois server on port 4321.
#  I originally started with a bunch of "if" statements checking for
#+ the larger providers.
#  This approach is unwieldy, and there's always another rwhois server
#+ that I didn't know about.
#  A more elegant approach is to check $OUTFILE for a reference
#+ to a whois server, parse that server name out of the comment section,
#+ and re-run the query against the appropriate rwhois server.
#  The parsing looks a bit ugly, with a long continued line inside
#+ backticks.
#  But it only has to be done once, and will work as new servers are added.
#@   ABS Guide author comment: it isn't all that ugly, and is, in fact,
#@+  an instructive use of Regular Expressions.

  if grep -E "^Comment: .*rwhois.[^ ]+" "$OUTFILE"
  then
    RWHOIS=`grep -e "^Comment:.*rwhois\.[^ ]\+" "$OUTFILE" | tail -n 1 |\
    sed "s/^\(.*\)\(rwhois\.[^ ]\+\)\(.*$\)/\2/"`
    echo "Searching for $IPADDR in ${RWHOIS}"
    whois -h ${RWHOIS}:${PORT} "$IPADDR" >> $OUTFILE
  fi
}

LACNICquery() {
  echo "Searching for $IPADDR in whois.lacnic.net"
```

```
  whois -h whois.lacnic.net "$IPADDR" > $OUTFILE

#  The  following if statement checks $OUTFILE (whois.txt) for
#+ the presence of "BR" (Brasil) in the country field.
#  If it is found, the query is re-run against whois.registro.br.

  if grep -E "^country:[ ]+BR$" "$OUTFILE"
  then
    echo "Searching for $IPADDR in whois.registro.br"
    whois -h whois.registro.br "$IPADDR" >> $OUTFILE
  fi
}

RIPEquery() {
  echo "Searching for $IPADDR in whois.ripe.net"
  whois -h whois.ripe.net "$IPADDR" > $OUTFILE
}

#  Initialize a few variables.
#  * slash8 is the most significant octet
#  * slash16 consists of the two most significant octets
#  * octet2 is the second most significant octet




slash8=`echo $IPADDR | cut -d. -f 1`
  if [ -z "$slash8" ]  # Yet another sanity check.
  then
    echo "Undefined error!"
    exit $E_UNDEF
  fi
slash16=`echo $IPADDR | cut -d. -f 1-2`
#                             ^ Period specified as 'cut" delimiter.
  if [ -z "$slash16" ]
  then
    echo "Undefined error!"
    exit $E_UNDEF
  fi
octet2=`echo $slash16 | cut -d. -f 2`
  if [ -z "$octet2" ]
  then
    echo "Undefined error!"
    exit $E_UNDEF
  fi


#  Check for various odds and ends of reserved space.
#  There is no point in querying for those addresses.

if [ $slash8 == 0 ]; then
  echo $IPADDR is '"This Network"' space\; Not querying
elif [ $slash8 == 10 ]; then
  echo $IPADDR is RFC1918 space\; Not querying
elif [ $slash8 == 14 ]; then
  echo $IPADDR is '"Public Data Network"' space\; Not querying
elif [ $slash8 == 127 ]; then
  echo $IPADDR is loopback space\; Not querying
elif [ $slash16 == 169.254 ]; then
  echo $IPADDR is link-local space\; Not querying
elif [ $slash8 == 172 ] && [ $octet2 -ge 16 ] && [ $octet2 -le 31 ];then
  echo $IPADDR is RFC1918 space\; Not querying
```

```
elif [ $slash16 == 192.168 ]; then
  echo $IPADDR is RFC1918 space\; Not querying
elif [ $slash8 -ge 224 ]; then
  echo $IPADDR is either Multicast or reserved space\; Not querying
elif [ $slash8 -ge 200 ] && [ $slash8 -le 201 ]; then LACNICquery "$IPADDR"
elif [ $slash8 -ge 202 ] && [ $slash8 -le 203 ]; then APNICquery "$IPADDR"
elif [ $slash8 -ge 210 ] && [ $slash8 -le 211 ]; then APNICquery "$IPADDR"
elif [ $slash8 -ge 218 ] && [ $slash8 -le 223 ]; then APNICquery "$IPADDR"

#  If we got this far without making a decision, query ARIN.
#  If a reference is found in $OUTFILE to APNIC, AFRINIC, LACNIC, or RIPE,
#+ query the appropriate whois server.

else
  ARINquery "$IPADDR"
  if grep "whois.afrinic.net" "$OUTFILE"; then
    AFRINICquery "$IPADDR"
  elif grep -E "^OrgID:[ ]+RIPE$" "$OUTFILE"; then
    RIPEquery "$IPADDR"
  elif grep -E "^OrgID:[ ]+APNIC$" "$OUTFILE"; then
    APNICquery "$IPADDR"
  elif grep -E "^OrgID:[ ]+LACNIC$" "$OUTFILE"; then
    LACNICquery "$IPADDR"
  fi
fi

#@  ---------------------------------------------------------------
#   Try also:
#   wget http://logi.cc/nw/whois.php3?ACTION=doQuery&DOMAIN=$IPADDR
#@  ---------------------------------------------------------------

#  We've  now  finished  the querying.
#  Echo a copy of the final result to the screen.

cat $OUTFILE
# Or "less $OUTFILE" . . .


exit 0

#@  ABS Guide author comments:
#@  Nothing fancy here, but still a very useful tool for hunting spammers.
#@  Sure, the script can be cleaned up some, and it's still a bit buggy,
#@+ (exercise for reader), but all the same, it's a nice piece of coding
#@+ by Walter Dnes.
#@  Thank you!
```

"Little Monster's" front end to wget.

Example A-30. Making *wget* easier to use

```
#!/bin/bash
# wgetter2.bash

# Author: Little Monster [monster@monstruum.co.uk]
# ==> Used in ABS Guide with permission of script author.
# ==> This script still needs debugging and fixups (exercise for reader).
# ==> It could also use some additional editing in the comments.


#  This is wgetter2 --
```

```
#+ a Bash script to make wget a bit more friendly, and save typing.

#  Carefully crafted by Little Monster.
#  More or less complete on 02/02/2005.
#  If you think this script can be improved,
#+ email me at: monster@monstruum.co.uk
# ==> and cc: to the author of the ABS Guide, please.
#  This script is licenced under the GPL.
#  You are free to copy, alter and re-use it,
#+ but please don't try to claim you wrote it.
#  Log your changes here instead.

# =======================================================================
# changelog:

# 07/02/2005.  Fixups by Little Monster.
# 02/02/2005.  Minor additions by Little Monster.
#              (See after # +++++++++++ )
# 29/01/2005.  Minor stylistic edits and cleanups by author of ABS Guide.
#              Added exit error codes.
# 22/11/2004.  Finished initial version of second version of wgetter:
#              wgetter2 is born.
# 01/12/2004.  Changed 'runn' function so it can be run 2 ways --
#              either ask for a file name or have one input on the CL.
# 01/12/2004.  Made sensible handling of no URL's given.
# 01/12/2004.  Made loop of main options, so you don't
#              have to keep calling wgetter 2 all the time.
#              Runs as a session instead.
# 01/12/2004.  Added looping to 'runn' function.
#              Simplified and improved.
# 01/12/2004.  Added state to recursion setting.
#              Enables re-use of previous value.
# 05/12/2004.  Modified the file detection routine in the 'runn' function
#              so it's not fooled by empty values, and is cleaner.
# 01/02/2004.  Added cookie finding routine from later version (which
#              isn't ready yet), so as not to have hard-coded paths.
# =======================================================================

# Error codes for abnormal exit.
E_USAGE=67        # Usage message, then quit.
E_NO_OPTS=68      # No command-line args entered.
E_NO_URLS=69      # No URLs passed to script.
E_NO_SAVEFILE=70  # No save filename passed to script.
E_USER_EXIT=71    # User decides to quit.


#  Basic default wget command we want to use.
#  This is the place to change it, if required.
#  NB: if using a proxy, set http_proxy = yourproxy in .wgetrc.
#  Otherwise delete --proxy=on, below.
# ====================================================================
CommandA="wget -nc -c -t 5 --progress=bar --random-wait --proxy=on -r"
# ====================================================================



# --------------------------------------------------------------------
# Set some other variables and explain them.

pattern=" -A .jpg,.JPG,.jpeg,.JPEG,.gif,.GIF,.htm,.html,.shtml,.php"
                    # wget's option to only get certain types of file.
                    # comment out if not using
```

```
today=`date +%F`    # Used for a filename.
home=$HOME          # Set HOME to an internal variable.
                    # In case some other path is used, change it here.
depthDefault=3      # Set a sensible default recursion.
Depth=$depthDefault # Otherwise user feedback doesn't tie in properly.
RefA=""             # Set blank referring page.
Flag=""             #  Default to not saving anything,
                    #+ or whatever else might be wanted in future.
lister=""           # Used for passing a list of urls directly to wget.
Woptions=""         # Used for passing wget some options for itself.
inFile=""           # Used for the run function.
newFile=""          # Used for the run function.
savePath="$home/w-save"
Config="$home/.wgetter2rc"
                    #  This is where some variables can be stored,
                    #+ if permanently changed from within the script.
Cookie_List="$home/.cookielist"
                    # So we know where the cookies are kept . . .
cFlag=""            # Part of the cookie file selection routine.

# Define the options available. Easy to change letters here if needed.
# These are the optional options; you don't just wait to be asked.

save=s   # Save command instead of executing it.
cook=c   # Change cookie file for this session.
help=h   # Usage guide.
list=l   # Pass wget the -i option and URL list.
runn=r   # Run saved commands as an argument to the option.
inpu=i   # Run saved commands interactively.
wopt=w   # Allow to enter options to pass directly to wget.
# ----------------------------------------------------------------------


if [ -z "$1" ]; then   # Make sure we get something for wget to eat.
   echo "You must at least enter a URL or option!"
   echo "-$help for usage."
   exit $E_NO_OPTS
fi



# +++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
# added added added added added added added added added added added added

if [ ! -e "$Config" ]; then   # See if configuration file exists.
   echo "Creating configuration file, $Config"
   echo "# This is the configuration file for wgetter2" > "$Config"
   echo "# Your customised settings will be saved in this file" >> "$Config"
else
   source $Config             # Import variables we set outside the script.
fi

if [ ! -e "$Cookie_List" ]; then
   # Set up a list of cookie files, if there isn't one.
   echo "Hunting for cookies . . ."
   find -name cookies.txt >> $Cookie_List # Create the list of cookie files.
fi #  Isolate this in its own 'if' statement,
   #+ in case we got interrupted while searching.

if [ -z "$cFlag" ]; then # If we haven't already done this . . .
   echo                  # Make a nice space after the command prompt.
   echo "Looks like you haven't set up your source of cookies yet."
```

```
   n=0                   #  Make sure the counter
                         #+ doesn't contain random values.
   while read; do
      Cookies[$n]=$REPLY # Put the cookie files we found into an array.
      echo "$n) ${Cookies[$n]}"  # Create a menu.
      n=$(( n + 1 ))     # Increment the counter.
   done < $Cookie_List   # Feed the read statement.
   echo "Enter the number of the cookie file you want to use."
   echo "If you won't be using cookies, just press RETURN."
   echo
   echo "I won't be asking this again. Edit $Config"
   echo "If you decide to change at a later date"
   echo "or use the -${cook} option for per session changes."
   read
   if [ ! -z $REPLY ]; then   # User didn't just press return.
      Cookie=" --load-cookies ${Cookies[$REPLY]}"
      # Set the variable here as well as in the config file.

      echo "Cookie=\" --load-cookies ${Cookies[$REPLY]}\"" >> $Config
   fi
   echo "cFlag=1" >> $Config  # So we know not to ask again.
fi

# end added section end added section end added section end added section
# +++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++



# Another variable.
# This one may or may not be subject to variation.
# A bit like the small print.
CookiesON=$Cookie
# echo "cookie file is $CookiesON" # For debugging.
# echo "home is ${home}"           # For debugging.
                                   # Got caught with this one!


wopts()
{
echo "Enter options to pass to wget."
echo "It is assumed you know what you're doing."
echo
echo "You can pass their arguments here too."
# That is to say, everything passed here is passed to wget.

read Wopts
# Read in the options to be passed to wget.

Woptions=" $Wopts"
#         ^  Why the leading space?
# Assign to another variable.
# Just for fun, or something . . .

echo "passing options ${Wopts} to wget"
# Mainly for debugging.
# Is cute.

return
}


save_func()
```

```
{
echo "Settings will be saved."
if [ ! -d $savePath ]; then  #  See if directory exists.
   mkdir $savePath           #  Create the directory to save things in
                             #+ if it isn't already there.
fi

Flag=S
# Tell the final bit of code what to do.
# Set a flag since stuff is done in main.

return
}


usage() # Tell them how it works.
{
    echo "Welcome to wgetter.  This is a front end to wget."
    echo "It will always run wget with these options:"
    echo "$CommandA"
    echo "and the pattern to match: $pattern \
(which you can change at the top of this script)."
    echo "It will also ask you for recursion depth, \
and if you want to use a referring page."
    echo "Wgetter accepts the following options:"
    echo ""
    echo "-$help : Display this help."
    echo "-$save : Save the command to a file $savePath/wget-($today) \
instead of running it."
    echo "-$runn : Run saved wget commands instead of starting a new one -"
    echo "Enter filename as argument to this option."
    echo "-$inpu : Run saved wget commands interactively --"
    echo "The script will ask you for the filename."
    echo "-$cook : Change the cookies file for this session."
    echo "-$list : Tell wget to use URL's from a list instead of \
from the command-line."
    echo "-$wopt : Pass any other options direct to wget."
    echo ""
    echo "See the wget man page for additional options \
you can pass to wget."
    echo ""

    exit $E_USAGE  # End here. Don't process anything else.
}



list_func() #  Gives the user the option to use the -i option to wget,
            #+ and a list of URLs.
{
while [ 1 ]; do
   echo "Enter the name of the file containing URL's (press q to change
your mind)."
   read urlfile
   if [ ! -e "$urlfile" ] && [ "$urlfile" != q ]; then
       # Look for a file, or the quit option.
       echo "That file does not exist!"
   elif [ "$urlfile" = q ]; then # Check quit option.
       echo "Not using a url list."
       return
   else
      echo "using $urlfile."
```

```
      echo "If you gave url's on the command-line, I'll use those first."
                              # Report wget standard behaviour to the user.
      lister=" -i $urlfile" # This is what we want to pass to wget.
      return
   fi
done
}


cookie_func() # Give the user the option to use a different cookie file.
{
while [ 1 ]; do
   echo "Change the cookies file. Press return if you don't want to change
it."
   read Cookies
   # NB: this is not the same as Cookie, earlier.
   # There is an 's' on the end.
   # Bit like chocolate chips.
   if [ -z "$Cookies" ]; then                 # Escape clause for wusses.
      return
   elif [ ! -e "$Cookies" ]; then
      echo "File does not exist.  Try again." # Keep em going . . .
   else
       CookiesON=" --load-cookies $Cookies"   # File is good -- use it!
       return
   fi
done
}



run_func()
{
if [ -z "$OPTARG" ]; then
# Test to see if we used the in-line option or the query one.
   if [ ! -d "$savePath" ]; then      # If directory doesn't exist . . .
      echo "$savePath does not appear to exist."
      echo "Please supply path and filename of saved wget commands:"
      read newFile
         until [ -f "$newFile" ]; do  # Keep going till we get something.
            echo "Sorry, that file does not exist.  Please try again."
            # Try really hard to get something.
            read newFile
         done


# -------------------------------------------------------------------------
#       if [ -z ( grep wget ${newfile} ) ]; then
        # Assume they haven't got the right file and bail out.
#       echo "Sorry, that file does not contain wget commands.  Aborting."
#       exit
#       fi
#
# This is bogus code.
# It doesn't actually work.
# If anyone wants to fix it, feel free!
# -------------------------------------------------------------------------


      filePath="${newFile}"
   else
   echo "Save path is $savePath"
```

```
     echo "Please enter name of the file which you want to use."
     echo "You have a choice of:"
     ls $savePath                                    # Give them a choice.
     read inFile
       until [ -f "$savePath/$inFile" ]; do         #  Keep going till
                                                    #+ we get something.
          if [ ! -f "${savePath}/${inFile}" ]; then # If file doesn't exist.
             echo "Sorry, that file does not exist.  Please choose from:"
             ls $savePath                           # If a mistake is made.
             read inFile
          fi
         done
      filePath="${savePath}/${inFile}"  # Make one variable . . .
   fi
else filePath="${savePath}/${OPTARG}"   # Which can be many things . . .
fi

if [ ! -f "$filePath" ]; then           # If a bogus file got through.
   echo "You did not specify a suitable file."
   echo "Run this script with the -${save} option first."
   echo "Aborting."
   exit $E_NO_SAVEFILE
fi
echo "Using: $filePath"
while read; do
    eval $REPLY
    echo "Completed: $REPLY"
done < $filePath  # Feed the actual file we are using into a 'while' loop.

exit
}



# Fish out any options we are using for the script.
# This is based on the demo in "Learning The Bash Shell" (O'Reilly).
while getopts ":$save$cook$help$list$runn:$inpu$wopt" opt
do
  case $opt in
     $save) save_func;;   #  Save some wgetter sessions for later.
     $cook) cookie_func;; #  Change cookie file.
     $help) usage;;       #  Get help.
     $list) list_func;;   #  Allow wget to use a list of URLs.
     $runn) run_func;;    #  Useful if you are calling wgetter from,
                          #+ for example, a cron script.
     $inpu) run_func;;    #  When you don't know what your files are named.
     $wopt) wopts;;       #  Pass options directly to wget.
        \?) echo "Not a valid option."
            echo "Use -${wopt} to pass options directly to wget,"
            echo "or -${help} for help";;      # Catch anything else.
  esac
done
shift $((OPTIND - 1))     # Do funky magic stuff with $#.


if [ -z "$1" ] && [ -z "$lister" ]; then
                          #  We should be left with at least one URL
                          #+ on the command-line, unless a list is
                          #+ being used -- catch empty CL's.
   echo "No URL's given! You must enter them on the same line as wgetter2."
   echo "E.g.,  wgetter2 http://somesite http://anothersite."
   echo "Use $help option for more information."
```

```
   exit $E_NO_URLS        # Bail out, with appropriate error code.
fi

URLS=" $@"
# Use this so that URL list can be changed if we stay in the option loop.

while [ 1 ]; do
   # This is where we ask for the most used options.
   # (Mostly unchanged from version 1 of wgetter)
   if [ -z $curDepth ]; then
      Current=""
   else Current=" Current value is $curDepth"
   fi
       echo "How deep should I go? \
(integer: Default is $depthDefault.$Current)"
       read Depth   # Recursion -- how far should we go?
       inputB=""    # Reset this to blank on each pass of the loop.
       echo "Enter the name of the referring page (default is none)."
       read inputB  # Need this for some sites.

       echo "Do you want to have the output logged to the terminal"
       echo "(y/n, default is yes)?"
       read noHide  # Otherwise wget will just log it to a file.

       case $noHide in    # Now you see me, now you don't.
          y|Y ) hide="";;
          n|N ) hide=" -b";;
            * ) hide="";;
       esac

       if [ -z ${Depth} ]; then
       #  User accepted either default or current depth,
       #+ in which case Depth is now empty.
          if [ -z ${curDepth} ]; then
          #  See if a depth was set on a previous iteration.
             Depth="$depthDefault"
             #  Set the default recursion depth if nothing
             #+ else to use.
          else Depth="$curDepth" #  Otherwise, set the one we used before.
          fi
       fi
   Recurse=" -l $Depth"          # Set how deep we want to go.
   curDepth=$Depth               # Remember setting for next time.

       if [ ! -z $inputB ]; then
          RefA=" --referer=$inputB"   # Option to use referring page.
       fi

   WGETTER="${CommandA}${pattern}${hide}${RefA}${Recurse}\
${CookiesON}${lister}${Woptions}${URLS}"
   #  Just string the whole lot together . . .
   #  NB: no embedded spaces.
   #  They are in the individual elements so that if any are empty,
   #+ we don't get an extra space.

   if [ -z "${CookiesON}" ] && [ "$cFlag" = "1" ] ; then
       echo "Warning -- can't find cookie file"
       #  This should be changed,
       #+ in case the user has opted to not use cookies.
   fi

   if [ "$Flag" = "S" ]; then
```

```
      echo "$WGETTER" >> $savePath/wget-${today}
      #  Create a unique filename for today, or append to it if it exists.
      echo "$inputB" >> $savePath/site-list-${today}
      #  Make a list, so it's easy to refer back to,
      #+ since the whole command is a bit confusing to look at.
      echo "Command saved to the file $savePath/wget-${today}"
           # Tell the user.
      echo "Referring page URL saved to the file$ \
savePath/site-list-${today}"
           # Tell the user.
      Saver=" with save option"
      # Stick this somewhere, so it appears in the loop if set.
   else
       echo "*****************"
       echo "*****Getting*****"
       echo "*****************"
       echo ""
       echo "$WGETTER"
       echo ""
       echo "*****************"
       eval "$WGETTER"
   fi

       echo ""
       echo "Starting over$Saver."
       echo "If you want to stop, press q."
       echo "Otherwise, enter some URL's:"
       # Let them go again. Tell about save option being set.

       read
       case $REPLY in
       # Need to change this to a 'trap' clause.
          q|Q ) exit $E_USER_EXIT;;  # Exercise for the reader?
            * ) URLS=" $REPLY";;
       esac

       echo ""
done


exit 0
```

Example A-31. A *podcasting* script

```
#!/bin/bash

#  bashpodder.sh:
#  By Linc 10/1/2004
#  Find the latest script at
#+ http://linc.homeunix.org:8080/scripts/bashpodder
#  Last revision 12/14/2004 - Many Contributors!
#  If you use this and have made improvements or have comments
#+ drop me an email at linc dot fessenden at gmail dot com
#  I'd appreciate it!

# ==>  ABS Guide extra comments.

# ==>  Author of this script has kindly granted permission
# ==>+ for inclusion in ABS Guide.
```

```
# ==> ################################################################
#
# ==> What is "podcasting"?

# ==> It's broadcasting "radio shows" over the Internet.
# ==> These shows can be played on iPods and other music file players.

# ==> This script makes it possible.
# ==> See documentation at the script author's site, above.

# ==> ################################################################


# Make script crontab friendly:
cd $(dirname $0)
# ==> Change to directory where this script lives.

# datadir is the directory you want podcasts saved to:
datadir=$(date +%Y-%m-%d)
# ==> Will create a date-labeled directory, named: YYYY-MM-DD

# Check for and create datadir if necessary:
if test ! -d $datadir
        then
        mkdir $datadir
fi

# Delete any temp file:
rm -f temp.log

#  Read the bp.conf file and wget any url not already
#+ in the podcast.log file:
while read podcast
  do # ==> Main action follows.
  file=$(wget -q $podcast -O - | tr '\r' '\n' | tr \' \" | \
sed -n 's/.*url="\([^"]*\)".*/\1/p')
  for url in $file
                do
                echo $url >> temp.log
                if ! grep "$url" podcast.log > /dev/null
                        then
                        wget -q -P $datadir "$url"
                fi
                done
    done < bp.conf

# Move dynamically created log file to permanent log file:
cat podcast.log >> temp.log
sort temp.log | uniq > podcast.log
rm temp.log
# Create an m3u playlist:
ls $datadir | grep -v m3u > $datadir/podcast.m3u


exit 0

#################################################
For a different scripting approach to Podcasting,
see Phil Salkie's article,
"Internet Radio to Podcast with Shell Tools"
in the September, 2005 issue of LINUX JOURNAL,
http://www.linuxjournal.com/article/8171
```

```
################################################
```

Example A-32. Nightly backup to a firewire HD

```
#!/bin/bash
# nightly-backup.sh
# http://www.richardneill.org/source.php#nightly-backup-rsync
# Copyright (c) 2005 Richard Neill <backup@richardneill.org>.
# This is Free Software licensed under the GNU GPL.
# ==> Included in ABS Guide with script author's kind permission.
# ==> (Thanks!)

#  This does a backup from the host computer to a locally connected
#+ firewire HDD using rsync and ssh.
#  (Script should work with USB-connected device (see lines 40-43).
#  It then rotates the backups.
#  Run it via cron every night at 5am.
#  This only backs up the home directory.
#  If ownerships (other than the user's) should be preserved,
#+ then run the rsync process as root (and re-instate the -o).
#  We save every day for 7 days, then every week for 4 weeks,
#+ then every month for 3 months.

#  See: http://www.mikerubel.org/computers/rsync_snapshots/
#+ for more explanation of the theory.
#  Save as: $HOME/bin/nightly-backup_firewire-hdd.sh

#  Known bugs:
#  ----------
#  i)  Ideally, we want to exclude ~/.tmp and the browser caches.

#  ii) If the user is sitting at the computer at 5am,
#+     and files are modified while the rsync is occurring,
#+     then the BACKUP_JUSTINCASE branch gets triggered.
#      To some extent, this is a
#+     feature, but it also causes a "disk-space leak".





##### BEGIN CONFIGURATION SECTION ############################################
LOCAL_USER=rjn                # User whose home directory should be backed up.
MOUNT_POINT=/backup           # Mountpoint of backup drive.
                              # NO trailing slash!
                              # This must be unique (eg using a udev symlink)
# MOUNT_POINT=/media/disk     # For USB-connected device.
SOURCE_DIR=/home/$LOCAL_USER  # NO trailing slash - it DOES matter to rsync.
BACKUP_DEST_DIR=$MOUNT_POINT/backup/`hostname -s`.${LOCAL_USER}.nightly_backup
DRY_RUN=false                 #If true, invoke rsync with -n, to do a dry run.
                              # Comment out or set to false for normal use.
VERBOSE=false                 # If true, make rsync verbose.
                              # Comment out or set to false otherwise.
COMPRESS=false                # If true, compress.
                              # Good for internet, bad on LAN.
                              # Comment out or set to false otherwise.

### Exit Codes ###
E_VARS_NOT_SET=64
E_COMMANDLINE=65
E_MOUNT_FAIL=70
```

```
E_NOSOURCEDIR=71
E_UNMOUNTED=72
E_BACKUP=73
##### END CONFIGURATION SECTION ##############################################


# Check that all the important variables have been set:
if [ -z "$LOCAL_USER" ] ||
   [ -z "$SOURCE_DIR" ] ||
   [ -z "$MOUNT_POINT" ]  ||
   [ -z "$BACKUP_DEST_DIR" ]
then
   echo 'One of the variables is not set! Edit the file: $0. BACKUP FAILED.'
   exit $E_VARS_NOT_SET
fi

if [ "$#" != 0 ]  # If command-line param(s) . . .
then              # Here document(ation).
  cat <<-ENDOFTEXT
    Automatic Nightly backup run from cron.
    Read the source for more details: $0
    The backup directory is $BACKUP_DEST_DIR .
    It will be created if necessary; initialisation is no longer required.

    WARNING: Contents of $BACKUP_DEST_DIR are rotated.
    Directories named 'backup.\$i' will eventually be DELETED.
    We keep backups from every day for 7 days (1-8),
    then every week for 4 weeks (9-12),
    then every month for 3 months (13-15).

    You may wish to add this to your crontab using 'crontab -e'
    #  Back up files: $SOURCE_DIR to $BACKUP_DEST_DIR
    #+ every night at 3:15 am
         15 03 * * * /home/$LOCAL_USER/bin/nightly-backup_firewire-hdd.sh

    Don't forget to verify the backups are working,
    especially if you don't read cron's mail!"
        ENDOFTEXT
   exit $E_COMMANDLINE
fi


# Parse the options.
# ==================

if [ "$DRY_RUN" == "true" ]; then
  DRY_RUN="-n"
  echo "WARNING:"
  echo "THIS IS A 'DRY RUN'!"
  echo "No data will actually be transferred!"
else
  DRY_RUN=""
fi

if [ "$VERBOSE" == "true" ]; then
  VERBOSE="-v"
else
  VERBOSE=""
fi

if [ "$COMPRESS" == "true" ]; then
  COMPRESS="-z"
```

```
else
  COMPRESS=""
fi


#  Every week (actually of 8 days) and every month,
#+ extra backups are preserved.
DAY_OF_MONTH=`date +%d`            # Day of month (01..31).
if [ $DAY_OF_MONTH = 01 ]; then    # First of month.
  MONTHSTART=true
elif [ $DAY_OF_MONTH = 08 \
    -o $DAY_OF_MONTH = 16 \
    -o $DAY_OF_MONTH = 24 ]; then
    # Day 8,16,24  (use 8, not 7 to better handle 31-day months)
      WEEKSTART=true
fi



#  Check that the HDD is mounted.
#  At least, check that *something* is mounted here!
#  We can use something unique to the device, rather than just guessing
#+ the scsi-id by having an appropriate udev rule in
#+ /etc/udev/rules.d/10-rules.local
#+ and by putting a relevant entry in /etc/fstab.
#  Eg: this udev rule:
# BUS="scsi", KERNEL="sd*", SYSFS{vendor}="WDC WD16",
# SYSFS{model}="00JB-00GVA0     ", NAME="%k", SYMLINK="lacie_1394d%n"

if mount | grep $MOUNT_POINT >/dev/null; then
  echo "Mount point $MOUNT_POINT is indeed mounted. OK"
else
  echo -n "Attempting to mount $MOUNT_POINT..."
           # If it isn't mounted, try to mount it.
  sudo mount $MOUNT_POINT 2>/dev/null

  if mount | grep $MOUNT_POINT >/dev/null; then
    UNMOUNT_LATER=TRUE
    echo "OK"
    #  Note: Ensure that this is also unmounted
    #+ if we exit prematurely with failure.
  else
    echo "FAILED"
    echo -e "Nothing is mounted at $MOUNT_POINT. BACKUP FAILED!"
    exit $E_MOUNT_FAIL
  fi
fi


# Check that source dir exists and is readable.
if [ ! -r  $SOURCE_DIR ] ; then
  echo "$SOURCE_DIR does not exist, or cannot be read. BACKUP FAILED."
  exit $E_NOSOURCEDIR
fi


# Check that the backup directory structure is as it should be.
# If not, create it.
# Create the subdirectories.
# Note that backup.0 will be created as needed by rsync.

for ((i=1;i<=15;i++)); do
```

```
  if [ ! -d $BACKUP_DEST_DIR/backup.$i ]; then
    if /bin/mkdir -p $BACKUP_DEST_DIR/backup.$i ; then
    #  ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^  No [ ] test brackets. Why?
      echo "Warning: directory $BACKUP_DEST_DIR/backup.$i is missing,"
      echo "or was not initialised. (Re-)creating it."
    else
      echo "ERROR: directory $BACKUP_DEST_DIR/backup.$i"
      echo "is missing and could not be created."
    if  [ "$UNMOUNT_LATER" == "TRUE" ]; then
        # Before we exit, unmount the mount point if necessary.
        cd
        sudo umount $MOUNT_POINT &&
        echo "Unmounted $MOUNT_POINT again. Giving up."
    fi
      exit $E_UNMOUNTED
  fi
fi
done


#  Set the permission to 700 for security
#+ on an otherwise permissive multi-user system.
if ! /bin/chmod 700 $BACKUP_DEST_DIR ; then
  echo "ERROR: Could not set permissions on $BACKUP_DEST_DIR to 700."

  if  [ "$UNMOUNT_LATER" == "TRUE" ]; then
  # Before we exit, unmount the mount point if necessary.
     cd ; sudo umount $MOUNT_POINT \
     && echo "Unmounted $MOUNT_POINT again. Giving up."
  fi

  exit $E_UNMOUNTED
fi

# Create the symlink: current -> backup.1 if required.
# A failure here is not critical.
cd $BACKUP_DEST_DIR
if [ ! -h current ] ; then
  if ! /bin/ln -s backup.1 current ; then
    echo "WARNING: could not create symlink current -> backup.1"
  fi
fi


# Now, do the rsync.
echo "Now doing backup with rsync..."
echo "Source dir: $SOURCE_DIR"
echo -e "Backup destination dir: $BACKUP_DEST_DIR\n"


/usr/bin/rsync $DRY_RUN $VERBOSE -a -S --delete --modify-window=60 \
--link-dest=../backup.1 $SOURCE_DIR $BACKUP_DEST_DIR/backup.0/

#  Only warn, rather than exit if the rsync failed,
#+ since it may only be a minor problem.
#  E.g., if one file is not readable, rsync will fail.
#  This shouldn't prevent the rotation.
#  Not using, e.g., `date +%a`  since these directories
#+ are just full of links and don't consume *that much* space.

if [ $? != 0 ]; then
  BACKUP_JUSTINCASE=backup.`date +%F_%T`.justincase
```

```
  echo "WARNING: the rsync process did not entirely succeed."
  echo "Something might be wrong."
  echo "Saving an extra copy at: $BACKUP_JUSTINCASE"
  echo "WARNING: if this occurs regularly, a LOT of space will be consumed,"
  echo "even though these are just hard-links!"
fi

# Save a readme in the backup parent directory.
# Save another one in the recent subdirectory.
echo "Backup of $SOURCE_DIR on `hostname` was last run on \
`date`" > $BACKUP_DEST_DIR/README.txt
echo "This backup of $SOURCE_DIR on `hostname` was created on \
`date`" > $BACKUP_DEST_DIR/backup.0/README.txt

# If we are not in a dry run, rotate the backups.
[ -z "$DRY_RUN" ] &&

  #  Check how full the backup disk is.
  #  Warn if 90%. if 98% or more, we'll probably fail, so give up.
  #  (Note: df can output to more than one line.)
  #  We test this here, rather than before
  #+ so that rsync may possibly have a chance.
  DISK_FULL_PERCENT=`/bin/df $BACKUP_DEST_DIR |
  tr "\n" ' ' | awk '{print $12}' | grep -oE [0-9]+ `
  echo "Disk space check on backup partition \
  $MOUNT_POINT $DISK_FULL_PERCENT% full."
  if [ $DISK_FULL_PERCENT -gt 90 ]; then
    echo "Warning: Disk is greater than 90% full."
  fi
  if [ $DISK_FULL_PERCENT -gt 98 ]; then
    echo "Error: Disk is full! Giving up."
      if  [ "$UNMOUNT_LATER" == "TRUE" ]; then
        # Before we exit, unmount the mount point if necessary.
        cd; sudo umount $MOUNT_POINT &&
        echo "Unmounted $MOUNT_POINT again. Giving up."
      fi
    exit $E_UNMOUNTED
  fi


 # Create an extra backup.
 # If this copy fails, give up.
 if [ -n "$BACKUP_JUSTINCASE" ]; then
   if ! /bin/cp -al $BACKUP_DEST_DIR/backup.0 \
      $BACKUP_DEST_DIR/$BACKUP_JUSTINCASE
   then
     echo "ERROR: Failed to create extra copy \
     $BACKUP_DEST_DIR/$BACKUP_JUSTINCASE"
     if  [ "$UNMOUNT_LATER" == "TRUE" ]; then
       # Before we exit, unmount the mount point if necessary.
       cd ;sudo umount $MOUNT_POINT &&
       echo "Unmounted $MOUNT_POINT again. Giving up."
     fi
     exit $E_UNMOUNTED
   fi
 fi


 # At start of month, rotate the oldest 8.
 if [ "$MONTHSTART" == "true" ]; then
   echo -e "\nStart of month. \
   Removing oldest backup: $BACKUP_DEST_DIR/backup.15"  &&
```

```
    /bin/rm -rf  $BACKUP_DEST_DIR/backup.15  &&
    echo "Rotating monthly,weekly backups: \
    $BACKUP_DEST_DIR/backup.[8-14] -> $BACKUP_DEST_DIR/backup.[9-15]"  &&
      /bin/mv $BACKUP_DEST_DIR/backup.14 $BACKUP_DEST_DIR/backup.15  &&
      /bin/mv $BACKUP_DEST_DIR/backup.13 $BACKUP_DEST_DIR/backup.14  &&
      /bin/mv $BACKUP_DEST_DIR/backup.12 $BACKUP_DEST_DIR/backup.13  &&
      /bin/mv $BACKUP_DEST_DIR/backup.11 $BACKUP_DEST_DIR/backup.12  &&
      /bin/mv $BACKUP_DEST_DIR/backup.10 $BACKUP_DEST_DIR/backup.11  &&
      /bin/mv $BACKUP_DEST_DIR/backup.9 $BACKUP_DEST_DIR/backup.10  &&
      /bin/mv $BACKUP_DEST_DIR/backup.8 $BACKUP_DEST_DIR/backup.9

  # At start of week, rotate the second-oldest 4.
  elif [ "$WEEKSTART" == "true" ]; then
    echo -e "\nStart of week. \
    Removing oldest weekly backup: $BACKUP_DEST_DIR/backup.12"  &&
    /bin/rm -rf  $BACKUP_DEST_DIR/backup.12  &&

    echo "Rotating weekly backups: \
    $BACKUP_DEST_DIR/backup.[8-11] -> $BACKUP_DEST_DIR/backup.[9-12]"  &&
      /bin/mv $BACKUP_DEST_DIR/backup.11 $BACKUP_DEST_DIR/backup.12  &&
      /bin/mv $BACKUP_DEST_DIR/backup.10 $BACKUP_DEST_DIR/backup.11  &&
      /bin/mv $BACKUP_DEST_DIR/backup.9 $BACKUP_DEST_DIR/backup.10  &&
      /bin/mv $BACKUP_DEST_DIR/backup.8 $BACKUP_DEST_DIR/backup.9

  else
    echo -e "\nRemoving oldest daily backup: $BACKUP_DEST_DIR/backup.8"  &&
      /bin/rm -rf  $BACKUP_DEST_DIR/backup.8

  fi  &&

  # Every day, rotate the newest 8.
  echo "Rotating daily backups: \
  $BACKUP_DEST_DIR/backup.[1-7] -> $BACKUP_DEST_DIR/backup.[2-8]"  &&
      /bin/mv $BACKUP_DEST_DIR/backup.7 $BACKUP_DEST_DIR/backup.8  &&
      /bin/mv $BACKUP_DEST_DIR/backup.6 $BACKUP_DEST_DIR/backup.7  &&
      /bin/mv $BACKUP_DEST_DIR/backup.5 $BACKUP_DEST_DIR/backup.6  &&
      /bin/mv $BACKUP_DEST_DIR/backup.4 $BACKUP_DEST_DIR/backup.5  &&
      /bin/mv $BACKUP_DEST_DIR/backup.3 $BACKUP_DEST_DIR/backup.4  &&
      /bin/mv $BACKUP_DEST_DIR/backup.2 $BACKUP_DEST_DIR/backup.3  &&
      /bin/mv $BACKUP_DEST_DIR/backup.1 $BACKUP_DEST_DIR/backup.2  &&
      /bin/mv $BACKUP_DEST_DIR/backup.0 $BACKUP_DEST_DIR/backup.1  &&

  SUCCESS=true


if  [ "$UNMOUNT_LATER" == "TRUE" ]; then
  # Unmount the mount point if it wasn't mounted to begin with.
  cd ; sudo umount $MOUNT_POINT && echo "Unmounted $MOUNT_POINT again."
fi


if [ "$SUCCESS" == "true" ]; then
  echo 'SUCCESS!'
  exit 0
fi

# Should have already exited if backup worked.
echo 'BACKUP FAILED! Is this just a dry run? Is the disk full?) '
exit $E_BACKUP
```

Example A-33. An expanded *cd* command

```
###########################################################################
#
#       cdll
#       by Phil Braham
#
#       ############################################
#       Latest version of this script available from
#       http://freshmeat.net/projects/cd/
#       ############################################
#
#       .cd_new
#
#       An enhancement of the Unix cd command
#
#       There are unlimited stack entries and special entries. The stack
#       entries keep the last cd_maxhistory
#       directories that have been used. The special entries can be
#       assigned to commonly used directories.
#
#       The special entries may be pre-assigned by setting the environment
#       variables CDSn or by using the -u or -U command.
#
#       The following is a suggestion for the .profile file:
#
#               . cdll              #  Set up the cd command
#       alias cd='cd_new'           #  Replace the cd command
#               cd -U               #  Upload pre-assigned entries for
#                                   #+ the stack and special entries
#               cd -D               #  Set non-default mode
#               alias @="cd_new @"  #  Allow @ to be used to get history
#
#       For help type:
#
#               cd -h or
#               cd -H
#
#
###########################################################################
#
#       Version 1.2.1
#
#       Written by Phil Braham - Realtime Software Pty Ltd
#       (realtime@mpx.com.au)
#       Please send any suggestions or enhancements to the author (also at
#       phil@braham.net)
#
############################################################################

cd_hm ()
{
        ${PRINTF} "%s" "cd [dir] [0-9] [@[s|h] [-g [<dir>]] [-d] \
[-D] [-r<n>] [dir|0-9] [-R<n>] [<dir>|0-9]
   [-s<n>] [-S<n>] [-u] [-U] [-f] [-F] [-h] [-H] [-v]
    <dir> Go to directory
    0-n         Go to previous directory (0 is previous, 1 is last but 1 etc)
                n is up to max history (default is 50)
    @           List history and special entries
    @h          List history entries
    @s          List special entries
    -g [<dir>]  Go to literal name (bypass special names)
                This is to allow access to dirs called '0','1','-h' etc
    -d          Change default action - verbose. (See note)
```

```
    -D          Change default action - silent. (See note)
    -s<n> Go to the special entry <n>*
    -S<n> Go to the special entry <n>
                and replace it with the current dir*
    -r<n> [<dir>] Go to directory <dir>
                              and then put it on special entry <n>*
    -R<n> [<dir>] Go to directory <dir>
                              and put current dir on special entry <n>*
    -a<n>       Alternative suggested directory. See note below.
    -f [<file>] File entries to <file>.
    -u [<file>] Update entries from <file>.
                If no filename supplied then default file
                (${CDPath}${2:-"$CDFile"}) is used
                -F and -U are silent versions
    -v          Print version number
    -h          Help
    -H          Detailed help

    *The special entries (0 - 9) are held until log off, replaced by another
     entry or updated with the -u command

    Alternative suggested directories:
    If a directory is not found then CD will suggest any
    possibilities. These are directories starting with the same letters
    and if any are found they are listed prefixed with -a<n>
    where <n> is a number.
    It's possible to go to the directory by entering cd -a<n>
    on the command line.

    The directory for -r<n> or -R<n> may be a number.
    For example:
        $ cd -r3 4  Go to history entry 4 and put it on special entry 3
        $ cd -R3 4  Put current dir on the special entry 3
                    and go to history entry 4
        $ cd -s3    Go to special entry 3

    Note that commands R,r,S and s may be used without a number
    and refer to 0:
        $ cd -s     Go to special entry 0
        $ cd -S     Go to special entry 0 and make special
                    entry 0 current dir
        $ cd -r 1   Go to history entry 1 and put it on special entry 0
        $ cd -r     Go to history entry 0 and put it on special entry 0
    "
        if ${TEST} "$CD_MODE" = "PREV"
        then
                ${PRINTF} "$cd_mnset"
        else
                ${PRINTF} "$cd_mset"
        fi
}

cd_Hm ()
{
        cd_hm
        ${PRINTF} "%s" "
        The previous directories (0-$cd_maxhistory) are stored in the
        environment variables CD[0] - CD[$cd_maxhistory]
        Similarly the special directories S0 - $cd_maxspecial are in
        the environment variable CDS[0] - CDS[$cd_maxspecial]
        and may be accessed from the command line
```

```
        The default pathname for the -f and -u commands is $CDPath
        The default filename for the -f and -u commands is $CDFile

        Set the following environment variables:
            CDL_PROMPTLEN  - Set to the length of prompt you require.
                Prompt string is set to the right characters of the
                current directory.
                If not set then prompt is left unchanged
            CDL_PROMPT_PRE - Set to the string to prefix the prompt.
                Default is:
                    non-root:  \"\\[\\e[01;34m\\]\"  (sets colour to blue).
                    root:      \"\\[\\e[01;31m\\]\"  (sets colour to red).
            CDL_PROMPT_POST    - Set to the string to suffix the prompt.
                Default is:
                    non-root:  \"\\[\\e[00m\\]$\"
                                (resets colour and displays $).
                    root:      \"\\[\\e[00m\\]#\"
                                (resets colour and displays #).
            CDPath - Set the default path for the -f & -u options.
                     Default is home directory
            CDFile - Set the default filename for the -f & -u options.
                     Default is cdfile

"
    cd_version

}

cd_version ()
{
 printf "Version: ${VERSION_MAJOR}.${VERSION_MINOR} Date: ${VERSION_DATE}\n"
}

#
# Truncate right.
#
# params:
#   p1 - string
#   p2 - length to truncate to
#
# returns string in tcd
#
cd_right_trunc ()
{
    local tlen=${2}
    local plen=${#1}
    local str="${1}"
    local diff
    local filler="<--"
    if ${TEST} ${plen} -le ${tlen}
    then
        tcd="${str}"
    else
        let diff=${plen}-${tlen}
        elen=3
        if ${TEST} ${diff} -le 2
        then
            let elen=${diff}
        fi
        tlen=-${tlen}
        let tlen=${tlen}+${elen}
        tcd=${filler:0:elen}${str:tlen}
```

```
    fi
}

#
# Three versions of do history:
#    cd_dohistory  - packs history and specials side by side
#    cd_dohistoryH - Shows only hstory
#    cd_dohistoryS - Shows only specials
#
cd_dohistory ()
{
    cd_getrc
        ${PRINTF} "History:\n"
    local -i count=${cd_histcount}
    while ${TEST} ${count} -ge 0
    do
        cd_right_trunc "${CD[count]}" ${cd_lchar}
            ${PRINTF} "%2d %-${cd_lchar}.${cd_lchar}s " ${count} "${tcd}"

        cd_right_trunc "${CDS[count]}" ${cd_rchar}
            ${PRINTF} "S%d %-${cd_rchar}.${cd_rchar}s\n" ${count} "${tcd}"
        count=${count}-1
    done
}

cd_dohistoryH ()
{
    cd_getrc
        ${PRINTF} "History:\n"
        local -i count=${cd_maxhistory}
        while ${TEST} ${count} -ge 0
        do
          ${PRINTF} "${count} %-${cd_flchar}.${cd_flchar}s\n" ${CD[$count]}
          count=${count}-1
        done
}

cd_dohistoryS ()
{
    cd_getrc
        ${PRINTF} "Specials:\n"
        local -i count=${cd_maxspecial}
        while ${TEST} ${count} -ge 0
        do
          ${PRINTF} "S${count} %-${cd_flchar}.${cd_flchar}s\n" ${CDS[$count]}
          count=${count}-1
        done
}

cd_getrc ()
{
    cd_flchar=$(stty -a | awk -F \;
    '/rows/ { print $2 $3 }' | awk -F \  '{ print $4 }')
    if ${TEST} ${cd_flchar} -ne 0
    then
        cd_lchar=${cd_flchar}/2-5
        cd_rchar=${cd_flchar}/2-5
            cd_flchar=${cd_flchar}-5
    else
            cd_flchar=${FLCHAR:=75}
            # cd_flchar is used for for the @s & @h history
            cd_lchar=${LCHAR:=35}
```

```
            cd_rchar=${RCHAR:=35}
    fi
}

cd_doselection ()
{
        local -i nm=0
        cd_doflag="TRUE"
        if ${TEST} "${CD_MODE}" = "PREV"
        then
                if ${TEST} -z "$cd_npwd"
                then
                        cd_npwd=0
                fi
        fi
        tm=$(echo "${cd_npwd}" | cut -b 1)
    if ${TEST} "${tm}" = "-"
    then
        pm=$(echo "${cd_npwd}" | cut -b 2)
        nm=$(echo "${cd_npwd}" | cut -d $pm -f2)
        case "${pm}" in
             a) cd_npwd=${cd_sugg[$nm]} ;;
             s) cd_npwd="${CDS[$nm]}" ;;
             S) cd_npwd="${CDS[$nm]}" ; CDS[$nm]=`pwd` ;;
             r) cd_npwd="$2" ; cd_specDir=$nm ; cd_doselection "$1" "$2";;
             R) cd_npwd="$2" ; CDS[$nm]=`pwd` ; cd_doselection "$1" "$2";;
        esac
    fi

    if ${TEST} "${cd_npwd}" != "." -a "${cd_npwd}" \
!= ".." -a "${cd_npwd}" -le ${cd_maxhistory} >>/dev/null 2>&1
    then
      cd_npwd=${CD[$cd_npwd]}
     else
       case "$cd_npwd" in
                @)  cd_dohistory ; cd_doflag="FALSE" ;;
               @h) cd_dohistoryH ; cd_doflag="FALSE" ;;
               @s) cd_dohistoryS ; cd_doflag="FALSE" ;;
               -h) cd_hm ; cd_doflag="FALSE" ;;
               -H) cd_Hm ; cd_doflag="FALSE" ;;
               -f) cd_fsave "SHOW" $2 ; cd_doflag="FALSE" ;;
               -u) cd_upload "SHOW" $2 ; cd_doflag="FALSE" ;;
               -F) cd_fsave "NOSHOW" $2 ; cd_doflag="FALSE" ;;
               -U) cd_upload "NOSHOW" $2 ; cd_doflag="FALSE" ;;
               -g) cd_npwd="$2" ;;
               -d) cd_chdefm 1; cd_doflag="FALSE" ;;
               -D) cd_chdefm 0; cd_doflag="FALSE" ;;
               -r) cd_npwd="$2" ; cd_specDir=0 ; cd_doselection "$1" "$2";;
               -R) cd_npwd="$2" ; CDS[0]=`pwd` ; cd_doselection "$1" "$2";;
               -s) cd_npwd="${CDS[0]}" ;;
               -S) cd_npwd="${CDS[0]}"  ; CDS[0]=`pwd` ;;
               -v) cd_version ; cd_doflag="FALSE";;
       esac
    fi
}

cd_chdefm ()
{
        if ${TEST} "${CD_MODE}" = "PREV"
        then
                CD_MODE=""
                if ${TEST} $1 -eq 1
```

```
                then
                        ${PRINTF} "${cd_mset}"
                fi
        else
                CD_MODE="PREV"
                if ${TEST} $1 -eq 1
                then
                        ${PRINTF} "${cd_mnset}"
                fi
        fi
}

cd_fsave ()
{
        local sfile=${CDPath}${2:-"$CDFile"}
        if ${TEST} "$1" = "SHOW"
        then
                ${PRINTF} "Saved to %s\n" $sfile
        fi
        ${RM} -f ${sfile}
        local -i count=0
        while ${TEST} ${count} -le ${cd_maxhistory}
        do
                echo "CD[$count]=\"${CD[$count]}\"" >> ${sfile}
                count=${count}+1
        done
        count=0
        while ${TEST} ${count} -le ${cd_maxspecial}
        do
                echo "CDS[$count]=\"${CDS[$count]}\"" >> ${sfile}
                count=${count}+1
        done
}

cd_upload ()
{
        local sfile=${CDPath}${2:-"$CDFile"}
        if ${TEST} "${1}" = "SHOW"
        then
                ${PRINTF} "Loading from %s\n" ${sfile}
        fi
        . ${sfile}
}

cd_new ()
{
    local -i count
    local -i choose=0

        cd_npwd="${1}"
        cd_specDir=-1
        cd_doselection "${1}" "${2}"

        if ${TEST} ${cd_doflag} = "TRUE"
        then
                if ${TEST} "${CD[0]}" != "`pwd`"
                then
                        count=$cd_maxhistory
                        while ${TEST} $count -gt 0
                        do
                                CD[$count]=${CD[$count-1]}
                                count=${count}-1
```

```
                        done
                        CD[0]=`pwd`
                fi
                command cd "${cd_npwd}" 2>/dev/null
        if ${TEST} $? -eq 1
        then
            ${PRINTF} "Unknown dir: %s\n" "${cd_npwd}"
            local -i ftflag=0
            for i in "${cd_npwd}"*
            do
                if ${TEST} -d "${i}"
                then
                    if ${TEST} ${ftflag} -eq 0
                    then
                        ${PRINTF} "Suggest:\n"
                        ftflag=1
                fi
                    ${PRINTF} "\t-a${choose} %s\n" "$i"
                                    cd_sugg[$choose]="${i}"
                    choose=${choose}+1
        fi
            done
        fi
        fi

        if ${TEST} ${cd_specDir} -ne -1
        then
                CDS[${cd_specDir}]=`pwd`
        fi

        if ${TEST} ! -z "${CDL_PROMPTLEN}"
        then
        cd_right_trunc "${PWD}" ${CDL_PROMPTLEN}
            cd_rp=${CDL_PROMPT_PRE}${tcd}${CDL_PROMPT_POST}
                export PS1="$(echo -ne ${cd_rp})"
        fi
}
#########################################################################
#                                                                       #
#                        Initialisation here                            #
#                                                                       #
#########################################################################
#
VERSION_MAJOR="1"
VERSION_MINOR="2.1"
VERSION_DATE="24-MAY-2003"
#
alias cd=cd_new
#
# Set up commands
RM=/bin/rm
TEST=test
PRINTF=printf              # Use builtin printf

#########################################################################
#                                                                       #
# Change this to modify the default pre- and post prompt strings.       #
# These only come into effect if CDL_PROMPTLEN is set.                  #
#                                                                       #
#########################################################################
if ${TEST} ${EUID} -eq 0
then
```

```
#   CDL_PROMPT_PRE=${CDL_PROMPT_PRE:="$HOSTNAME@"}
    CDL_PROMPT_PRE=${CDL_PROMPT_PRE:="\\[\\e[01;31m\\]"}  # Root is in red
    CDL_PROMPT_POST=${CDL_PROMPT_POST:="\\[\\e[00m\\]#"}
else
    CDL_PROMPT_PRE=${CDL_PROMPT_PRE:="\\[\\e[01;34m\\]"}  # Users in blue
    CDL_PROMPT_POST=${CDL_PROMPT_POST:="\\[\\e[00m\\]$"}
fi
########################################################################
#
# cd_maxhistory defines the max number of history entries allowed.
typeset -i cd_maxhistory=50

########################################################################
#
# cd_maxspecial defines the number of special entries.
typeset -i cd_maxspecial=9
#
#
########################################################################
#
#  cd_histcount defines the number of entries displayed in
#+ the history command.
typeset -i cd_histcount=9
#
########################################################################
export CDPath=${HOME}/
#  Change these to use a different                                     #
#+ default path and filename                                           #
export CDFile=${CDFILE:=cdfile}           # for the -u and -f commands  #
#
########################################################################
                                                                       #
typeset -i cd_lchar cd_rchar cd_flchar
                        #  This is the number of chars to allow for the #
cd_flchar=${FLCHAR:=75} #+ cd_flchar is used for for the @s & @h history#

typeset -ax CD CDS
#
cd_mset="\n\tDefault mode is now set - entering cd with no parameters \
has the default action\n\tUse cd -d or -D for cd to go to \
previous directory with no parameters\n"
cd_mnset="\n\tNon-default mode is now set - entering cd with no \
parameters is the same as entering cd 0\n\tUse cd -d or \
-D to change default cd action\n"

# ==================================================================== #



: <<DOCUMENTATION

Written by Phil Braham. Realtime Software Pty Ltd.
Released under GNU license. Free to use. Please pass any modifications
or comments to the author Phil Braham:

realtime@mpx.com.au
=======================================================================

cdll is a replacement for cd and incorporates similar functionality to
the bash pushd and popd commands but is independent of them.

This version of cdll has been tested on Linux using Bash. It will work
```

```
on most Linux versions but will probably not work on other shells without
modification.

Introduction
============

cdll allows easy moving about between directories. When changing to a new
directory the current one is automatically put onto a stack. By default
50 entries are kept, but this is configurable. Special directories can be
kept for easy access - by default up to 10, but this is configurable. The
most recent stack entries and the special entries can be easily viewed.

The directory stack and special entries can be saved to, and loaded from,
a file. This allows them to be set up on login, saved before logging out
or changed when moving project to project.

In addition, cdll provides a flexible command prompt facility that allows,
for example, a directory name in colour that is truncated from the left
if it gets too long.


Setting up cdll
===============

Copy cdll to either your local home directory or a central directory
such as /usr/bin (this will require root access).

Copy the file cdfile to your home directory. It will require read and
write access. This a default file that contains a directory stack and
special entries.

To replace the cd command you must add commands to your login script.
The login script is one or more of:

    /etc/profile
    ~/.bash_profile
    ~/.bash_login
    ~/.profile
    ~/.bashrc
    /etc/bash.bashrc.local

To setup your login, ~/.bashrc is recommended, for global (and root) setup
add the commands to /etc/bash.bashrc.local

To set up on login, add the command:
    . <dir>/cdll
For example if cdll is in your local home directory:
    . ~/cdll
If in /usr/bin then:
    . /usr/bin/cdll

If you want to use this instead of the buitin cd command then add:
    alias cd='cd_new'
We would also recommend the following commands:
    alias @='cd_new @'
    cd -U
    cd -D

If you want to use cdll's prompt facilty then add the following:
    CDL_PROMPTLEN=nn
Where nn is a number described below. Initially 99 would be suitable
number.
```

```
Thus the script looks something like this:

    ######################################################################
    # CD Setup
    ######################################################################
    CDL_PROMPTLEN=21        # Allow a prompt length of up to 21 characters
    . /usr/bin/cdll         # Initialise cdll
    alias cd='cd_new'       # Replace the built in cd command
    alias @='cd_new @'      # Allow @ at the prompt to display history
    cd -U                   # Upload directories
    cd -D                   # Set default action to non-posix
    ######################################################################

The full meaning of these commands will become clear later.

There are a couple of caveats. If another program changes the directory
without calling cdll, then the directory won't be put on the stack and
also if the prompt facility is used then this will not be updated. Two
programs that can do this are pushd and popd. To update the prompt and
stack simply enter:

    cd .

Note that if the previous entry on the stack is the current directory
then the stack is not updated.

Usage
=====
cd [dir] [0-9] [@[s|h] [-g <dir>] [-d] [-D] [-r<n>]
   [dir|0-9] [-R<n>] [<dir>|0-9] [-s<n>] [-S<n>]
   [-u] [-U] [-f] [-F] [-h] [-H] [-v]

    <dir>       Go to directory
    0-n         Goto previous directory (0 is previous,
                1 is last but 1, etc.)
                n is up to max history (default is 50)
    @           List history and special entries (Usually available as $ @)
    @h          List history entries
    @s          List special entries
    -g [<dir>]  Go to literal name (bypass special names)
                This is to allow access to dirs called '0','1','-h' etc
    -d          Change default action - verbose. (See note)
    -D          Change default action - silent. (See note)
    -s<n>       Go to the special entry <n>
    -S<n>       Go to the special entry <n>
                      and replace it with the current dir
    -r<n> [<dir>] Go to directory <dir>
                              and then put it on special entry <n>
    -R<n> [<dir>] Go to directory <dir>
                              and put current dir on special entry <n>
    -a<n>       Alternative suggested directory. See note below.
    -f [<file>] File entries to <file>.
    -u [<file>] Update entries from <file>.
                If no filename supplied then default file (~/cdfile) is used
                -F and -U are silent versions
    -v          Print version number
    -h          Help
    -H          Detailed help
```

```
Examples
========

These examples assume non-default mode is set (that is, cd with no
parameters will go to the most recent stack directory), that aliases
have been set up for cd and @ as described above and that cd's prompt
facility is active and the prompt length is 21 characters.

    /home/phil$ @
    # List the entries with the @
    History:
    # Output of the @ command
    .....
    # Skipped these entries for brevity
    1 /home/phil/ummdev               S1 /home/phil/perl
    # Most recent two history entries
    0 /home/phil/perl/eg              S0 /home/phil/umm/ummdev
    # and two special entries are shown

    /home/phil$ cd /home/phil/utils/Cdll
    # Now change directories
    /home/phil/utils/Cdll$ @
    # Prompt reflects the directory.
    History:
    # New history
    .....
    1 /home/phil/perl/eg              S1 /home/phil/perl
    # History entry 0 has moved to 1
    0 /home/phil                      S0 /home/phil/umm/ummdev
    # and the most recent has entered

To go to a history entry:

    /home/phil/utils/Cdll$ cd 1
    # Go to history entry 1.
    /home/phil/perl/eg$
    # Current directory is now what was 1

To go to a special entry:

    /home/phil/perl/eg$ cd -s1
    # Go to special entry 1
    /home/phil/umm/ummdev$
    # Current directory is S1

To go to a directory called, for example, 1:

    /home/phil$ cd -g 1
    # -g ignores the special meaning of 1
    /home/phil/1$

To put current directory on the special list as S1:
    cd -r1 .        #  OR
    cd -R1 .        #  These have the same effect if the directory is
                    #+ . (the current directory)

To go to a directory and add it as a special
  The directory for -r<n> or -R<n> may be a number.
  For example:
        $ cd -r3 4  Go to history entry 4 and put it on special entry 3
        $ cd -R3 4  Put current dir on the special entry 3 and go to
                    history entry 4
```

```
        $ cd -s3    Go to special entry 3

    Note that commands R,r,S and s may be used without a number and
    refer to 0:
        $ cd -s     Go to special entry 0
        $ cd -S     Go to special entry 0 and make special entry 0
                    current dir
        $ cd -r 1   Go to history entry 1 and put it on special entry 0
        $ cd -r     Go to history entry 0 and put it on special entry 0


    Alternative suggested directories:

    If a directory is not found, then CD will suggest any
    possibilities. These are directories starting with the same letters
    and if any are found they are listed prefixed with -a<n>
    where <n> is a number. It's possible to go to the directory
    by entering cd -a<n> on the command line.

        Use cd -d or -D to change default cd action. cd -H will show
        current action.

        The history entries (0-n) are stored in the environment variables
        CD[0] - CD[n]
        Similarly the special directories S0 - 9 are in the environment
        variable CDS[0] - CDS[9]
        and may be accessed from the command line, for example:

            ls -l ${CDS[3]}
            cat ${CD[8]}/file.txt

        The default pathname for the -f and -u commands is ~
        The default filename for the -f and -u commands is cdfile


Configuration
=============

    The following environment variables can be set:

        CDL_PROMPTLEN  - Set to the length of prompt you require.
            Prompt string is set to the right characters of the current
            directory. If not set, then prompt is left unchanged. Note
            that this is the number of characters that the directory is
            shortened to, not the total characters in the prompt.

            CDL_PROMPT_PRE - Set to the string to prefix the prompt.
                Default is:
                    non-root:  "\\[\\e[01;34m\\]"  (sets colour to blue).
                    root:      "\\[\\e[01;31m\\]"  (sets colour to red).

            CDL_PROMPT_POST    - Set to the string to suffix the prompt.
                Default is:
                    non-root:  "\\[\\e[00m\\]$"
                               (resets colour and displays $).
                    root:      "\\[\\e[00m\\]#"
                               (resets colour and displays #).

        Note:
            CDL_PROMPT_PRE & _POST only t

        CDPath - Set the default path for the -f & -u options.
```

```
                 Default is home directory
        CDFile - Set the default filename for the -f & -u options.
                 Default is cdfile


    There are three variables defined in the file cdll which control the
    number of entries stored or displayed. They are in the section labeled
    'Initialisation here' towards the end of the file.

        cd_maxhistory       - The number of history entries stored.
                              Default is 50.
        cd_maxspecial       - The number of special entries allowed.
                              Default is 9.
        cd_histcount        - The number of history and special entries
                              displayed. Default is 9.

    Note that cd_maxspecial should be >= cd_histcount to avoid displaying
    special entries that can't be set.


Version: 1.2.1 Date: 24-MAY-2003

DOCUMENTATION
```

Example A-34. A soundcard setup script

```
#!/bin/bash
# soundcard-on.sh

#  Script author: Mkarcher
#  http://www.thinkwiki.org/wiki  ...
#  /Script_for_configuring_the_CS4239_sound_chip_in_PnP_mode
#  ABS Guide author made minor changes and added comments.
#  Couldn't contact script author to ask for permission to use, but ...
#+ the script was released under the FDL,
#+ so its use here should be both legal and ethical.

#  Sound-via-pnp-script for Thinkpad 600E
#+ and possibly other computers with onboard CS4239/CS4610
#+ that do not work with the PCI driver
#+ and are not recognized by the PnP code of snd-cs4236.
#  Also for some 770-series Thinkpads, such as the 770x.
#  Run as root user, of course.
#
#  These are old and very obsolete laptop computers,
#+ but this particular script is very instructive,
#+ as it shows how to set up and hack device files.



#  Search for sound card pnp device:

for dev in /sys/bus/pnp/devices/*
do
  grep CSC0100 $dev/id > /dev/null && WSSDEV=$dev
  grep CSC0110 $dev/id > /dev/null && CTLDEV=$dev
done
# On 770x:
# WSSDEV = /sys/bus/pnp/devices/00:07
# CTLDEV = /sys/bus/pnp/devices/00:06
# These are symbolic links to /sys/devices/pnp0/ ...
```

```
#  Activate devices:
#  Thinkpad boots with devices disabled unless "fast boot" is turned off
#+ (in BIOS).

echo activate > $WSSDEV/resources
echo activate > $CTLDEV/resources


# Parse resource settings.

{ read # Discard "state = active" (see below).
  read bla port1
  read bla port2
  read bla port3
  read bla irq
  read bla dma1
  read bla dma2
 # The "bla's" are labels in the first field: "io," "state," etc.
 # These are discarded.

 #  Hack: with PnPBIOS: ports are: port1: WSS, port2:
 #+ OPL, port3: sb (unneeded)
 #       with ACPI-PnP:ports are: port1: OPL, port2: sb, port3: WSS
 #  (ACPI bios seems to be wrong here, the PnP-card-code in snd-cs4236.c
 #+  uses the PnPBIOS port order)
 #  Detect port order using the fixed OPL port as reference.
  if [ ${port2%%-*} = 0x388 ]
 #             ^^^^  Strip out everything following hyphen in port address.
 #                   So, if port1 is 0x530-0x537
 #+                  we're left with 0x530 -- the start address of the port.
 then
   # PnPBIOS: usual order
   port=${port1%%-*}
   oplport=${port2%%-*}
 else
   # ACPI: mixed-up order
   port=${port3%%-*}
   oplport=${port1%%-*}
 fi
 } < $WSSDEV/resources
# To see what's going on here:
# ---------------------------
#   cat /sys/devices/pnp0/00:07/resources
#
#   state = active
#   io 0x530-0x537
#   io 0x388-0x38b
#   io 0x220-0x233
#   irq 5
#   dma 1
#   dma 0
#   ^^^   "bla" labels in first field (discarded).


{ read # Discard first line, as above.
  read bla port1
  cport=${port1%%-*}
  #            ^^^^
  # Just want _start_ address of port.
} < $CTLDEV/resources
```

```
# Load the module:

modprobe --ignore-install snd-cs4236 port=$port cport=$cport\
fm_port=$oplport irq=$irq dma1=$dma1 dma2=$dma2 isapnp=0 index=0
# See the modprobe manpage.

exit $?
```

Example A-35. Locating split paragraphs in a text file

```
#!/bin/bash
# find-splitpara.sh
#  Finds split paragraphs in a text file,
#+ and tags the line numbers.


ARGCOUNT=1       # Expect one arg.
OFF=0            # Flag states.
ON=1
E_WRONGARGS=85

file="$1"        # Target filename.
lineno=1         # Line number. Start at 1.
Flag=$OFF        # Blank line flag.

if [ $# -ne "$ARGCOUNT" ]
then
  echo "Usage: `basename $0` FILENAME"
  exit $E_WRONGARGS
fi

file_read ()     # Scan file for pattern, then print line.
{
while read line
do

  if [[ "$line" =~ ^[a-z] && $Flag -eq $ON ]]
     then  # Line begins with lowercase character, following blank line.
     echo -n "$lineno::   "
     echo "$line"
  fi


  if [[ "$line" =~ ^$ ]]
     then       #  If blank line,
     Flag=$ON   #+ set flag.
  else
     Flag=$OFF
  fi

  ((lineno++))

done
} < $file  # Redirect file into function's stdin.

file_read
```

```
exit $?


# ---------------------------------------------------------------
This is line one of an example paragraph, bla, bla, bla.
This is line two, and line three should follow on next line, but

there is a blank line separating the two parts of the paragraph.
# ---------------------------------------------------------------

Running this script on a file containing the above paragraph
yields:

4::   there is a blank line separating the two parts of the paragraph.


There will be additional output for all the other split paragraphs
in the target file.
```

Example A-36. Insertion sort

```
#!/bin/bash
# insertion-sort.bash: Insertion sort implementation in Bash
#                      Heavy use of Bash array features:
#+                     (string) slicing, merging, etc
# URL: http://www.lugmen.org.ar/~jjo/jjotip/insertion-sort.bash.d
#+          /insertion-sort.bash.sh
#
# Author: JuanJo Ciarlante <jjo@irrigacion.gov.ar>
# Lightly reformatted by ABS Guide author.
# License: GPLv2
# Used in ABS Guide with author's permission (thanks!).
#
# Test with:   ./insertion-sort.bash -t
# Or:          bash insertion-sort.bash -t
# The following *doesn't* work:
#              sh insertion-sort.bash -t
#  Why not? Hint: which Bash-specific features are disabled
#+ when running a script by 'sh script.sh'?
#
: ${DEBUG:=0}  # Debug, override with:  DEBUG=1 ./scriptname . . .
# Parameter substitution -- set DEBUG to 0 if not previously set.

# Global array: "list"
typeset -a list
# Load whitespace-separated numbers from stdin.
if [ "$1" = "-t" ]; then
DEBUG=1
        read -a list < <( od -Ad -w24 -t u2 /dev/urandom ) # Random list.
#                    ^ ^  process substition
else
        read -a list
fi
numelem=${#list[*]}

#  Shows the list, marking the element whose index is $1
#+ by surrounding it with the two chars passed as $2.
#  Whole line prefixed with $3.
showlist()
  {
```

```
  echo "$3"${list[@]:0:$1} ${2:0:1}${list[$1]}${2:1:1} ${list[@]:$1+1};
  }

# Loop _pivot_ -- from second element to end of list.
for(( i=1; i<numelem; i++ )) do
        ((DEBUG))&&showlist i "[]" " "
        # From current _pivot_, back to first element.
        for(( j=i; j; j-- )) do
                # Search for the 1st elem. less than current "pivot" . . .
                [[ "${list[j-1]}" -le "${list[i]}" ]] && break
        done
        (( i==j )) && continue ## No insertion was needed for this element.
        # . . . Move list[i] (pivot) to the left of list[j]:
        list=(${list[@]:0:j} ${list[i]} ${list[j]}\
        #         {0,j-1}        {i}       {j}
              ${list[@]:j+1:i-(j+1)} ${list[@]:i+1})
        #         {j+1,i-1}              {i+1,last}
        ((DEBUG))&&showlist j "<>" "*"
done


echo
echo  "------"
echo $'Result:\n'${list[@]}

exit $?
```

Example A-37. Standard Deviation

```
#!/bin/bash
# sd.sh: Standard Deviation

#  The Standard Deviation indicates how consistent a set of data is.
#  It shows to what extent the individual data points deviate from the
#+ arithmetic mean, i.e., how much they "bounce around" (or cluster).
#  It is essentially the average deviation-distance of the
#+ data points from the mean.

# =========================================================== #
#    To calculate the Standard Deviation:
#
# 1  Find the arithmetic mean (average) of all the data points.
# 2  Subtract each data point from the arithmetic mean,
#    and square that difference.
# 3  Add all of the individual difference-squares in # 2.
# 4  Divide the sum in # 3 by the number of data points.
#    This is known as the "variance."
# 5  The square root of # 4 gives the Standard Deviation.
# =========================================================== #

count=0         # Number of data points; global.
SC=9            # Scale to be used by bc. Nine decimal places.
E_DATAFILE=90   # Data file error.

# ----------------- Set data file ---------------------
if [ ! -z "$1" ]  # Specify filename as cmd-line arg?
then
  datafile="$1" #  ASCII text file,
else            #+ one (numerical) data point per line!
  datafile=sample.dat
fi              #  See example data file, below.
```

```
if [ ! -e "$datafile" ]
then
  echo "\""$datafile"\" does not exist!"
  exit $E_DATAFILE
fi
# -----------------------------------------------------


arith_mean ()
{
  local rt=0         # Running total.
  local am=0         # Arithmetic mean.
  local ct=0         # Number of data points.

  while read value   # Read one data point at a time.
  do
    rt=$(echo "scale=$SC; $rt + $value" | bc)
    (( ct++ ))
  done

  am=$(echo "scale=$SC; $rt / $ct" | bc)

  echo $am; return $ct   # This function "returns" TWO values!
  #  Caution: This little trick will not work if $ct > 255!
  #  To handle a larger number of data points,
  #+ simply comment out the "return $ct" above.
} <"$datafile"   # Feed in data file.

sd ()
{
  mean1=$1  # Arithmetic mean (passed to function).
  n=$2      # How many data points.
  sum2=0    # Sum of squared differences ("variance").
  avg2=0    # Average of $sum2.
  sdev=0    # Standard Deviation.

  while read value   # Read one line at a time.
  do
    diff=$(echo "scale=$SC; $mean1 - $value" | bc)
    # Difference between arith. mean and data point.
    dif2=$(echo "scale=$SC; $diff * $diff" | bc) # Squared.
    sum2=$(echo "scale=$SC; $sum2 + $dif2" | bc) # Sum of squares.
  done

    avg2=$(echo "scale=$SC; $sum2 / $n" | bc)  # Avg. of sum of squares.
    sdev=$(echo "scale=$SC; sqrt($avg2)" | bc) # Square root =
    echo $sdev                                 # Standard Deviation.

} <"$datafile"   # Rewinds data file.


# ======================================================= #
mean=$(arith_mean); count=$?   # Two returns from function!
std_dev=$(sd $mean $count)

echo
echo "Number of data points in \""$datafile"\" = $count"
echo "Arithmetic mean (average) = $mean"
echo "Standard Deviation = $std_dev"
echo
# ======================================================= #
```

```
exit

#  This script could stand some drastic streamlining,
#+ but not at the cost of reduced legibility, please.


# ++++++++++++++++++++++++++++++++++++++++ #
# A sample data file (sample1.dat):

# 18.35
# 19.0
# 18.88
# 18.91
# 18.64


# $ sh sd.sh sample1.dat

# Number of data points in "sample1.dat" = 5
# Arithmetic mean (average) = 18.756000000
# Standard Deviation = .235338054
# ++++++++++++++++++++++++++++++++++++++++ #
```

Example A-38. A *pad* file generator for shareware authors

```
#!/bin/bash
# pad.sh

#######################################################
#               PAD (xml) file creator
#+ Written by Mendel Cooper <thegrendel.abs@gmail.com>.
#+ Released to the Public Domain.
#
#  Generates a "PAD" descriptor file for shareware
#+ packages, according to the specifications
#+ of the ASP.
#  http://www.asp-shareware.org/pad
#######################################################


# Accepts (optional) save filename as a command-line argument.
if [ -n "$1" ]
then
  savefile=$1
else
  savefile=save_file.xml               # Default save_file name.
fi


# ===== PAD file headers =====
HDR1="<?xml version=\"1.0\" encoding=\"Windows-1252\" ?>"
HDR2="<XML_DIZ_INFO>"
HDR3="<MASTER_PAD_VERSION_INFO>"
HDR4="\t<MASTER_PAD_VERSION>1.15</MASTER_PAD_VERSION>"
HDR5="\t<MASTER_PAD_INFO>Portable Application Description, or PAD
for short, is a data set that is used by shareware authors to
disseminate information to anyone interested in their software products.
To find out more go to http://www.asp-shareware.org/pad</MASTER_PAD_INFO>"
HDR6="</MASTER_PAD_VERSION_INFO>"
# ============================
```

```
fill_in ()
{
  if [ -z "$2" ]
  then
    echo -n "$1? "     # Get user input.
  else
    echo -n "$1 $2? "  # Additional query?
  fi

  read var             # May paste to fill in field.
                       # This shows how flexible "read" can be.

  if [ -z "$var" ]
  then
    echo -e "\t\t<$1 />" >>$savefile    # Indent with 2 tabs.
    return
  else
    echo -e "\t\t<$1>$var</$1>" >>$savefile
    return ${#var}     # Return length of input string.
  fi
}

check_field_length ()  # Check length of program description fields.
{
  # $1 = maximum field length
  # $2 = actual field length
  if [ "$2" -gt "$1" ]
  then
    echo "Warning: Maximum field length of $1 characters exceeded!"
  fi
}

clear                  # Clear screen.
echo "PAD File Creator"
echo "--- ---- -------"
echo

# Write File Headers to file.
echo $HDR1 >$savefile
echo $HDR2 >>$savefile
echo $HDR3 >>$savefile
echo -e $HDR4 >>$savefile
echo -e $HDR5 >>$savefile
echo $HDR6 >>$savefile


# Company_Info
echo "COMPANY INFO"
CO_HDR="Company_Info"
echo "<$CO_HDR>" >>$savefile

fill_in Company_Name
fill_in Address_1
fill_in Address_2
fill_in City_Town
fill_in State_Province
fill_in Zip_Postal_Code
fill_in Country

# If applicable:
```

```
# fill_in ASP_Member "[Y/N]"
# fill_in ASP_Member_Number
# fill_in ESC_Member "[Y/N]"

fill_in Company_WebSite_URL

clear   # Clear screen between sections.

   # Contact_Info
echo "CONTACT INFO"
CONTACT_HDR="Contact_Info"
echo "<$CONTACT_HDR>" >>$savefile
fill_in Author_First_Name
fill_in Author_Last_Name
fill_in Author_Email
fill_in Contact_First_Name
fill_in Contact_Last_Name
fill_in Contact_Email
echo -e "\t</$CONTACT_HDR>" >>$savefile
   # END Contact_Info

clear

   # Support_Info
echo "SUPPORT INFO"
SUPPORT_HDR="Support_Info"
echo "<$SUPPORT_HDR>" >>$savefile
fill_in Sales_Email
fill_in Support_Email
fill_in General_Email
fill_in Sales_Phone
fill_in Support_Phone
fill_in General_Phone
fill_in Fax_Phone
echo -e "\t</$SUPPORT_HDR>" >>$savefile
   # END Support_Info

echo "</$CO_HDR>" >>$savefile
# END Company_Info

clear

# Program_Info
echo "PROGRAM INFO"
PROGRAM_HDR="Program_Info"
echo "<$PROGRAM_HDR>" >>$savefile
fill_in Program_Name
fill_in Program_Version
fill_in Program_Release_Month
fill_in Program_Release_Day
fill_in Program_Release_Year
fill_in Program_Cost_Dollars
fill_in Program_Cost_Other
fill_in Program_Type "[Shareware/Freeware/GPL]"
fill_in Program_Release_Status "[Beta, Major Upgrade, etc.]"
fill_in Program_Install_Support
fill_in Program_OS_Support "[Win9x/Win2k/Linux/etc.]"
fill_in Program_Language "[English/Spanish/etc.]"

echo; echo

  # File_Info
```

```
echo "FILE INFO"
FILEINFO_HDR="File_Info"
echo "<$FILEINFO_HDR>" >>$savefile
fill_in Filename_Versioned
fill_in Filename_Previous
fill_in Filename_Generic
fill_in Filename_Long
fill_in File_Size_Bytes
fill_in File_Size_K
fill_in File_Size_MB
echo -e "\t</$FILEINFO_HDR>" >>$savefile
  # END File_Info

clear

  # Expire_Info
echo "EXPIRE INFO"
EXPIRE_HDR="Expire_Info"
echo "<$EXPIRE_HDR>" >>$savefile
fill_in Has_Expire_Info "Y/N"
fill_in Expire_Count
fill_in Expire_Based_On
fill_in Expire_Other_Info
fill_in Expire_Month
fill_in Expire_Day
fill_in Expire_Year
echo -e "\t</$EXPIRE_HDR>" >>$savefile
  # END Expire_Info

clear

  # More Program_Info
echo "ADDITIONAL PROGRAM INFO"
fill_in Program_Change_Info
fill_in Program_Specific_Category
fill_in Program_Categories
fill_in Includes_JAVA_VM "[Y/N]"
fill_in Includes_VB_Runtime "[Y/N]"
fill_in Includes_DirectX "[Y/N]"
  # END More Program_Info

echo "</$PROGRAM_HDR>" >>$savefile
# END Program_Info

clear

# Program Description
echo "PROGRAM DESCRIPTIONS"
PROGDESC_HDR="Program_Descriptions"
echo "<$PROGDESC_HDR>" >>$savefile

LANG="English"
echo "<$LANG>" >>$savefile

fill_in Keywords "[comma + space separated]"
echo
echo "45, 80, 250, 450, 2000 word program descriptions"
echo "(may cut and paste into field)"
#  It would be highly appropriate to compose the following
#+ "Char_Desc" fields with a text editor,
#+ then cut-and-paste the text into the answer fields.
echo
```

```
echo "              |---------------45 characters---------------|"
fill_in Char_Desc_45
check_field_length 45 "$?"
echo
fill_in Char_Desc_80
check_field_length 80 "$?"

fill_in Char_Desc_250
check_field_length 250 "$?"

fill_in Char_Desc_450
fill_in Char_Desc_2000

echo "</$LANG>" >>$savefile
echo "</$PROGDESC_HDR>" >>$savefile
# END Program Description

clear
echo "Done."; echo; echo
echo "Save file is:  \""$savefile"\""

exit 0
```

Example A-39. A *man page* editor

```
#!/bin/bash
# maned.sh
# A rudimentary man page editor

# Version: 0.1 (Alpha, probably buggy)
# Author: Mendel Cooper <thegrendel.abs@gmail.com>
# Reldate: 16 June 2008
# License: GPL3


savefile=      # Global, used in multiple functions.
E_NOINPUT=90   # User input missing (error). May or may not be critical.

# =========== Markup Tags ============ #
TopHeader=".TH"
NameHeader=".SH NAME"
SyntaxHeader=".SH SYNTAX"
SynopsisHeader=".SH SYNOPSIS"
InstallationHeader=".SH INSTALLATION"
DescHeader=".SH DESCRIPTION"
OptHeader=".SH OPTIONS"
FilesHeader=".SH FILES"
EnvHeader=".SH ENVIRONMENT"
AuthHeader=".SH AUTHOR"
BugsHeader=".SH BUGS"
SeeAlsoHeader=".SH SEE ALSO"
BOLD=".B"
# Add more tags, as needed.
# See groff docs for markup meanings.
# ==================================== #

start ()
{
clear                  # Clear screen.
echo "ManEd"
echo "-----"
```

```
echo
echo "Simple man page creator"
echo "Author: Mendel Cooper"
echo "License: GPL3"
echo; echo; echo
}

progname ()
{
  echo -n "Program name? "
  read name

  echo -n "Manpage section? [Hit RETURN for default (\"1\") ]  "
  read section
  if [ -z "$section" ]
  then
    section=1   # Most man pages are in section 1.
  fi

  if [ -n "$name" ]
  then
    savefile=""$name"."$section""       #  Filename suffix = section.
    echo -n "$1 " >>$savefile
    name1=$(echo "$name" | tr a-z A-Z)  #  Change to uppercase,
                                        #+ per man page convention.
    echo -n "$name1" >>$savefile
  else
    echo "Error! No input."             # Mandatory input.
    exit $E_NOINPUT                     # Critical!
    #  Exercise: The script-abort if no filename input is a bit clumsy.
    #            Rewrite this section so a default filename is used
    #+           if no input.
  fi

  echo -n "  \"$section\"">>$savefile   # Append, always append.

  echo -n "Version? "
  read ver
  echo -n " \"Version $ver \"">>$savefile
  echo >>$savefile

  echo -n "Short description [0 - 5 words]? "
  read sdesc
  echo "$NameHeader">>$savefile
  echo ""$BOLD" "$name"">>$savefile
  echo "\- "$sdesc"">>$savefile

}

fill_in ()
{ # This function more or less copied from "pad.sh" script.
  echo -n "$2? "       # Get user input.
  read var             # May paste (a single line only!) to fill in field.

  if [ -n "$var" ]
  then
    echo "$1 " >>$savefile
    echo -n "$var" >>$savefile
  else                 # Don't append empty field to file.
    return $E_NOINPUT  # Not critical here.
  fi
```

```
  echo >>$savefile

}


end ()
{
clear
echo -n "Would you like to view the saved man page (y/n)? "
read ans
if [ "$ans" = "n" -o "$ans" = "N" ]; then exit; fi
exec less "$savefile"  #  Exit script and hand off control to "less" ...
                       #+ ... which formats for viewing man page source.
}


# ---------------------------------------- #
start
progname "$TopHeader"
fill_in "$SynopsisHeader" "Synopsis"
fill_in "$DescHeader" "Long description"
# May paste in *single line* of text.
fill_in "$OptHeader" "Options"
fill_in "$FilesHeader" "Files"
fill_in "$AuthHeader" "Author"
fill_in "$BugsHeader" "Bugs"
fill_in "$SeeAlsoHeader" "See also"
# fill_in "$OtherHeader" ... as necessary.
end    # ... exit not needed.
# ---------------------------------------- #

#  Note that the generated man page will usually
#+ require manual fine-tuning with a text editor.
#  However, it's a distinct improvement upon
#+ writing man source from scratch
#+ or even editing a blank man page template.

#  The main deficiency of the script is that it permits
#+ pasting only a single text line into the input fields.
#  This may be a long, cobbled-together line, which groff
#  will automatically wrap and hyphenate.
#  However, if you want multiple (newline-separated) paragraphs,
#+ these must be inserted by manual text editing on the
#+ script-generated man page.
#  Exercise (difficult): Fix this!

#  This script is not nearly as elaborate as the
#+ full-featured "manedit" package
#+ http://freshmeat.net/projects/manedit/
#+ but it's much easier to use.
```

Example A-40. Petals Around the Rose

```
#!/bin/bash -i
# petals.sh

#########################################################################
# Petals Around the Rose                                                #
#                                                                       #
# Version 0.1 Created by Serghey Rodin                                  #
# Version 0.2 Modded by ABS Guide Author                                #
```

```
#                                                                       #
# License: GPL3                                                         #
# Used in ABS Guide with permission.                                    #
# ##################################################################### #

hits=0      # Correct guesses.
WIN=6       # Mastered the game.
ALMOST=5    # One short of mastery.
EXIT=exit   # Give up early?

RANDOM=$$   # Seeds the random number generator from PID of script.


# Bones (ASCII graphics for dice)
bone1[1]="|         |"
bone1[2]="|       o |"
bone1[3]="|       o |"
bone1[4]="| o     o |"
bone1[5]="| o     o |"
bone1[6]="| o     o |"
bone2[1]="|    o    |"
bone2[2]="|         |"
bone2[3]="|    o    |"
bone2[4]="|         |"
bone2[5]="|    o    |"
bone2[6]="| o     o |"
bone3[1]="|         |"
bone3[2]="| o       |"
bone3[3]="| o       |"
bone3[4]="| o     o |"
bone3[5]="| o     o |"
bone3[6]="| o     o |"
bone="+---------+"



# Functions

instructions () {

  clear
  echo -n "Do you need instructions? (y/n) "; read ans
  if [ "$ans" = "y" -o "$ans" = "Y" ]; then
    clear
    echo -e '\E[34;47m'  # Blue type.

#  "cat document"
    cat <<INSTRUCTIONSZZZ
The name of the game is Petals Around the Rose,
and that name is significant.
Five dice will roll and you must guess the "answer" for each roll.
It will be zero or an even number.
After your guess, you will be told the answer for the roll, but . . .
that's ALL the information you will get.

Six consecutive correct guesses admits you to the
Fellowship of the Rose.
INSTRUCTIONSZZZ

    echo -e "\033[0m"    # Turn off blue.
    else clear
  fi
```

```
}


fortune ()
{
  RANGE=7
  FLOOR=0
  number=0
  while [ "$number" -le $FLOOR ]
  do
    number=$RANDOM
    let "number %= $RANGE"   # 1 - 6.
  done

  return $number
}



throw () { # Calculate each individual die.
  fortune; B1=$?
  fortune; B2=$?
  fortune; B3=$?
  fortune; B4=$?
  fortune; B5=$?

  calc () { # Function embedded within a function!
    case "$1" in
       3   ) rose=2;;
       5   ) rose=4;;
       *   ) rose=0;;
    esac    # Simplified algorithm.
            # Doesn't really get to the heart of the matter.
    return $rose
  }

  answer=0
  calc "$B1"; answer=$(expr $answer + $(echo $?))
  calc "$B2"; answer=$(expr $answer + $(echo $?))
  calc "$B3"; answer=$(expr $answer + $(echo $?))
  calc "$B4"; answer=$(expr $answer + $(echo $?))
  calc "$B5"; answer=$(expr $answer + $(echo $?))
}



game ()
{ # Generate graphic display of dice throw.
  throw
    echo -e "\033[1m"    # Bold.
  echo -e "\n"
  echo -e "$bone\t$bone\t$bone\t$bone\t$bone"
  echo -e \
 "${bone1[$B1]}\t${bone1[$B2]}\t${bone1[$B3]}\t${bone1[$B4]}\t${bone1[$B5]}"
  echo -e \
 "${bone2[$B1]}\t${bone2[$B2]}\t${bone2[$B3]}\t${bone2[$B4]}\t${bone2[$B5]}"
  echo -e \
 "${bone3[$B1]}\t${bone3[$B2]}\t${bone3[$B3]}\t${bone3[$B4]}\t${bone3[$B5]}"
  echo -e "$bone\t$bone\t$bone\t$bone\t$bone"
  echo -e "\n\n\t\t"
    echo -e "\033[0m"    # Turn off bold.
```

```
  echo -n "There are how many petals around the rose? "
}



# ============================================================== #

instructions

while [ "$petal" != "$EXIT" ]    # Main loop.
do
  game
  read petal
  echo "$petal" | grep [0-9] >/dev/null  # Filter response for digit.
                                         # Otherwise just roll dice again.
  if [ "$?" -eq 0 ]   # If-loop #1.
  then
    if [ "$petal" == "$answer" ]; then    # If-loop #2.
        echo -e "\nCorrect. There are $petal petals around the rose.\n"
        (( hits++ ))

        if [ "$hits" -eq "$WIN" ]; then   # If-loop #3.
          echo -e '\E[31;47m'  # Red type.
          echo -e "\033[1m"    # Bold.
          echo "You have unraveled the mystery of the Rose Petals!"
          echo "Welcome to the Fellowship of the Rose!!!"
          echo "(You are herewith sworn to secrecy.)"; echo
          echo -e "\033[0m"    # Turn off red & bold.
          break                # Exit!
        else echo "You have $hits correct so far."; echo

        if [ "$hits" -eq "$ALMOST" ]; then
          echo "Just one more gets you to the heart of the mystery!"; echo
        fi

      fi                                  # Close if-loop #3.

    else
      echo -e "\nWrong. There are $answer petals around the rose.\n"
      hits=0   # Reset number of correct guesses.
    fi                                    # Close if-loop #2.

    echo -n "Hit ENTER for the next roll, or type \"exit\" to end. "
    read
    if [ "$REPLY" = "$EXIT" ]; then exit
    fi

  fi                  # Close if-loop #1.

  clear
done                  # End of main (while) loop.

###

exit $?

# Resources:
# ---------
# 1) http://en.wikipedia.org/wiki/Petals_Around_the_Rose
#    (Wikipedia entry.)
# 2) http://www.borrett.id.au/computing/petals-bg.htm
#    (How Bill Gates coped with the Petals Around the Rose challenge.)
```

Example A-41. Quacky: a Perquackey-type word game

```
#!/bin/bash
# qky.sh

##############################################################
# QUACKEY: a somewhat simplified version of Perquackey [TM]. #
#                                                            #
# Author: Mendel Cooper  <thegrendel.abs@gmail.com>          #
# version 0.1.02      03 May, 2008                           #
# License: GPL3                                              #
##############################################################

WLIST=/usr/share/dict/word.lst
#                     ^^^^^^^^  Word list file found here.
#  ASCII word list, one word per line, UNIX format.
#  A suggested list is the script author's "yawl" word list package.
#  http://bash.deta.in/yawl-0.3.2.tar.gz
#    or
#  http://ibiblio.org/pub/Linux/libs/yawl-0.3.2.tar.gz

NONCONS=0     # Word not constructable from letter set.
CONS=1        # Constructable.
SUCCESS=0
NG=1
FAILURE=''
NULL=0        # Zero out value of letter (if found).
MINWLEN=3     # Minimum word length.
MAXCAT=5      # Maximum number of words in a given category.
PENALTY=200   # General-purpose penalty for unacceptable words.
total=
E_DUP=70      # Duplicate word error.

TIMEOUT=10    # Time for word input.

NVLET=10      # 10 letters for non-vulnerable.
VULET=13      # 13 letters for vulnerable (not yet implemented!).

declare -a Words
declare -a Status
declare -a Score=( 0 0 0 0 0 0 0 0 0 0 0 )


letters=( a n s r t m l k p r b c i d s i d z e w u e t f
e y e r e f e g t g h h i t r s c i t i d i j a t a o l a
m n a n o v n w o s e l n o s p a q e e r a b r s a o d s
t g t i t l u e u v n e o x y m r k )
#  Letter distribution table shamelessly borrowed from "Wordy" game,
#+ ca. 1992, written by a certain fine fellow named Mendel Cooper.

declare -a LS

numelements=${#letters[@]}
randseed="$1"

instructions ()
{
  clear
  echo "Welcome to QUACKEY, the anagramming word construction game."; echo
  echo -n "Do you need instructions? (y/n) "; read ans

   if [ "$ans" = "y" -o "$ans" = "Y" ]; then
```

```
     clear
     echo -e '\E[31;47m'  # Red foreground. '\E[34;47m' for blue.
     cat <<INSTRUCTION1

QUACKEY is a variant of Perquackey [TM].
The rules are the same, but the scoring is simplified
and plurals of previously played words are allowed.
"Vulnerable" play is not yet implemented,
but it is otherwise feature-complete.

As the game begins, the player gets 10 letters.
The object is to construct valid dictionary words
of at least 3-letter length from the letterset.
Each word-length category
-- 3-letter, 4-letter, 5-letter, ... --
fills up with the fifth word entered,
and no further words in that category are accepted.

The penalty for too-short (two-letter), duplicate, unconstructable,
and invalid (not in dictionary) words is -200. The same penalty applies
to attempts to enter a word in a filled-up category.

INSTRUCTION1

  echo -n "Hit ENTER for next page of instructions. "; read az1

     cat <<INSTRUCTION2

The scoring mostly corresponds to classic Perquackey:
The first 3-letter word scores    60, plus   10 for each additional one.
The first 4-letter word scores   120, plus   20 for each additional one.
The first 5-letter word scores   200, plus   50 for each additional one.
The first 6-letter word scores   300, plus  100 for each additional one.
The first 7-letter word scores   500, plus  150 for each additional one.
The first 8-letter word scores   750, plus  250 for each additional one.
The first 9-letter word scores  1000, plus  500 for each additional one.
The first 10-letter word scores 2000, plus 2000 for each additional one.

Category completion bonuses are:
3-letter words   100
4-letter words   200
5-letter words   400
6-letter words   800
7-letter words  2000
8-letter words 10000
This is a simplification of the absurdly baroque Perquackey bonus
scoring system.

INSTRUCTION2

  echo -n "Hit ENTER for final page of instructions. "; read az1

     cat <<INSTRUCTION3


Hitting just ENTER for a word entry ends the game.

Individual word entry is timed to a maximum of 10 seconds.
*** Timing out on an entry ends the game. ***
Aside from that, the game is untimed.

--------------------------------------------------
```

```
Game statistics are automatically saved to a file.
--------------------------------------------------

For competitive ("duplicate") play, a previous letterset
may be duplicated by repeating the script's random seed,
command-line parameter \$1.
For example, "qky 7633" specifies the letterset
c a d i f r h u s k ...
INSTRUCTION3

  echo; echo -n "Hit ENTER to begin game. "; read az1

       echo -e "\033[0m"    # Turn off red.
     else clear
  fi

  clear

}



seed_random ()
{                          #  Seed random number generator.
  if [ -n "$randseed" ]    #  Can specify random seed.
  then                     #+ for play in competitive mode.
#   RANDOM="$randseed"
    echo "RANDOM seed set to "$randseed""
  else
    randseed="$$"          # Or get random seed from process ID.
    echo "RANDOM seed not specified, set to Process ID of script ($$)."
  fi

  RANDOM="$randseed"

  echo
}


get_letset ()
{
  element=0
  echo -n "Letterset:"

  for lset in $(seq $NVLET)
  do  # Pick random letters to fill out letterset.
    LS[element]="${letters[$((RANDOM%numelements))]}"
    ((element++))
  done

  echo
  echo "${LS[@]}"

}


add_word ()
{
  wrd="$1"
  local idx=0

  Status[0]=""
```

```
  Status[3]=""
  Status[4]=""

  while [ "${Words[idx]}" != '' ]
  do
    if [ "${Words[idx]}" = "$wrd" ]
    then
      Status[3]="Duplicate-word-PENALTY"
      let "Score[0]= 0 - $PENALTY"
      let "Score[1]-=$PENALTY"
      return $E_DUP
    fi

    ((idx++))
  done

  Words[idx]="$wrd"
  get_score

}

get_score()
{
  local wlen=0
  local score=0
  local bonus=0
  local first_word=0
  local add_word=0
  local numwords=0

  wlen=${#wrd}
  numwords=${Score[wlen]}
  Score[2]=0
  Status[4]=""   # Initialize "bonus" to 0.

  case "$wlen" in
    3) first_word=60
       add_word=10;;
    4) first_word=120
       add_word=20;;
    5) first_word=200
       add_word=50;;
    6) first_word=300
       add_word=100;;
    7) first_word=500
       add_word=150;;
    8) first_word=750
       add_word=250;;
    9) first_word=1000
       add_word=500;;
   10) first_word=2000
       add_word=2000;;   # This category modified from original rules!
      esac

  ((Score[wlen]++))
  if [ ${Score[wlen]} -eq $MAXCAT ]
  then   # Category completion bonus scoring simplified!
    case $wlen in
      3 ) bonus=100;;
      4 ) bonus=200;;
      5 ) bonus=400;;
      6 ) bonus=800;;
```

```
      7 ) bonus=2000;;
      8 ) bonus=10000;;
    esac  # Needn't worry about 9's and 10's.
    Status[4]="Category-$wlen-completion***BONUS***"
    Score[2]=$bonus
  else
    Status[4]=""   # Erase it.
  fi


    let "score =  $first_word +   $add_word * $numwords"
    if [ "$numwords" -eq 0 ]
    then
      Score[0]=$score
    else
      Score[0]=$add_word
    fi   #  All this to distinguish last-word score
         #+ from total running score.
  let "Score[1] += ${Score[0]}"
  let "Score[1] += ${Score[2]}"

}



get_word ()
{
  local wrd=''
  read -t $TIMEOUT wrd   # Timed read.
  echo $wrd
}

is_constructable ()
{ # This is the most complex and difficult-to-write function.
  local -a local_LS=( "${LS[@]}" )  # Local copy of letter set.
  local is_found=0
  local idx=0
  local pos
  local strlen
  local local_word=( "$1" )
  strlen=${#local_word}

  while [ "$idx" -lt "$strlen" ]
  do
    is_found=$(expr index "${local_LS[*]}" "${local_word:idx:1}")
    if [ "$is_found" -eq "$NONCONS" ] # Not constructable!
    then
      echo "$FAILURE"; return
    else
      ((pos = ($is_found - 1) / 2))   # Compensate for spaces betw. letters!
      local_LS[pos]=$NULL             # Zero out used letters.
      ((idx++))                       # Bump index.
    fi
  done

  echo "$SUCCESS"
  return
}

is_valid ()
{ # Surprisingly easy to check if word in dictionary ...
  fgrep -qw "$1" "$WLIST"   # ... courtesy of 'grep' ...
```

```
  echo $?
}

check_word ()
{
  if [ -z "$1" ]
  then
    return
  fi

  Status[1]=""
  Status[2]=""
  Status[3]=""
  Status[4]=""

  iscons=$(is_constructable "$1")
  if [ "$iscons" ]
  then
    Status[1]="constructable"
    v=$(is_valid "$1")
    if [ "$v" -eq "$SUCCESS" ]
    then
      Status[2]="valid"
      strlen=${#1}

      if [ ${Score[strlen]} -eq "$MAXCAT" ]   # Category full!
      then
        Status[3]="Category-$strlen-overflow-PENALTY"
        return $NG
      fi

      case "$strlen" in
        1 | 2 )
        Status[3]="Two-letter-word-PENALTY"
        return $NG;;
        * )
        Status[3]=""
        return $SUCCESS;;
      esac
    else
      Status[3]="Not-valid-PENALTY"
      return $NG
    fi
  else
    Status[3]="Not-constructable-PENALTY"
      return $NG
  fi

  ### FIXME: Streamline the above code block.

}


display_words ()
{
  local idx=0
  local wlen0

  clear
  echo "Letterset:   ${LS[@]}"
  echo "Threes:    Fours:    Fives:     Sixes:    Sevens:    Eights:"
  echo "------------------------------------------------------------"
```

```
  while [ "${Words[idx]}" != '' ]
  do
   wlen0=${#Words[idx]}
   case "$wlen0" in
     3) ;;
     4) echo -n "           " ;;
     5) echo -n "                     " ;;
     6) echo -n "                                " ;;
     7) echo -n "                                          " ;;
     8) echo -n "                                                     " ;;
   esac
   echo "${Words[idx]}"
   ((idx++))
  done

  ### FIXME: The word display is pretty crude.
}


play ()
{
  word="Start game"   # Dummy word, to start ...

  while [ "$word" ]   #  If player just hits return (null word),
  do                  #+ then game ends.
    echo "$word: "${Status[@]}""
    echo -n "Last score: [${Score[0]}]   TOTAL score: [${Score[1]}]:     Next word: "
    total=${Score[1]}
    word=$(get_word)
    check_word "$word"

    if [ "$?" -eq "$SUCCESS" ]
    then
      add_word "$word"
    else
      let "Score[0]= 0 - $PENALTY"
      let "Score[1]-=$PENALTY"
    fi

  display_words
  done   # Exit game.

  ### FIXME: The play () function calls too many other functions.
  ### This verges on "spaghetti code" !!!
}

end_of_game ()
{ # Save and display stats.

  #######################Autosave##########################
  savefile=qky.save.$$
  #                 ^^ PID of script
  echo `date` >> $savefile
  echo "Letterset # $randseed  (random seed) ">> $savefile
  echo -n "Letterset: " >> $savefile
  echo "${LS[@]}" >> $savefile
  echo "---------" >> $savefile
  echo "Words constructed:" >> $savefile
  echo "${Words[@]}" >> $savefile
```

```
  echo >> $savefile
  echo "Score: $total" >> $savefile

  echo "Statistics for this round saved in \""$savefile"\""
  #########################################################

  echo "Score for this round: $total"
  echo "Words:  ${Words[@]}"
}

# ---------#
instructions
seed_random
get_letset
play
end_of_game
# ---------#

exit $?

# TODO:
#
# 1) Clean up code!
# 2) Prettify the display_words () function (maybe with widgets?).
# 3) Improve the time-out ... maybe change to untimed entry,
#+   but with a time limit for the overall round.
# 4) An on-screen countdown timer would be nice.
# 5) Implement "vulnerable" mode of play for compatibility with classic
#+   version of the game.
# 6) Improve save-to-file capability (and maybe make it optional).
# 7) Fix bugs!!!

# For more info, reference:
# http://bash.deta.in/qky.README.html
```

Example A-42. Nim

```
#!/bin/bash
# nim.sh: Game of Nim

# Author: Mendel Cooper
# Reldate: 15 July 2008
# License: GPL3

ROWS=5     # Five rows of pegs (or matchsticks).
WON=91     # Exit codes to keep track of wins/losses.
LOST=92    # Possibly useful if running in batch mode.
QUIT=99
peg_msg=   # Peg/Pegs?
Rows=( 0 5 4 3 2 1 )    # Array holding play info.
# ${Rows[0]} holds total number of pegs, updated after each turn.
# Other array elements hold number of pegs in corresponding row.

instructions ()
{
  clear
  tput bold
  echo "Welcome to the game of Nim."; echo
  echo -n "Do you need instructions? (y/n) "; read ans

   if [ "$ans" = "y" -o "$ans" = "Y" ]; then
```

```
     clear
     echo -e '\E[33;41m'  # Yellow fg., over red bg.; bold.
     cat <<INSTRUCTIONS

Nim is a game with roots in the distant past.
This particular variant starts with five rows of pegs.

1:    | | | | |
2:     | | | |
3:      | | |
4:       | |
5:        |

The number at the left identifies the row.

The human player moves first, and alternates turns with the bot.
A turn consists of removing at least one peg from a single row.
It is permissable to remove ALL the pegs from a row.
For example, in row 2, above, the player can remove 1, 2, 3, or 4 pegs.
The player who removes the last peg loses.

The strategy consists of trying to be the one who removes
the next-to-last peg(s), leaving the loser with the final peg.

To exit the game early, hit ENTER during your turn.
INSTRUCTIONS

echo; echo -n "Hit ENTER to begin game. "; read azx

      echo -e "\033[0m"    # Restore display.
      else tput sgr0; clear
  fi

clear

}


tally_up ()
{
  let "Rows[0] = ${Rows[1]} + ${Rows[2]} + ${Rows[3]} + ${Rows[4]} + \
  ${Rows[5]}"    # Add up how many pegs remaining.
}


display ()
{
  index=1   # Start with top row.
  echo

  while [ "$index" -le "$ROWS" ]
  do
    p=${Rows[index]}
    echo -n "$index:   "          # Show row number.

  # ------------------------------------------------
  # Two concurrent inner loops.

      indent=$index
      while [ "$indent" -gt 0 ]
      do
        echo -n " "               # Staggered rows.
```

```
        ((indent--))              # Spacing between pegs.
      done

    while [ "$p" -gt 0 ]
    do
      echo -n "| "
      ((p--))
    done
  # -----------------------------------------------

  echo
  ((index++))
  done

  tally_up

  rp=${Rows[0]}

  if [ "$rp" -eq 1 ]
  then
    peg_msg=peg
    final_msg="Game over."
  else             # Game not yet over . . .
    peg_msg=pegs
    final_msg=""   # . . . So "final message" is blank.
  fi

  echo "      $rp $peg_msg remaining."
  echo "      "$final_msg""


  echo
}

player_move ()
{

  echo "Your move:"

  echo -n "Which row? "
  while read idx
  do                   # Validity check, etc.

    if [ -z "$idx" ]   # Hitting return quits.
    then
        echo "Premature exit."; echo
        tput sgr0      # Restore display.
        exit $QUIT
    fi

    if [ "$idx" -gt "$ROWS" -o "$idx" -lt 1 ]   # Bounds check.
    then
      echo "Invalid row number!"
      echo -n "Which row? "
    else
      break
    fi
    # TODO:
    # Add check for non-numeric input.
    # Also, script crashes on input outside of range of long double.
    # Fix this.
```

```
  done

  echo -n "Remove how many? "
  while read num
  do                    # Validity check.

  if [ -z "$num" ]
  then
    echo "Premature exit."; echo
    tput sgr0           # Restore display.
    exit $QUIT
  fi

    if [ "$num" -gt ${Rows[idx]} -o "$num" -lt 1 ]
    then
      echo "Cannot remove $num!"
      echo -n "Remove how many? "
    else
      break
    fi
  done
  # TODO:
  # Add check for non-numeric input.
  # Also, script crashes on input outside of range of long double.
  # Fix this.

  let "Rows[idx] -= $num"

  display
  tally_up

  if [ ${Rows[0]} -eq 1 ]
  then
   echo "      Human wins!"
   echo "      Congratulations!"
   tput sgr0   # Restore display.
   echo
   exit $WON
  fi

  if [ ${Rows[0]} -eq 0 ]
  then          # Snatching defeat from the jaws of victory . . .
    echo "      Fool!"
    echo "      You just removed the last peg!"
    echo "      Bot wins!"
    tput sgr0   # Restore display.
    echo
    exit $LOST
  fi
}


bot_move ()
{

  row_b=0
  while [[ $row_b -eq 0 || ${Rows[row_b]} -eq 0 ]]
  do
    row_b=$RANDOM          # Choose random row.
    let "row_b %= $ROWS"
  done

```

```
  num_b=0
  r0=${Rows[row_b]}

  if [ "$r0" -eq 1 ]
  then
    num_b=1
  else
    let "num_b = $r0 - 1"
         #  Leave only a single peg in the row.
  fi     #  Not a very strong strategy,
         #+ but probably a bit better than totally random.

  let "Rows[row_b] -= $num_b"
  echo -n "Bot:  "
  echo "Removing from row $row_b ... "

  if [ "$num_b" -eq 1 ]
  then
    peg_msg=peg
  else
    peg_msg=pegs
  fi

  echo "      $num_b $peg_msg."

  display
  tally_up

  if [ ${Rows[0]} -eq 1 ]
  then
   echo "      Bot wins!"
   tput sgr0   # Restore display.
   exit $WON
  fi

}


# ================================================== #
instructions     # If human player needs them . . .
tput bold        # Bold characters for easier viewing.
display          # Show game board.

while [ true ]   # Main loop.
do               # Alternate human and bot turns.
  player_move
  bot_move
done
# ================================================== #

# Exercise:
# --------
# Improve the bot's strategy.
# There is, in fact, a Nim strategy that can force a win.
# See the Wikipedia article on Nim:  http://en.wikipedia.org/wiki/Nim
# Recode the bot to use this strategy (rather difficult).

#  Curiosities:
#  -----------
#  Nim played a prominent role in Alain Resnais' 1961 New Wave film,
#+ Last Year at Marienbad.
```

```
#
#  In 1978, Leo Christopherson wrote an animated version of Nim,
#+ Android Nim, for the TRS-80 Model I.
```

Example A-43. A command-line stopwatch

```
#!/bin/sh
# sw.sh
# A command-line Stopwatch

# Author: Pádraig Brady
#    http://www.pixelbeat.org/scripts/sw
#    (Minor reformatting by ABS Guide author.)
#    Used in ABS Guide with script author's permission.
# Notes:
#    This script starts a few processes per lap, in addition to
#    the shell loop processing, so the assumption is made that
#    this takes an insignificant amount of time compared to
#    the response time of humans (~.1s) (or the keyboard
#    interrupt rate (~.05s)).
#    '?' for splits must be entered twice if characters
#    (erroneously) entered before it (on the same line).
#    '?' since not generating a signal may be slightly delayed
#    on heavily loaded systems.
#    Lap timings on ubuntu may be slightly delayed due to:
#    https://bugs.launchpad.net/bugs/62511
# Changes:
#    V1.0, 23 Aug 2005, Initial release
#    V1.1, 26 Jul 2007, Allow both splits and laps from single invocation.
#                       Only start timer after a key is pressed.
#                       Indicate lap number
#                       Cache programs at startup so there is less error
#                       due to startup delays.
#    V1.2, 01 Aug 2007, Work around `date` commands that don't have
#                       nanoseconds.
#                       Use stty to change interrupt keys to space for
#                       laps etc.
#                       Ignore other input as it causes problems.
#    V1.3, 01 Aug 2007, Testing release.
#    V1.4, 02 Aug 2007, Various tweaks to get working under ubuntu
#                       and Mac OS X.
#    V1.5, 27 Jun 2008, set LANG=C as got vague bug report about it.

export LANG=C

ulimit -c 0   # No coredumps from SIGQUIT.
trap '' TSTP  # Ignore Ctrl-Z just in case.
save_tty=`stty -g` && trap "stty $save_tty" EXIT  # Restore tty on exit.
stty quit ' ' # Space for laps rather than Ctrl-\.
stty eof  '?' # ? for splits rather than Ctrl-D.
stty -echo    # Don't echo input.

cache_progs() {
    stty > /dev/null
    date > /dev/null
    grep . < /dev/null
    (echo "import time" | python) 2> /dev/null
    bc < /dev/null
    sed '' < /dev/null
    printf '1' > /dev/null
    /usr/bin/time false 2> /dev/null
```

```
    cat < /dev/null
}
cache_progs   # To minimise startup delay.

date +%s.%N | grep -qF 'N' && use_python=1 # If `date` lacks nanoseconds.
now() {
    if [ "$use_python" ]; then
        echo "import time; print time.time()" 2>/dev/null | python
    else
        printf "%.2f" `date +%s.%N`
    fi
}

fmt_seconds() {
    seconds=$1
    mins=`echo $seconds/60 | bc`
    if [ "$mins" != "0" ]; then
        seconds=`echo "$seconds - ($mins*60)" | bc`
        echo "$mins:$seconds"
    else
        echo "$seconds"
    fi
}

total() {
    end=`now`
    total=`echo "$end - $start" | bc`
    fmt_seconds $total
}

stop() {
    [ "$lapped" ] && lap "$laptime" "display"
    total
    exit
}

lap() {
    laptime=`echo "$1" | sed -n 's/.*real[^0-9.]*\(.*\)/\1/p'`
    [ ! "$laptime" -o "$laptime" = "0.00" ] && return
    # Signals too frequent.
    laptotal=`echo $laptime+0$laptotal | bc`
    if [ "$2" = "display" ]; then
        lapcount=`echo 0$lapcount+1 | bc`
        laptime=`fmt_seconds $laptotal`
        echo $laptime "($lapcount)"
        lapped="true"
        laptotal="0"
    fi
}

echo -n "Space for lap | ? for split | Ctrl-C to stop | Space to start...">&2

while true; do
    trap true INT QUIT  # Set signal handlers.
    laptime=`/usr/bin/time -p 2>&1 cat >/dev/null`
    ret=$?
    trap '' INT QUIT    # Ignore signals within this script.
    if [ $ret -eq 1 -o $ret -eq 2 -o $ret -eq 130 ]; then # SIGINT = stop
        [ ! "$start" ] && { echo >&2; exit; }
        stop
    elif [ $ret -eq 3 -o $ret -eq 131 ]; then             # SIGQUIT = lap
        if [ ! "$start" ]; then
```

```
            start=`now` || exit 1
            echo >&2
            continue
        fi
        lap "$laptime" "display"
    else                # eof = split
        [ ! "$start" ] && continue
        total
        lap "$laptime"  # Update laptotal.
    fi
done

exit $?
```

Example A-44. An all-purpose shell scripting homework assignment solution

```
#!/bin/bash
#  homework.sh: All-purpose homework assignment solution.
#  Author: M. Leo Cooper
#  If you substitute your own name as author, then it is plagiarism,
#+ possibly a lesser sin than cheating on your homework!
#  License: Public Domain

#  This script may be turned in to your instructor
#+ in fulfillment of ALL shell scripting homework assignments.
#  It's sparsely commented, but you, the student, can easily remedy that.
#  The script author repudiates all responsibility!

DLA=1
P1=2
P2=4
P3=7
PP1=0
PP2=8
MAXL=9
E_LZY=99

declare -a L
L[0]="3 4 0 17 29 8 13 18 19 17 20 2 19 14 17 28"
L[1]="8 29 12 14 18 19 29 4 12 15 7 0 19 8 2 0 11 11 24 29 17 4 6 17 4 19"
L[2]="29 19 7 0 19 29 8 29 7 0 21 4 29 13 4 6 11 4 2 19 4 3"
L[3]="19 14 29 2 14 12 15 11 4 19 4 29 19 7 8 18 29"
L[4]="18 2 7 14 14 11 22 14 17 10 29 0 18 18 8 6 13 12 4 13 19 26"
L[5]="15 11 4 0 18 4 29 0 2 2 4 15 19 29 12 24 29 7 20 12 1 11 4 29"
L[6]="4 23 2 20 18 4 29 14 5 29 4 6 17 4 6 8 14 20 18 29"
L[7]="11 0 25 8 13 4 18 18 27"
L[8]="0 13 3 29 6 17 0 3 4 29 12 4 29 0 2 2 14 17 3 8 13 6 11 24 26"
L[9]="19 7 0 13 10 29 24 14 20 26"

declare -a \
alph=( A B C D E F G H I J K L M N O P Q R S T U V W X Y Z . , : ' ' )


pt_lt ()
{
  echo -n "${alph[$1]}"
  echo -n -e "\a"
  sleep $DLA
}

b_r ()
```

```
{
 echo -e '\E[31;48m\033[1m'
}

cr ()
{
 echo -e "\a"
 sleep $DLA
}

restore ()
{
  echo -e '\033[0m'             # Bold off.
  tput sgr0                     # Normal.
}


p_l ()
{
  for ltr in $1
  do
    pt_lt "$ltr"
  done
}

# ----------------------
b_r

for i in $(seq 0 $MAXL)
do
  p_l "${L[i]}"
  if [[ "$i" -eq "$P1" || "$i" -eq "$P2" || "$i" -eq "$P3" ]]
  then
    cr
  elif [[ "$i" -eq "$PP1" || "$i" -eq "$PP2" ]]
  then
    cr; cr
  fi
done

restore
# ----------------------

echo

exit $E_LZY

#  A typical example of an obfuscated script that is difficult
#+ to understand, and frustrating to maintain.
#  In your career as a sysadmin, you'll run into these critters
#+ all too often.
```

Example A-45. The Knight's Tour

```
#!/bin/bash
# ktour.sh

# author: mendel cooper
# reldate: 12 Jan 2009
# license: public domain
```

```
# (Not much sense GPLing something that's pretty much in the common
#+ domain anyhow.)

###################################################################
#             The Knight's Tour, a classic problem.               #
#             =====================================               #
#  The knight must move onto every square of the chess board,     #
#  but cannot revisit any square he has already visited.          #
#                                                                 #
#  And just why is Sir Knight unwelcome for a return visit?       #
#  Could it be that he has a habit of partying into the wee hours #
#+ of the morning?                                                #
#  Possibly he leaves pizza crusts in the bed, empty beer bottles #
#+ all over the floor, and clogs the plumbing. . . .              #
#                                                                 #
#  -------------------------------------------------------------  #
#                                                                 #
#  Usage: ktour.sh [start-square] [stupid]                        #
#                                                                 #
#  Note that start-square can be a square number                  #
#+ in the range 0 - 63 ... or                                     #
#  a square designator in conventional chess notation,            #
#  such as a1, f5, h3, etc.                                       #
#                                                                 #
#  If start-square-number not supplied,                           #
#+ then starts on a random square somewhere on the board.         #
#                                                                 #
# "stupid" as second parameter sets the stupid strategy.          #
#                                                                 #
#  Examples:                                                      #
#  ktour.sh 23          starts on square #23 (h3)                 #
#  ktour.sh g6 stupid   starts on square #46,                     #
#                       using "stupid" (non-Warnsdorff) strategy. #
###################################################################

DEBUG=      # Set this to echo debugging info to stdout.
SUCCESS=0
FAIL=99
BADMOVE=-999
FAILURE=1
LINELEN=21  # How many moves to display per line.
# ---------------------------------------- #
# Board array params
ROWS=8   # 8 x 8 board.
COLS=8
let "SQUARES = $ROWS * $COLS"
let "MAX = $SQUARES - 1"
MIN=0
# 64 squares on board, indexed from 0 to 63.

VISITED=1
UNVISITED=-1
UNVSYM="##"
# ---------------------------------------- #
# Global variables.
startpos=    # Starting position (square #, 0 - 63).
currpos=     # Current position.
movenum=     # Move number.
CRITPOS=37   # Have to patch for f5 starting position!

declare -i board
# Use a one-dimensional array to simulate a two-dimensional one.
```

```
# This can make life difficult and result in ugly kludges; see below.
declare -i moves  # Offsets from current knight position.


initialize_board ()
{
  local idx

  for idx in {0..63}
  do
    board[$idx]=$UNVISITED
  done
}



print_board ()
{
  local idx

  echo "    _____________________________________"
  for row in {7..0}               #  Reverse order of rows ...
  do                              #+ so it prints in chessboard order.
    let "rownum = $row + 1"       #  Start numbering rows at 1.
    echo -n "$rownum  |"          #  Mark board edge with border and
    for column in {0..7}          #+ "algebraic notation."
    do
      let "idx = $ROWS*$row + $column"
      if [ ${board[idx]} -eq $UNVISITED ]
      then
        echo -n "$UNVSYM   "      ##
      else                        # Mark square with move number.
        printf "%02d " "${board[idx]}"; echo -n "  "
      fi
    done
    echo -e -n "\b\b\b|"  # \b is a backspace.
    echo                  # -e enables echoing escaped chars.
  done

  echo "    -------------------------------------"
  echo "     a    b    c    d    e    f    g    h"
}



failure()
{ # Whine, then bail out.
  echo
  print_board
  echo
  echo    "   Waah!!! Ran out of squares to move to!"
  echo -n "   Knight's Tour attempt ended"
  echo    " on $(to_algebraic $currpos) [square #$currpos]"
  echo    "   after just $movenum moves!"
  echo
  exit $FAIL
}



xlat_coords ()   #  Translate x/y coordinates to board position
{                #+ (board-array element #).
```

```
  #  For user input of starting board position as x/y coords.
  #  This function not used in initial release of ktour.sh.
  #  May be used in an updated version, for compatibility with
  #+ standard implementation of the Knight's Tour in C, Python, etc.
  if [ -z "$1" -o -z "$2" ]
  then
    return $FAIL
  fi

  local xc=$1
  local yc=$2

  let "board_index = $xc * $ROWS + yc"

  if [ $board_index -lt $MIN -o $board_index -gt $MAX ]
  then
    return $FAIL    # Strayed off the board!
  else
    return $board_index
  fi
}



to_algebraic ()   #  Translate board position (board-array element #)
{                 #+ to standard algebraic notation used by chess players.
  if [ -z "$1" ]
  then
    return $FAIL
  fi

  local element_no=$1   # Numerical board position.
  local col_arr=( a b c d e f g h )
  local row_arr=( 1 2 3 4 5 6 7 8 )

  let "row_no = $element_no / $ROWS"
  let "col_no = $element_no % $ROWS"
  t1=${col_arr[col_no]}; t2=${row_arr[row_no]}
  local apos=$t1$t2   # Concatenate.
  echo $apos
}



from_algebraic ()   #  Translate standard algebraic chess notation
{                   #+ to numerical board position (board-array element #).
                    #  Or recognize numerical input & return it unchanged.
  if [ -z "$1" ]
  then
    return $FAIL
  fi   # If no command-line arg, then will default to random start pos.

  local ix
  local ix_count=0
  local b_index     # Board index [0-63]
  local alpos="$1"

  arow=${alpos:0:1} # position = 0, length = 1
  acol=${alpos:1:1}

  if [[ $arow =~ [[:digit:]] ]]   #  Numerical input?
  then       #  POSIX char class
```

```
    if [[ $acol =~ [[:alpha:]] ]] # Number followed by a letter? Illegal!
      then return $FAIL
    else if [ $alpos -gt $MAX ]   # Off board?
      then return $FAIL
    else return $alpos            #  Return digit(s) unchanged . . .
      fi                          #+ if within range.
    fi
  fi

  if [[ $acol -eq $MIN || $acol -gt $ROWS ]]
  then        # Outside of range 1 - 8?
    return $FAIL
  fi

  for ix in a b c d e f g h
  do  # Convert column letter to column number.
   if [ "$arow" = "$ix" ]
   then
     break
   fi
  ((ix_count++))    # Find index count.
  done

  ((acol--))        # Decrementing converts to zero-based array.
  let "b_index = $ix_count + $acol * $ROWS"

  if [ $b_index -gt $MAX ]   # Off board?
  then
    return $FAIL
  fi

  return $b_index

}


generate_moves ()   #  Calculate all valid knight moves,
{                   #+ relative to current position ($1),
                    #+ and store in ${moves} array.
  local kt_hop=1    #  One square  :: short leg of knight move.
  local kt_skip=2   #  Two squares :: long leg  of knight move.
  local valmov=0    #  Valid moves.
  local row_pos; let "row_pos = $1 % $COLS"


  let "move1 = -$kt_skip + $ROWS"           # 2 sideways to-the-left,  1 up
    if [[ `expr $row_pos - $kt_skip` -lt $MIN ]]   # An ugly, ugly kludge!
    then                                           # Can't move off board.
      move1=$BADMOVE                               # Not even temporarily.
    else
      ((valmov++))
    fi
  let "move2 = -$kt_hop + $kt_skip * $ROWS" # 1 sideways to-the-left,  2 up
    if [[ `expr $row_pos - $kt_hop` -lt $MIN ]]    # Kludge continued ...
    then
      move2=$BADMOVE
    else
      ((valmov++))
    fi
  let "move3 =  $kt_hop + $kt_skip * $ROWS" # 1 sideways to-the-right, 2 up
    if [[ `expr $row_pos + $kt_hop` -ge $COLS ]]
    then
```

```
      move3=$BADMOVE
    else
      ((valmov++))
    fi
  let "move4 =  $kt_skip + $ROWS"           # 2 sideways to-the-right, 1 up
    if [[ `expr $row_pos + $kt_skip` -ge $COLS ]]
    then
      move4=$BADMOVE
    else
      ((valmov++))
    fi
  let "move5 =  $kt_skip - $ROWS"           # 2 sideways to-the-right, 1 dn
    if [[ `expr $row_pos + $kt_skip` -ge $COLS ]]
    then
      move5=$BADMOVE
    else
      ((valmov++))
    fi
  let "move6 =  $kt_hop - $kt_skip * $ROWS" # 1 sideways to-the-right, 2 dn
    if [[ `expr $row_pos + $kt_hop` -ge $COLS ]]
    then
      move6=$BADMOVE
    else
      ((valmov++))
    fi
  let "move7 = -$kt_hop - $kt_skip * $ROWS" # 1 sideways to-the-left,  2 dn
    if [[ `expr $row_pos - $kt_hop` -lt $MIN ]]
    then
      move7=$BADMOVE
    else
      ((valmov++))
    fi
  let "move8 = -$kt_skip - $ROWS"           # 2 sideways to-the-left,  1 dn
    if [[ `expr $row_pos - $kt_skip` -lt $MIN ]]
    then
      move8=$BADMOVE
    else
      ((valmov++))
    fi   # There must be a better way to do this.

  local m=( $valmov $move1 $move2 $move3 $move4 $move5 $move6 $move7 $move8 )
  # ${moves[0]} = number of valid moves.
  # ${moves[1]} ... ${moves[8]} = possible moves.
  echo "${m[*]}"    # Elements of array to stdout for capture in a var.

}



is_on_board ()  # Is position actually on the board?
{
  if [[ "$1" -lt "$MIN" || "$1" -gt "$MAX" ]]
  then
    return $FAILURE
  else
    return $SUCCESS
  fi
}



do_move ()      # Move the knight!
```

```
{
  local valid_moves=0
  local aapos
  currposl="$1"
  lmin=$ROWS
  iex=0
  squarel=
  mpm=
  mov=
  declare -a p_moves

  ########################## DECIDE-MOVE #############################
  if [ $startpos -ne $CRITPOS ]
  then   # CRITPOS = square #37
    decide_move
  else                     # Needs a special patch for startpos=37 !!!
    decide_move_patched    # Why this particular move and no other ???
  fi
  ####################################################################

  (( ++movenum ))          # Increment move count.
  let "square = $currposl + ${moves[iex]}"

  ##################    DEBUG    ###############
  if [ "$DEBUG" ]
    then debug   # Echo debugging information.
  fi
  ##############################################

  if [[ "$square" -gt $MAX || "$square" -lt $MIN ||
        ${board[square]} -ne $UNVISITED ]]
  then
    (( --movenum ))              #  Decrement move count,
    echo "RAN OUT OF SQUARES!!!" #+ since previous one was invalid.
    return $FAIL
  fi

  board[square]=$movenum
  currpos=$square       # Update current position.
  ((valid_moves++));    # moves[0]=$valid_moves
  aapos=$(to_algebraic $square)
  echo -n "$aapos "
  test $(( $Moves % $LINELEN )) -eq 0 && echo
  # Print LINELEN=21 moves per line. A valid tour shows 3 complete lines.
  return $valid_moves   # Found a square to move to!
}



do_move_stupid()   #  Dingbat algorithm,
{                  #+ courtesy of script author, *not* Warnsdorff.
  local valid_moves=0
  local movloc
  local squareloc
  local aapos
  local cposloc="$1"

  for movloc in {1..8}
  do       # Move to first-found unvisited square.
    let "squareloc = $cposloc + ${moves[movloc]}"
    is_on_board $squareloc
    if [ $? -eq $SUCCESS ] && [ ${board[squareloc]} -eq $UNVISITED ]
```

```
    then   # Add conditions to above if-test to improve algorithm.
      (( ++movenum ))
      board[squareloc]=$movenum
      currpos=$squareloc     # Update current position.
      ((valid_moves++));     # moves[0]=$valid_moves
      aapos=$(to_algebraic $squareloc)
      echo -n "$aapos "
      test $(( $Moves % $LINELEN )) -eq 0 && echo   # Print 21 moves/line.
      return $valid_moves    # Found a square to move to!
    fi
  done

  return $FAIL
  #  If no square found in all 8 loop iterations,
  #+ then Knight's Tour attempt ends in failure.

  #  Dingbat algorithm will typically fail after about 30 - 40 moves,
  #+ but executes _much_ faster than Warnsdorff's in do_move() function.
}



decide_move ()         #  Which move will we make?
{                      #  But, fails on startpos=37 !!!
  for mov in {1..8}
  do
    let "squarel = $currposl + ${moves[mov]}"
    is_on_board $squarel
    if [[ $? -eq $SUCCESS && ${board[squarel]} -eq $UNVISITED ]]
    then   #  Find accessible square with least possible future moves.
           #  This is Warnsdorff's algorithm.
           #  What happens is that the knight wanders toward the outer edge
           #+ of the board, then pretty much spirals inward.
           #  Given two or more possible moves with same value of
           #+ least-possible-future-moves, this implementation chooses
           #+ the _first_ of those moves.
           #  This means that there is not necessarily a unique solution
           #+ for any given starting position.

      possible_moves $squarel
      mpm=$?
      p_moves[mov]=$mpm

      if [ $mpm -lt $lmin ]  # If less than previous minimum ...
      then #     ^^
        lmin=$mpm            # Update minimum.
        iex=$mov             # Save index.
      fi

    fi
  done
}



decide_move_patched ()         #  Decide which move to make,
{  #        ^^^^^^^            #+ but only if startpos=37 !!!
  for mov in {1..8}
  do
    let "squarel = $currposl + ${moves[mov]}"
    is_on_board $squarel
    if [[ $? -eq $SUCCESS && ${board[squarel]} -eq $UNVISITED ]]
```

```
    then
      possible_moves $squarel
      mpm=$?
      p_moves[mov]=$mpm

      if [ $mpm -le $lmin ]  # If less-than-or equal to prev. minimum!
      then #     ^^
        lmin=$mpm
        iex=$mov
      fi

    fi
  done                       # There has to be a better way to do this.
}



possible_moves ()            #  Calculate number of possible moves,
{                            #+ given the current position.

  if [ -z "$1" ]
  then
    return $FAIL
  fi

  local curr_pos=$1
  local valid_movl=0
  local icx=0
  local movl
  local sq
  declare -a movesloc

  movesloc=( $(generate_moves $curr_pos) )

  for movl in {1..8}
  do
    let "sq = $curr_pos + ${movesloc[movl]}"
    is_on_board $sq
    if [ $? -eq $SUCCESS ] && [ ${board[sq]} -eq $UNVISITED ]
    then
      ((valid_movl++));
    fi
  done

  return $valid_movl         # Found a square to move to!
}


strategy ()
{
  echo

  if [ -n "$STUPID" ]
  then
    for Moves in {1..63}
    do
      cposl=$1
      moves=( $(generate_moves $currpos) )
      do_move_stupid "$currpos"
      if [ $? -eq $FAIL ]
      then
        failure
```

```
      fi
      done
  fi

  #  Don't need an "else" clause here,
  #+ because Stupid Strategy will always fail and exit!
  for Moves in {1..63}
  do
    cposl=$1
    moves=( $(generate_moves $currpos) )
    do_move "$currpos"
    if [ $? -eq $FAIL ]
    then
      failure
    fi

  done
        #  Could have condensed above two do-loops into a single one,
  echo  #+ but this would have slowed execution.

  print_board
  echo
  echo "Knight's Tour ends on $(to_algebraic $currpos) [square #$currpos]."
  return $SUCCESS
}

debug ()
{       # Enable this by setting DEBUG=1 near beginning of script.
  local n

  echo "================================="
  echo "  At move number  $movenum:"
  echo " *** possible moves = $mpm ***"
# echo "### square = $square ###"
  echo "lmin = $lmin"
  echo "${moves[@]}"

  for n in {1..8}
  do
    echo -n "($n):${p_moves[n]} "
  done

  echo
  echo "iex = $iex :: moves[iex] = ${moves[iex]}"
  echo "square = $square"
  echo "================================="
  echo
} # Gives pretty complete status after ea. move.



# =============================================================== #
# int main () {
from_algebraic "$1"
startpos=$?
if [ "$startpos" -eq "$FAIL" ]          # Okay even if no $1.
then   #         ^^^^^^^^^^^              Okay even if input -lt 0.
  echo "No starting square specified (or illegal input)."
  let "startpos = $RANDOM % $SQUARES"   # 0 - 63 permissable range.
fi
```

```
if [ "$2" = "stupid" ]
then
  STUPID=1
  echo -n "     ### Stupid Strategy ###"
else
  STUPID=''
  echo -n "  *** Warnsdorff's Algorithm ***"
fi


initialize_board

movenum=0
board[startpos]=$movenum   # Mark each board square with move number.
currpos=$startpos
algpos=$(to_algebraic $startpos)

echo; echo "Starting from $algpos [square #$startpos] ..."; echo
echo -n "Moves:"

strategy "$currpos"

echo

exit 0   # return 0;

# }      # End of main() pseudo-function.
# =============================================================== #


# Exercises:
# ---------
#
# 1) Extend this example to a 10 x 10 board or larger.
# 2) Improve the "stupid strategy" by modifying the
#    do_move_stupid function.
#    Hint: Prevent straying into corner squares in early moves
#          (the exact opposite of Warnsdorff's algorithm!).
# 3) This script could stand considerable improvement and
#    streamlining, especially in the poorly-written
#    generate_moves() function
#    and in the DECIDE-MOVE patch in the do_move() function.
#    Must figure out why standard algorithm fails for startpos=37 ...
#+   but _not_ on any other, including symmetrical startpos=26.
#    Possibly, when calculating possible moves, counts the move back
#+   to the originating square. If so, it might be a relatively easy fix.
```

Example A-46. Magic Squares

```
#!/bin/bash
# msquare.sh
# Magic Square generator (odd-order squares only!)

# Author: mendel cooper
# reldate: 19 Jan. 2009
# License: Public Domain
# A C-program by the very talented Kwon Young Shin inspired this script.
#     http://user.chollian.net/~brainstm/MagicSquare.htm

# Definition: A "magic square" is a two-dimensional array
#             of integers in which all the rows, columns,
```

```
#             and *long* diagonals add up to the same number.
#             Being "square," the array has the same number
#             of rows and columns. That number is the "order."
# An example of a magic square of order 3 is:
#   8  1  6
#   3  5  7
#   4  9  2
# All the rows, columns, and the two long diagonals add up to 15.


# Globals
EVEN=2
MAXSIZE=31   # 31 rows x 31 cols.
E_usage=90   # Invocation error.
dimension=
declare -i square

usage_message ()
{
  echo "Usage: $0 order"
  echo "   ... where \"order\" (square size) is an ODD integer"
  echo "       in the range 3 - 31."
  #  Actually works for squares up to order 159,
  #+ but large squares will not display pretty-printed in a term window.
  #  Try increasing MAXSIZE, above.
  exit $E_usage
}


calculate ()       # Here's where the actual work gets done.
{
  local row col index dimadj j k cell_val=1
  dimension=$1

  let "dimadj = $dimension * 3"; let "dimadj /= 2"   # x 1.5, then truncate.

  for ((j=0; j < dimension; j++))
  do
    for ((k=0; k < dimension; k++))
    do  # Calculate indices, then convert to 1-dim. array index.
        # Bash doesn't support multidimensional arrays. Pity.
      let "col = $k - $j + $dimadj"; let "col %= $dimension"
      let "row = $j * 2 - $k + $dimension"; let "row %= $dimension"
      let "index = $row*($dimension) + $col"
      square[$index]=cell_val; ((cell_val++))
    done
  done
}     # Plain math, visualization not required.


print_square ()               # Output square, one row at a time.
{
  local row col idx d1
  let "d1 = $dimension - 1"   # Adjust for zero-indexed array.

  for row in $(seq 0 $d1)
  do

    for col in $(seq 0 $d1)
    do
      let "idx = $row * $dimension + $col"
      printf "%3d " "${square[idx]}"; echo -n "  "
```

```
    done   # Displays up to 13th order neatly in 80-column term window.

    echo   # Newline after each row.
  done
}


#################################################
if [[ -z "$1" ]] || [[ "$1" -gt $MAXSIZE ]]
then
  usage_message
fi

let "test_even = $1 % $EVEN"
if [ $test_even -eq 0 ]
then           # Can't handle even-order squares.
  usage_message
fi

calculate $1
print_square   # echo "${square[@]}"   # DEBUG

exit $?
#################################################


# Exercises:
# ---------
# 1) Add a function to calculate the sum of each row, column,
#    and *long* diagonal. The sums must match.
#    This is the "magic constant" of that particular order square.
# 2) Have the print_square function auto-calculate how much space
#    to allot between square elements for optimized display.
#    This might require parameterizing the "printf" line.
# 3) Add appropriate functions for generating magic squares
#    with an *even* number of rows/columns.
#    This is non-trivial(!).
#    See the URL for Kwon Young Shin, above, for help.
```

Example A-47. Fifteen Puzzle

```
#!/bin/bash
# fifteen.sh

# Classic "Fifteen Puzzle"
# Author: Antonio Macchi
# Lightly edited and commented by ABS Guide author.
# Used in ABS Guide with permission. (Thanks!)

#  The invention of the Fifteen Puzzle is attributed to either
#+ Sam Loyd or Noyes Palmer Chapman.
#  The puzzle was wildly popular in the late 19th-century.

#  Object: Rearrange the numbers so they read in order,
#+ from 1 - 15:   ________________
#                |  1   2   3   4 |
#                |  5   6   7   8 |
#                |  9  10  11  12 |
#                | 13  14  15     |
#                 ----------------
```

```
######################
# Constants          #
  SQUARES=16         #
  FAIL=70            #
  E_PREMATURE_EXIT=80 #
######################


########
# Data #
########

Puzzle=( 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 " " )


#############
# Functions #
#############

function swap
{
  local tmp

  tmp=${Puzzle[$1]}
  Puzzle[$1]=${Puzzle[$2]}
  Puzzle[$2]=$tmp
}


function Jumble
{ # Scramble the pieces at beginning of round.
  local i pos1 pos2

  for i in {1..100}
  do
    pos1=$(( $RANDOM % $SQUARES))
    pos2=$(( $RANDOM % $SQUARES ))
    swap $pos1 $pos2
  done
}


function PrintPuzzle
{
  local i1 i2 puzpos
  puzpos=0

  clear
  echo "Enter  quit  to exit."; echo   # Better that than Ctl-C.

  echo ",----.----.----.----."   # Top border.
  for i1 in {1..4}
  do
    for i2 in {1..4}
    do
      printf "| %2s " "${Puzzle[$puzpos]}"
      (( puzpos++ ))
    done
    echo "|"                     # Right-side border.
    test $i1 = 4 || echo "+----+----+----+----+"
  done
```

```
  echo "'----'----'----'----'"   # Bottom border.
}


function GetNum
{ # Test for valid input.
  local puznum garbage

  while true
  do
          echo "Moves: $moves" # Also counts invalid moves.
    read -p "Number to move: " puznum garbage
      if [ "$puznum" = "quit" ]; then echo; exit $E_PREMATURE_EXIT; fi
    test -z "$puznum" -o -n "${puznum//[0-9]/}" && continue
    test $puznum -gt 0 -a $puznum -lt $SQUARES && break
  done
  return $puznum
}


function GetPosFromNum
{ # $1 = puzzle-number
  local puzpos

  for puzpos in {0..15}
  do
    test "${Puzzle[$puzpos]}" = "$1" && break
  done
  return $puzpos
}


function Move
{ # $1=Puzzle-pos
  test $1 -gt 3 && test "${Puzzle[$(( $1 - 4 ))]}" = " "\
       && swap $1 $(( $1 - 4 )) && return 0
  test $(( $1%4 )) -ne 3 && test "${Puzzle[$(( $1 + 1 ))]}" = " "\
       && swap $1 $(( $1 + 1 )) && return 0
  test $1 -lt 12 && test "${Puzzle[$(( $1 + 4 ))]}" = " "\
       && swap $1 $(( $1 + 4 )) && return 0
  test $(( $1%4 )) -ne 0 && test "${Puzzle[$(( $1 - 1 ))]}" = " " &&\
       swap $1 $(( $1 - 1 )) && return 0
  return 1
}


function Solved
{
  local pos

  for pos in {0..14}
  do
    test "${Puzzle[$pos]}" = $(( $pos + 1 )) || return $FAIL
    # Check whether number in each square = square number.
  done
  return 0   # Successful solution.
}


################## MAIN () #######################{
moves=0
Jumble
```

```
while true   # Loop continuously until puzzle solved.
do
  echo; echo
  PrintPuzzle
  echo
  while true
  do
    GetNum
    puznum=$?
    GetPosFromNum $puznum
    puzpos=$?
    ((moves++))
    Move $puzpos && break
  done
  Solved && break
done

echo;echo
PrintPuzzle
echo; echo "BRAVO!"; echo

exit 0
###################################################}

#  Exercise:
#  --------
#  Rewrite the script to display the letters A - O,
#+ rather than the numbers 1 - 15.
```

Example A-48. ***The Towers of Hanoi, graphic version***

```
#! /bin/bash
# The Towers Of Hanoi
# Original script (hanoi.bash) copyright (C) 2000 Amit Singh.
# All Rights Reserved.
# http://hanoi.kernelthread.com

#  hanoi2.bash
#  Version 2.00: modded for ASCII-graphic display.
#  Version 2.01: fixed no command-line param bug.
#  Uses code contributed by Antonio Macchi,
#+ with heavy editing by ABS Guide author.
#  This variant falls under the original copyright, see above.
#  Used in ABS Guide with Amit Singh's permission (thanks!).


###   Variables && sanity check   ###

E_NOPARAM=86
E_BADPARAM=87            # Illegal no. of disks passed to script.
E_NOEXIT=88

DISKS=${1:-$E_NOPARAM}   # Must specify how many disks.
Moves=0

MWIDTH=7
MARGIN=2
# Arbitrary "magic" constants; work okay for relatively small # of disks.
# BASEWIDTH=51   # Original code.
```

```
let "basewidth = $MWIDTH * $DISKS + $MARGIN"       # "Base" beneath rods.
# Above "algorithm" could likely stand improvement.

###   Display variables   ###
let "disks1 = $DISKS - 1"
let "spaces1 = $DISKS"
let "spaces2 = 2 * $DISKS"

let "lastmove_t = $DISKS - 1"                      # Final move?


declare -a Rod1 Rod2 Rod3

###   #########################   ###


function repeat  {  # $1=char $2=number of repetitions
  local n           # Repeat-print a character.

  for (( n=0; n<$2; n++ )); do
    echo -n "$1"
  done
}

function FromRod  {
  local rod summit weight sequence

  while true; do
    rod=$1
    test ${rod/[^123]/} || continue

    sequence=$(echo $(seq 0 $disks1 | tac))
    for summit in $sequence; do
      eval weight=\${Rod${rod}[$summit]}
      test $weight -ne 0 &&
           { echo "$rod $summit $weight"; return; }
    done
  done
}


function ToRod  { # $1=previous (FromRod) weight
  local rod firstfree weight sequence

  while true; do
    rod=$2
    test ${rod/[^123]} || continue

    sequence=$(echo $(seq 0 $disks1 | tac))
    for firstfree in $sequence; do
      eval weight=\${Rod${rod}[$firstfree]}
      test $weight -gt 0 && { (( firstfree++ )); break; }
    done
    test $weight -gt $1 -o $firstfree = 0 &&
         { echo "$rod $firstfree"; return; }
  done
}


function PrintRods  {
  local disk rod empty fill sp sequence
```

```
  repeat " " $spaces1
  echo -n "|"
  repeat " " $spaces2
  echo -n "|"
  repeat " " $spaces2
  echo "|"

  sequence=$(echo $(seq 0 $disks1 | tac))
  for disk in $sequence; do
    for rod in {1..3}; do
      eval empty=$(( $DISKS - (Rod${rod}[$disk] / 2) ))
      eval fill=\${Rod${rod}[$disk]}
      repeat " " $empty
      test $fill -gt 0 && repeat "*" $fill || echo -n "|"
      repeat " " $empty
    done
    echo
  done
  repeat "=" $basewidth   # Print "base" beneath rods.
  echo
}


display ()
{
  echo
  PrintRods

  # Get rod-number, summit and weight
  first=( `FromRod $1` )
  eval Rod${first[0]}[${first[1]}]=0

  # Get rod-number and first-free position
  second=( `ToRod ${first[2]} $2` )
  eval Rod${second[0]}[${second[1]}]=${first[2]}


echo; echo; echo
if [ "${Rod3[lastmove_t]}" = 1 ]
then   # Last move? If yes, then display final position.
    echo "+  Final Position: $Moves moves"; echo
    PrintRods
  fi
}


# From here down, almost the same as original (hanoi.bash) script.

dohanoi() {   # Recursive function.
    case $1 in
    0)
        ;;
    *)
        dohanoi "$(($1-1))" $2 $4 $3
        if [ "$Moves" -ne 0 ]
        then
          echo "+  Position after move $Moves"
        fi
        ((Moves++))
        echo -n "   Next move will be:  "
        echo $2 "-->" $3
```

```
          display $2 $3
        dohanoi "$(($1-1))" $4 $3 $2
        ;;
    esac
}


setup_arrays ()
{
  local dim n elem

  let "dim1 = $1 - 1"
  elem=$dim1

  for n in $(seq 0 $dim1)
  do
   let "Rod1[$elem] = 2 * $n + 1"
   Rod2[$n]=0
   Rod3[$n]=0
   ((elem--))
  done
}


###   Main   ###

setup_arrays $DISKS
echo; echo "+  Start Position"

case $# in
    1) case $(($1>0)) in     # Must have at least one disk.
       1)
           disks=$1
           dohanoi $1 1 3 2
#          Total moves = 2^n - 1, where n = number of disks.
           echo
           exit 0;
           ;;
       *)
           echo "$0: Illegal value for number of disks";
           exit $E_BADPARAM;
           ;;
       esac
    ;;
    *)
       clear
       echo "usage: $0 N"
       echo "       Where \"N\" is the number of disks."
       exit $E_NOPARAM;
       ;;
esac

exit $E_NOEXIT   # Shouldn't exit here.

# Note:
# Redirect script output to a file, otherwise it scrolls off display.
```

Example A-49. *The Towers of Hanoi, alternate graphic version*

```
#! /bin/bash
# The Towers Of Hanoi
# Original script (hanoi.bash) copyright (C) 2000 Amit Singh.
# All Rights Reserved.
# http://hanoi.kernelthread.com

#  hanoi2.bash
#  Version 2: modded for ASCII-graphic display.
#  Uses code contributed by Antonio Macchi,
#+ with heavy editing by ABS Guide author.
#  This variant also falls under the original copyright, see above.
#  Used in ABS Guide with Amit Singh's permission (thanks!).


#   Variables   #
E_NOPARAM=86
E_BADPARAM=87   # Illegal no. of disks passed to script.
E_NOEXIT=88
DELAY=2         # Interval, in seconds, between moves. Change, if desired.
DISKS=$1
Moves=0

MWIDTH=7
MARGIN=2
# Arbitrary "magic" constants, work okay for relatively small # of disks.
# BASEWIDTH=51   # Original code.
let "basewidth = $MWIDTH * $DISKS + $MARGIN" # "Base" beneath rods.
# Above "algorithm" could likely stand improvement.

# Display variables.
let "disks1 = $DISKS - 1"
let "spaces1 = $DISKS"
let "spaces2 = 2 * $DISKS"

let "lastmove_t = $DISKS - 1"                # Final move?


declare -a Rod1 Rod2 Rod3

#################


function repeat  {  # $1=char $2=number of repetitions
  local n           # Repeat-print a character.

  for (( n=0; n<$2; n++ )); do
    echo -n "$1"
  done
}

function FromRod  {
  local rod summit weight sequence

  while true; do
    rod=$1
    test ${rod/[^123]/} || continue

    sequence=$(echo $(seq 0 $disks1 | tac))
    for summit in $sequence; do
      eval weight=\${Rod${rod}[$summit]}
      test $weight -ne 0 &&
           { echo "$rod $summit $weight"; return; }
```

```
    done
  done
}


function ToRod  { # $1=previous (FromRod) weight
  local rod firstfree weight sequence

  while true; do
    rod=$2
    test ${rod/[^123]} || continue

    sequence=$(echo $(seq 0 $disks1 | tac))
    for firstfree in $sequence; do
      eval weight=\${Rod${rod}[$firstfree]}
      test $weight -gt 0 && { (( firstfree++ )); break; }
    done
    test $weight -gt $1 -o $firstfree = 0 &&
         { echo "$rod $firstfree"; return; }
  done
}


function PrintRods  {
  local disk rod empty fill sp sequence

  tput cup 5 0

  repeat " " $spaces1
  echo -n "|"
  repeat " " $spaces2
  echo -n "|"
  repeat " " $spaces2
  echo "|"

  sequence=$(echo $(seq 0 $disks1 | tac))
  for disk in $sequence; do
    for rod in {1..3}; do
      eval empty=$(( $DISKS - (Rod${rod}[$disk] / 2) ))
      eval fill=\${Rod${rod}[$disk]}
      repeat " " $empty
      test $fill -gt 0 && repeat "*" $fill || echo -n "|"
      repeat " " $empty
    done
    echo
  done
  repeat "=" $basewidth   # Print "base" beneath rods.
  echo
}


display ()
{
  echo
  PrintRods

  # Get rod-number, summit and weight
  first=( `FromRod $1` )
  eval Rod${first[0]}[${first[1]}]=0

  # Get rod-number and first-free position
  second=( `ToRod ${first[2]} $2` )
```

```
  eval Rod${second[0]}[${second[1]}]=${first[2]}


  if [ "${Rod3[lastmove_t]}" = 1 ]
  then   # Last move? If yes, then display final position.
    tput cup 0 0
    echo; echo "+  Final Position: $Moves moves"
    PrintRods
  fi

  sleep $DELAY
}

# From here down, almost the same as original (hanoi.bash) script.

dohanoi() {   # Recursive function.
    case $1 in
    0)
        ;;
    *)
        dohanoi "$(($1-1))" $2 $4 $3
        if [ "$Moves" -ne 0 ]
        then
          tput cup 0 0
          echo; echo "+  Position after move $Moves"
        fi
        ((Moves++))
        echo -n "   Next move will be:  "
        echo $2 "-->" $3
        display $2 $3
        dohanoi "$(($1-1))" $4 $3 $2
        ;;
    esac
}

setup_arrays ()
{
  local dim n elem

  let "dim1 = $1 - 1"
  elem=$dim1

  for n in $(seq 0 $dim1)
  do
   let "Rod1[$elem] = 2 * $n + 1"
   Rod2[$n]=0
   Rod3[$n]=0
   ((elem--))
  done
}


###   Main   ###

trap "tput cnorm" 0
tput civis
clear

setup_arrays $DISKS

tput cup 0 0
echo; echo "+  Start Position"
```

```
case $# in
    1) case $(($1>0)) in     # Must have at least one disk.
       1)
           disks=$1
           dohanoi $1 1 3 2
#          Total moves = 2^n - 1, where n = # of disks.
           echo
           exit 0;
           ;;
       *)
           echo "$0: Illegal value for number of disks";
           exit $E_BADPARAM;
           ;;
       esac
    ;;
    *)
       echo "usage: $0 N"
       echo "       Where \"N\" is the number of disks."
       exit $E_NOPARAM;
       ;;
esac

exit $E_NOEXIT   # Shouldn't exit here.

#  Exercise:
#  --------
#  There is a minor bug in the script that causes the display of
#+ the next-to-last move to be skipped.
#+ Fix this.
```

Example A-50. An alternate version of the getopt-simple.sh script

```
#!/bin/bash
# UseGetOpt.sh

# Author: Peggy Russell <prusselltechgroup@gmail.com>

UseGetOpt () {
  declare inputOptions
  declare -r E_OPTERR=85
  declare -r ScriptName=${0##*/}
  declare -r ShortOpts="adf:hlt"
  declare -r LongOpts="aoption,debug,file:,help,log,test"

DoSomething () {
    echo "The function name is '${FUNCNAME}'"
    #  Recall that $FUNCNAME is an internal variable
    #+ holding the name of the function it is in.
  }

  inputOptions=$(getopt -o "${ShortOpts}" --long \
              "${LongOpts}" --name "${ScriptName}" -- "${@}")

  if [[ ($? -ne 0) || ($# -eq 0) ]]; then
    echo "Usage: ${ScriptName} [-dhlt] {OPTION...}"
    exit $E_OPTERR
  fi

  eval set -- "${inputOptions}"
```

```
# Only for educational purposes. Can be removed.
#-----------------------------------------------
echo "++ Test: Number of arguments: [$#]"
echo '++ Test: Looping through "$@"'
for a in "$@"; do
  echo "  ++ [$a]"
done
#-----------------------------------------------

while true; do
  case "${1}" in
    --aoption | -a)  # Argument found.
      echo "Option [$1]"
      ;;

    --debug | -d)    # Enable informational messages.
      echo "Option [$1] Debugging enabled"
      ;;

    --file | -f)     #  Check for optional argument.
      case "$2" in   #+ Double colon is optional argument.
        "")          #  Not there.
            echo "Option [$1] Use default"
            shift
            ;;

        *) # Got it
           echo "Option [$1] Using input [$2]"
           shift
           ;;

      esac
      DoSomething
      ;;

    --log | -l) # Enable Logging.
      echo "Option [$1] Logging enabled"
      ;;

    --test | -t) # Enable testing.
      echo "Option [$1] Testing enabled"
      ;;

    --help | -h)
      echo "Option [$1] Display help"
      break
      ;;

    --)   # Done! $# is argument number for "--", $@ is "--"
      echo "Option [$1] Dash Dash"
      break
      ;;

     *)
      echo "Major internal error!"
      exit 8
      ;;

  esac
  echo "Number of arguments: [$#]"
  shift
done
```

```
  shift
  # Only for educational purposes. Can be removed.
  #-----------------------------------------------------------------------
  echo "++ Test: Number of arguments after \"--\" is [$#] They are: [$@]"
  echo '++ Test: Looping through "$@"'
  for a in "$@"; do
    echo "  ++ [$a]"
  done
  #-----------------------------------------------------------------------

}

################################### M A I N ########################
#  If you remove "function UseGetOpt () {" and corresponding "}",
#+ you can uncomment the "exit 0" line below, and invoke this script
#+ with the various options from the command-line.
#-------------------------------------------------------------------
# exit 0

echo "Test 1"
UseGetOpt -f myfile one "two three" four

echo;echo "Test 2"
UseGetOpt -h

echo;echo "Test 3 - Short Options"
UseGetOpt -adltf myfile  anotherfile

echo;echo "Test 4 - Long Options"
UseGetOpt --aoption --debug --log --test --file myfile anotherfile

exit
```

Example A-51. The version of the *UseGetOpt.sh* example used in the Tab Expansion appendix

```
#!/bin/bash

#  UseGetOpt-2.sh
#  Modified version of the script for illustrating tab-expansion
#+ of command-line options.
#  See the "Introduction to Tab Expansion" appendix.

#  Possible options: -a -d -f -l -t -h
#+                   --aoption, --debug --file --log --test -- help --

#  Author of original script: Peggy Russell <prusselltechgroup@gmail.com>


# UseGetOpt () {
  declare inputOptions
  declare -r E_OPTERR=85
  declare -r ScriptName=${0##*/}
  declare -r ShortOpts="adf:hlt"
  declare -r LongOpts="aoption,debug,file:,help,log,test"

DoSomething () {
    echo "The function name is '${FUNCNAME}'"
  }

  inputOptions=$(getopt -o "${ShortOpts}" --long \
```

```
            "${LongOpts}" --name "${ScriptName}" -- "${@}")

if [[ ($? -ne 0) || ($# -eq 0) ]]; then
  echo "Usage: ${ScriptName} [-dhlt] {OPTION...}"
  exit $E_OPTERR
fi

eval set -- "${inputOptions}"


while true; do
  case "${1}" in
    --aoption | -a)  # Argument found.
      echo "Option [$1]"
      ;;

    --debug | -d)    # Enable informational messages.
      echo "Option [$1] Debugging enabled"
      ;;

    --file | -f)     #  Check for optional argument.
      case "$2" in   #+ Double colon is optional argument.
        "")          #  Not there.
            echo "Option [$1] Use default"
            shift
            ;;

        *) # Got it
           echo "Option [$1] Using input [$2]"
           shift
           ;;

      esac
      DoSomething
      ;;

    --log | -l) # Enable Logging.
      echo "Option [$1] Logging enabled"
      ;;

    --test | -t) # Enable testing.
      echo "Option [$1] Testing enabled"
      ;;

    --help | -h)
      echo "Option [$1] Display help"
      break
      ;;

    --)   # Done! $# is argument number for "--", $@ is "--"
      echo "Option [$1] Dash Dash"
      break
      ;;

     *)
      echo "Major internal error!"
      exit 8
      ;;

  esac
  echo "Number of arguments: [$#]"
  shift
```

```
  done

  shift

#  }

exit
```

Example A-52. Cycling through all the possible color backgrounds

```
#!/bin/bash

# show-all-colors.sh
# Displays all 256 possible background colors, using ANSI escape sequences.
# Author: Chetankumar Phulpagare
# Used in ABS Guide with permission.

T1=8
T2=6
T3=36
offset=0

for num1 in {0..7}
do {
   for num2 in {0,1}
       do {
          shownum=`echo "$offset + $T1 * ${num2} + $num1" | bc`
          echo -en "\E[0;48;5;${shownum}m color ${shownum} \E[0m"
          }
       done
   echo
   }
done

offset=16
for num1 in {0..5}
do {
   for num2 in {0..5}
       do {
          for num3 in {0..5}
              do {
                 shownum=`echo "$offset + $T2 * ${num3} \
                 + $num2 + $T3 * ${num1}" | bc`
                 echo -en "\E[0;48;5;${shownum}m color ${shownum} \E[0m"
                 }
               done
          echo
          }
       done
}
done

offset=232
for num1 in {0..23}
do {
   shownum=`expr $offset + $num1`
   echo -en "\E[0;48;5;${shownum}m ${shownum}\E[0m"
}
done

echo
```

Example A-53. Morse Code Practice

```
#!/bin/bash
# sam.sh, v. .01a
# Still Another Morse (code training script)
# With profuse apologies to Sam (F.B.) Morse.
# Author: Mendel Cooper
# License: GPL3
# Reldate: 05/25/11

# Morse code training script.
# Converts arguments to audible dots and dashes.
# Note: lowercase input only at this time.



# Get the wav files from the source tarball:
# http://bash.deta.in/abs-guide-latest.tar.bz2
DOT='soundfiles/dot.wav'
DASH='soundfiles/dash.wav'
# Maybe move soundfiles to /usr/local/sounds?

LETTERSPACE=300000  # Microseconds.
WORDSPACE=980000
# Nice and slow, for beginners. Maybe 5 wpm?

EXIT_MSG="May the Morse be with you!"
E_NOARGS=75         # No command-line args?



declare -A morse    # Associative array!
# ======================================= #
morse[a]="dot; dash"
morse[b]="dash; dot; dot; dot"
morse[c]="dash; dot; dash; dot"
morse[d]="dash; dot; dot"
morse[e]="dot"
morse[f]="dot; dot; dash; dot"
morse[g]="dash; dash; dot"
morse[h]="dot; dot; dot; dot"
morse[i]="dot; dot;"
morse[j]="dot; dash; dash; dash"
morse[k]="dash; dot; dash"
morse[l]="dot; dash; dot; dot"
morse[m]="dash; dash"
morse[n]="dash; dot"
morse[o]="dash; dash; dash"
morse[p]="dot; dash; dash; dot"
morse[q]="dash; dash; dot; dash"
morse[r]="dot; dash; dot"
morse[s]="dot; dot; dot"
morse[t]="dash"
morse[u]="dot; dot; dash"
morse[v]="dot; dot; dot; dash"
morse[w]="dot; dash; dash"
morse[x]="dash; dot; dot; dash"
morse[y]="dash; dot; dash; dash"
morse[z]="dash; dash; dot; dot"
morse[0]="dash; dash; dash; dash; dash"
morse[1]="dot; dash; dash; dash; dash"
morse[2]="dot; dot; dash; dash; dash"
```

```
morse[3]="dot; dot; dot; dash; dash"
morse[4]="dot; dot; dot; dot; dash"
morse[5]="dot; dot; dot; dot; dot"
morse[6]="dash; dot; dot; dot; dot"
morse[7]="dash; dash; dot; dot; dot"
morse[8]="dash; dash; dash; dot; dot"
morse[9]="dash; dash; dash; dash; dot"
# The following must be escaped or quoted.
morse[?]="dot; dot; dash; dash; dot; dot"
morse[.]="dot; dash; dot; dash; dot; dash"
morse[,]="dash; dash; dot; dot; dash; dash"
morse[/]="dash; dot; dot; dash; dot"
morse[\@]="dot; dash; dash; dot; dash; dot"
# ======================================= #

play_letter ()
{
  eval ${morse[$1]}   # Play dots, dashes from appropriate sound files.
  # Why is 'eval' necessary here?
  usleep $LETTERSPACE # Pause in between letters.
}

extract_letters ()
{                     # Slice string apart, letter by letter.
  local pos=0         # Starting at left end of string.
  local len=1         # One letter at a time.
  strlen=${#1}

  while [ $pos -lt $strlen ]
  do
    letter=${1:pos:len}
    #      ^^^^^^^^^^^^    See Chapter 10.1.
    play_letter $letter
    echo -n "*"       #    Mark letter just played.
    ((pos++))
  done
}

######### Play the sounds ############
dot()  { aplay "$DOT" 2&>/dev/null;  }
dash() { aplay "$DASH" 2&>/dev/null; }
######################################

no_args ()
{
    declare -a usage
    usage=( $0 word1 word2 ... )

    echo "Usage:"; echo
    echo ${usage[*]}
    for index in 0 1 2 3
    do
      extract_letters ${usage[index]}
      usleep $WORDSPACE
      echo -n " "     # Print space between words.
    done
#   echo "Usage: $0 word1 word2 ... "
    echo; echo
}


# int main()
```

```
# {

clear                  # Clear the terminal screen.
echo "            SAM"
echo "Still Another Morse code trainer"
echo "    Author: Mendel Cooper"
echo; echo;

if [ -z "$1" ]
then
  no_args
  echo; echo; echo "$EXIT_MSG"; echo
  exit $E_NOARGS
fi

echo; echo "$*"       # Print text that will be played.

until [ -z "$1" ]
do
  extract_letters $1
  shift           # On to next word.
  usleep $WORDSPACE
  echo -n " "     # Print space between words.
done

echo; echo; echo "$EXIT_MSG"; echo

exit 0
# }

#  Exercises:
#  ---------
#  1) Have the script accept either lowercase or uppercase words
#+    as arguments. Hint: Use 'tr' . . .
#  2) Have the script optionally accept input from a text file.
```

Example A-54. Base64 encoding/decoding

```
#!/bin/bash
# base64.sh: Bash implementation of Base64 encoding and decoding.
#
# Copyright (c) 2011 vladz <vladz@devzero.fr>
# Used in ABSG with permission (thanks!).
#
#  Encode or decode original Base64 (and also Base64url)
#+ from STDIN to STDOUT.
#
#    Usage:
#
#    Encode
#    $ ./base64.sh < binary-file > binary-file.base64
#    Decode
#    $ ./base64.sh -d < binary-file.base64 > binary-file
#
# Reference:
#
#    [1]  RFC4648 - "The Base16, Base32, and Base64 Data Encodings"
#         http://tools.ietf.org/html/rfc4648#section-5


# The base64_charset[] array contains entire base64 charset,
```

```
# and additionally the character "=" ...
base64_charset=( {A..Z} {a..z} {0..9} + / = )
                # Nice illustration of brace expansion.

#  Uncomment the ### line below to use base64url encoding instead of
#+ original base64.
### base64_charset=( {A..Z} {a..z} {0..9} - _ = )

#  Output text width when encoding
#+ (64 characters, just like openssl output).
text_width=64

function display_base64_char {
#  Convert a 6-bit number (between 0 and 63) into its corresponding values
#+ in Base64, then display the result with the specified text width.
  printf "${base64_charset[$1]}"; (( width++ ))
  (( width % text_width == 0 )) && printf "\n"
}

function encode_base64 {
# Encode three 8-bit hexadecimal codes into four 6-bit numbers.
  #    We need two local int array variables:
  #    c8[]: to store the codes of the 8-bit characters to encode
  #    c6[]: to store the corresponding encoded values on 6-bit
  declare -a -i c8 c6

  #  Convert hexadecimal to decimal.
  c8=( $(printf "ibase=16; ${1:0:2}\n${1:2:2}\n${1:4:2}\n" | bc) )

  #  Let's play with bitwise operators
  #+ (3x8-bit into 4x6-bits conversion).
  (( c6[0] = c8[0] >> 2 ))
  (( c6[1] = ((c8[0] &  3) << 4) | (c8[1] >> 4) ))

  # The following operations depend on the c8 element number.
  case ${#c8[*]} in
    3) (( c6[2] = ((c8[1] & 15) << 2) | (c8[2] >> 6) ))
       (( c6[3] = c8[2] & 63 )) ;;
    2) (( c6[2] = (c8[1] & 15) << 2 ))
       (( c6[3] = 64 )) ;;
    1) (( c6[2] = c6[3] = 64 )) ;;
  esac

  for char in ${c6[@]}; do
    display_base64_char ${char}
  done
}

function decode_base64 {
# Decode four base64 characters into three hexadecimal ASCII characters.
  #  c8[]: to store the codes of the 8-bit characters
  #  c6[]: to store the corresponding Base64 values on 6-bit
  declare -a -i c8 c6

  # Find decimal value corresponding to the current base64 character.
  for current_char in ${1:0:1} ${1:1:1} ${1:2:1} ${1:3:1}; do
     [ "${current_char}" = "=" ] && break

     position=0
     while [ "${current_char}" != "${base64_charset[${position}]}" ]; do
        (( position++ ))
     done
```

```
    c6=( ${c6[*]} ${position} )
  done

  #  Let's play with bitwise operators
  #+ (4x8-bit into 3x6-bits conversion).
  (( c8[0] = (c6[0] << 2) | (c6[1] >> 4) ))

  # The next operations depends on the c6 elements number.
  case ${#c6[*]} in
    3) (( c8[1] = ( (c6[1] & 15) << 4) | (c6[2] >> 2) ))
       (( c8[2] = (c6[2] & 3) << 6 )); unset c8[2] ;;
    4) (( c8[1] = ( (c6[1] & 15) << 4) | (c6[2] >> 2) ))
       (( c8[2] = ( (c6[2] &  3) << 6) |  c6[3] )) ;;
  esac

  for char in ${c8[*]}; do
    printf "\x$(printf "%x" ${char})"
  done
}


# main ()

if [ "$1" = "-d" ]; then   # decode

  # Reformat STDIN in pseudo 4x6-bit groups.
  content=$(cat - | tr -d "\n" | sed -r "s/(.{4})/\1 /g")

  for chars in ${content}; do decode_base64 ${chars}; done

else
  # Make a hexdump of stdin and reformat in 3-byte groups.
  content=$(cat - | xxd -ps -u | sed -r "s/(\w{6})/\1 /g" |
            tr -d "\n")

  for chars in ${content}; do encode_base64 ${chars}; done

  echo

fi
```

Example A-55. Inserting text in a file using *sed*

```
#!/bin/bash
#  Prepends a string at a specified line
#+ in files with names ending in "sample"
#+ in the current working directory.
#  000000000000000000000000000000000000
#  This script overwrites files!
#  Be careful running it in a directory
#+ where you have important files!!!
#  000000000000000000000000000000000000

#  Create a couple of files to operate on ...
#  01sample
#  02sample
#  ... etc.
#  These files must not be empty, else the prepend will not work.

lineno=1            # Append at line 1 (prepend).
```

```
filespec="*sample"  # Filename pattern to operate on.

string=$(whoami)    # Will set your username as string to insert.
                    # It could just as easily be any other string.

for file in $filespec # Specify which files to alter.
do #        ^^^^^^^^^
 sed -i ""$lineno"i "$string"" $file
#    ^^ -i option edits files in-place.
#                 ^ Insert (i) command.
 echo ""$file" altered!"
done

echo "Warning: files possibly clobbered!"

exit 0

# Exercise:
# Add error checking to this script.
# It needs it badly.
```

Example A-56. The Gronsfeld Cipher

```
#!/bin/bash
# gronsfeld.bash

# License: GPL3
# Reldate 06/23/11

#  This is an implementation of the Gronsfeld Cipher.
#  It's essentially a stripped-down variant of the
#+ polyalphabetic Vigenère Tableau, but with only 10 alphabets.
#  The classic Gronsfeld has a numeric sequence as the key word,
#+ but here we substitute a letter string, for ease of use.
#  Allegedly, this cipher was invented by the eponymous Count Gronsfeld
#+ in the 17th Century. It was at one time considered to be unbreakable.
#  Note that this is ###not### a secure cipher by modern standards.

#  Global Variables  #
Enc_suffix="29379"   #  Encrypted text output with this 5-digit suffix.
                     #  This functions as a decryption flag,
                     #+ and when used to generate passwords adds security.
Default_key="gronsfeldk"
                     #  The script uses this if key not entered below
                     #  (at "Keychain").
                     #  Change the above two values frequently
                     #+ for added security.

GROUPLEN=5           #  Output in groups of 5 letters, per tradition.
alpha1=( abcdefghijklmnopqrstuvwxyz )
alpha2=( {A..Z} )    #  Output in all caps, per tradition.
                     #  Use   alpha2=( {a..z} )   for password generator.
wraplen=26           #  Wrap around if past end of alphabet.
dflag=               #  Decrypt flag (set if $Enc_suffix present).
E_NOARGS=76          #  Missing command-line args?
DEBUG=77             #  Debugging flag.
declare -a offsets   #  This array holds the numeric shift values for
                     #+ encryption/decryption.

########Keychain#########
key=  ### Put key here!!!
```

```
      # 10 characters!
#########################



# Function
: ()
{ # Encrypt or decrypt, depending on whether $dflag is set.
  # Why ": ()" as a function name? Just to prove that it can be done.

  local idx keydx mlen off1 shft
  local plaintext="$1"
  local mlen=${#plaintext}

for (( idx=0; idx<$mlen; idx++ ))
do
  let "keydx = $idx % $keylen"
  shft=${offsets[keydx]}

  if [ -n "$dflag" ]
  then                  # Decrypt!
    let "off1 = $(expr index "${alpha1[*]}" ${plaintext:idx:1}) - $shft"
    # Shift backward to decrypt.
  else                  # Encrypt!
    let "off1 = $(expr index "${alpha1[*]}" ${plaintext:idx:1}) + $shft"
    # Shift forward to encrypt.
    test $(( $idx % $GROUPLEN)) = 0 && echo -n " "  # Groups of 5 letters.
    #  Comment out above line for output as a string without whitespace,
    #+ for example, if using the script as a password generator.
  fi

  ((off1--))   # Normalize. Why is this necessary?

      if [ $off1 -lt 0 ]
      then     # Catch negative indices.
        let "off1 += $wraplen"
      fi

  ((off1 %= $wraplen))   # Wrap around if past end of alphabet.

  echo -n "${alpha2[off1]}"

done

  if [ -z "$dflag" ]
  then
    echo " $Enc_suffix"
#   echo "$Enc_suffix"  # For password generator.
  else
    echo
  fi
} # End encrypt/decrypt function.



# int main () {

# Check for command-line args.
if [ -z "$1" ]
then
   echo "Usage: $0 TEXT TO ENCODE/DECODE"
   exit $E_NOARGS
```

```
fi

if [ ${!#} == "$Enc_suffix" ]
#    ^^^^^ Final command-line arg.
then
  dflag=ON
  echo -n "+"           # Flag decrypted text with a "+" for easy ID.
fi

if [ -z "$key" ]
then
  key="$Default_key"    # "gronsfeldk" per above.
fi

keylen=${#key}

for (( idx=0; idx<$keylen; idx++ ))
do  # Calculate shift values for encryption/decryption.
  offsets[idx]=$(expr index "${alpha1[*]}" ${key:idx:1})   # Normalize.
  ((offsets[idx]--))  #  Necessary because "expr index" starts at 1,
                      #+ whereas array count starts at 0.
  # Generate array of numerical offsets corresponding to the key.
  # There are simpler ways to accomplish this.
done

args=$(echo "$*" | sed -e 's/ //g' | tr A-Z a-z | sed -e 's/[0-9]//g')
# Remove whitespace and digits from command-line args.
# Can modify to also remove punctuation characters, if desired.

         # Debug:
         # echo "$args"; exit $DEBUG

: "$args"               # Call the function named ":".
# : is a null operator, except . . . when it's a function name!

exit $?    # } End-of-script


#   **************************************************************   #
#   This script can function as a  password generator,
#+  with several minor mods, see above.
#   That would allow an easy-to-remember password, even the word
#+ "password" itself, which encrypts to vrgfotvo29379
#+  a fairly secure password not susceptible to a dictionary attack.
#   Or, you could use your own name (surely that's easy to remember!).
#   For example, Bozo Bozeman encrypts to hfnbttdppkt29379.
#   **************************************************************   #
```

Example A-57. Bingo Number Generator

```
#!/bin/bash
# bingo.sh
# Bingo number generator
# Reldate 20Aug12, License: Public Domain

#######################################################################
# This script generates bingo numbers.
# Hitting a key generates a new number.
# Hitting 'q' terminates the script.
# In a given run of the script, there will be no duplicate numbers.
# When the script terminates, it prints a log of the numbers generated.
```

```
######################################################################

MIN=1       # Lowest allowable bingo number.
MAX=75      # Highest allowable bingo number.
COLS=15     # Numbers in each column (B I N G O).
SINGLE_DIGIT_MAX=9

declare -a Numbers
Prefix=(B I N G O)

initialize_Numbers ()
{  # Zero them out to start.
   # They'll be incremented if chosen.
   local index=0
   until [ "$index" -gt $MAX ]
   do
     Numbers[index]=0
     ((index++))
   done

   Numbers[0]=1   # Flag zero, so it won't be selected.
}


generate_number ()
{
   local number

   while [ 1 ]
   do
     let "number = $(expr $RANDOM % $MAX)"
     if [ ${Numbers[number]} -eq 0 ]    # Number not yet called.
     then
       let "Numbers[number]+=1"         # Flag it in the array.
       break                            # And terminate loop.
     fi   # Else if already called, loop and generate another number.
   done
   # Exercise: Rewrite this more elegantly as an until-loop.

   return $number
}


print_numbers_called ()
{   # Print out the called number log in neat columns.
    # echo ${Numbers[@]}

local pre2=0                #  Prefix a zero, so columns will align
                            #+ on single-digit numbers.

echo "Number Stats"

for (( index=1; index<=MAX; index++))
do
  count=${Numbers[index]}
  let "t = $index - 1"      # Normalize, since array begins with index 0.
  let "column = $(expr $t / $COLS)"
  pre=${Prefix[column]}
# echo -n "${Prefix[column]} "

if [ $(expr $t % $COLS) -eq 0 ]
then
```

```
  echo   # Newline at end of row.
fi

  if [ "$index" -gt $SINGLE_DIGIT_MAX ]  # Check for single-digit number.
  then
    echo -n "$pre$index#$count "
  else    # Prefix a zero.
    echo -n "$pre$pre2$index#$count "
  fi

done
}



# main () {
RANDOM=$$   # Seed random number generator.

initialize_Numbers   # Zero out the number tracking array.

clear
echo "Bingo Number Caller"; echo

while [[ "$key" != "q" ]]   # Main loop.
do
  read -s -n1 -p "Hit a key for the next number [q to exit] " key
  # Usually 'q' exits, but not always.
  # Can always hit Ctl-C if q fails.
  echo

  generate_number; new_number=$?

  let "column = $(expr $new_number / $COLS)"
  echo -n "${Prefix[column]} "   # B-I-N-G-O

  echo $new_number
done

echo; echo

# Game over ...
print_numbers_called
echo; echo "[#0 = not called . . . #1 = called]"

echo

exit 0
# }


# Certainly, this script could stand some improvement.
#See also the author's Instructable:
#www.instructables.com/id/Binguino-An-Arduino-based-Bingo-Number-Generato/
```

To end this section, a review of the basics . . . and more.

Example A-58. Basics Reviewed

```
#!/bin/bash
# basics-reviewed.bash

```

```
# File extension == *.bash == specific to Bash

#   Copyright (c) Michael S. Zick, 2003; All rights reserved.
#   License: Use in any form, for any purpose.
#   Revision: $ID$
#
#              Edited for layout by M.C.
#   (author of the "Advanced Bash Scripting Guide")
#   Fixes and updates (04/08) by Cliff Bamford.


#  This script tested under Bash versions 2.04, 2.05a and 2.05b.
#  It may not work with earlier versions.
#  This demonstration script generates one --intentional--
#+ "command not found" error message. See line 436.

#  The current Bash maintainer, Chet Ramey, has fixed the items noted
#+ for later versions of Bash.



        ###-------------------------------------------###
        ###  Pipe the output of this script to 'more' ###
        ###+ else it will scroll off the page.        ###
        ###                                           ###
        ###  You may also redirect its output         ###
        ###+ to a file for examination.               ###
        ###-------------------------------------------###



#  Most of the following points are described at length in
#+ the text of the foregoing "Advanced Bash Scripting Guide."
#  This demonstration script is mostly just a reorganized presentation.
#      -- msz

# Variables are not typed unless otherwise specified.

#  Variables are named. Names must contain a non-digit.
#  File descriptor names (as in, for example: 2>&1)
#+ contain ONLY digits.

# Parameters and Bash array elements are numbered.
# (Parameters are very similar to Bash arrays.)

# A variable name may be undefined (null reference).
unset VarNull

# A variable name may be defined but empty (null contents).
VarEmpty=''                         # Two, adjacent, single quotes.

# A variable name may be defined and non-empty.
VarSomething='Literal'

# A variable may contain:
#   * A whole number as a signed 32-bit (or larger) integer
#   * A string
# A variable may also be an array.

#  A string may contain embedded blanks and may be treated
#+ as if it where a function name with optional arguments.
```

```
#  The names of variables and the names of functions
#+ are in different namespaces.


#  A variable may be defined as a Bash array either explicitly or
#+ implicitly by the syntax of the assignment statement.
#  Explicit:
declare -a ArrayVar



# The echo command is a builtin.
echo $VarSomething

# The printf command is a builtin.
# Translate %s as: String-Format
printf %s $VarSomething         # No linebreak specified, none output.
echo                            # Default, only linebreak output.




# The Bash parser word breaks on whitespace.
# Whitespace, or the lack of it is significant.
# (This holds true in general; there are, of course, exceptions.)




# Translate the DOLLAR_SIGN character as: Content-Of.

# Extended-Syntax way of writing Content-Of:
echo ${VarSomething}

#  The ${ ... } Extended-Syntax allows more than just the variable
#+ name to be specified.
#  In general, $VarSomething can always be written as: ${VarSomething}.

# Call this script with arguments to see the following in action.



#  Outside of double-quotes, the special characters @ and *
#+ specify identical behavior.
#  May be pronounced as: All-Elements-Of.

#  Without specification of a name, they refer to the
#+ pre-defined parameter Bash-Array.



# Glob-Pattern references
echo $*                         # All parameters to script or function
echo ${*}                       # Same

# Bash disables filename expansion for Glob-Patterns.
# Only character matching is active.


# All-Elements-Of references
echo $@                         # Same as above
echo ${@}                       # Same as above
```

```
#  Within double-quotes, the behavior of Glob-Pattern references
#+ depends on the setting of IFS (Input Field Separator).
#  Within double-quotes, All-Elements-Of references behave the same.


#  Specifying only the name of a variable holding a string refers
#+ to all elements (characters) of a string.


#  To specify an element (character) of a string,
#+ the Extended-Syntax reference notation (see below) MAY be used.




#  Specifying only the name of a Bash array references
#+ the subscript zero element,
#+ NOT the FIRST DEFINED nor the FIRST WITH CONTENTS element.

#  Additional qualification is needed to reference other elements,
#+ which means that the reference MUST be written in Extended-Syntax.
#  The general form is: ${name[subscript]}.

#  The string forms may also be used: ${name:subscript}
#+ for Bash-Arrays when referencing the subscript zero element.


# Bash-Arrays are implemented internally as linked lists,
#+ not as a fixed area of storage as in some programming languages.


#   Characteristics of Bash arrays (Bash-Arrays):
#   --------------------------------------------

#   If not otherwise specified, Bash-Array subscripts begin with
#+  subscript number zero. Literally: [0]
#   This is called zero-based indexing.
###
#   If not otherwise specified, Bash-Arrays are subscript packed
#+  (sequential subscripts without subscript gaps).
###
#   Negative subscripts are not allowed.
###
#   Elements of a Bash-Array need not all be of the same type.
###
#   Elements of a Bash-Array may be undefined (null reference).
#       That is, a Bash-Array may be "subscript sparse."
###
#   Elements of a Bash-Array may be defined and empty (null contents).
###
#   Elements of a Bash-Array may contain:
#     * A whole number as a signed 32-bit (or larger) integer
#     * A string
#     * A string formated so that it appears to be a function name
#     + with optional arguments
###
#   Defined elements of a Bash-Array may be undefined (unset).
#       That is, a subscript packed Bash-Array may be changed
```

```
#   +   into a subscript sparse Bash-Array.
###
#   Elements may be added to a Bash-Array by defining an element
#+  not previously defined.
###
# For these reasons, I have been calling them "Bash-Arrays".
# I'll return to the generic term "array" from now on.
#     -- msz


echo "========================================================="

#  Lines 202 - 334 supplied by Cliff Bamford. (Thanks!)
#  Demo --- Interaction with Arrays, quoting, IFS, echo, * and @   ---
#+ all affect how things work

ArrayVar[0]='zero'                    # 0 normal
ArrayVar[1]=one                       # 1 unquoted literal
ArrayVar[2]='two'                     # 2 normal
ArrayVar[3]='three'                   # 3 normal
ArrayVar[4]='I am four'               # 4 normal with spaces
ArrayVar[5]='five'                    # 5 normal
unset ArrayVar[6]                     # 6 undefined
ArrayValue[7]='seven'                 # 7 normal
ArrayValue[8]=''                      # 8 defined but empty
ArrayValue[9]='nine'                  # 9 normal


echo '--- Here is the array we are using for this test'
echo
echo "ArrayVar[0]='zero'             # 0 normal"
echo "ArrayVar[1]=one                # 1 unquoted literal"
echo "ArrayVar[2]='two'              # 2 normal"
echo "ArrayVar[3]='three'            # 3 normal"
echo "ArrayVar[4]='I am four'        # 4 normal with spaces"
echo "ArrayVar[5]='five'             # 5 normal"
echo "unset ArrayVar[6]              # 6 undefined"
echo "ArrayValue[7]='seven'          # 7 normal"
echo "ArrayValue[8]=''               # 8 defined but empty"
echo "ArrayValue[9]='nine'           # 9 normal"
echo


echo
echo '---Case0: No double-quotes, Default IFS of space,tab,newline ---'
IFS=$'\x20'$'\x09'$'\x0A'            # In exactly this order.
echo 'Here is: printf %q {${ArrayVar[*]}'
printf %q ${ArrayVar[*]}
echo
echo 'Here is: printf %q {${ArrayVar[@]}'
printf %q ${ArrayVar[@]}
echo
echo 'Here is: echo ${ArrayVar[*]}'
echo  ${ArrayVar[@]}
echo 'Here is: echo {${ArrayVar[@]}'
echo ${ArrayVar[@]}

echo
echo '---Case1: Within double-quotes - Default IFS of space-tab-
newline ---'
IFS=$'\x20'$'\x09'$'\x0A'           #  These three bytes,
echo 'Here is: printf %q "{${ArrayVar[*]}"'
```

```
printf %q "${ArrayVar[*]}"
echo
echo 'Here is: printf %q "{${ArrayVar[@]}"'
printf %q "${ArrayVar[@]}"
echo
echo 'Here is: echo "${ArrayVar[*]}"'
echo  "${ArrayVar[@]}"
echo 'Here is: echo "{${ArrayVar[@]}"'
echo "${ArrayVar[@]}"

echo
echo '---Case2: Within double-quotes - IFS is q'
IFS='q'
echo 'Here is: printf %q "{${ArrayVar[*]}"'
printf %q "${ArrayVar[*]}"
echo
echo 'Here is: printf %q "{${ArrayVar[@]}"'
printf %q "${ArrayVar[@]}"
echo
echo 'Here is: echo "${ArrayVar[*]}"'
echo  "${ArrayVar[@]}"
echo 'Here is: echo "{${ArrayVar[@]}"'
echo "${ArrayVar[@]}"

echo
echo '---Case3: Within double-quotes - IFS is ^'
IFS='^'
echo 'Here is: printf %q "{${ArrayVar[*]}"'
printf %q "${ArrayVar[*]}"
echo
echo 'Here is: printf %q "{${ArrayVar[@]}"'
printf %q "${ArrayVar[@]}"
echo
echo 'Here is: echo "${ArrayVar[*]}"'
echo  "${ArrayVar[@]}"
echo 'Here is: echo "{${ArrayVar[@]}"'
echo "${ArrayVar[@]}"

echo
echo '---Case4: Within double-quotes - IFS is ^ followed by
space,tab,newline'
IFS=$'^'$'\x20'$'\x09'$'\x0A'       # ^ + space tab newline
echo 'Here is: printf %q "{${ArrayVar[*]}"'
printf %q "${ArrayVar[*]}"
echo
echo 'Here is: printf %q "{${ArrayVar[@]}"'
printf %q "${ArrayVar[@]}"
echo
echo 'Here is: echo "${ArrayVar[*]}"'
echo  "${ArrayVar[@]}"
echo 'Here is: echo "{${ArrayVar[@]}"'
echo "${ArrayVar[@]}"

echo
echo '---Case6: Within double-quotes - IFS set and empty '
IFS=''
echo 'Here is: printf %q "{${ArrayVar[*]}"'
printf %q "${ArrayVar[*]}"
echo
echo 'Here is: printf %q "{${ArrayVar[@]}"'
printf %q "${ArrayVar[@]}"
echo
```

```
echo 'Here is: echo "${ArrayVar[*]}"'
echo  "${ArrayVar[@]}"
echo 'Here is: echo "{${ArrayVar[@]}"'
echo "${ArrayVar[@]}"

echo
echo '---Case7: Within double-quotes - IFS is unset'
unset IFS
echo 'Here is: printf %q "{${ArrayVar[*]}"'
printf %q "${ArrayVar[*]}"
echo
echo 'Here is: printf %q "{${ArrayVar[@]}"'
printf %q "${ArrayVar[@]}"
echo
echo 'Here is: echo "${ArrayVar[*]}"'
echo  "${ArrayVar[@]}"
echo 'Here is: echo "{${ArrayVar[@]}"'
echo "${ArrayVar[@]}"

echo
echo '---End of Cases---'
echo "========================================================"; echo



# Put IFS back to the default.
# Default is exactly these three bytes.
IFS=$'\x20'$'\x09'$'\x0A'           # In exactly this order.

# Interpretation of the above outputs:
#   A Glob-Pattern is I/O; the setting of IFS matters.
###
#   An All-Elements-Of does not consider IFS settings.
###
#   Note the different output using the echo command and the
#+  quoted format operator of the printf command.


#  Recall:
#   Parameters are similar to arrays and have the similar behaviors.
###
#  The above examples demonstrate the possible variations.
#  To retain the shape of a sparse array, additional script
#+ programming is required.
###
#  The source code of Bash has a routine to output the
#+ [subscript]=value   array assignment format.
#  As of version 2.05b, that routine is not used,
#+ but that might change in future releases.



# The length of a string, measured in non-null elements (characters):
echo
echo '- - Non-quoted references - -'
echo 'Non-Null character count: '${#VarSomething}' characters.'

# test='Lit'$'\x00''eral'           # $'\x00' is a null character.
# echo ${#test}                     # See that?
```

```
#  The length of an array, measured in defined elements,
#+ including null content elements.
echo
echo 'Defined content count: '${#ArrayVar[@]}' elements.'
# That is NOT the maximum subscript (4).
# That is NOT the range of the subscripts (1 . . 4 inclusive).
# It IS the length of the linked list.
###
#  Both the maximum subscript and the range of the subscripts may
#+ be found with additional script programming.

# The length of a string, measured in non-null elements (characters):
echo
echo '- - Quoted, Glob-Pattern references - -'
echo 'Non-Null character count: '"${#VarSomething}"' characters.'

#  The length of an array, measured in defined elements,
#+ including null-content elements.
echo
echo 'Defined element count: '"${#ArrayVar[*]}"' elements.'

#  Interpretation: Substitution does not effect the ${# ... } operation.
#  Suggestion:
#  Always use the All-Elements-Of character
#+ if that is what is intended (independence from IFS).



#  Define a simple function.
#  I include an underscore in the name
#+ to make it distinctive in the examples below.
###
#  Bash separates variable names and function names
#+ in different namespaces.
#  The Mark-One eyeball isn't that advanced.
###
_simple() {
    echo -n 'SimpleFunc'$@          #  Newlines are swallowed in
}                                   #+ result returned in any case.


# The ( ... ) notation invokes a command or function.
# The $( ... ) notation is pronounced: Result-Of.


# Invoke the function _simple
echo
echo '- - Output of function _simple - -'
_simple                             # Try passing arguments.
echo
# or
(_simple)                           # Try passing arguments.
echo

echo '- Is there a variable of that name? -'
echo $_simple not defined           # No variable by that name.

# Invoke the result of function _simple (Error msg intended)

###
$(_simple)                          # Gives an error message:
#                          line 436: SimpleFunc: command not found
```

```
#                           ---------------------------------------

echo
###

#  The first word of the result of function _simple
#+ is neither a valid Bash command nor the name of a defined function.
###
# This demonstrates that the output of _simple is subject to evaluation.
###
# Interpretation:
#   A function can be used to generate in-line Bash commands.


# A simple function where the first word of result IS a bash command:
###
_print() {
    echo -n 'printf %q '$@
}

echo '- - Outputs of function _print - -'
_print parm1 parm2                  # An Output NOT A Command.
echo

$(_print parm1 parm2)               #  Executes: printf %q parm1 parm2
                                    #  See above IFS examples for the
                                    #+ various possibilities.
echo

$(_print $VarSomething)             # The predictable result.
echo



# Function variables
# ------------------

echo
echo '- - Function variables - -'
# A variable may represent a signed integer, a string or an array.
# A string may be used like a function name with optional arguments.

# set -vx                           #  Enable if desired
declare -f funcVar                  #+ in namespace of functions

funcVar=_print                      # Contains name of function.
$funcVar parm1                      # Same as _print at this point.
echo

funcVar=$(_print )                  # Contains result of function.
$funcVar                            # No input, No output.
$funcVar $VarSomething              # The predictable result.
echo

funcVar=$(_print $VarSomething)     #  $VarSomething replaced HERE.
$funcVar                            #  The expansion is part of the
echo                                #+ variable contents.

funcVar="$(_print $VarSomething)"   #  $VarSomething replaced HERE.
$funcVar                            #  The expansion is part of the
echo                                #+ variable contents.
```

```
#  The difference between the unquoted and the double-quoted versions
#+ above can be seen in the "protect_literal.sh" example.
#  The first case above is processed as two, unquoted, Bash-Words.
#  The second case above is processed as one, quoted, Bash-Word.




# Delayed replacement
# -------------------

echo
echo '- - Delayed replacement - -'
funcVar="$(_print '$VarSomething')" # No replacement, single Bash-Word.
eval $funcVar                       # $VarSomething replaced HERE.
echo

VarSomething='NewThing'
eval $funcVar                       # $VarSomething replaced HERE.
echo

# Restore the original setting trashed above.
VarSomething=Literal

#  There are a pair of functions demonstrated in the
#+ "protect_literal.sh" and "unprotect_literal.sh" examples.
#  These are general purpose functions for delayed replacement literals
#+ containing variables.




# REVIEW:
# ------

#  A string can be considered a Classic-Array of elements (characters).
#  A string operation applies to all elements (characters) of the string
#+ (in concept, anyway).
###
#  The notation: ${array_name[@]} represents all elements of the
#+ Bash-Array: array_name.
###
#  The Extended-Syntax string operations can be applied to all
#+ elements of an array.
###
#  This may be thought of as a For-Each operation on a vector of strings.
###
#  Parameters are similar to an array.
#  The initialization of a parameter array for a script
#+ and a parameter array for a function only differ
#+ in the initialization of ${0}, which never changes its setting.
###
#  Subscript zero of the script's parameter array contains
#+ the name of the script.
###
#  Subscript zero of a function's parameter array DOES NOT contain
#+ the name of the function.
#  The name of the current function is accessed by the $FUNCNAME variable.
###
#  A quick, review list follows (quick, not short).
```

```
echo
echo '- - Test (but not change) - -'
echo '- null reference -'
echo -n ${VarNull-'NotSet'}' '          # NotSet
echo ${VarNull}                         # NewLine only
echo -n ${VarNull:-'NotSet'}' '         # NotSet
echo ${VarNull}                         # Newline only

echo '- null contents -'
echo -n ${VarEmpty-'Empty'}' '          # Only the space
echo ${VarEmpty}                        # Newline only
echo -n ${VarEmpty:-'Empty'}' '         # Empty
echo ${VarEmpty}                        # Newline only

echo '- contents -'
echo ${VarSomething-'Content'}          # Literal
echo ${VarSomething:-'Content'}         # Literal

echo '- Sparse Array -'
echo ${ArrayVar[@]-'not set'}

# ASCII-Art time
# State     Y==yes, N==no
#           -       :-
# Unset     Y       Y       ${# ... } == 0
# Empty     N       Y       ${# ... } == 0
# Contents  N       N       ${# ... } > 0

#  Either the first and/or the second part of the tests
#+ may be a command or a function invocation string.
echo
echo '- - Test 1 for undefined - -'
declare -i t
_decT() {
    t=$t-1
}

# Null reference, set: t == -1
t=${#VarNull}                           # Results in zero.
${VarNull- _decT }                      # Function executes, t now -1.
echo $t

# Null contents, set: t == 0
t=${#VarEmpty}                          # Results in zero.
${VarEmpty- _decT }                     # _decT function NOT executed.
echo $t

# Contents, set: t == number of non-null characters
VarSomething='_simple'                  # Set to valid function name.
t=${#VarSomething}                      # non-zero length
${VarSomething- _decT }                 # Function _simple executed.
echo $t                                 # Note the Append-To action.

# Exercise: clean up that example.
unset t
unset _decT
VarSomething=Literal

echo
echo '- - Test and Change - -'
echo '- Assignment if null reference -'
echo -n ${VarNull='NotSet'}' '          # NotSet NotSet
```

```
echo ${VarNull}
unset VarNull

echo '- Assignment if null reference -'
echo -n ${VarNull:='NotSet'}' '         # NotSet NotSet
echo ${VarNull}
unset VarNull

echo '- No assignment if null contents -'
echo -n ${VarEmpty='Empty'}' '          # Space only
echo ${VarEmpty}
VarEmpty=''

echo '- Assignment if null contents -'
echo -n ${VarEmpty:='Empty'}' '         # Empty Empty
echo ${VarEmpty}
VarEmpty=''

echo '- No change if already has contents -'
echo ${VarSomething='Content'}          # Literal
echo ${VarSomething:='Content'}         # Literal


# "Subscript sparse" Bash-Arrays
###
#  Bash-Arrays are subscript packed, beginning with
#+ subscript zero unless otherwise specified.
###
#  The initialization of ArrayVar was one way
#+ to "otherwise specify".  Here is the other way:
###
echo
declare -a ArraySparse
ArraySparse=( [1]=one [2]='' [4]='four' )
# [0]=null reference, [2]=null content, [3]=null reference

echo '- - Array-Sparse List - -'
# Within double-quotes, default IFS, Glob-Pattern

IFS=$'\x20'$'\x09'$'\x0A'
printf %q "${ArraySparse[*]}"
echo

#  Note that the output does not distinguish between "null content"
#+ and "null reference".
#  Both print as escaped whitespace.
###
#  Note also that the output does NOT contain escaped whitespace
#+ for the "null reference(s)" prior to the first defined element.
###
# This behavior of 2.04, 2.05a and 2.05b has been reported
#+ and may change in a future version of Bash.

#  To output a sparse array and maintain the [subscript]=value
#+ relationship without change requires a bit of programming.
#  One possible code fragment:
###
# local l=${#ArraySparse[@]}        # Count of defined elements
# local f=0                         # Count of found subscripts
# local i=0                         # Subscript to test
(                                   # Anonymous in-line function
    for (( l=${#ArraySparse[@]}, f = 0, i = 0 ; f < l ; i++ ))
```

```
    do
        # 'if defined then...'
        ${ArraySparse[$i]+ eval echo '\ ['$i']='${ArraySparse[$i]} ; (( f++ )) }
    done
)

# The reader coming upon the above code fragment cold
#+ might want to review "command lists" and "multiple commands on a line"
#+ in the text of the foregoing "Advanced Bash Scripting Guide."
###
#  Note:
#  The "read -a array_name" version of the "read" command
#+ begins filling array_name at subscript zero.
#  ArraySparse does not define a value at subscript zero.
###
#  The user needing to read/write a sparse array to either
#+ external storage or a communications socket must invent
#+ a read/write code pair suitable for their purpose.
###
# Exercise: clean it up.

unset ArraySparse

echo
echo '- - Conditional alternate (But not change)- -'
echo '- No alternate if null reference -'
echo -n ${VarNull+'NotSet'}' '
echo ${VarNull}
unset VarNull

echo '- No alternate if null reference -'
echo -n ${VarNull:+'NotSet'}' '
echo ${VarNull}
unset VarNull

echo '- Alternate if null contents -'
echo -n ${VarEmpty+'Empty'}' '              # Empty
echo ${VarEmpty}
VarEmpty=''

echo '- No alternate if null contents -'
echo -n ${VarEmpty:+'Empty'}' '             # Space only
echo ${VarEmpty}
VarEmpty=''

echo '- Alternate if already has contents -'

# Alternate literal
echo -n ${VarSomething+'Content'}' '        # Content Literal
echo ${VarSomething}

# Invoke function
echo -n ${VarSomething:+ $(_simple) }' '    # SimpleFunc Literal
echo ${VarSomething}
echo

echo '- - Sparse Array - -'
echo ${ArrayVar[@]+'Empty'}                 # An array of 'Empty'(ies)
echo

echo '- - Test 2 for undefined - -'
```

```
declare -i t
_incT() {
    t=$t+1
}

#  Note:
#  This is the same test used in the sparse array
#+ listing code fragment.

# Null reference, set: t == -1
t=${#VarNull}-1                     # Results in minus-one.
${VarNull+ _incT }                  # Does not execute.
echo $t' Null reference'

# Null contents, set: t == 0
t=${#VarEmpty}-1                    # Results in minus-one.
${VarEmpty+ _incT }                 # Executes.
echo $t'  Null content'

# Contents, set: t == (number of non-null characters)
t=${#VarSomething}-1                # non-null length minus-one
${VarSomething+ _incT }             # Executes.
echo $t'  Contents'

# Exercise: clean up that example.
unset t
unset _incT

# ${name?err_msg} ${name:?err_msg}
#  These follow the same rules but always exit afterwards
#+ if an action is specified following the question mark.
#  The action following the question mark may be a literal
#+ or a function result.
###
#  ${name?} ${name:?} are test-only, the return can be tested.




# Element operations
# ------------------

echo
echo '- - Trailing sub-element selection - -'

#  Strings, Arrays and Positional parameters

#  Call this script with multiple arguments
#+ to see the parameter selections.

echo '- All -'
echo ${VarSomething:0}              # all non-null characters
echo ${ArrayVar[@]:0}               # all elements with content
echo ${@:0}                         # all parameters with content;
                                    # ignoring parameter[0]

echo
echo '- All after -'
echo ${VarSomething:1}              # all non-null after character[0]
echo ${ArrayVar[@]:1}               # all after element[0] with content
echo ${@:2}                         # all after param[1] with content
```

```
echo
echo '- Range after -'
echo ${VarSomething:4:3}            # ral
                                    # Three characters after
                                    # character[3]

echo '- Sparse array gotch -'
echo ${ArrayVar[@]:1:2}     #  four - The only element with content.
                            #  Two elements after (if that many exist).
                            #  the FIRST WITH CONTENTS
                            #+ (the FIRST WITH  CONTENTS is being
                            #+ considered as if it
                            #+ were subscript zero).
#  Executed as if Bash considers ONLY array elements with CONTENT
#  printf %q "${ArrayVar[@]:0:3}"    # Try this one

#  In versions 2.04, 2.05a and 2.05b,
#+ Bash does not handle sparse arrays as expected using this notation.
#
#  The current Bash maintainer, Chet Ramey, has corrected this.


echo '- Non-sparse array -'
echo ${@:2:2}               # Two parameters following parameter[1]

# New victims for string vector examples:
stringZ=abcABC123ABCabc
arrayZ=( abcabc ABCABC 123123 ABCABC abcabc )
sparseZ=( [1]='abcabc' [3]='ABCABC' [4]='' [5]='123123' )

echo
echo ' - - Victim string - -'$stringZ'- - '
echo ' - - Victim array - -'${arrayZ[@]}'- - '
echo ' - - Sparse array - -'${sparseZ[@]}'- - '
echo ' - [0]==null ref, [2]==null ref, [4]==null content - '
echo ' - [1]=abcabc [3]=ABCABC [5]=123123 - '
echo ' - non-null-reference count: '${#sparseZ[@]}' elements'

echo
echo '- - Prefix sub-element removal - -'
echo '- - Glob-Pattern match must include the first character. - -'
echo '- - Glob-Pattern may be a literal or a function result. - -'
echo


# Function returning a simple, Literal, Glob-Pattern
_abc() {
    echo -n 'abc'
}

echo '- Shortest prefix -'
echo ${stringZ#123}                 # Unchanged (not a prefix).
echo ${stringZ#$(_abc)}             # ABC123ABCabc
echo ${arrayZ[@]#abc}               # Applied to each element.

# echo ${sparseZ[@]#abc}            # Version-2.05b core dumps.
# Has since been fixed by Chet Ramey.

# The -it would be nice- First-Subscript-Of
# echo ${#sparseZ[@]#*}             # This is NOT valid Bash.

echo
```

```
echo '- Longest prefix -'
echo ${stringZ##1*3}                # Unchanged (not a prefix)
echo ${stringZ##a*C}                # abc
echo ${arrayZ[@]##a*c}              # ABCABC 123123 ABCABC

# echo ${sparseZ[@]##a*c}           # Version-2.05b core dumps.
# Has since been fixed by Chet Ramey.

echo
echo '- - Suffix sub-element removal - -'
echo '- - Glob-Pattern match must include the last character. - -'
echo '- - Glob-Pattern may be a literal or a function result. - -'
echo
echo '- Shortest suffix -'
echo ${stringZ%1*3}                 # Unchanged (not a suffix).
echo ${stringZ%$(_abc)}             # abcABC123ABC
echo ${arrayZ[@]%abc}               # Applied to each element.

# echo ${sparseZ[@]%abc}            # Version-2.05b core dumps.
# Has since been fixed by Chet Ramey.

# The -it would be nice- Last-Subscript-Of
# echo ${#sparseZ[@]%*}             # This is NOT valid Bash.

echo
echo '- Longest suffix -'
echo ${stringZ%%1*3}                # Unchanged (not a suffix)
echo ${stringZ%%b*c}                # a
echo ${arrayZ[@]%%b*c}              # a ABCABC 123123 ABCABC a

# echo ${sparseZ[@]%%b*c}           # Version-2.05b core dumps.
# Has since been fixed by Chet Ramey.

echo
echo '- - Sub-element replacement - -'
echo '- - Sub-element at any location in string. - -'
echo '- - First specification is a Glob-Pattern - -'
echo '- - Glob-Pattern may be a literal or Glob-Pattern function result. - -'
echo '- - Second specification may be a literal or function result. - -'
echo '- - Second specification may be unspecified. Pronounce that'
echo '    as: Replace-With-Nothing (Delete) - -'
echo



# Function returning a simple, Literal, Glob-Pattern
_123() {
    echo -n '123'
}

echo '- Replace first occurrence -'
echo ${stringZ/$(_123)/999}         # Changed (123 is a component).
echo ${stringZ/ABC/xyz}             # xyzABC123ABCabc
echo ${arrayZ[@]/ABC/xyz}           # Applied to each element.
echo ${sparseZ[@]/ABC/xyz}          # Works as expected.

echo
echo '- Delete first occurrence -'
echo ${stringZ/$(_123)/}
echo ${stringZ/ABC/}
echo ${arrayZ[@]/ABC/}
echo ${sparseZ[@]/ABC/}
```

```
#  The replacement need not be a literal,
#+ since the result of a function invocation is allowed.
#  This is general to all forms of replacement.
echo
echo '- Replace first occurrence with Result-Of -'
echo ${stringZ/$(_123)/$(_simple)}  # Works as expected.
echo ${arrayZ[@]/ca/$(_simple)}     # Applied to each element.
echo ${sparseZ[@]/ca/$(_simple)}    # Works as expected.

echo
echo '- Replace all occurrences -'
echo ${stringZ//[b2]/X}             # X-out b's and 2's
echo ${stringZ//abc/xyz}            # xyzABC123ABCxyz
echo ${arrayZ[@]//abc/xyz}          # Applied to each element.
echo ${sparseZ[@]//abc/xyz}         # Works as expected.

echo
echo '- Delete all occurrences -'
echo ${stringZ//[b2]/}
echo ${stringZ//abc/}
echo ${arrayZ[@]//abc/}
echo ${sparseZ[@]//abc/}

echo
echo '- - Prefix sub-element replacement - -'
echo '- - Match must include the first character. - -'
echo

echo '- Replace prefix occurrences -'
echo ${stringZ/#[b2]/X}             # Unchanged (neither is a prefix).
echo ${stringZ/#$(_abc)/XYZ}        # XYZABC123ABCabc
echo ${arrayZ[@]/#abc/XYZ}          # Applied to each element.
echo ${sparseZ[@]/#abc/XYZ}         # Works as expected.

echo
echo '- Delete prefix occurrences -'
echo ${stringZ/#[b2]/}
echo ${stringZ/#$(_abc)/}
echo ${arrayZ[@]/#abc/}
echo ${sparseZ[@]/#abc/}

echo
echo '- - Suffix sub-element replacement - -'
echo '- - Match must include the last character. - -'
echo

echo '- Replace suffix occurrences -'
echo ${stringZ/%[b2]/X}             # Unchanged (neither is a suffix).
echo ${stringZ/%$(_abc)/XYZ}        # abcABC123ABCXYZ
echo ${arrayZ[@]/%abc/XYZ}          # Applied to each element.
echo ${sparseZ[@]/%abc/XYZ}         # Works as expected.

echo
echo '- Delete suffix occurrences -'
echo ${stringZ/%[b2]/}
echo ${stringZ/%$(_abc)/}
echo ${arrayZ[@]/%abc/}
echo ${sparseZ[@]/%abc/}

echo
echo '- - Special cases of null Glob-Pattern - -'
```

```
echo

echo '- Prefix all -'
# null substring pattern means 'prefix'
echo ${stringZ/#/NEW}              # NEWabcABC123ABCabc
echo ${arrayZ[@]/#/NEW}            # Applied to each element.
echo ${sparseZ[@]/#/NEW}           # Applied to null-content also.
                                   # That seems reasonable.

echo
echo '- Suffix all -'
# null substring pattern means 'suffix'
echo ${stringZ/%/NEW}              # abcABC123ABCabcNEW
echo ${arrayZ[@]/%/NEW}            # Applied to each element.
echo ${sparseZ[@]/%/NEW}           # Applied to null-content also.
                                   # That seems reasonable.

echo
echo '- - Special case For-Each Glob-Pattern - -'
echo '- - - - This is a nice-to-have dream - - - -'
echo

_GenFunc() {
    echo -n ${0}                   # Illustration only.
    # Actually, that would be an arbitrary computation.
}

# All occurrences, matching the AnyThing pattern.
# Currently //*/ does not match null-content nor null-reference.
# /#/ and /%/ does match null-content but not null-reference.
echo ${sparseZ[@]//*/$(_GenFunc)}


#  A possible syntax would be to make
#+ the parameter notation used within this construct mean:
#   ${1} - The full element
#   ${2} - The prefix, if any, to the matched sub-element
#   ${3} - The matched sub-element
#   ${4} - The suffix, if any, to the matched sub-element
#
# echo ${sparseZ[@]//*/$(_GenFunc ${3})}   # Same as ${1} here.
# Perhaps it will be implemented in a future version of Bash.


exit 0
```

Example A-59. Testing execution times of various commands

```
#!/bin/bash
#  test-execution-time.sh
#  Example by Erik Brandsberg, for testing execution time
#+ of certain operations.
#  Referenced in the "Optimizations" section of "Miscellany" chapter.

count=50000
echo "Math tests"
echo "Math via \$(( ))"
time for (( i=0; i< $count; i++))
do
  result=$(( $i%2 ))
done
```

```
echo "Math via *expr*:"
time for (( i=0; i< $count; i++))
do
  result=`expr "$i%2"`
done

echo "Math via *let*:"
time for (( i=0; i< $count; i++))
do
  let result=$i%2
done

echo
echo "Conditional testing tests"

echo "Test via case:"
time for (( i=0; i< $count; i++))
do
  case $(( $i%2 )) in
    0) : ;;
    1) : ;;
  esac
done

echo "Test with if [], no quotes:"
time for (( i=0; i< $count; i++))
do
  if [ $(( $i%2 )) = 0 ]; then
     :
  else
     :
  fi
done

echo "Test with if [], quotes:"
time for (( i=0; i< $count; i++))
do
  if [ "$(( $i%2 ))" = "0" ]; then
     :
  else
     :
  fi
done

echo "Test with if [], using -eq:"
time for (( i=0; i< $count; i++))
do
  if [ $(( $i%2 )) -eq 0 ]; then
     :
  else
     :
  fi
done

exit $?
```

Example A-60. Associative arrays vs. conventional arrays (execution times)

```
#!/bin/bash
#  assoc-arr-test.sh
```

```
#  Benchmark test script to compare execution times of
#  numeric-indexed array vs. associative array.
#     Thank you, Erik Brandsberg.

count=100000       # May take a while for some of the tests below.
declare simple     # Can change to 20000, if desired.
declare -a array1
declare -A array2
declare -a array3
declare -A array4

echo "===Assignment tests==="
echo

echo "Assigning a simple variable:"
# References $i twice to equalize lookup times.
time for (( i=0; i< $count; i++)); do
        simple=$i$i
done

echo "---"

echo "Assigning a numeric index array entry:"
time for (( i=0; i< $count; i++)); do
        array1[$i]=$i
done

echo "---"

echo "Overwriting a numeric index array entry:"
time for (( i=0; i< $count; i++)); do
        array1[$i]=$i
done

echo "---"

echo "Linear reading of numeric index array:"
time for (( i=0; i< $count; i++)); do
        simple=array1[$i]
done

echo "---"

echo "Assigning an associative array entry:"
time for (( i=0; i< $count; i++)); do
        array2[$i]=$i
done

echo "---"

echo "Overwriting an associative array entry:"
time for (( i=0; i< $count; i++)); do
        array2[$i]=$i
done

echo "---"

echo "Linear reading an associative array entry:"
time for (( i=0; i< $count; i++)); do
        simple=array2[$i]
done
```

```
echo "---"

echo "Assigning a random number to a simple variable:"
time for (( i=0; i< $count; i++)); do
        simple=$RANDOM
done

echo "---"

echo "Assign a sparse numeric index array entry randomly into 64k cells:"
time for (( i=0; i< $count; i++)); do
        array3[$RANDOM]=$i
done

echo "---"

echo "Reading sparse numeric index array entry:"
time for value in "${array3[@]}"i; do
        simple=$value
done

echo "---"

echo "Assigning a sparse associative array entry randomly into 64k cells:"
time for (( i=0; i< $count; i++)); do
        array4[$RANDOM]=$i
done

echo "---"

echo "Reading sparse associative index array entry:"
time for value in "${array4[@]}"; do
        simple=$value
done

exit $?
```

Appendix B. Reference Cards

The following reference cards provide a useful *summary* of certain scripting concepts. The foregoing text treats these matters in more depth, as well as giving usage examples.

Table B-1. Special Shell Variables

Variable	Meaning
`$0`	Filename of script
`$1`	Positional parameter #1
`$2 - $9`	Positional parameters #2 - #9
`${10}`	Positional parameter #10
`$#`	Number of positional parameters
`"$*"`	All the positional parameters (as a single word) *
`"$@"`	All the positional parameters (as separate strings)
`${#*}`	Number of positional parameters
`${#@}`	Number of positional parameters
`$?`	Return value
`$$`	Process ID (PID) of script
`$-`	Flags passed to script (using *set*)
`$_`	Last argument of previous command
`$!`	Process ID (PID) of last job run in background

***** *Must be quoted*, otherwise it defaults to `$@`.

Table B-2. TEST Operators: Binary Comparison

Operator	Meaning	-----	Operator	Meaning
Arithmetic Comparison			String Comparison	
`-eq`	Equal to		`=`	Equal to
			`==`	Equal to
`-ne`	Not equal to		`!=`	Not equal to
`-lt`	Less than		`\<`	Less than (ASCII) *
`-le`	Less than or equal to			
`-gt`	Greater than		`\>`	Greater than (ASCII) *
`-ge`	Greater than or equal to			
			`-z`	String is empty

			`-n`	String is not empty
Arithmetic Comparison	within double parentheses ((...))			
>	Greater than			
>=	Greater than or equal to			
<	Less than			
<=	Less than or equal to			

* *If within a double-bracket* [[...]] *test construct, then no escape \ is needed.*

Table B-3. TEST Operators: Files

Operator	**Tests Whether**	**-----**	**Operator**	**Tests Whether**
`-e`	File exists		`-s`	File is not zero size
`-f`	File is a *regular* file			
`-d`	File is a *directory*		`-r`	File has *read* permission
`-h`	File is a symbolic link		`-w`	File has *write* permission
`-L`	File is a *symbolic link*		`-x`	File has *execute* permission
`-b`	File is a block device			
`-c`	File is a character device		`-g`	*sgid* flag set
`-p`	File is a pipe		`-u`	*suid* flag set
`-S`	File is a socket		`-k`	"sticky bit" set
`-t`	File is associated with a *terminal*			
`-N`	File modified since it was last read		`F1 -nt F2`	File F1 is *newer* than F2 *
`-O`	You own the file		`F1 -ot F2`	File F1 is *older* than F2 *
`-G`	*Group id* of file same as yours		`F1 -ef F2`	Files F1 and F2 are *hard links* to the same file *
`!`	NOT (inverts sense of above tests)			

* *Binary* operator (requires two operands).

Table B-4. Parameter Substitution and Expansion

Expression	**Meaning**
`${var}`	Value of `var` (same as `$var`)
`${var-$DEFAULT}`	If `var` not set, evaluate expression as *`$DEFAULT`* *
`${var:-$DEFAULT}`	If `var` not set or is empty, *evaluate* expression as *`$DEFAULT`* *
`${var=$DEFAULT}`	If `var` not set, evaluate expression as *`$DEFAULT`* *

`${var:=$DEFAULT}`	If *var* not set or is empty, evaluate expression as *$DEFAULT* *
`${var+$OTHER}`	If *var* set, evaluate expression as *$OTHER*, otherwise as null string
`${var:+$OTHER}`	If *var* set, evaluate expression as *$OTHER*, otherwise as null string
`${var?$ERR_MSG}`	If *var* not set, print *$ERR_MSG* and abort script with an exit status of 1.*
`${var:?$ERR_MSG}`	If *var* not set, print *$ERR_MSG* and abort script with an exit status of 1.*
`${!varprefix*}`	Matches all previously declared variables beginning with *varprefix*
`${!varprefix@}`	Matches all previously declared variables beginning with *varprefix*

***** If *var is* set, evaluate the expression as *$var* with no side-effects.

 # Note that some of the above behavior of operators has changed from earlier versions of Bash.

Table B-5. String Operations

Expression	**Meaning**
`${#string}`	Length of *$string*
`${string:position}`	Extract substring from *$string* at *$position*
`${string:position:length}`	Extract *$length* characters substring from *$string* at *$position* [zero-indexed, first character is at position 0]
`${string#substring}`	Strip shortest match of *$substring* from front of *$string*
`${string##substring}`	Strip longest match of *$substring* from front of *$string*
`${string%substring}`	Strip shortest match of *$substring* from back of *$string*
`${string%%substring}`	Strip longest match of *$substring* from back of *$string*
`${string/substring/replacement}`	Replace first match of *$substring* with *$replacement*
`${string//substring/replacement}`	Replace *all* matches of *$substring* with *$replacement*
`${string/#substring/replacement}`	If *$substring* matches *front* end of *$string*, substitute *$replacement* for *$substring*
`${string/%substring/replacement}`	If *$substring* matches *back* end of *$string*, substitute *$replacement* for *$substring*

`expr match "$string" '$substring'`	Length of matching *$substring** at beginning of *$string*
`expr "$string" : '$substring'`	Length of matching *$substring** at beginning of *$string*
`expr index "$string" $substring`	Numerical position in *$string* of first character in *$substring** that matches [0 if no match, first character counts as position 1]
`expr substr $string $position $length`	Extract *$length* characters from *$string* starting at *$position* [0 if no match, first character counts as position 1]
`expr match "$string" '\($substring\)'`	Extract *$substring**, searching from beginning of *$string*
`expr "$string" : '\($substring\)'`	Extract *$substring** , searching from beginning of *$string*
`expr match "$string" '.*\($substring\)'`	Extract *$substring**, searching from end of *$string*
`expr "$string" : '.*\($substring\)'`	Extract *$substring**, searching from end of *$string*

* Where *$substring* is a Regular Expression.

Table B-6. Miscellaneous Constructs

Expression	**Interpretation**
Brackets	
`if [ CONDITION ]`	Test construct
`if [[ CONDITION ]]`	Extended test construct
`Array[1]=element1`	Array initialization
`[a-z]`	Range of characters within a Regular Expression
Curly Brackets	
`${variable}`	Parameter substitution
`${!variable}`	Indirect variable reference
`{ command1; command2; . . . commandN; }`	Block of code
`{string1,string2,string3,...}`	Brace expansion
`{a..z}`	Extended brace expansion
`{}`	Text replacement, after find and xargs
Parentheses	
`( command1; command2 )`	Command group executed within a subshell
`Array=(element1 element2 element3)`	Array initialization

`result=$(COMMAND)`	Command substitution, new style
`>(COMMAND)`	Process substitution
`<(COMMAND)`	Process substitution
Double Parentheses	
`(( var = 78 ))`	Integer arithmetic
`var=$(( 20 + 5 ))`	Integer arithmetic, with variable assignment
`(( var++ ))`	*C-style* variable increment
`(( var-- ))`	*C-style* variable decrement
`(( var0 = var1<98?9:21 ))`	*C-style* ternary operation
Quoting	
`"$variable"`	"Weak" quoting
`'string'`	'Strong' quoting
Back Quotes	
``result=`COMMAND` ``	Command substitution, classic style

Appendix C. A Sed and Awk Micro-Primer

This is a very brief introduction to the **sed** and **awk** text processing utilities. We will deal with only a few basic commands here, but that will suffice for understanding simple sed and awk constructs within shell scripts.

sed: a non-interactive text file editor

awk: a field-oriented pattern processing language with a **C**-style syntax

For all their differences, the two utilities share a similar invocation syntax, use regular expressions , read input by default from `stdin`, and output to `stdout`. These are well-behaved UNIX tools, and they work together well. The output from one can be piped to the other, and their combined capabilities give shell scripts some of the power of Perl.

One important difference between the utilities is that while shell scripts can easily pass arguments to sed, it is more cumbersome for awk (see Example 36-5 and Example 28-2).

C.1. Sed

Sed is a non-interactive [141] stream **ed**itor. It receives text input, whether from `stdin` or from a file, performs certain operations on specified lines of the input, one line at a time, then outputs the result to `stdout` or to a file. Within a shell script, *sed* is usually one of several tool components in a pipe.

Sed determines which lines of its input that it will operate on from the *address range* passed to it. [142] Specify this address range either by line number or by a pattern to match. For example, *`3d`* signals *sed* to delete line 3 of the input, and *`/Windows/d`* tells sed that you want every line of the input containing a match to "Windows" deleted.

Of all the operations in the *sed* toolkit, we will focus primarily on the three most commonly used ones. These are **p**rinting (to `stdout`), **d**eletion, and **s**ubstitution.

Table C-1. Basic sed operators

Operator	Name	Effect
`[address-range]/p`	print	Print [specified address range]
`[address-range]/d`	delete	Delete [specified address range]
`s/pattern1/pattern2/`	substitute	Substitute pattern2 for first instance of pattern1 in a line
`[address-range]/s/pattern1/pattern2/`	substitute	Substitute pattern2 for first instance of pattern1 in a line, over *`address-range`*
`[address-range]/y/pattern1/pattern2/`	transform	replace any character in pattern1 with the corresponding character in pattern2, over *`address-range`* (equivalent of **tr**)

`[address] i pattern Filename`	insert	Insert pattern at address indicated in file Filename. Usually used with `-i` *`in-place`* option.
`g`	global	Operate on *every* pattern match within each matched line of input

Unless the `g` (*global*) operator is appended to a *substitute* command, the substitution operates only on the *first* instance of a pattern match within each line.

From the command-line and in a shell script, a sed operation may require quoting and certain options.

```
sed -e '/^$/d' $filename
# The -e option causes the next string to be interpreted as an editing instruction.
#  (If passing only a single instruction to sed, the "-e" is optional.)
#  The "strong" quotes ('') protect the RE characters in the instruction
#+ from reinterpretation as special characters by the body of the script.
# (This reserves RE expansion of the instruction for sed.)
#
# Operates on the text contained in file $filename.
```

In certain cases, a *sed* editing command will not work with single quotes.

```
filename=file1.txt
pattern=BEGIN

  sed "/^$pattern/d" "$filename"  # Works as specified.
# sed '/^$pattern/d' "$filename"    has unexpected results.
#        In this instance, with strong quoting (' ... '),
#+      "$pattern" will not expand to "BEGIN".
```

Sed uses the `-e` option to specify that the following string is an instruction or set of instructions. If there is only a single instruction contained in the string, then this may be omitted.

```
sed -n '/xzy/p' $filename
# The -n option tells sed to print only those lines matching the pattern.
# Otherwise all input lines would print.
# The -e option not necessary here since there is only a single editing instruction.
```

Table C-2. Examples of sed operators

Notation	**Effect**
`8d`	Delete 8th line of input.
`/^$/d`	Delete all blank lines.
`1,/^$/d`	Delete from beginning of input up to, and including first blank line.
`/Jones/p`	Print only lines containing "Jones" (with -n option).
`s/Windows/Linux/`	Substitute "Linux" for first instance of "Windows" found in each input line.
`s/BSOD/stability/g`	Substitute "stability" for every instance of "BSOD" found in each input line.
`s/ *$//`	Delete all spaces at the end of every line.

`s/00*/0/g`	Compress all consecutive sequences of zeroes into a single zero.
`echo "Working on it." | sed -e '1i How far are you along?'`	Prints "How far are you along?" as first line, "Working on it" as second.
`5i 'Linux is great.' file.txt`	Inserts 'Linux is great.' at line 5 of the file file.txt.
`/GUI/d`	Delete all lines containing "GUI".
`s/GUI//g`	Delete all instances of "GUI", leaving the remainder of each line intact.

Substituting a zero-length string for another is equivalent to deleting that string within a line of input. This leaves the remainder of the line intact. Applying **`s/GUI//`** to the line

```
The most important parts of any application are its GUI and sound effects
```

results in

```
The most important parts of any application are its  and sound effects
```

A backslash forces the **sed** replacement command to continue on to the next line. This has the effect of using the *newline* at the end of the first line as the *replacement string*.

```
s/^  */\
/g
```

This substitution replaces line-beginning spaces with a newline. The net result is to replace paragraph indents with a blank line between paragraphs.

An address range followed by one or more operations may require open and closed curly brackets, with appropriate newlines.

```
/[0-9A-Za-z]/,/^$/{
/^$/d
}
```

This deletes only the first of each set of consecutive blank lines. That might be useful for single-spacing a text file, but retaining the blank line(s) between paragraphs.

The usual delimiter that *sed* uses is /. However, *sed* allows other delimiters, such as %. This is useful when / is part of a replacement string, as in a file pathname. See Example 11-10 and Example 16-32.

A quick way to double-space a text file is **`sed G filename`**.

For illustrative examples of sed within shell scripts, see:

1. Example 36-1
2. Example 36-2
3. Example 16-3
4. Example A-2
5. Example 16-17
6. Example 16-27
7. Example A-12
8. Example A-16
9. Example A-17
10. Example 16-32
11. Example 11-10

12. Example 16-48
13. Example A-1
14. Example 16-14
15. Example 16-12
16. Example A-10
17. Example 19-12
18. Example 16-19
19. Example A-29
20. Example A-31
21. Example A-24
22. Example A-43
23. Example A-55

For a more extensive treatment of *sed*, refer to the pertinent references in the *Bibliography*.

C.2. Awk

Awk [143] is a full-featured text processing language with a syntax reminiscent of *C*. While it possesses an extensive set of operators and capabilities, we will cover only a few of these here - the ones most useful in shell scripts.

Awk breaks each line of input passed to it into fields. By default, a field is a string of consecutive characters delimited by whitespace, though there are options for changing this. Awk parses and operates on each separate field. This makes it ideal for handling structured text files -- especially tables -- data organized into consistent chunks, such as rows and columns.

Strong quoting and curly brackets enclose blocks of awk code within a shell script.

```
# $1 is field #1, $2 is field #2, etc.

echo one two | awk '{print $1}'
# one

echo one two | awk '{print $2}'
# two

# But what is field #0 ($0)?
echo one two | awk '{print $0}'
# one two
# All the fields!


awk '{print $3}' $filename
# Prints field #3 of file $filename to stdout.

awk '{print $1 $5 $6}' $filename
# Prints fields #1, #5, and #6 of file $filename.

awk '{print $0}' $filename
# Prints the entire file!
# Same effect as:   cat $filename . . . or . . . sed '' $filename
```

We have just seen the awk *print* command in action. The only other feature of awk we need to deal with here is variables. Awk handles variables similarly to shell scripts, though a bit more flexibly.

```
{ total += ${column_number} }
```

This adds the value of *column_number* to the running total of *total*>. Finally, to print "total", there is an **END** command block, executed after the script has processed all its input.

```
END { print total }
```

Corresponding to the **END**, there is a **BEGIN**, for a code block to be performed before awk starts processing its input.

The following example illustrates how **awk** can add text-parsing tools to a shell script.

Example C-1. Counting Letter Occurrences

```
#! /bin/sh
# letter-count2.sh: Counting letter occurrences in a text file.
#
# Script by nyal [nyal@voila.fr].
# Used in ABS Guide with permission.
# Recommented and reformatted by ABS Guide author.
# Version 1.1: Modified to work with gawk 3.1.3.
#              (Will still work with earlier versions.)


INIT_TAB_AWK=""
# Parameter to initialize awk script.
count_case=0
FILE_PARSE=$1

E_PARAMERR=85

usage()
{
    echo "Usage: letter-count.sh file letters" 2>&1
    # For example:   ./letter-count2.sh filename.txt a b c
    exit $E_PARAMERR  # Too few arguments passed to script.
}

if [ ! -f "$1" ] ; then
    echo "$1: No such file." 2>&1
    usage                 # Print usage message and exit.
fi

if [ -z "$2" ] ; then
    echo "$2: No letters specified." 2>&1
    usage
fi

shift                      # Letters specified.
for letter in `echo $@`    # For each one . . .
  do
  INIT_TAB_AWK="$INIT_TAB_AWK tab_search[${count_case}] = \
  \"$letter\"; final_tab[${count_case}] = 0; "
  # Pass as parameter to awk script below.
  count_case=`expr $count_case + 1`
done

# DEBUG:
# echo $INIT_TAB_AWK;

cat $FILE_PARSE |
```

```
# Pipe the target file to the following awk script.

# ----------------------------------------------------------------------
# Earlier version of script:
# awk -v tab_search=0 -v final_tab=0 -v tab=0 -v \
# nb_letter=0 -v chara=0 -v chara2=0 \

awk \
"BEGIN { $INIT_TAB_AWK } \
{ split(\$0, tab, \"\"); \
for (chara in tab) \
{ for (chara2 in tab_search) \
{ if (tab_search[chara2] == tab[chara]) { final_tab[chara2]++ } } } } \
END { for (chara in final_tab) \
{ print tab_search[chara] \" => \" final_tab[chara] } }"
# ----------------------------------------------------------------------
#  Nothing all that complicated, just . . .
#+ for-loops, if-tests, and a couple of specialized functions.

exit $?

# Compare this script to letter-count.sh.
```

For simpler examples of awk within shell scripts, see:

1. Example 15-14
2. Example 20-8
3. Example 16-32
4. Example 36-5
5. Example 28-2
6. Example 15-20
7. Example 29-3
8. Example 29-4
9. Example 11-3
10. Example 16-61
11. Example 9-16
12. Example 16-4
13. Example 10-6
14. Example 36-19
15. Example 11-9
16. Example 36-4
17. Example 16-53
18. Example T-3

That's all the awk we'll cover here, folks, but there's lots more to learn. See the appropriate references in the *Bibliography*.

Appendix D. Parsing and Managing Pathnames

Emmanual Rouat contributed the following example of parsing and transforming *filenames* and, in particular, pathnames. It draws heavily on the functionality of *sed*.

```
#!/usr/bin/env bash
#-----------------------------------------------------------
# Management of PATH, LD_LIBRARY_PATH, MANPATH variables...
# By Emmanuel Rouat <no-email>
# (Inspired by the bash documentation 'pathfuncs' and on
# discussions found on stackoverflow:
# http://stackoverflow.com/questions/370047/
# http://stackoverflow.com/questions/273909/#346860 )
# Last modified: Sat Sep 22 12:01:55 CEST 2012
#
# The following functions handle spaces correctly.
# These functions belong in .bash_profile rather than in
# .bashrc, I guess.
#
# The modular aspect of these functions should make it easy
# to expand them to handle path substitutions instead
# of path removal etc....
#
# See http://www.catonmat.net/blog/awk-one-liners-explained-part-two/
# (item 43) for an explanation of the 'duplicate-entries' removal
# (it's a nice trick!)
#-----------------------------------------------------------

# Show $@ (usually PATH) as list.
function p_show() { local p="$@" && for p; do [[ ${!p} ]] &&
echo -e ${!p//:/\\n}; done }

# Filter out empty lines, multiple/trailing slashes, and duplicate entries.
function p_filter()
{ awk '/^[ \t]*$/ {next} {sub(/\/+$/, "");gsub(/\/+/, "/")}!x[$0]++' ;}

# Rebuild list of items into ':' separated word (PATH-like).
function p_build() { paste -sd: ;}

# Clean $1 (typically PATH) and rebuild it
function p_clean()
{ local p=${1} && eval ${p}='$(p_show ${p} | p_filter | p_build)' ;}

# Remove $1 from $2 (found on stackoverflow, with modifications).
function p_rm()
{ local d=$(echo $1 | p_filter) p=${2} &&
  eval ${p}='$(p_show ${p} | p_filter | grep -xv "${d}" | p_build)' ;}

#  Same as previous, but filters on a pattern (dangerous...
#+ don't use 'bin' or '/' as pattern!).
function p_rmpat()
{ local d=$(echo $1 | p_filter) p=${2} && eval ${p}='$(p_show ${p} |
  p_filter | grep -v "${d}" | p_build)' ;}

# Delete $1 from $2 and append it cleanly.
function p_append()
{ local d=$(echo $1 | p_filter) p=${2} && p_rm "${d}" ${p} &&
  eval ${p}='$(p_show ${p} d | p_build)' ;}

# Delete $1 from $2 and prepend it cleanly.
```

```
function p_prepend()
{ local d=$(echo $1 | p_filter) p=${2} && p_rm "${d}" ${p} &&
  eval ${p}='$(p_show d ${p} | p_build)' ;}

# Some tests:
echo
MYPATH="/bin:/usr/bin/:/bin://bin/"
p_append "/project//my project/bin" MYPATH
echo "Append '/project//my project/bin' to '/bin:/usr/bin/:/bin://bin/'"
echo "(result should be: /bin:/usr/bin:/project/my project/bin)"
echo $MYPATH

echo
MYOTHERPATH="/bin:/usr/bin/:/bin:/project//my project/bin"
p_prepend "/project//my project/bin" MYOTHERPATH
echo "Prepend '/project//my project/bin' \
to '/bin:/usr/bin/:/bin:/project//my project/bin/'"
echo "(result should be: /project/my project/bin:/bin:/usr/bin)"
echo $MYOTHERPATH

echo
p_prepend "/project//my project/bin" FOOPATH  # FOOPATH doesn't exist.
echo "Prepend '/project//my project/bin' to an unset variable"
echo "(result should be: /project/my project/bin)"
echo $FOOPATH

echo
BARPATH="/a:/b/://b c://a:/my local pub"
p_clean BARPATH
echo "Clean BARPATH='/a:/b/://b c://a:/my local pub'"
echo "(result should be: /a:/b:/b c:/my local pub)"
echo $BARPATH
```

***

David Wheeler kindly permitted me to use his instructive examples.

```
Doing it correctly: A quick summary
by David Wheeler
http://www.dwheeler.com/essays/filenames-in-shell.html

So, how can you process filenames correctly in shell? Here's a quick
summary about how to do it correctly, for the impatient who "just want the
answer". In short: Double-quote to use "$variable" instead of $variable,
set IFS to just newline and tab, prefix all globs/filenames so they cannot
begin with "-" when expanded, and use one of a few templates that work
correctly. Here are some of those templates that work correctly:


 IFS="$(printf '\n\t')"
 # Remove SPACE, so filenames with spaces work well.

 #  Correct glob use:
 #+ always use "for" loop, prefix glob, check for existence:
 for file in ./* ; do          # Use "./*" ... NEVER bare "*" ...
   if [ -e "$file" ] ; then    # Make sure it isn't an empty match.
     COMMAND ... "$file" ...
   fi
 done
```

```
# Correct glob use, but requires nonstandard bash extension.
shopt -s nullglob  #  Bash extension,
                   #+ so that empty glob matches will work.
for file in ./* ; do        # Use "./*", NEVER bare "*"
  COMMAND ... "$file" ...
done



#  These handle all filenames correctly;
#+ can be unwieldy if COMMAND is large:
find ... -exec COMMAND... {} \;
find ... -exec COMMAND... {} \+ # If multiple files are okay for COMMAND.



#  This skips filenames with control characters
#+ (including tab and newline).
IFS="$(printf '\n\t')"
controlchars="$(printf '*[\001-\037\177]*')"
for file in $(find . ! -name "$controlchars"') ; do
  COMMAND "$file" ...
done



#  Okay if filenames can't contain tabs or newlines --
#+ beware the assumption.
IFS="$(printf '\n\t')"
for file in $(find .) ; do
  COMMAND "$file" ...
done



# Requires nonstandard but common extensions in find and xargs:
find . -print0 | xargs -0 COMMAND

# Requires nonstandard extensions to find and to shell (bash works).
# variables might not stay set once the loop ends:
find . -print0 | while IFS="" read -r -d "" file ; do ...
  COMMAND "$file" # Use quoted "$file", not $file, everywhere.
done



#  Requires nonstandard extensions to find and to shell (bash works).
#  Underlying system must include named pipes (FIFOs)
#+ or the /dev/fd mechanism.
#  In this version, variables *do* stay set after the loop ends,
#  and you can read from stdin.
#+ (Change the 4 to another number if fd 4 is needed.)

while IFS="" read -r -d "" file <&4 ; do
  COMMAND "$file"   # Use quoted "$file" -- not $file, everywhere.
done 4< <(find . -print0)


#  Named pipe version.
#  Requires nonstandard extensions to find and to shell's read (bash ok).
#  Underlying system must include named pipes (FIFOs).
#  Again, in this version, variables *do* stay set after the loop ends,
```

```
#  and you can read from stdin.
# (Change the 4 to something else if fd 4 needed).

mkfifo mypipe

find . -print0 > mypipe &
while IFS="" read -r -d "" file <&4 ; do
  COMMAND "$file" # Use quoted "$file", not $file, everywhere.
done 4< mypipe
```

Appendix E. Exit Codes With Special Meanings

Table E-1. ***Reserved*** **Exit Codes**

Exit Code Number	**Meaning**	**Example**	**Comments**
1	Catchall for general errors	let "var1 = 1/0"	Miscellaneous errors, such as "divide by zero" and other impermissible operations
2	Misuse of shell builtins (according to Bash documentation)	empty_function() {}	Missing keyword or command, or permission problem (and *diff* return code on a failed binary file comparison).
126	Command invoked cannot execute	/dev/null	Permission problem or command is not an executable
127	"command not found"	illegal_command	Possible problem with `$PATH` or a typo
128	Invalid argument to exit	exit 3.14159	**exit** takes only integer args in the range 0 - 255 (see first footnote)
128+n	Fatal error signal "n"	*kill -9* `$PPID` of script	**`$?`** returns 137 (128 + 9)
130	Script terminated by Control-C	*Ctl-C*	Control-C is fatal error signal 2, (130 = 128 + 2, see above)
255*	Exit status out of range	exit -1	**exit** takes only integer args in the range 0 - 255

According to the above table, exit codes 1 - 2, 126 - 165, and 255 [144] have special meanings, and should therefore be avoided for user-specified exit parameters. Ending a script with *exit 127* would certainly cause confusion when troubleshooting (is the error code a "command not found" or a user-defined one?). However, many scripts use an *exit 1* as a general bailout-upon-error. Since exit code 1 signifies so many possible errors, it is not particularly useful in debugging.

There has been an attempt to systematize exit status numbers (see `/usr/include/sysexits.h`), but this is intended for C and C++ programmers. A similar standard for scripting might be appropriate. The author of this document proposes restricting user-defined exit codes to the range 64 - 113 (in addition to 0, for success), to conform with the C/C++ standard. This would allot 50 valid codes, and make troubleshooting scripts more straightforward. [145] All user-defined exit codes in the accompanying examples to this document conform to this standard, except where overriding circumstances exist, as in Example 9-2.

Issuing a $? from the command-line after a shell script exits gives results consistent with the table above only from the Bash or *sh* prompt. Running the *C-shell* or *tcsh* may give different values in some cases.

Appendix F. A Detailed Introduction to I/O and I/O Redirection

written by Stéphane Chazelas, and revised by the document author

A command expects the first three file descriptors to be available. The first, *fd 0* (standard input, `stdin`), is for reading. The other two (*fd 1*, `stdout` and *fd 2*, `stderr`) are for writing.

There is a `stdin`, `stdout`, and a `stderr` associated with each command. **`ls 2>&1`** means temporarily connecting the `stderr` of the **ls** command to the same "resource" as the shell's `stdout`.

By convention, a command reads its input from fd 0 (`stdin`), prints normal output to fd 1 (`stdout`), and error ouput to fd 2 (`stderr`). If one of those three fd's is not open, you may encounter problems:

```
bash$ cat /etc/passwd >&-
cat: standard output: Bad file descriptor
```

For example, when **xterm** runs, it first initializes itself. Before running the user's shell, **xterm** opens the terminal device (/dev/pts/<n> or something similar) three times.

At this point, Bash inherits these three file descriptors, and each command (child process) run by Bash inherits them in turn, except when you redirect the command. Redirection means reassigning one of the file descriptors to another file (or a pipe, or anything permissible). File descriptors may be reassigned locally (for a command, a command group, a subshell, a while or if or case or for loop...), or globally, for the remainder of the shell (using exec).

`ls > /dev/null` means running **ls** with its fd 1 connected to `/dev/null`.

```
bash$ lsof -a -p $$ -d0,1,2
COMMAND PID     USER   FD   TYPE DEVICE SIZE NODE NAME
 bash    363 bozo        0u   CHR  136,1         3 /dev/pts/1
 bash    363 bozo        1u   CHR  136,1         3 /dev/pts/1
 bash    363 bozo        2u   CHR  136,1         3 /dev/pts/1


bash$ exec 2> /dev/null
bash$ lsof -a -p $$ -d0,1,2
COMMAND PID     USER   FD   TYPE DEVICE SIZE NODE NAME
 bash    371 bozo        0u   CHR  136,1         3 /dev/pts/1
 bash    371 bozo        1u   CHR  136,1         3 /dev/pts/1
 bash    371 bozo        2w   CHR    1,3       120 /dev/null


bash$ bash -c 'lsof -a -p $$ -d0,1,2' | cat
COMMAND PID USER   FD   TYPE DEVICE SIZE NODE NAME
 lsof    379 root    0u   CHR  136,1         3 /dev/pts/1
 lsof    379 root    1w  FIFO    0,0      7118 pipe
 lsof    379 root    2u   CHR  136,1         3 /dev/pts/1


bash$ echo "$(bash -c 'lsof -a -p $$ -d0,1,2' 2>&1)"
COMMAND PID USER   FD   TYPE DEVICE SIZE NODE NAME
 lsof    426 root    0u   CHR  136,1         3 /dev/pts/1
```

```
lsof    426 root    1w  FIFO    0,0      7520 pipe
lsof    426 root    2w  FIFO    0,0      7520 pipe
```

This works for different types of redirection.

Exercise: Analyze the following script.

```
#! /usr/bin/env bash

mkfifo /tmp/fifo1 /tmp/fifo2
while read a; do echo "FIFO1: $a"; done < /tmp/fifo1 & exec 7> /tmp/fifo1
exec 8> >(while read a; do echo "FD8: $a, to fd7"; done >&7)

exec 3>&1
(
 (
  (
   while read a; do echo "FIFO2: $a"; done < /tmp/fifo2 | tee /dev/stderr \
   | tee /dev/fd/4 | tee /dev/fd/5 | tee /dev/fd/6 >&7 & exec 3> /tmp/fifo2

   echo 1st, to stdout
   sleep 1
   echo 2nd, to stderr >&2
   sleep 1
   echo 3rd, to fd 3 >&3
   sleep 1
   echo 4th, to fd 4 >&4
   sleep 1
   echo 5th, to fd 5 >&5
   sleep 1
   echo 6th, through a pipe | sed 's/.*/PIPE: &, to fd 5/' >&5
   sleep 1
   echo 7th, to fd 6 >&6
   sleep 1
   echo 8th, to fd 7 >&7
   sleep 1
   echo 9th, to fd 8 >&8

  ) 4>&1 >&3 3>&- | while read a; do echo "FD4: $a"; done 1>&3 5>&- 6>&-
 ) 5>&1 >&3 | while read a; do echo "FD5: $a"; done 1>&3 6>&-
) 6>&1 >&3 | while read a; do echo "FD6: $a"; done 3>&-

rm -f /tmp/fifo1 /tmp/fifo2


# For each command and subshell, figure out which fd points to what.
# Good luck!

exit 0
```

Appendix G. Command-Line Options

Many executables, whether binaries or script files, accept options to modify their run-time behavior. For example: from the command-line, typing **command -o** would invoke *command*, with option `o`.

G.1. Standard Command-Line Options

Over time, there has evolved a loose standard for the meanings of command-line option flags. The GNU utilities conform more closely to this "standard" than older UNIX utilities.

Traditionally, UNIX command-line options consist of a dash, followed by one or more lowercase letters. The GNU utilities added a double-dash, followed by a complete word or compound word.

The two most widely-accepted options are:

- `-h`

  `--help`

  Help: Give usage message and exit.
- `-v`

  `--version`

  Version: Show program version and exit.

Other common options are:

- `-a`

  `--all`

  All: show *all* information or operate on *all* arguments.
- `-l`

  `--list`

  List: list files or arguments without taking other action.
- `-o`

  Output filename
- `-q`

  `--quiet`

  Quiet: suppress `stdout`.
- `-r`

  `-R`

`--recursive`

Recursive: Operate recursively (down directory tree).
- `-v`

  `--verbose`

  Verbose: output additional information to `stdout` or `stderr`.
- `-z`

  `--compress`

  Compress: apply compression (usually gzip).

However:

- In **tar** and **gawk**:

  `-f`

  `--file`

  File: filename follows.
- In **cp**, **mv**, **rm**:

  `-f`

  `--force`

  Force: force overwrite of target file(s).

Many UNIX and Linux utilities deviate from this "standard," so it is dangerous to *assume* that a given option will behave in a standard way. Always check the man page for the command in question when in doubt.

A complete table of recommended options for the GNU utilities is available at the GNU standards page.

G.2. Bash Command-Line Options

Bash itself has a number of command-line options. Here are some of the more useful ones.

- `-c`

  Read commands from the following string and assign any arguments to the positional parameters.

```
bash$ bash -c 'set a b c d; IFS="+-;"; echo "$*"'
a+b+c+d
```

- `-r`

`--restricted`

Runs the shell, or a script, in restricted mode.

- `--posix`

  Forces Bash to conform to POSIX mode.
- `--version`

  Display Bash version information and exit.
- `--`

  End of options. Anything further on the command line is an argument, not an option.

Appendix H. Important Files

startup files

These files contain the aliases and environmental variables made available to Bash running as a user shell and to all Bash scripts invoked after system initialization.

`/etc/profile`

Systemwide defaults, mostly setting the environment (all Bourne-type shells, not just Bash [146])

`/etc/bashrc`

systemwide functions and aliases for Bash

`$HOME/.bash_profile`

user-specific Bash environmental default settings, found in each user's home directory (the local counterpart to `/etc/profile`)

`$HOME/.bashrc`

user-specific Bash init file, found in each user's home directory (the local counterpart to `/etc/bashrc`). Only interactive shells and user scripts read this file. See Appendix M for a sample `.bashrc` file.

logout file

`$HOME/.bash_logout`

user-specific instruction file, found in each user's home directory. Upon exit from a login (Bash) shell, the commands in this file execute.

data files

`/etc/passwd`

A listing of all the user accounts on the system, their identities, their home directories, the groups they belong to, and their default shell. Note that the user passwords are *not* stored in this file, [147] but in `/etc/shadow` in encrypted form.

system configuration files

`/etc/sysconfig/hwconf`

Listing and description of attached hardware devices. This information is in text form and can be extracted and parsed.

```
bash$ grep -A 5 AUDIO /etc/sysconfig/hwconf
class: AUDIO
 bus: PCI
 detached: 0
 driver: snd-intel8x0
 desc: "Intel Corporation 82801CA/CAM AC'97 Audio Controller"
 vendorId: 8086
```

This file is present on Red Hat and Fedora Core installations, but may be missing from other distros.

Appendix I. Important System Directories

Sysadmins and anyone else writing administrative scripts should be intimately familiar with the following system directories.

- `/bin`

  Binaries (executables). Basic system programs and utilities (such as **bash**).
- `/usr/bin` [148]

  More system binaries.
- `/usr/local/bin`

  Miscellaneous binaries local to the particular machine.
- `/sbin`

  System binaries. Basic system administrative programs and utilities (such as **fsck**).
- `/usr/sbin`

  More system administrative programs and utilities.
- `/etc`

  Et cetera. Systemwide configuration scripts.

  Of particular interest are the `/etc/fstab` (filesystem table), `/etc/mtab` (mounted filesystem table), and the `/etc/inittab` files.
- `/etc/rc.d`

  Boot scripts, on Red Hat and derivative distributions of Linux.
- `/usr/share/doc`

  Documentation for installed packages.
- `/usr/man`

  The systemwide manpages.
- `/dev`

  Device directory. Entries (but *not* mount points) for physical and virtual devices. See Chapter 29.
- `/proc`

  Process directory. Contains information and statistics about running processes and kernel parameters. See Chapter 29.
- `/sys`

  Systemwide device directory. Contains information and statistics about device and device names. This is newly added to Linux with the 2.6.X kernels.
- `/mnt`

  Mount. Directory for mounting hard drive partitions, such as `/mnt/dos`, and physical devices. In newer Linux distros, the `/media` directory has taken over as the preferred mount point for I/O devices.

* `/media`

  In newer Linux distros, the preferred mount point for I/O devices, such as CD/DVD drives or USB flash drives.
* `/var`

  Variable (changeable) system files. This is a catchall "scratchpad" directory for data generated while a Linux/UNIX machine is running.
* `/var/log`

  Systemwide log files.
* `/var/spool/mail`

  User mail spool.
* `/lib`

  Systemwide library files.
* `/usr/lib`

  More systemwide library files.
* `/tmp`

  System temporary files.
* `/boot`

  System *boot* directory. The kernel, module links, system map, and boot manager reside here.

  Altering files in this directory may result in an unbootable system.

Appendix J. An Introduction to Programmable Completion

The *programmable completion* feature in Bash permits typing a partial command, then pressing the **[Tab]** key to auto-complete the command sequence. [149] If multiple completions are possible, then **[Tab]** lists them all. Let's see how it works.

```
bash$ xtra[Tab]
xtraceroute       xtrapin           xtrapproto
 xtraceroute.real  xtrapinfo         xtrapreset
 xtrapchar         xtrapout          xtrapstats


bash$ xtrac[Tab]
xtraceroute       xtraceroute.real


bash$ xtraceroute.r[Tab]
xtraceroute.real
```

Tab completion also works for variables and path names.

```
bash$ echo $BASH[Tab]
$BASH                 $BASH_COMPLETION      $BASH_SUBSHELL
 $BASH_ARGC            $BASH_COMPLETION_DIR  $BASH_VERSINFO
 $BASH_ARGV            $BASH_LINENO          $BASH_VERSION
 $BASH_COMMAND         $BASH_SOURCE


bash$ echo /usr/local/[Tab]
bin/     etc/     include/ libexec/ sbin/    src/
 doc/     games/   lib/     man/     share/
```

The Bash **complete** and **compgen** builtins make it possible for *tab completion* to recognize partial *parameters* and *options* to commands. In a very simple case, we can use **complete** from the command-line to specify a short list of acceptable parameters.

```
bash$ touch sample_command
bash$ touch file1.txt file2.txt file2.doc file30.txt file4.zzz
bash$ chmod +x sample_command
bash$ complete -f -X '!*.txt' sample_command


bash$ ./sample[Tab][Tab]
sample_command
file1.txt   file2.txt   file30.txt
```

The `-f` option to *complete* specifies filenames, and `-X` the filter pattern.

For anything more complex, we could write a script that specifies a list of acceptable command-line parameters. The **compgen** builtin expands a list of *arguments* to *generate* completion matches.

Let us take a modified version of the *UseGetOpt.sh* script as an example command. This script accepts a number of command-line parameters, preceded by either a single or double dash. And here is the corresponding *completion script*, by convention given a filename corresponding to its associated command.

Example J-1. Completion script for *UseGetOpt.sh*

```
# file: UseGetOpt-2
# UseGetOpt-2.sh parameter-completion

_UseGetOpt-2 ()   #  By convention, the function name
{                 #+ starts with an underscore.
  local cur
  # Pointer to current completion word.
  # By convention, it's named "cur" but this isn't strictly necessary.

  COMPREPLY=()   # Array variable storing the possible completions.
  cur=${COMP_WORDS[COMP_CWORD]}

  case "$cur" in
    -*)
    COMPREPLY=( $( compgen -W '-a -d -f -l -t -h --aoption --debug \
                               --file --log --test --help --' -- $cur ) );;
#   Generate the completion matches and load them into $COMPREPLY array.
#   xx) May add more cases here.
#   yy)
#   zz)
  esac

  return 0
}

complete -F _UseGetOpt-2 -o filenames ./UseGetOpt-2.sh
#        ^^ ^^^^^^^^^^^^  Invokes the function _UseGetOpt-2.
```

Now, let's try it.

```
bash$ source UseGetOpt-2

bash$ ./UseGetOpt-2.sh -[Tab]
--         --aoption  --debug    --file     --help     --log     --test
 -a         -d         -f         -h         -l         -t


bash$ ./UseGetOpt-2.sh --[Tab]
--         --aoption  --debug    --file     --help     --log     --test
```

We begin by sourcing the "completion script." This sets the command-line parameters. [150]

In the first instance, hitting **[Tab]** after a single dash, the output is all the possible parameters preceded by *one or more* dashes. Hitting **[Tab]** after *two* dashes gives the possible parameters preceded by *two or more* dashes.

Now, just what is the point of having to jump through flaming hoops to enable command-line tab completion? *It saves keystrokes.* [151]

--

Resources:

Bash programmable completion project

Mitch Frazier's *Linux Journal* article, *More on Using the Bash Complete Command*

Steve's excellent two-part article, "An Introduction to Bash Completion": Part 1 and Part 2

Appendix K. Localization

Localization is an undocumented Bash feature.

A localized shell script echoes its text output in the language defined as the system's locale. A Linux user in Berlin, Germany, would get script output in German, whereas his cousin in Berlin, Maryland, would get output from the same script in English.

To create a localized script, use the following template to write all messages to the user (error messages, prompts, etc.).

```
#!/bin/bash
# localized.sh
#  Script by Stéphane Chazelas,
#+ modified by Bruno Haible, bugfixed by Alfredo Pironti.

. gettext.sh

E_CDERROR=65

error()
{
  printf "$@" >&2
  exit $E_CDERROR
}

cd $var || error "`eval_gettext \"Can\'t cd to \\\$var.\"`"
#  The triple backslashes (escapes) in front of $var needed
#+ "because eval_gettext expects a string
#+ where the variable values have not yet been substituted."
#    -- per Bruno Haible
read -p "`gettext \"Enter the value: \"`" var
#  ...


#  ------------------------------------------------------------------
#  Alfredo Pironti comments:

#  This script has been modified to not use the $"..." syntax in
#+ favor of the "`gettext \"...\"`" syntax.
#  This is ok, but with the new localized.sh program, the commands
#+ "bash -D filename" and "bash --dump-po-string filename"
#+ will produce no output
#+ (because those command are only searching for the $"..." strings)!
#  The ONLY way to extract strings from the new file is to use the
# 'xgettext' program. However, the xgettext program is buggy.

# Note that 'xgettext' has another bug.
#
# The shell fragment:
#    gettext -s "I like Bash"
# will be correctly extracted, but . . .
#    xgettext -s "I like Bash"
# . . . fails!
#  'xgettext' will extract "-s" because
#+ the command only extracts the
#+ very first argument after the 'gettext' word.
```

```
#  Escape characters:
#
#  To localize a sentence like
#     echo -e "Hello\tworld!"
#+ you must use
#     echo -e "`gettext \"Hello\\tworld\"`"
#  The "double escape character" before the `t' is needed because
#+ 'gettext' will search for a string like: 'Hello\tworld'
#  This is because gettext will read one literal `\')
#+ and will output a string like "Bonjour\tmonde",
#+ so the 'echo' command will display the message correctly.
#
#  You may not use
#     echo "`gettext -e \"Hello\tworld\"`"
#+ due to the xgettext bug explained above.



# Let's localize the following shell fragment:
#     echo "-h display help and exit"
#
# First, one could do this:
#     echo "`gettext \"-h display help and exit\"`"
#  This way 'xgettext' will work ok,
#+ but the 'gettext' program will read "-h" as an option!
#
# One solution could be
#     echo "`gettext -- \"-h display help and exit\"`"
#  This way 'gettext' will work,
#+ but 'xgettext' will extract "--", as referred to above.
#
# The workaround you may use to get this string localized is
#     echo -e "`gettext \"\\0-h display help and exit\"`"
#  We have added a \0 (NULL) at the beginning of the sentence.
#  This way 'gettext' works correctly, as does 'xgettext.'
#  Moreover, the NULL character won't change the behavior
#+ of the 'echo' command.
#  ------------------------------------------------------------------
```

```
bash$ bash -D localized.sh
"Can't cd to %s."
 "Enter the value: "
```

This lists all the localized text. (The `-D` option lists double-quoted strings prefixed by a $, without executing the script.)

```
bash$ bash --dump-po-strings localized.sh
#: a:6
 msgid "Can't cd to %s."
 msgstr ""
 #: a:7
 msgid "Enter the value: "
 msgstr ""
```

The `--dump-po-strings` option to Bash resembles the `-D` option, but uses gettext "po" format.

Bruno Haible points out:

> Starting with gettext-0.12.2, **xgettext -o - localized.sh** is recommended instead of **bash --dump-po-strings localized.sh**, because **xgettext** . . .

1. understands the gettext and eval_gettext commands (whereas bash --dump-po-strings understands only its deprecated $"..." syntax)

2. can extract comments placed by the programmer, intended to be read by the translator.

This shell code is then not specific to Bash any more; it works the same way with Bash 1.x and other /bin/sh implementations.

Now, build a `language.po` file for each language that the script will be translated into, specifying the *msgstr*. Alfredo Pironti gives the following example:

fr.po:

```
#: a:6
msgid "Can't cd to $var."
msgstr "Impossible de se positionner dans le repertoire $var."
#: a:7
msgid "Enter the value: "
msgstr "Entrez la valeur : "

#  The string are dumped with the variable names, not with the %s syntax,
#+ similar to C programs.
#+ This is a very cool feature if the programmer uses
#+ variable names that make sense!
```

Then, run msgfmt.

msgfmt -o localized.sh.mo fr.po

Place the resulting `localized.sh.mo` file in the `/usr/local/share/locale/fr/LC_MESSAGES` directory, and at the beginning of the script, insert the lines:

```
TEXTDOMAINDIR=/usr/local/share/locale
TEXTDOMAIN=localized.sh
```

If a user on a French system runs the script, she will get French messages.

With older versions of Bash or other shells, localization requires gettext, using the `-s` option. In this case, the script becomes:

```
#!/bin/bash
# localized.sh

E_CDERROR=65

error() {
  local format=$1
  shift
  printf "$(gettext -s "$format")" "$@" >&2
  exit $E_CDERROR
}
cd $var || error "Can't cd to %s." "$var"
read -p "$(gettext -s "Enter the value: ")" var
# ...
```

The `TEXTDOMAIN` and `TEXTDOMAINDIR` variables need to be set and exported to the environment. This should be done within the script itself.

This appendix written by Stéphane Chazelas, with modifications suggested by Alfredo Pironti, and by Bruno Haible, maintainer of GNU gettext.

Appendix L. History Commands

The Bash shell provides command-line tools for editing and manipulating a user's *command history*. This is primarily a convenience, a means of saving keystrokes.

Bash history commands:

1. **history**
2. **fc**

```
bash$ history
   1  mount /mnt/cdrom
    2  cd /mnt/cdrom
    3  ls
     ...
```

Internal variables associated with Bash history commands:

1. $HISTCMD
2. $HISTCONTROL
3. $HISTIGNORE
4. $HISTFILE
5. $HISTFILESIZE
6. $HISTSIZE
7. $HISTTIMEFORMAT (Bash, ver. 3.0 or later)
8. !!
9. !$
10. !#
11. !N
12. !-N
13. !STRING
14. !?STRING?
15. ^STRING^string^

Unfortunately, the Bash history tools find no use in scripting.

```
#!/bin/bash
# history.sh
# A (vain) attempt to use the 'history' command in a script.

history                       # No output.

var=$(history); echo "$var"  # $var is empty.

#  History commands are, by default, disabled within a script.
#  However, as dhw points out,
#+ set -o history
#+ enables the history mechanism.

set -o history
var=$(history); echo "$var"   # 1  var=$(history)
```

```
bash$ ./history.sh
(no output)
```

The Advancing in the Bash Shell site gives a good introduction to the use of history commands in Bash.

Appendix M. Sample .bashrc and .bash_profile Files

The ~/.bashrc file determines the behavior of interactive shells. A good look at this file can lead to a better understanding of Bash.

Emmanuel Rouat contributed the following very elaborate .bashrc file, written for a Linux system. He welcomes reader feedback on it.

Study the file carefully, and feel free to reuse code snippets and functions from it in your own .bashrc file or even in your scripts.

Example M-1. Sample .bashrc file

```
# =============================================================== #
#
# PERSONAL $HOME/.bashrc FILE for bash-3.0 (or later)
# By Emmanuel Rouat [no-email]
#
# Last modified: Tue Nov 20 22:04:47 CET 2012

#  This file is normally read by interactive shells only.
#+ Here is the place to define your aliases, functions and
#+ other interactive features like your prompt.
#
#  The majority of the code here assumes you are on a GNU
#+ system (most likely a Linux box) and is often based on code
#+ found on Usenet or Internet.
#
#  See for instance:
#  http://tldp.org/LDP/abs/html/index.html
#  http://www.caliban.org/bash
#  http://www.shelldorado.com/scripts/categories.html
#  http://www.dotfiles.org
#
#  The choice of colors was done for a shell with a dark background
#+ (white on black), and this is usually also suited for pure text-mode
#+ consoles (no X server available). If you use a white background,
#+ you'll have to do some other choices for readability.
#
#  This bashrc file is a bit overcrowded.
#  Remember, it is just just an example.
#  Tailor it to your needs.
#
# =============================================================== #

# --> Comments added by HOWTO author.

# If not running interactively, don't do anything
[ -z "$PS1" ] && return


#-------------------------------------------------------------
# Source global definitions (if any)
#-------------------------------------------------------------
```

```
if [ -f /etc/bashrc ]; then
      . /etc/bashrc   # --> Read /etc/bashrc, if present.
fi


#--------------------------------------------------------------
#  Automatic setting of $DISPLAY (if not set already).
#  This works for me - your mileage may vary. . . .
#  The problem is that different types of terminals give
#+ different answers to 'who am i' (rxvt in particular can be
#+ troublesome) - however this code seems to work in a majority
#+ of cases.
#--------------------------------------------------------------

function get_xserver ()
{
    case $TERM in
        xterm )
            XSERVER=$(who am i | awk '{print $NF}' | tr -d ')''(' )
            # Ane-Pieter Wieringa suggests the following alternative:
            #  I_AM=$(who am i)
            #  SERVER=${I_AM#*(}
            #  SERVER=${SERVER%*)}
            XSERVER=${XSERVER%%:*}
            ;;
            aterm | rxvt)
            # Find some code that works here. ...
            ;;
    esac
}

if [ -z ${DISPLAY:=""} ]; then
    get_xserver
    if [[ -z ${XSERVER}  || ${XSERVER} == $(hostname) ||
       ${XSERVER} == "unix" ]]; then
          DISPLAY=":0.0"          # Display on local host.
    else
       DISPLAY=${XSERVER}:0.0     # Display on remote host.
    fi
fi

export DISPLAY

#-------------------------------------------------------------
# Some settings
#-------------------------------------------------------------

#set -o nounset     # These  two options are useful for debugging.
#set -o xtrace
alias debug="set -o nounset; set -o xtrace"

ulimit -S -c 0      # Don't want coredumps.
set -o notify
set -o noclobber
set -o ignoreeof


# Enable options:
shopt -s cdspell
shopt -s cdable_vars
shopt -s checkhash
```

```
shopt -s checkwinsize
shopt -s sourcepath
shopt -s no_empty_cmd_completion
shopt -s cmdhist
shopt -s histappend histreedit histverify
shopt -s extglob       # Necessary for programmable completion.

# Disable options:
shopt -u mailwarn
unset MAILCHECK        # Don't want my shell to warn me of incoming mail.


#-------------------------------------------------------------
# Greeting, motd etc. ...
#-------------------------------------------------------------

# Color definitions (taken from Color Bash Prompt HowTo).
# Some colors might look different of some terminals.
# For example, I see 'Bold Red' as 'orange' on my screen,
# hence the 'Green' 'BRed' 'Red' sequence I often use in my prompt.


# Normal Colors
Black='\e[0;30m'        # Black
Red='\e[0;31m'          # Red
Green='\e[0;32m'        # Green
Yellow='\e[0;33m'       # Yellow
Blue='\e[0;34m'         # Blue
Purple='\e[0;35m'       # Purple
Cyan='\e[0;36m'         # Cyan
White='\e[0;37m'        # White

# Bold
BBlack='\e[1;30m'       # Black
BRed='\e[1;31m'         # Red
BGreen='\e[1;32m'       # Green
BYellow='\e[1;33m'      # Yellow
BBlue='\e[1;34m'        # Blue
BPurple='\e[1;35m'      # Purple
BCyan='\e[1;36m'        # Cyan
BWhite='\e[1;37m'       # White

# Background
On_Black='\e[40m'       # Black
On_Red='\e[41m'         # Red
On_Green='\e[42m'       # Green
On_Yellow='\e[43m'      # Yellow
On_Blue='\e[44m'        # Blue
On_Purple='\e[45m'      # Purple
On_Cyan='\e[46m'        # Cyan
On_White='\e[47m'       # White

NC="\e[m"               # Color Reset


ALERT=${BWhite}${On_Red} # Bold White on red background



echo -e "${BCyan}This is BASH ${BRed}${BASH_VERSION%.*}${BCyan}\
- DISPLAY on ${BRed}$DISPLAY${NC}\n"
date
```

```
if [ -x /usr/games/fortune ]; then
    /usr/games/fortune -s     # Makes our day a bit more fun.... :-)
fi

function _exit()              # Function to run upon exit of shell.
{
    echo -e "${BRed}Hasta la vista, baby${NC}"
}
trap _exit EXIT

#-------------------------------------------------------------
# Shell Prompt - for many examples, see:
#       http://www.debian-administration.org/articles/205
#       http://www.askapache.com/linux/bash-power-prompt.html
#       http://tldp.org/HOWTO/Bash-Prompt-HOWTO
#       https://github.com/nojhan/liquidprompt
#-------------------------------------------------------------
# Current Format: [TIME USER@HOST PWD] >
# TIME:
#    Green     == machine load is low
#    Orange    == machine load is medium
#    Red       == machine load is high
#    ALERT     == machine load is very high
# USER:
#    Cyan      == normal user
#    Orange    == SU to user
#    Red       == root
# HOST:
#    Cyan      == local session
#    Green     == secured remote connection (via ssh)
#    Red       == unsecured remote connection
# PWD:
#    Green     == more than 10% free disk space
#    Orange    == less than 10% free disk space
#    ALERT     == less than 5% free disk space
#    Red       == current user does not have write privileges
#    Cyan      == current filesystem is size zero (like /proc)
# >:
#    White     == no background or suspended jobs in this shell
#    Cyan      == at least one background job in this shell
#    Orange    == at least one suspended job in this shell
#
#    Command is added to the history file each time you hit enter,
#    so it's available to all shells (using 'history -a').


# Test connection type:
if [ -n "${SSH_CONNECTION}" ]; then
    CNX=${Green}        # Connected on remote machine, via ssh (good).
elif [[ "${DISPLAY%%:0*}" != "" ]]; then
    CNX=${ALERT}        # Connected on remote machine, not via ssh (bad).
else
    CNX=${BCyan}        # Connected on local machine.
fi

# Test user type:
if [[ ${USER} == "root" ]]; then
    SU=${Red}           # User is root.
elif [[ ${USER} != $(logname) ]]; then
    SU=${BRed}          # User is not login user.
else
    SU=${BCyan}         # User is normal (well ... most of us are).
```

```
fi



NCPU=$(grep -c 'processor' /proc/cpuinfo)    # Number of CPUs
SLOAD=$(( 100*${NCPU} ))        # Small load
MLOAD=$(( 200*${NCPU} ))        # Medium load
XLOAD=$(( 400*${NCPU} ))        # Xlarge load

# Returns system load as percentage, i.e., '40' rather than '0.40)'.
function load()
{
    local SYSLOAD=$(cut -d " " -f1 /proc/loadavg | tr -d '.')
    # System load of the current host.
    echo $((10#$SYSLOAD))       # Convert to decimal.
}

# Returns a color indicating system load.
function load_color()
{
    local SYSLOAD=$(load)
    if [ ${SYSLOAD} -gt ${XLOAD} ]; then
        echo -en ${ALERT}
    elif [ ${SYSLOAD} -gt ${MLOAD} ]; then
        echo -en ${Red}
    elif [ ${SYSLOAD} -gt ${SLOAD} ]; then
        echo -en ${BRed}
    else
        echo -en ${Green}
    fi
}

# Returns a color according to free disk space in $PWD.
function disk_color()
{
    if [ ! -w "${PWD}" ] ; then
        echo -en ${Red}
        # No 'write' privilege in the current directory.
    elif [ -s "${PWD}" ] ; then
        local used=$(command df -P "$PWD" |
                   awk 'END {print $5} {sub(/%/,"")}')
        if [ ${used} -gt 95 ]; then
            echo -en ${ALERT}           # Disk almost full (>95%).
        elif [ ${used} -gt 90 ]; then
            echo -en ${BRed}            # Free disk space almost gone.
        else
            echo -en ${Green}           # Free disk space is ok.
        fi
    else
        echo -en ${Cyan}
        # Current directory is size '0' (like /proc, /sys etc).
    fi
}

# Returns a color according to running/suspended jobs.
function job_color()
{
    if [ $(jobs -s | wc -l) -gt "0" ]; then
        echo -en ${BRed}
    elif [ $(jobs -r | wc -l) -gt "0" ] ; then
        echo -en ${BCyan}
    fi
```

```
}

# Adds some text in the terminal frame (if applicable).


# Now we construct the prompt.
PROMPT_COMMAND="history -a"
case ${TERM} in
  *term | rxvt | linux)
        PS1="\[\$(load_color)\][\A\[${NC}\] "
        # Time of day (with load info):
        PS1="\[\$(load_color)\][\A\[${NC}\] "
        # User@Host (with connection type info):
        PS1=${PS1}"\[${SU}\]\u\[${NC}\]@\[${CNX}\]\h\[${NC}\] "
        # PWD (with 'disk space' info):
        PS1=${PS1}"\[\$(disk_color)\]\W]\[${NC}\] "
        # Prompt (with 'job' info):
        PS1=${PS1}"\[\$(job_color)\]>\[${NC}\] "
        # Set title of current xterm:
        PS1=${PS1}"\[\e]0;[\u@\h] \w\a\]"
        ;;
    *)
        PS1="(\A \u@\h \W) > " # --> PS1="(\A \u@\h \w) > "
                               # --> Shows full pathname of current dir.
        ;;
esac



export TIMEFORMAT=$'\nreal %3R\tuser %3U\tsys %3S\tpcpu %P\n'
export HISTIGNORE="&:bg:fg:ll:h"
export HISTTIMEFORMAT="$(echo -e ${BCyan})[%d/%m %H:%M:%S]$(echo -e ${NC}) "
export HISTCONTROL=ignoredups
export HOSTFILE=$HOME/.hosts    # Put a list of remote hosts in ~/.hosts


#============================================================
#
#  ALIASES AND FUNCTIONS
#
#  Arguably, some functions defined here are quite big.
#  If you want to make this file smaller, these functions can
#+ be converted into scripts and removed from here.
#
#============================================================

#-------------------
# Personnal Aliases
#-------------------

alias rm='rm -i'
alias cp='cp -i'
alias mv='mv -i'
# -> Prevents accidentally clobbering files.
alias mkdir='mkdir -p'

alias h='history'
alias j='jobs -l'
alias which='type -a'
alias ..='cd ..'

# Pretty-print of some PATH variables:
```

```
alias path='echo -e ${PATH//:/\\n}'
alias libpath='echo -e ${LD_LIBRARY_PATH//:/\\n}'


alias du='du -kh'    # Makes a more readable output.
alias df='df -kTh'

#-------------------------------------------------------------
# The 'ls' family (this assumes you use a recent GNU ls).
#-------------------------------------------------------------
# Add colors for filetype and  human-readable sizes by default on 'ls':
alias ls='ls -h --color'
alias lx='ls -lXB'         #  Sort by extension.
alias lk='ls -lSr'         #  Sort by size, biggest last.
alias lt='ls -ltr'         #  Sort by date, most recent last.
alias lc='ls -ltcr'        #  Sort by/show change time,most recent last.
alias lu='ls -ltur'        #  Sort by/show access time,most recent last.

# The ubiquitous 'll': directories first, with alphanumeric sorting:
alias ll="ls -lv --group-directories-first"
alias lm='ll |more'        #  Pipe through 'more'
alias lr='ll -R'           #  Recursive ls.
alias la='ll -A'           #  Show hidden files.
alias tree='tree -Csuh'    #  Nice alternative to 'recursive ls' ...


#-------------------------------------------------------------
# Tailoring 'less'
#-------------------------------------------------------------

alias more='less'
export PAGER=less
export LESSCHARSET='latin1'
export LESSOPEN='|/usr/bin/lesspipe.sh %s 2>&-'
                # Use this if lesspipe.sh exists.
export LESS='-i -N -w  -z-4 -g -e -M -X -F -R -P%t?f%f \
:stdin .?pb%pb\%:?lbLine %lb:?bbByte %bb:-...'

# LESS man page colors (makes Man pages more readable).
export LESS_TERMCAP_mb=$'\E[01;31m'
export LESS_TERMCAP_md=$'\E[01;31m'
export LESS_TERMCAP_me=$'\E[0m'
export LESS_TERMCAP_se=$'\E[0m'
export LESS_TERMCAP_so=$'\E[01;44;33m'
export LESS_TERMCAP_ue=$'\E[0m'
export LESS_TERMCAP_us=$'\E[01;32m'


#-------------------------------------------------------------
# Spelling typos - highly personnal and keyboard-dependent :-)
#-------------------------------------------------------------

alias xs='cd'
alias vf='cd'
alias moer='more'
alias moew='more'
alias kk='ll'


#-------------------------------------------------------------
# A few fun ones
#-------------------------------------------------------------
```

```
# Adds some text in the terminal frame (if applicable).

function xtitle()
{
    case "$TERM" in
    *term* | rxvt)
        echo -en  "\e]0;$*\a" ;;
    *)  ;;
    esac
}


# Aliases that use xtitle
alias top='xtitle Processes on $HOST && top'
alias make='xtitle Making $(basename $PWD) ; make'

# .. and functions
function man()
{
    for i ; do
        xtitle The $(basename $1|tr -d .[:digit:]) manual
        command man -a "$i"
    done
}


#-------------------------------------------------------------
# Make the following commands run in background automatically:
#-------------------------------------------------------------

function te()  # wrapper around xemacs/gnuserv
{
    if [ "$(gnuclient -batch -eval t 2>&-)" == "t" ]; then
       gnuclient -q "$@";
    else
       ( xemacs "$@" &);
    fi
}

function soffice() { command soffice "$@" & }
function firefox() { command firefox "$@" & }
function xpdf() { command xpdf "$@" & }


#-------------------------------------------------------------
# File & strings related functions:
#-------------------------------------------------------------


# Find a file with a pattern in name:
function ff() { find . -type f -iname '*'"$*"'*' -ls ; }

# Find a file with pattern $1 in name and Execute $2 on it:
function fe() { find . -type f -iname '*'"${1:-}"'*' \
-exec ${2:-file} {} \;  ; }

#  Find a pattern in a set of files and highlight them:
#+ (needs a recent version of egrep).
function fstr()
{
    OPTIND=1
```

```
    local mycase=""
    local usage="fstr: find string in files.
Usage: fstr [-i] \"pattern\" [\"filename pattern\"] "
    while getopts :it opt
    do
        case "$opt" in
           i) mycase="-i " ;;
           *) echo "$usage"; return ;;
        esac
    done
    shift $(( $OPTIND - 1 ))
    if [ "$#" -lt 1 ]; then
        echo "$usage"
        return;
    fi
    find . -type f -name "${2:-*}" -print0 | \
xargs -0 egrep --color=always -sn ${case} "$1" 2>&- | more

}


function swap()
{ # Swap 2 filenames around, if they exist (from Uzi's bashrc).
    local TMPFILE=tmp.$$

    [ $# -ne 2 ] && echo "swap: 2 arguments needed" && return 1
    [ ! -e $1 ] && echo "swap: $1 does not exist" && return 1
    [ ! -e $2 ] && echo "swap: $2 does not exist" && return 1

    mv "$1" $TMPFILE
    mv "$2" "$1"
    mv $TMPFILE "$2"
}

function extract()      # Handy Extract Program
{
    if [ -f $1 ] ; then
        case $1 in
            *.tar.bz2)   tar xvjf $1     ;;
            *.tar.gz)    tar xvzf $1     ;;
            *.bz2)       bunzip2 $1      ;;
            *.rar)       unrar x $1      ;;
            *.gz)        gunzip $1       ;;
            *.tar)       tar xvf $1      ;;
            *.tbz2)      tar xvjf $1     ;;
            *.tgz)       tar xvzf $1     ;;
            *.zip)       unzip $1        ;;
            *.Z)         uncompress $1   ;;
            *.7z)        7z x $1         ;;
            *)           echo "'$1' cannot be extracted via >extract<" ;;
        esac
    else
        echo "'$1' is not a valid file!"
    fi
}


# Creates an archive (*.tar.gz) from given directory.
function maketar() { tar cvzf "${1%%/}.tar.gz"  "${1%%/}/"; }

# Create a ZIP archive of a file or folder.
function makezip() { zip -r "${1%%/}.zip" "$1" ; }
```

```
# Make your directories and files access rights sane.
function sanitize() { chmod -R u=rwX,g=rX,o= "$@" ;}

#-------------------------------------------------------------
# Process/system related functions:
#-------------------------------------------------------------


function my_ps() { ps $@ -u $USER -o pid,%cpu,%mem,bsdtime,command ; }
function pp() { my_ps f | awk '!/awk/ && $0~var' var=${1:-".*"} ; }


function killps()   # kill by process name
{
    local pid pname sig="-TERM"   # default signal
    if [ "$#" -lt 1 ] || [ "$#" -gt 2 ]; then
        echo "Usage: killps [-SIGNAL] pattern"
        return;
    fi
    if [ $# = 2 ]; then sig=$1 ; fi
    for pid in $(my_ps| awk '!/awk/ && $0~pat { print $1 }' pat=${!#} )
    do
        pname=$(my_ps | awk '$1~var { print $5 }' var=$pid )
        if ask "Kill process $pid <$pname> with signal $sig?"
            then kill $sig $pid
        fi
    done
}

function mydf()         # Pretty-print of 'df' output.
{                       # Inspired by 'dfc' utility.
    for fs ; do

        if [ ! -d $fs ]
        then
          echo -e $fs" :No such file or directory" ; continue
        fi

        local info=( $(command df -P $fs | awk 'END{ print $2,$3,$5 }') )
        local free=( $(command df -Pkh $fs | awk 'END{ print $4 }') )
        local nbstars=$(( 20 * ${info[1]} / ${info[0]} ))
        local out="["
        for ((j=0;j<20;j++)); do
            if [ ${j} -lt ${nbstars} ]; then
               out=$out"*"
            else
               out=$out"-"
            fi
        done
        out=${info[2]}" "$out"] ("$free" free on "$fs")"
        echo -e $out
    done
}


function my_ip() # Get IP adress on ethernet.
{
    MY_IP=$(/sbin/ifconfig eth0 | awk '/inet/ { print $2 } ' |
      sed -e s/addr://)
    echo ${MY_IP:-"Not connected"}
}
```

```
function ii()   # Get current host related info.
{
    echo -e "\nYou are logged on ${BRed}$HOST"
    echo -e "\n${BRed}Additionnal information:$NC " ; uname -a
    echo -e "\n${BRed}Users logged on:$NC " ; w -hs |
             cut -d " " -f1 | sort | uniq
    echo -e "\n${BRed}Current date :$NC " ; date
    echo -e "\n${BRed}Machine stats :$NC " ; uptime
    echo -e "\n${BRed}Memory stats :$NC " ; free
    echo -e "\n${BRed}Diskspace :$NC " ; mydf / $HOME
    echo -e "\n${BRed}Local IP Address :$NC" ; my_ip
    echo -e "\n${BRed}Open connections :$NC "; netstat -pan --inet;
    echo
}

#-------------------------------------------------------------
# Misc utilities:
#-------------------------------------------------------------

function repeat()       # Repeat n times command.
{
    local i max
    max=$1; shift;
    for ((i=1; i <= max ; i++)); do  # --> C-like syntax
        eval "$@";
    done
}


function ask()          # See 'killps' for example of use.
{
    echo -n "$@" '[y/n] ' ; read ans
    case "$ans" in
        y*|Y*) return 0 ;;
        *) return 1 ;;
    esac
}

function corename()   # Get name of app that created a corefile.
{
    for file ; do
        echo -n $file : ; gdb --core=$file --batch | head -1
    done
}



#=========================================================================
#
#  PROGRAMMABLE COMPLETION SECTION
#  Most are taken from the bash 2.05 documentation and from Ian McDonald's
# 'Bash completion' package (http://www.caliban.org/bash/#completion)
#  You will in fact need bash more recent then 3.0 for some features.
#
#  Note that most linux distributions now provide many completions
# 'out of the box' - however, you might need to make your own one day,
#  so I kept those here as examples.
#=========================================================================

if [ "${BASH_VERSION%.*}" \< "3.0" ]; then
    echo "You will need to upgrade to version 3.0 for full \
```

```
          programmable completion features"
    return
fi

shopt -s extglob        # Necessary.

complete -A hostname   rsh rcp telnet rlogin ftp ping disk
complete -A export     printenv
complete -A variable   export local readonly unset
complete -A enabled    builtin
complete -A alias      alias unalias
complete -A function   function
complete -A user       su mail finger

complete -A helptopic  help     # Currently same as builtins.
complete -A shopt      shopt
complete -A stopped -P '%' bg
complete -A job -P '%'     fg jobs disown

complete -A directory  mkdir rmdir
complete -A directory   -o default cd

# Compression
complete -f -o default -X '*.+(zip|ZIP)'  zip
complete -f -o default -X '!*.+(zip|ZIP)' unzip
complete -f -o default -X '*.+(z|Z)'      compress
complete -f -o default -X '!*.+(z|Z)'     uncompress
complete -f -o default -X '*.+(gz|GZ)'    gzip
complete -f -o default -X '!*.+(gz|GZ)'   gunzip
complete -f -o default -X '*.+(bz2|BZ2)'  bzip2
complete -f -o default -X '!*.+(bz2|BZ2)' bunzip2
complete -f -o default -X '!*.+(zip|ZIP|z|Z|gz|GZ|bz2|BZ2)' extract


# Documents - Postscript,pdf,dvi.....
complete -f -o default -X '!*.+(ps|PS)'  gs ghostview ps2pdf ps2ascii
complete -f -o default -X \
'!*.+(dvi|DVI)' dvips dvipdf xdvi dviselect dvitype
complete -f -o default -X '!*.+(pdf|PDF)' acroread pdf2ps
complete -f -o default -X '!*.@(@(?(e)ps|?(E)PS|pdf|PDF)?\
(.gz|.GZ|.bz2|.BZ2|.Z))' gv ggv
complete -f -o default -X '!*.texi*' makeinfo texi2dvi texi2html texi2pdf
complete -f -o default -X '!*.tex' tex latex slitex
complete -f -o default -X '!*.lyx' lyx
complete -f -o default -X '!*.+(htm*|HTM*)' lynx html2ps
complete -f -o default -X \
'!*.+(doc|DOC|xls|XLS|ppt|PPT|sx?|SX?|csv|CSV|od?|OD?|ott|OTT)' soffice

# Multimedia
complete -f -o default -X \
'!*.+(gif|GIF|jp*g|JP*G|bmp|BMP|xpm|XPM|png|PNG)' xv gimp ee gqview
complete -f -o default -X '!*.+(mp3|MP3)' mpg123 mpg321
complete -f -o default -X '!*.+(ogg|OGG)' ogg123
complete -f -o default -X \
'!*.@(mp[23]|MP[23]|ogg|OGG|wav|WAV|pls|\
m3u|xm|mod|s[3t]m|it|mtm|ult|flac)' xmms
complete -f -o default -X '!*.@(mp?(e)g|MP?(E)G|wma|avi|AVI|\
asf|vob|VOB|bin|dat|vcd|ps|pes|fli|viv|rm|ram|yuv|mov|MOV|qt|\
QT|wmv|mp3|MP3|ogg|OGG|ogm|OGM|mp4|MP4|wav|WAV|asx|ASX)' xine
```

```
complete -f -o default -X '!*.pl'  perl perl5


#  This is a 'universal' completion function - it works when commands have
#+ a so-called 'long options' mode , ie: 'ls --all' instead of 'ls -a'
#  Needs the '-o' option of grep
#+ (try the commented-out version if not available).

#  First, remove '=' from completion word separators
#+ (this will allow completions like 'ls --color=auto' to work correctly).

COMP_WORDBREAKS=${COMP_WORDBREAKS/=/}


_get_longopts()
{
  #$1 --help | sed  -e '/--/!d' -e 's/.*--\([^[:space:].,]*\).*/--\1/'| \
  #grep ^"$2" |sort -u ;
    $1 --help | grep -o -e "--[^[:space:].,]*" | grep -e "$2" |sort -u
}

_longopts()
{
    local cur
    cur=${COMP_WORDS[COMP_CWORD]}

    case "${cur:-*}" in
       -*)      ;;
        *)      return ;;
    esac

    case "$1" in
       \~*)     eval cmd="$1" ;;
         *)     cmd="$1" ;;
    esac
    COMPREPLY=( $(_get_longopts ${1} ${cur} ) )
}
complete  -o default -F _longopts configure bash
complete  -o default -F _longopts wget id info a2ps ls recode

_tar()
{
    local cur ext regex tar untar

    COMPREPLY=()
    cur=${COMP_WORDS[COMP_CWORD]}

    # If we want an option, return the possible long options.
    case "$cur" in
        -*)     COMPREPLY=( $(_get_longopts $1 $cur ) ); return 0;;
    esac

    if [ $COMP_CWORD -eq 1 ]; then
        COMPREPLY=( $( compgen -W 'c t x u r d A' -- $cur ) )
        return 0
    fi

    case "${COMP_WORDS[1]}" in
        ?(-)c*f)
            COMPREPLY=( $( compgen -f $cur ) )
            return 0
            ;;
```

```
        +([^Izjy])f)
            ext='tar'
            regex=$ext
            ;;
        *z*f)
            ext='tar.gz'
            regex='t\(ar\.\)\(gz\|Z\)'
            ;;
        *[Ijy]*f)
            ext='t?(ar.)bz?(2)'
            regex='t\(ar\.\)bz2\?'
            ;;
        *)
            COMPREPLY=( $( compgen -f $cur ) )
            return 0
            ;;

    esac

    if [[ "$COMP_LINE" == tar*.$ext' '* ]]; then
        # Complete on files in tar file.
        #
        # Get name of tar file from command line.
        tar=$( echo "$COMP_LINE" | \
                        sed -e 's|^.* \([^ ]*'$regex'\) .*$|\1|' )
        # Devise how to untar and list it.
        untar=t${COMP_WORDS[1]//[^Izjyf]/}

        COMPREPLY=( $( compgen -W "$( echo $( tar $untar $tar \
                                2>/dev/null ) )" -- "$cur" ) )
        return 0

    else
        # File completion on relevant files.
        COMPREPLY=( $( compgen -G $cur\*.$ext ) )

    fi

    return 0

}

complete -F _tar -o default tar

_make()
{
    local mdef makef makef_dir="." makef_inc gcmd cur prev i;
    COMPREPLY=();
    cur=${COMP_WORDS[COMP_CWORD]};
    prev=${COMP_WORDS[COMP_CWORD-1]};
    case "$prev" in
        -*f)
            COMPREPLY=($(compgen -f $cur ));
            return 0
            ;;
    esac;
    case "$cur" in
        -*)
            COMPREPLY=($(_get_longopts $1 $cur ));
            return 0
            ;;
    esac;
```

```
    # ... make reads
    #          GNUmakefile,
    #     then makefile
    #     then Makefile ...
    if [ -f ${makef_dir}/GNUmakefile ]; then
        makef=${makef_dir}/GNUmakefile
    elif [ -f ${makef_dir}/makefile ]; then
        makef=${makef_dir}/makefile
    elif [ -f ${makef_dir}/Makefile ]; then
        makef=${makef_dir}/Makefile
    else
       makef=${makef_dir}/*.mk         # Local convention.
    fi


    #  Before we scan for targets, see if a Makefile name was
    #+ specified with -f.
    for (( i=0; i < ${#COMP_WORDS[@]}; i++ )); do
        if [[ ${COMP_WORDS[i]} == -f ]]; then
            # eval for tilde expansion
            eval makef=${COMP_WORDS[i+1]}
            break
        fi
    done
    [ ! -f $makef ] && return 0

    # Deal with included Makefiles.
    makef_inc=$( grep -E '^-?include' $makef |
                 sed -e "s,^.* ,"$makef_dir"/," )
    for file in $makef_inc; do
        [ -f $file ] && makef="$makef $file"
    done


    #  If we have a partial word to complete, restrict completions
    #+ to matches of that word.
    if [ -n "$cur" ]; then gcmd='grep "^$cur"' ; else gcmd=cat ; fi

    COMPREPLY=( $( awk -F':' '/^[a-zA-Z0-9][^$#\/\t=]*:([^=]|$)/ \
                               {split($1,A,/ /);for(i in A)print A[i]}' \
                                $makef 2>/dev/null | eval $gcmd  ))

}

complete -F _make -X '+($*|*.[cho])' make gmake pmake




_killall()
{
    local cur prev
    COMPREPLY=()
    cur=${COMP_WORDS[COMP_CWORD]}

    #  Get a list of processes
    #+ (the first sed evaluation
    #+ takes care of swapped out processes, the second
    #+ takes care of getting the basename of the process).
    COMPREPLY=( $( ps -u $USER -o comm  | \
        sed -e '1,1d' -e 's#[]\[]##g' -e 's#^.*/##'| \
```

```
        awk '{if ($0 ~ /^'$cur'/) print $0}' ))

    return 0
}

complete -F _killall killall killps



# Local Variables:
# mode:shell-script
# sh-shell:bash
# End:
```

And, here is a snippet from Andrzej Szelachowski's instructive `.bash_profile` file.

Example M-2. `.bash_profile` file

```
# From Andrzej Szelachowski's ~/.bash_profile:


#  Note that a variable may require special treatment
#+ if it will be exported.

DARKGRAY='\e[1;30m'
LIGHTRED='\e[1;31m'
GREEN='\e[32m'
YELLOW='\e[1;33m'
LIGHTBLUE='\e[1;34m'
NC='\e[m'

PCT="\`if [[ \$EUID -eq 0 ]]; then T='$LIGHTRED' ; else T='$LIGHTBLUE'; fi;
echo \$T \`"

#  For "literal" command substitution to be assigned to a variable,
#+ use escapes and double quotes:
#+       PCT="\` ... \`" . . .
#  Otherwise, the value of PCT variable is assigned only once,
#+ when the variable is exported/read from .bash_profile,
#+ and it will not change afterwards even if the user ID changes.


PS1="\n$GREEN[\w] \n$DARKGRAY($PCT\t$DARKGRAY)-($PCT\u$DARKGRAY)-($PCT\!
$DARKGRAY)$YELLOW-> $NC"

#  Escape a variables whose value changes:
#        if [[ \$EUID -eq 0 ]],
#  Otherwise the value of the EUID variable will be assigned only once,
#+ as above.

#  When a variable is assigned, it should be called escaped:
#+       echo \$T,
#  Otherwise the value of the T variable is taken from the moment the PCT
#+ variable is exported/read from .bash_profile.
#  So, in this example it would be null.

#  When a variable's value contains a semicolon it should be strong quoted:
#        T='$LIGHTRED',
#  Otherwise, the semicolon will be interpreted as a command separator.
```

```
#  Variables PCT and PS1 can be merged into a new PS1 variable:

PS1="\`if [[ \$EUID -eq 0 ]]; then PCT='$LIGHTRED';
else PCT='$LIGHTBLUE'; fi;
echo '\n$GREEN[\w] \n$DARKGRAY('\$PCT'\t$DARKGRAY)-\
('\$PCT'\u$DARKGRAY)-('\$PCT'\!$DARKGRAY)$YELLOW-> $NC'\`"

# The trick is to use strong quoting for parts of old PS1 variable.
```

Appendix N. Converting DOS Batch Files to Shell Scripts

Quite a number of programmers learned scripting on a PC running DOS. Even the crippled DOS batch file language allowed writing some fairly powerful scripts and applications, though they often required extensive kludges and workarounds. Occasionally, the need still arises to convert an old DOS batch file to a UNIX shell script. This is generally not difficult, as DOS batch file operators are only a limited subset of the equivalent shell scripting ones.

Table N-1. Batch file keywords / variables / operators, and their shell equivalents

Batch File Operator	Shell Script Equivalent	Meaning
`%`	$	command-line parameter prefix
`/`	-	command option flag
`\`	/	directory path separator
`==`	=	(equal-to) string comparison test
`!==!`	!=	(not equal-to) string comparison test
`|`	|	pipe
`@`	set `+v`	do not echo current command
`*`	*	filename "wild card"
`>`	>	file redirection (overwrite)
`>>`	>>	file redirection (append)
`<`	<	redirect `stdin`
`%VAR%`	$VAR	environmental variable
`REM`	#	comment
`NOT`	!	negate following test
`NUL`	`/dev/null`	"black hole" for burying command output
`ECHO`	echo	echo (many more option in Bash)
`ECHO.`	echo	echo blank line
`ECHO OFF`	set `+v`	do not echo command(s) following
`FOR %%VAR IN (LIST) DO`	for var in [list]; do	"for" loop
`:LABEL`	none (unnecessary)	label
`GOTO`	none (use a function)	jump to another location in the script
`PAUSE`	sleep	pause or wait an interval
`CHOICE`	case or select	menu choice
`IF`	if	if-test

`IF EXIST` *`FILENAME`*	if [-e filename]	test if file exists
`IF !%N==!`	if [-z "$N"]	if replaceable parameter "N" not present
`CALL`	source or . (dot operator)	"include" another script
`COMMAND /C`	source or . (dot operator)	"include" another script (same as CALL)
`SET`	export	set an environmental variable
`SHIFT`	shift	left shift command-line argument list
`SGN`	-lt or -gt	sign (of integer)
`ERRORLEVEL`	$?	exit status
`CON`	`stdin`	"console" (`stdin`)
`PRN`	`/dev/lp0`	(generic) printer device
`LPT1`	`/dev/lp0`	first printer device
`COM1`	`/dev/ttyS0`	first serial port

Batch files usually contain DOS commands. These must be translated into their UNIX equivalents in order to convert a batch file into a shell script.

Table N-2. DOS commands and their UNIX equivalents

DOS Command	UNIX Equivalent	Effect
`ASSIGN`	ln	link file or directory
`ATTRIB`	chmod	change file permissions
`CD`	cd	change directory
`CHDIR`	cd	change directory
`CLS`	clear	clear screen
`COMP`	diff, comm, cmp	file compare
`COPY`	cp	file copy
`Ctl-C`	Ctl-C	break (signal)
`Ctl-Z`	Ctl-D	EOF (end-of-file)
`DEL`	rm	delete file(s)
`DELTREE`	rm -rf	delete directory recursively
`DIR`	ls -l	directory listing
`ERASE`	rm	delete file(s)
`EXIT`	exit	exit current process
`FC`	comm, cmp	file compare
`FIND`	grep	find strings in files
`MD`	mkdir	make directory

MKDIR	mkdir	make directory
MORE	more	text file paging filter
MOVE	mv	move
PATH	$PATH	path to executables
REN	mv	rename (move)
RENAME	mv	rename (move)
RD	rmdir	remove directory
RMDIR	rmdir	remove directory
SORT	sort	sort file
TIME	date	display system time
TYPE	cat	output file to stdout
XCOPY	cp	(extended) file copy

Virtually all UNIX and shell operators and commands have many more options and enhancements than their DOS and batch file counterparts. Many DOS batch files rely on auxiliary utilities, such as **ask.com**, a crippled counterpart to read.

DOS supports only a very limited and incompatible subset of filename wild-card expansion, recognizing just the * and ? characters.

Converting a DOS batch file into a shell script is generally straightforward, and the result ofttimes reads better than the original.

Example N-1. VIEWDATA.BAT: DOS Batch File

```
REM VIEWDATA

REM INSPIRED BY AN EXAMPLE IN "DOS POWERTOOLS"
REM                           BY PAUL SOMERSON


@ECHO OFF

IF !%1==! GOTO VIEWDATA
REM  IF NO COMMAND-LINE ARG...
FIND "%1" C:\BOZO\BOOKLIST.TXT
GOTO EXIT0
REM  PRINT LINE WITH STRING MATCH, THEN EXIT.

:VIEWDATA
TYPE C:\BOZO\BOOKLIST.TXT | MORE
REM  SHOW ENTIRE FILE, 1 PAGE AT A TIME.

:EXIT0
```

The script conversion is somewhat of an improvement. [152]

Example N-2. *viewdata.sh*: Shell Script Conversion of VIEWDATA.BAT

```
#!/bin/bash
# viewdata.sh
# Conversion of VIEWDATA.BAT to shell script.

DATAFILE=/home/bozo/datafiles/book-collection.data
ARGNO=1

# @ECHO OFF                  Command unnecessary here.

if [ $# -lt "$ARGNO" ]    # IF !%1==! GOTO VIEWDATA
then
  less $DATAFILE          # TYPE C:\MYDIR\BOOKLIST.TXT | MORE
else
  grep "$1" $DATAFILE     # FIND "%1" C:\MYDIR\BOOKLIST.TXT
fi

exit 0                    # :EXIT0

#  GOTOs, labels, smoke-and-mirrors, and flimflam unnecessary.
#  The converted script is short, sweet, and clean,
#+ which is more than can be said for the original.
```

Ted Davis' Shell Scripts on the PC site had a set of comprehensive tutorials on the old-fashioned art of batch file programming. Unfortunately the page has vanished without a trace.

Appendix O. Exercises

The exercises that follow test and extend your knowledge of scripting. Think of them as a challenge, as an entertaining way to take you further along the stony path toward UNIX wizardry.

```
      On a dingy side street in a run-down section of Hoboken, New Jersey,
      there sits a nondescript squat two-story brick building with an inscripti
      incised on a marble plate in its wall:

                       Bash Scripting Hall of Fame.

      Inside, among various dusty uninteresting exhibits is a corroding,
      cobweb-festooned brass plaque inscribed with a short, very short
      list of those few persons who have successfully mastered the material
      in the Advanced Bash Scripting
Guide, as evidenced by their performance
      on the following Exercise sections.

      (Alas, the author of the ABS
Guide is not represented among the exhibits.
      This is possibly due to malicious rumors about lack of
credentials and
      deficient scripting skills.)
```

O.1. Analyzing Scripts

Examine the following script. Run it, then explain what it does. Annotate the script and rewrite it in a more compact and elegant manner.

```
#!/bin/bash

MAX=10000


  for((nr=1; nr<$MAX; nr++))
  do

    let "t1 = nr % 5"
    if [ "$t1" -ne 3 ]
    then
      continue
    fi

    let "t2 = nr % 7"
    if [ "$t2" -ne 4 ]
    then
      continue
    fi

    let "t3 = nr % 9"
    if [ "$t3" -ne 5 ]
    then
```

```
      continue
    fi

  break   # What happens when you comment out this line? Why?

  done

  echo "Number = $nr"


exit 0
```

Explain what the following script does. It is really just a parameterized command-line pipe.

```
#!/bin/bash

DIRNAME=/usr/bin
FILETYPE="shell script"
LOGFILE=logfile

file "$DIRNAME"/* | fgrep "$FILETYPE" | tee $LOGFILE | wc -l

exit 0
```

Examine and explain the following script. For hints, you might refer to the listings for find and stat.

```
#!/bin/bash

# Author:  Nathan Coulter
# This code is released to the public domain.
# The author gave permission to use this code snippet in the ABS Guide.

find -maxdepth 1 -type f -printf '%f\000'  | {
   while read -d $'\000'; do
      mv "$REPLY" "$(date -d "$(stat -c '%y' "$REPLY") " '+%Y%m%d%H%M%S'
      )-$REPLY"
   done
}

# Warning: Test-drive this script in a "scratch" directory.
# It will somehow affect all the files there.
```

A reader sent in the following code snippet.

```
while read LINE
do
  echo $LINE
done < `tail -f /var/log/messages`
```

He wished to write a script tracking changes to the system log file, `/var/log/messages`. Unfortunately, the above code block hangs and does nothing useful. Why? Fix this so it does work. (Hint: rather than redirecting the `stdin` of the loop, try a pipe.)

Analyze the following "one-liner" (here split into two lines for clarity) contributed by Rory Winston:

```
export SUM=0; for f in $(find src -name "*.java");
do export SUM=$(($SUM + $(wc -l $f | awk '{ print $1 }'))); done; echo $SUM
```

Hint: First, break the script up into bite-sized sections. Then, carefully examine its use of double-parentheses arithmetic, the export command, the find command, the wc command, and awk.

Analyze Example A-10, and reorganize it in a simplified and more logical style. See how many of the variables can be eliminated, and try to optimize the script to speed up its execution time.

Alter the script so that it accepts any ordinary ASCII text file as input for its initial "generation". The script will read the first *$ROW*$COL* characters, and set the occurrences of vowels as "living" cells. Hint: be sure to translate the spaces in the input file to underscore characters.

O.2. Writing Scripts

Write a script to carry out each of the following tasks.

EASY

Self-reproducing Script
: Write a script that backs itself up, that is, copies itself to a file named `backup.sh`.

  Hint: Use the cat command and the appropriate positional parameter.

Home Directory Listing
: Perform a recursive directory listing on the user's home directory and save the information to a file. Compress the file, have the script prompt the user to insert a USB flash drive, then press **ENTER**. Finally, save the file to the flash drive after making certain the flash drive has properly mounted by parsing the output of df. Note that the flash drive must be *unmounted* before it is removed.

Converting for loops to while and until loops
: Convert the *for loops* in Example 11-1 to *while loops*. Hint: store the data in an array and step through the array elements.

  Having already done the "heavy lifting," now convert the loops in the example to *until loops*.

Changing the line spacing of a text file
: Write a script that reads each line of a target file, then writes the line back to `stdout`, but with an extra blank line following. This has the effect of *double-spacing* the file.

  Include all necessary code to check whether the script gets the necessary command-line argument (a filename), and whether the specified file exists.

  When the script runs correctly, modify it to *triple-space* the target file.

  Finally, write a script to remove all blank lines from the target file, *single-spacing* it.

Backwards Listing
: Write a script that echoes itself to `stdout`, but *backwards*.

Automatically Decompressing Files

Given a list of filenames as input, this script queries each target file (parsing the output of the file command) for the type of compression used on it. Then the script automatically invokes the appropriate decompression command (**gunzip**, **bunzip2**, **unzip**, **uncompress**, or whatever). If a target file is not compressed, the script emits a warning message, but takes no other action on that particular file.

Unique System ID

Generate a "unique" 6-digit hexadecimal identifier for your computer. Do *not* use the flawed hostid command. Hint: **md5sum `/etc/passwd`**, then select the first 6 digits of output.

Backup

Archive as a "tarball" (`*.tar.gz` file) all the files in your home directory tree (`/home/your-name`) that have been modified in the last 24 hours. Hint: use find.

Optional: you may use this as the basis of a *backup* script.

Checking whether a process is still running

Given a process ID (*PID*) as an argument, this script will check, at user-specified intervals, whether the given process is still running. You may use the ps and sleep commands.

Primes

Print (to `stdout`) all prime numbers between 60000 and 63000. The output should be nicely formatted in columns (hint: use printf).

Lottery Numbers

One type of lottery involves picking five different numbers, in the range of 1 - 50. Write a script that generates five pseudorandom numbers in this range, *with no duplicates*. The script will give the option of echoing the numbers to `stdout` or saving them to a file, along with the date and time the particular number set was generated. (If your script consistently generates *winning* lottery numbers, then you can retire on the proceeds and leave shell scripting to those of us who have to work for a living.)

INTERMEDIATE

Integer or String

Write a script function that determines if an argument passed to it is an integer or a string. The function will return TRUE (0) if passed an integer, and FALSE (1) if passed a string.

Hint: What does the following expression return when `$1` is *not* an integer?

```
expr $1 + 0
```

ASCII to Integer

The *atoi* function in **C** converts a string character to an integer. Write a shell script function that performs the same operation. Likewise, write a shell script function that does the inverse, mirroring the **C** *itoa* function which converts an integer into an ASCII character.

Managing Disk Space

List, one at a time, all files larger than 100K in the `/home/username` directory tree. Give the user the option to delete or compress the file, then proceed to show the next one. Write to a logfile the names of all deleted files and the deletion times.

Banner

Simulate the functionality of the deprecated banner command in a script.

Removing Inactive Accounts

Inactive accounts on a network server waste disk space and may become a security risk. Write an administrative script (to be invoked by *root* or the cron daemon) that checks for and deletes user accounts that have not been accessed within the last 90 days.

Enforcing Disk Quotas

Write a script for a multi-user system that checks users' disk usage. If a user surpasses a preset limit (500 MB, for example) in her `/home/username` directory, then the script automatically sends her a "pigout" warning e-mail.

The script will use the du and mail commands. As an option, it will allow setting and enforcing quotas using the quota and setquota commands.

Logged in User Information

For all logged in users, show their real names and the time and date of their last login.

Hint: use who, lastlog, and parse `/etc/passwd`.

Safe Delete

Implement, as a script, a "safe" delete command, `sdel.sh`. Filenames passed as command-line arguments to this script are not deleted, but instead gzipped if not already compressed (use file to check), then moved to a `~/TRASH` directory. Upon invocation, the script checks the `~/TRASH` directory for files older than 48 hours and permanently deletes them. (An better alternative might be to have a second script handle this, periodically invoked by the cron daemon.)

Extra credit: Write the script so it can handle files and directories recursively. This would give it the capability of "safely deleting" entire directory structures.

Making Change

What is the most efficient way to make change for $1.68, using only coins in common circulations (up to 25c)? It's 6 quarters, 1 dime, a nickel, and three cents.

Given any arbitrary command-line input in dollars and cents ($*.??), calculate the change, using the minimum number of coins. If your home country is not the United States, you may use your local currency units instead. The script will need to parse the command-line input, then change it to multiples of the smallest monetary unit (cents or whatever). Hint: look at Example 24-8.

Quadratic Equations

Solve a *quadratic* equation of the form *`Ax^2 + Bx + C = 0`*. Have a script take as arguments the coefficients, **`A`**, **`B`**, and **`C`**, and return the solutions to five decimal places.

Hint: pipe the coefficients to bc, using the well-known formula, *`x = ( -B +/- sqrt( B^2 - 4AC ) ) / 2A`*.

Table of Logarithms

Using the bc and printf commands, print out a nicely-formatted table of eight-place natural logarithms in the interval between 0.00 and 100.00, in steps of .01.

Hint: *bc* requires the `-l` option to load the math library.

Unicode Table

Using Example T-1 as a template, write a script that prints to a file a complete Unicode table.

Hint: Use the `-e` option to echo: **echo -e '\uXXXX'**, where *XXXX* is the Unicode numerical character designation. This requires version 4.2 or later of Bash.

Sum of Matching Numbers

Find the sum of all five-digit numbers (in the range 10000 - 99999) containing *exactly two* out of the following set of digits: { 4, 5, 6 }. These may repeat within the same number, and if so, they count once for each occurrence.

Some examples of *matching numbers* are 42057, 74638, and 89515.

Lucky Numbers

A *lucky number* is one whose individual digits add up to 7, in successive additions. For example, 62431 is a *lucky number* (6 + 2 + 4 + 3 + 1 = 16, 1 + 6 = 7). Find all the *lucky numbers* between 1000 and 10000.

Craps

Borrowing the ASCII graphics from Example A-40, write a script that plays the well-known gambling game of *craps*. The script will accept bets from one or more players, roll the dice, and keep track of wins and losses, as well as of each player's bankroll.

Tic-tac-toe

Write a script that plays the child's game of *tic-tac-toe* against a human player. The script will let the human choose whether to take the first move. The script will follow an optimal strategy, and therefore never lose. To simplify matters, you may use ASCII graphics:

```
   o | x |
   ----------
     | x |
   ----------
     | o |

   Your move, human (row, column)?
```

Alphabetizing a String

Alphabetize (in ASCII order) an arbitrary string read from the command-line.

Parsing

Parse `/etc/passwd`, and output its contents in nice, easy-to-read tabular form.

Logging Logins

Parse `/var/log/messages` to produce a nicely formatted file of user logins and login times. The script may need to run as *root*. (Hint: Search for the string "LOGIN.")

Pretty-Printing a Data File

Certain database and spreadsheet packages use save-files with the fields separated by commas, commonly referred to as *comma-separated values* or CSVs. Other applications often need to parse these files.

Given a data file with comma-separated fields, of the form:

```
Jones,Bill,235 S. Williams St.,Denver,CO,80221,(303) 244-7989
Smith,Tom,404 Polk Ave.,Los Angeles,CA,90003,(213) 879-5612
...
```

Reformat the data and print it out to `stdout` in labeled, evenly-spaced columns.

Justification

Given ASCII text input either from `stdin` or a file, adjust the word spacing to right-justify each line to a user-specified line-width, then send the output to `stdout`.

Mailing List

Using the mail command, write a script that manages a simple mailing list. The script automatically e-mails the monthly company newsletter, read from a specified text file, and sends it to all the addresses on the mailing list, which the script reads from another specified file.

Generating Passwords

Generate pseudorandom 8-character passwords, using characters in the ranges [0-9], [A-Z], [a-z]. Each password must contain at least two digits.

Monitoring a User

You suspect that one particular user on the network has been abusing her privileges and possibly attempting to hack the system. Write a script to automatically monitor and log her activities when she's signed on. The log file will save entries for the previous week, and delete those entries more than seven days old.

You may use last, lastlog, and lastcomm to aid your surveillance of the suspected fiend.

Checking for Broken Links

Using lynx with the `-traversal` option, write a script that checks a Web site for broken links.

DIFFICULT

Testing Passwords

Write a script to check and validate passwords. The object is to flag "weak" or easily guessed password candidates.

A trial password will be input to the script as a command-line parameter. To be considered acceptable, a password must meet the following minimum qualifications:

◊ Minimum length of 8 characters
◊ Must contain at least one numeric character
◊ Must contain at least one of the following non-alphabetic characters: @, #, $, %, &, *, +, -, =

Optional:

◊ Do a dictionary check on every sequence of at least four consecutive alphabetic characters in the password under test. This will eliminate passwords containing embedded "words" found in a standard dictionary.
◊ Enable the script to check all the passwords on your system. These do not reside in `/etc/passwd`.

This exercise tests mastery of Regular Expressions.

Cross Reference

Write a script that generates a *cross-reference* (*concordance*) on a target file. The output will be a listing of all word occurrences in the target file, along with the line numbers in which each word occurs. Traditionally, *linked list* constructs would be used in such applications. Therefore, you should investigate arrays in the course of this exercise. Example 16-12 is probably *not* a good place to start.

Square Root

Write a script to calculate square roots of numbers using *Newton's Method.*

The algorithm for this, expressed as a snippet of Bash pseudo-code is:

```
#  (Isaac) Newton's Method for speedy extraction
#+ of square roots.

guess = $argument
#  $argument is the number to find the square root of.
#  $guess is each successive calculated "guess" -- or trial solution --
#+ of the square root.
#  Our first "guess" at a square root is the argument itself.

oldguess = 0
# $oldguess is the previous $guess.

tolerance = .000001
# To how close a tolerance we wish to calculate.

loopcnt = 0
# Let's keep track of how many times through the loop.
# Some arguments will require more loop iterations than others.
```

```
while [ ABS( $guess $oldguess ) -gt $tolerance ]
#       ^^^^^^^^^^^^^^^^^^^^^^^ Fix up syntax, of course.

#      "ABS" is a (floating point) function to find the absolute value
#+      of the difference between the two terms.
#             So, as long as difference between current and previous
#+            trial solution (guess) exceeds the tolerance, keep looping.

do
   oldguess = $guess  # Update $oldguess to previous $guess.

#  =======================================================
   guess = ( $oldguess + ( $argument / $oldguess ) ) / 2.0
#        = 1/2 ( ($oldguess **2 + $argument) / $oldguess )
#  equivalent to:
#        = 1/2 ( $oldguess + $argument / $oldguess )
#  that is, "averaging out" the trial solution and
#+ the proportion of argument deviation
#+ (in effect, splitting the error in half).
#  This converges on an accurate solution
#+ with surprisingly few loop iterations . . .
#+ for arguments > $tolerance, of course.
#  =======================================================

   (( loopcnt++ ))     # Update loop counter.
done
```

It's a simple enough recipe, and *seems* at first glance easy enough to convert into a working Bash script. The problem, though, is that Bash has no native support for floating point numbers. So, the script writer needs to use bc or possibly awk to convert the numbers and do the calculations. It could get rather messy . . .

Logging File Accesses

Log all accesses to the files in `/etc` during the course of a single day. This information should include the filename, user name, and access time. If any alterations to the files take place, that will be flagged. Write this data as tabular (tab-separated) formatted records in a logfile.

Monitoring Processes

Write a script to continually monitor all running processes and to keep track of how many child processes each parent spawns. If a process spawns more than five children, then the script sends an e-mail to the system administrator (or *root*) with all relevant information, including the time, PID of the parent, PIDs of the children, etc. The script appends a report to a log file every ten minutes.

Strip Comments

Strip all comments from a shell script whose name is specified on the command-line. Note that the initial #! line must not be stripped out.

Strip HTML Tags

Strip all the HTML tags from a specified HTML file, then reformat it into lines between 60 and 75 characters in length. Reset paragraph and block spacing, as appropriate, and convert HTML tables to their approximate text equivalent.

XML Conversion

Convert an XML file to both HTML and text format.

Optional: A script that converts Docbook/SGML to XML.

Chasing Spammers

Write a script that analyzes a spam e-mail by doing DNS lookups on the IP addresses in the headers to identify the relay hosts as well as the originating ISP. The script will forward the unaltered spam message to the responsible ISPs. Of course, it will be necessary to filter out *your own ISP's IP address*, so you don't end up complaining about yourself.

As necessary, use the appropriate network analysis commands.

For some ideas, see Example 16-41 and Example A-28.

Optional: Write a script that searches through a list of e-mail messages and deletes the spam according to specified filters.

Creating man pages

Write a script that automates the process of creating man pages.

Given a text file which contains information to be formatted into a *man page*, the script will read the file, then invoke the appropriate groff commands to output the corresponding *man page* to `stdout`. The text file contains blocks of information under the standard *man page* headings, i.e., NAME, SYNOPSIS, DESCRIPTION, etc.

Example A-39 is an instructive first step.

Hex Dump

Do a hex(adecimal) dump on a binary file specified as an argument to the script. The output should be in neat tabular fields, with the first field showing the address, each of the next 8 fields a 4-byte hex number, and the final field the ASCII equivalent of the previous 8 fields.

The obvious followup to this is to extend the hex dump script into a disassembler. Using a lookup table, or some other clever gimmick, convert the hex values into 80x86 op codes.

Emulating a Shift Register

Using Example 27-15 as an inspiration, write a script that emulates a 64-bit shift register as an array. Implement functions to *load* the register, *shift left*, *shift right*, and *rotate* it. Finally, write a function that interprets the register contents as eight 8-bit ASCII characters.

Calculating Determinants

Write a script that calculates determinants [153] by recursively expanding the *minors*. Use a 4 x 4 determinant as a test case.

Hidden Words

Write a "word-find" puzzle generator, a script that hides 10 input words in a 10 x 10 array of random letters. The words may be hidden across, down, or diagonally.

Optional: Write a script that *solves* word-find puzzles. To keep this from becoming too difficult, the solution script will find only horizontal and vertical words. (Hint: Treat each row and column as a string, and search for substrings.)

Anagramming

Anagram 4-letter input. For example, the anagrams of *word* are: *do or rod row word*. You may use `/usr/share/dict/linux.words` as the reference list.

Word Ladders

A "word ladder" is a sequence of words, with each successive word in the sequence differing from the previous one by a single letter.

For example, to "ladder" from *mark* to *vase*:

```
mark --> park --> part --> past --> vast --> vase
         ^           ^       ^      ^           ^
```

Write a script that solves word ladder puzzles. Given a starting and an ending word, the script will list all intermediate steps in the "ladder." Note that *all* words in the sequence must be legitimate dictionary words.

Fog Index

The "fog index" of a passage of text estimates its reading difficulty, as a number corresponding roughly to a school grade level. For example, a passage with a fog index of 12 should be comprehensible to anyone with 12 years of schooling.

The Gunning version of the fog index uses the following algorithm.

1. Choose a section of the text at least 100 words in length.
2. Count the number of sentences (a portion of a sentence truncated by the boundary of the text section counts as one).
3. Find the average number of words per sentence.

   AVE_WDS_SEN = TOTAL_WORDS / SENTENCES
4. Count the number of "difficult" words in the segment -- those containing at least 3 syllables. Divide this quantity by total words to get the proportion of difficult words.

   PRO_DIFF_WORDS = LONG_WORDS / TOTAL_WORDS
5. The Gunning fog index is the sum of the above two quantities, multiplied by 0.4, then rounded to the nearest integer.

   G_FOG_INDEX = int (0.4 * (AVE_WDS_SEN + PRO_DIFF_WORDS))

Step 4 is by far the most difficult portion of the exercise. There exist various algorithms for estimating the syllable count of a word. A rule-of-thumb formula might consider the number of letters in a word and the vowel-consonant mix.

A strict interpretation of the Gunning fog index does not count compound words and proper nouns as "difficult" words, but this would enormously complicate the script.

Calculating PI using Buffon's Needle

The Eighteenth Century French mathematician de Buffon came up with a novel experiment. Repeatedly drop a needle of length `n` onto a wooden floor composed of long and narrow parallel boards. The cracks separating the equal-width floorboards are a fixed distance `d` apart. Keep track of the total drops and the number of times the needle intersects a crack on the floor. The ratio of these two quantities turns out to be a fractional multiple of PI.

In the spirit of Example 16-50, write a script that runs a Monte Carlo simulation of *Buffon's Needle*. To simplify matters, set the needle length equal to the distance between the cracks, `n = d`.

Hint: there are actually two critical variables: the distance from the center of the needle to the nearest crack, and the inclination angle of the needle to that crack. You may use bc to handle the calculations.

Playfair Cipher

Implement the Playfair (Wheatstone) Cipher in a script.

The Playfair Cipher encrypts text by substitution of *digrams* (2-letter groupings). It is traditional to use a 5 x 5 letter scrambled-alphabet *key square* for the encryption and decryption.

```
   C O D E S
   A B F G H
   I K L M N
   P Q R T U
   V W X Y Z

Each letter of the alphabet appears once, except "I" also represents
"J". The arbitrarily chosen key word, "CODES" comes first, then all
```

```
the rest of the alphabet, in order from left to right, skipping letters
already used.

To encrypt, separate the plaintext message into digrams (2-letter
groups). If a group has two identical letters, delete the second, and
form a new group. If there is a single letter left over at the end,
insert a "null" character, typically an "X."

THIS IS A TOP SECRET MESSAGE

TH IS IS AT OP SE CR ET ME SA GE



For each digram, there are three possibilities.
-----------------------------------------------

1) Both letters will be on the same row of the key square:
   For each letter, substitute the one immediately to the right, in that
   row. If necessary, wrap around left to the beginning of the row.

or

2) Both letters will be in the same column of the key square:
   For each letter, substitute the one immediately below it, in that
   row. If necessary, wrap around to the top of the column.

or

3) Both letters will form the corners of a rectangle within the key square:
   For each letter, substitute the one on the other corner the rectangle
   which lies on the same row.


The "TH" digram falls under case #3.
G H
M N
T U           (Rectangle with "T" and "H" at corners)

T --> U
H --> G


The "SE" digram falls under case #1.
C O D E S     (Row containing "S" and "E")

S --> C  (wraps around left to beginning of row)
E --> S

=========================================================================

To decrypt encrypted text, reverse the above procedure under cases #1
and #2 (move in opposite direction for substitution). Under case #3,
just take the remaining two corners of the rectangle.


Helen Fouche Gaines' classic work, ELEMENTARY CRYPTANALYSIS (1939), gives a
fairly detailed description of the Playfair Cipher and its solution methods.
```

This script will have three main sections

I. Generating the *key square*, based on a user-input keyword.

II. Encrypting a *plaintext* message.
III. Decrypting encrypted text.

The script will make extensive use of arrays and functions. You may use Example A-56 as an inspiration.

--

Please do not send the author your solutions to these exercises. There are more appropriate ways to impress him with your cleverness, such as submitting bugfixes and suggestions for improving the book.

Appendix P. Revision History

> This document first appeared as a 60-page HOWTO in the late spring of 2000. Since then, it has gone through quite a number of updates and revisions. This book could not have been written without the assistance of the Linux community, and especially of the volunteers of the Linux Documentation Project.

Here is the e-mail to the LDP requesting permission to submit version 0.1.

```
From thegrendel@theriver.com Sat Jun 10 09:05:33 2000 -0700
Date: Sat, 10 Jun 2000 09:05:28 -0700 (MST)
From: "M. Leo Cooper" <thegrendel@theriver.com>
X-Sender: thegrendel@localhost
To: ldp-discuss@lists.linuxdoc.org
Subject: Permission to submit HOWTO

Dear HOWTO Coordinator,

I am working on and would like to submit to the LDP a HOWTO on the subject
of "Bash Scripting" (shell scripting, using 'bash'). As it happens,
I have been writing this document, off and on, for about the last eight
months or so, and I could produce a first draft in ASCII text format in
a matter of just a few more days.

I began writing this out of frustration at being unable to find a
decent book on shell scripting. I managed to locate some pretty good
articles on various aspects of scripting, but nothing like a complete,
beginning-to-end tutorial.  Well, in keeping with my philosophy, if all
else fails, do it yourself.

As it stands, this proposed "Bash-Scripting HOWTO" would serve as a
combination tutorial and reference, with the heavier emphasis on the
tutorial. It assumes Linux experience, but only a very basic level
of programming skills. Interspersed with the text are 79 illustrative
example scripts of varying complexity, all liberally commented. There
are even exercises for the reader.

At this stage, I'm up to 18,000+ words (124k), and that's over 50 pages of
text (whew!).


I haven't mentioned that I've previously authored an LDP HOWTO, the
"Software-Building HOWTO", which I wrote in Linuxdoc/SGML. I don't know
if I could handle Docbook/SGML, and I'm glad you have volunteers to do
the conversion. You people seem to have gotten on a more organized basis
these last few months. Working with Greg Hankins and Tim Bynum was nice,
but a professional team is even nicer.

Anyhow, please advise.


Mendel Cooper
thegrendel@theriver.com
```

Table P-1. Revision History

Release	Date	Comments

0.1	14 Jun 2000	Initial release.
0.2	30 Oct 2000	Bugs fixed, plus much additional material and more example scripts.
0.3	12 Feb 2001	Major update.
0.4	08 Jul 2001	Complete revision and expansion of the book.
0.5	03 Sep 2001	Major update: Bugfixes, material added, sections reorganized.
1.0	14 Oct 2001	Stable release: Bugfixes, reorganization, material added.
1.1	06 Jan 2002	Bugfixes, material and scripts added.
1.2	31 Mar 2002	Bugfixes, material and scripts added.
1.3	02 Jun 2002	TANGERINE release: A few bugfixes, much more material and scripts added.
1.4	16 Jun 2002	MANGO release: A number of typos fixed, more material and scripts.
1.5	13 Jul 2002	PAPAYA release: A few bugfixes, much more material and scripts added.
1.6	29 Sep 2002	POMEGRANATE release: Bugfixes, more material, one more script.
1.7	05 Jan 2003	COCONUT release: A couple of bugfixes, more material, one more script.
1.8	10 May 2003	BREADFRUIT release: A number of bugfixes, more scripts and material.
1.9	21 Jun 2003	PERSIMMON release: Bugfixes, and more material.
2.0	24 Aug 2003	GOOSEBERRY release: Major update.
2.1	14 Sep 2003	HUCKLEBERRY release: Bugfixes, and more material.
2.2	31 Oct 2003	CRANBERRY release: Major update.
2.3	03 Jan 2004	STRAWBERRY release: Bugfixes and more material.
2.4	25 Jan 2004	MUSKMELON release: Bugfixes.
2.5	15 Feb 2004	STARFRUIT release: Bugfixes and more material.
2.6	15 Mar 2004	SALAL release: Minor update.
2.7	18 Apr 2004	MULBERRY release: Minor update.
2.8	11 Jul 2004	ELDERBERRY release: Minor update.
3.0	03 Oct 2004	LOGANBERRY release: Major update.
3.1	14 Nov 2004	BAYBERRY release: Bugfix update.
3.2	06 Feb 2005	BLUEBERRY release: Minor update.
3.3	20 Mar 2005	RASPBERRY release: Bugfixes, much material added.
3.4	08 May 2005	TEABERRY release: Bugfixes, stylistic revisions.
3.5	05 Jun 2005	BOXBERRY release: Bugfixes, some material added.
3.6	28 Aug 2005	POKEBERRY release: Bugfixes, some material added.
3.7	23 Oct 2005	WHORTLEBERRY release: Bugfixes, some material added.
3.8	26 Feb 2006	BLAEBERRY release: Bugfixes, some material added.
3.9	15 May 2006	SPICEBERRY release: Bugfixes, some material added.
4.0	18 Jun 2006	WINTERBERRY release: Major reorganization.
4.1	08 Oct 2006	WAXBERRY release: Minor update.
4.2	10 Dec 2006	SPARKLEBERRY release: Important update.
4.3	29 Apr 2007	INKBERRY release: Bugfixes, material added.
5.0	24 Jun 2007	SERVICEBERRY release: Major update.
5.1	10 Nov 2007	LINGONBERRY release: Minor update.
5.2	16 Mar 2008	SILVERBERRY release: Important update.
5.3	11 May 2008	GOLDENBERRY release: Minor update.

5.4	21 Jul 2008	ANGLEBERRY release: Major update.
5.5	23 Nov 2008	FARKLEBERRY release: Minor update.
5.6	26 Jan 2009	WORCESTERBERRY release: Minor update.
6.0	23 Mar 2009	THIMBLEBERRY release: Major update.
6.1	30 Sep 2009	BUFFALOBERRY release: Minor update.
6.2	17 Mar 2010	ROWANBERRY release: Minor update.
6.3	30 Apr 2011	SWOZZLEBERRY release: Major update.
6.4	30 Aug 2011	VORTEXBERRY release: Minor update.
6.5	05 Apr 2012	TUNGSTENBERRY release: Minor update.
6.6	27 Nov 2012	YTTERBIUMBERRY release: Minor update.
10	10 Mar 2014	YTTERBIUMBERRY release: License change.

Appendix Q. Download and Mirror Sites

The latest update of this document, as an archived, bzip2-ed "tarball" including both the SGML source and rendered HTML, may be downloaded from the author's home site). A pdf version is also available (mirror site). There is likewise an epub version, courtesy of Craig Barnes and Michael Satke. The change log gives a detailed revision history. The *ABS Guide* even has its own `freshmeat.net/freecode` page to keep track of major updates, user comments, and popularity ratings for the project.

The legacy hosting site for this document is the Linux Documentation Project, which maintains many other Guides and HOWTOs as well.

Many thanks to Ronny Bangsund for donating server space to host this project.

Appendix R. To Do List

- A comprehensive survey of incompatibilities between Bash and the classic Bourne shell.
- Same as above, but for the Korn shell (*ksh*).

Appendix S. Copyright

The *Advanced Bash Scripting Guide* is herewith granted to the PUBLIC DOMAIN. This has the following implications and consequences.

```
A.  All previous releases of the Advanced Bash Scripting Guide
    are as well granted to the Public Domain.

A1. All printed editions, whether authorized by the author or not,
    are as well granted to the Public Domain. This legally overrides
    any stated intention or wishes of the publishers. Any statement
    of copyright is void and invalid.
    THERE ARE NO EXCEPTIONS TO THIS.

A2. Any release of the Advanced Bash Scripting Guide, whether in
    electronic or print form is granted to the Public Domain by the
    express directive of the author and previous copyright holder, Mendel
    Cooper. No other person(s) or entities have ever held a valid copyright.

B.  As a Public Domain document, unlimited copying and distribution rights
    are granted. There can be NO restrictions. If anyone has published or will
    in the future publish an original or modified version of this document,
    then only additional original material may be copyrighted. The core
    work will remain in the Public Domain.
```

By law, distributors and publishers (including on-line publishers) are prohibited from imposing any conditions, strictures, or provisions on this document, any previous versions, or any derivative versions. The author asserts that he has *not* entered into any contractual obligations that would alter the foregoing declarations.

Essentially, you may freely distribute this book or any derivative thereof in electronic or printed form. If you have previously purchased or are in possession of a printed copy of a current or previous edition, you have the LEGAL RIGHT to copy and/or redistribute it, regardless of any copyright notice. Any copyright notice is void.

Additionally, the author wishes to state his intention that:

```
If you copy or distribute this book, kindly DO NOT
use the materials within, or any portion thereof, in a patent or copyright
lawsuit against the Open Source community, its developers, its
distributors, or against any of its associated software or documentation
including, but not limited to, the Linux kernel, Open Office, Samba,
and Wine. Kindly DO NOT use any of the materials within
this book in testimony or depositions as a plaintiff's "expert witness" in
any lawsuit against the Open Source community, any of its developers, its
distributors, or any of its associated software or documentation.
```

A Public Domain license essentially does not restrict ANY legitimate distribution or use of this book. The author especially encourages its (royalty-free!) use for classroom and instructional purposes.

To date, limited print rights (Lulu edition) have been granted to one individual and to *no one else*. Neither that individual nor Lulu holds or ever has held a valid copyright.

It has come to the attention of the author that *unauthorized* electronic and print editions of this book are being sold commercially on itunes®, *amazon.com* and elsewhere. These are illegal and pirated editions produced without the author's permission, and readers of this book are strongly urged not to purchase them. In fact, these pirated editions are now legal, but necessarily fall into the Public Domain, and any copyright notices contained within them are invalid and void.

The author produced this book in a manner consistent with the spirit of the LDP Manifesto.

Hyun Jin Cha has done a Korean translation of version 1.0.11 of this book. Spanish, Portuguese, French, German, Italian, Russian, Czech, Chinese, Indonesian, Dutch, Romanian, Bulgarian, and Turkish translations are also available or in progress. If you wish to translate this document into another language, please feel free to do so, subject to the terms stated above. The author wishes to be notified of such efforts.

Those generous readers desiring to make a donation to the author may contribute a small amount via Paypal to my e-mail address, <`thegrendel.abs@gmail.com`>. (An **`Honor Roll of Supporters`** is given at the beginning of the Change Log.) This is *not* a requirement. The *ABS Guide* is a free and freely distributed document for the use and enjoyment of the Linux community. However, in these difficult times, showing support for voluntary projects and especially to authors of limited means is more critically important than ever.

Appendix T. ASCII Table

Traditionally, a book of this sort has an ASCII Table appendix. This book does not. Instead, here are several short scripts, each of which generates a complete ASCII table.

Example T-1. A script that generates an ASCII table

```
#!/bin/bash
# ascii.sh
# ver. 0.2, reldate 26 Aug 2008
# Patched by ABS Guide author.

# Original script by Sebastian Arming.
# Used with permission (thanks!).

exec >ASCII.txt         #  Save stdout to file,
                        #+ as in the example scripts
                        #+ reassign-stdout.sh and upperconv.sh.

MAXNUM=256
COLUMNS=5
OCT=8
OCTSQU=64
LITTLESPACE=-3
BIGSPACE=-5

i=1 # Decimal counter
o=1 # Octal counter

while [ "$i" -lt "$MAXNUM" ]; do  # We don't have to count past 400 octal.
        paddi="    $i"
        echo -n "${paddi: $BIGSPACE}  "       # Column spacing.
        paddo="00$o"
#       echo -ne "\\${paddo: $LITTLESPACE}"   # Original.
        echo -ne "\\0${paddo: $LITTLESPACE}"  # Fixup.
#                   ^
        echo -n "     "
        if (( i % $COLUMNS == 0)); then       # New line.
           echo
        fi
        ((i++, o++))
        # The octal notation for 8 is 10, and 64 decimal is 100 octal.
        (( i % $OCT == 0))    && ((o+=2))
        (( i % $OCTSQU == 0)) && ((o+=20))
done

exit $?

# Compare this script with the "pr-asc.sh" example.
# This one handles "unprintable" characters.

# Exercise:
# Rewrite this script to use decimal numbers, rather than octal.
```

Example T-2. Another ASCII table script

```
#!/bin/bash
```

```
# Script author: Joseph Steinhauser
# Lightly edited by ABS Guide author, but not commented.
# Used in ABS Guide with permission.

#-------------------------------------------------------------------------
#-- File:  ascii.sh    Print ASCII chart, base 10/8/16         (JETS-2012)
#-------------------------------------------------------------------------
#-- Usage: ascii [oct|dec|hex|help|8|10|16]
#--
#-- This script prints out a summary of ASCII char codes from Zero to 127.
#-- Numeric values may be printed in Base10, Octal, or Hex.
#--
#-- Format Based on: /usr/share/lib/pub/ascii with base-10 as default.
#-- For more detail, man ascii . . .
#-------------------------------------------------------------------------

[ -n "$BASH_VERSION" ] && shopt -s extglob

case "$1" in
   oct|[Oo]?([Cc][Tt])|8)       Obase=Octal;  Numy=3o;;
   hex|[Hh]?([Ee][Xx])|16|[Xx]) Obase=Hex;    Numy=2X;;
   help|?(-)[h?])        sed -n '2,/^[ ]*$/p' $0;exit;;
   code|[Cc][Oo][Dd][Ee])sed -n '/case/,$p'   $0;exit;;
   *) Obase=Decimal
esac # CODE is actually shorter than the chart!

printf "\t\t## $Obase ASCII Chart ##\n\n"; FM1="|%0${Numy:-3d}"; LD=-1

AB="nul soh stx etx eot enq ack bel bs tab nl vt np cr so si dle"
AD="dc1 dc2 dc3 dc4 nak syn etb can em sub esc fs gs rs us sp"

for TOK in $AB $AD; do ABR[$((LD+=1))]=$TOK; done;
ABR[127]=del

IDX=0
while [ $IDX -le 127 ] && CHR="${ABR[$IDX]}"
   do (($ {#CHR}))&& FM2='%-3s'|| FM2=`printf '\\\\%o  ' $IDX`
      printf "$FM1 $FM2" "$IDX" $CHR; (( (IDX+=1)%8))||echo '|'
   done

exit $?
```

Example T-3. A third ASCII table script, using *awk*

```
#!/bin/bash
# ASCII table script, using awk.
# Author: Joseph Steinhauser
# Used in ABS Guide with permission.


#-------------------------------------------------------------------------
#-- File:  ascii     Print ASCII chart, base 10/8/16         (JETS-2010)
#-------------------------------------------------------------------------
#-- Usage: ascii [oct|dec|hex|help|8|10|16]
#--
#-- This script prints a summary of ASCII char codes from Zero to 127.
#-- Numeric values may be printed in Base10, Octal, or Hex (Base16).
#--
#-- Format Based on: /usr/share/lib/pub/ascii with base-10 as default.
#-- For more detail, man ascii
#-------------------------------------------------------------------------
```

```
[ -n "$BASH_VERSION" ] && shopt -s extglob

case "$1" in
   oct|[Oo]?([Cc][Tt])|8)       Obase=Octal;  Numy=3o;;
   hex|[Hh]?([Ee][Xx])|16|[Xx]) Obase=Hex;    Numy=2X;;
   help|?(-)[h?])        sed -n '2,/^[ ]*$/p' $0;exit;;
   code|[Cc][Oo][Dd][Ee])sed -n '/case/,$p'   $0;exit;;
   *) Obase=Decimal
esac
export Obase   # CODE is actually shorter than the chart!

awk 'BEGIN{print "\n\t\t## "ENVIRON["Obase"]" ASCII Chart ##\n"
           ab="soh,stx,etx,eot,enq,ack,bel,bs,tab,nl,vt,np,cr,so,si,dle,"
           ad="dc1,dc2,dc3,dc4,nak,syn,etb,can,em,sub,esc,fs,gs,rs,us,sp"
           split(ab ad,abr,",");abr[0]="nul";abr[127]="del";
           fm1="|%0'"${Numy:- 4d}"' %-3s"
           for(idx=0;idx<128;idx++){fmt=fm1 (++colz%8?"":"|\n")
           printf(fmt,idx,(idx in abr)?abr[idx]:sprintf("%c",idx))} }'

exit $?
```

Index

This index / glossary / quick-reference lists many of the important topics covered in the text. Terms are arranged in *approximate* ASCII sorting order, *modified as necessary* for enhanced clarity.

Note that *commands* are indexed in Part 4.

* * *

Example script

< Left angle bracket

- Is-less-than

  String comparison

  Integer comparison within double parentheses
- Redirection

  < `stdin`

  << *Here document*

  <<< *Here string*

  <> Opening a file for *both* reading and writing

> Right angle bracket

- Is-greater-than

  String comparison

  Integer comparison, within *double parentheses*
- Redirection

  > Redirect `stdout` to a file

  >> Redirect `stdout` to a file, but *append*

  i>&j Redirect *file descriptor* `i` to *file descriptor* `j`

  >&j Redirect `stdout` to *file descriptor* `j`

  >&2 Redirect `stdout` of a command to `stderr`

  2>&1 Redirect `stderr` to `stdout`

  &> Redirect *both* `stdout` and `stderr` of a command to a file

  :> `file` Truncate file to zero length

| Pipe, a device for passing the output of a command to another command or to the shell

|| Logical OR test operator

- (dash)

- Prefix to *default parameter*, in *parameter substitution*

- Prefix to *option flag*
- Indicating *redirection* from `stdin` or `stdout`
- **--** (double-dash)

  Prefix to *long* command options

  C-style variable decrement within double parentheses

; (semicolon)

- As command separator
- **\;** *Escaped* semicolon, terminates a find command
- **;;** Double-semicolon, terminator in a case option

  Required when ...

  do keyword is on the first line of *loop*

  terminating *curly-bracketed* code block
- **;;& ;&** Terminators in a *case* option (version 4+ of Bash).

: Colon

- **`:> filename`** Truncate file to zero length
- *null* command, equivalent to the true Bash builtin
- Used in an anonymous here document
- Used in an otherwise empty function
- Used as a function name

! Negation operator, inverts exit status of a test or command

- **!=** not-equal-to String comparison operator

? (question mark)

- Match zero or one characters, in an Extended Regular Expression
- Single-character *wild card*, in globbing
- In a *C*-style Trinary operator

// Double forward slash, behavior of cd command toward

. (dot / period)

- **.** Load a file (into a script), equivalent to source command
- **.** Match single character, in a Regular Expression
- **.** Current working directory

  ./ Current working directory
- **..** *Parent* directory

' ... ' (single quotes) *strong* quoting

" ... " (double quotes) *weak* quoting

- *Double-quoting* the *backslash* (\) character

,

- Comma operator
- **,**

  ,,

  Lowercase conversion in *parameter substitution*

() Parentheses

- **(...)** Command group; starts a subshell
- **(...)** Enclose group of *Extended Regular Expressions*
- **>(...)**

  <(...) Process substitution
- **...)** Terminates test-condition in *case* construct
- **((...))** Double parentheses, in arithmetic expansion

[Left bracket, *test* construct

[ ]Brackets

- *Array* element
- Enclose character set to match in a *Regular Expression*
- *Test* construct

[[...]] Double brackets, extended *test* construct

$ *Anchor*, in a Regular Expression

$ Prefix to a variable name

$(...) Command substitution, setting a variable with output of a command, using parentheses notation

` ... ` Command substitution, using backquotes notation

$[...] Integer expansion (deprecated)

${ ... } Variable manipulation / evaluation

- **${var}** Value of a variable
- **${#var}** Length of a variable
- **${#@}**

  ${#*} Number of *positional parameters*
- **${parameter?err_msg}** Parameter-unset message

- **${parameter-default}**

  ${parameter:-default}

  ${parameter=default}

  ${parameter:=default} Set default parameter
- **${parameter+alt_value}**

  ${parameter:+alt_value}

  Alternate value of parameter, if set
- **${!var}**

  Indirect referencing of a variable, new notation
- **${!#}**

  Final *positional parameter*. (This is an *indirect reference* to $#.)
- **${!varprefix*}**

  ${!varprefix@}

  Match *names* of all previously declared variables beginning with `varprefix`
- **${string:position}**

  ${string:position:length} Substring extraction
- **${var#Pattern}**

  ${var##Pattern} Substring removal
- **${var%Pattern}**

  ${var%%Pattern} Substring removal
- **${string/substring/replacement}**

  ${string//substring/replacement}

  ${string/#substring/replacement}

  ${string/%substring/replacement} Substring replacement

$' ... ' String expansion, using *escaped* characters.

**** Escape the character following

- **\< ... \>** Angle brackets, *escaped*, word boundary in a Regular Expression
- **\{ N \}** "Curly" brackets, *escaped*, number of character sets to match in an Extended RE
- **\;** *Semicolon*, *escaped*, terminates a find command
- **\$$** Indirect reverencing of a variable, old-style notation
- Escaping a *newline*, to write a multi-line command

&

- **&>** Redirect *both* `stdout` and `stderr` of a command to a file
- **>&j** Redirect `stdout` to *file descriptor j*

  >&2 Redirect `stdout` of a command to `stderr`
- **i>&j** Redirect *file descriptor* *i* to *file descriptor j*

  2>&1 Redirect `stderr` to `stdout`
- Closing *file descriptors*

  n<&- Close input file descriptor *n*

  0<&-, **<&-** Close `stdin`

  n>&- Close output file descriptor *n*

  1>&-, **>&-** Close `stdout`
- **&&** Logical AND test operator
- **Command &** Run job in *background*

Hashmark, special symbol beginning a script *comment*

#! Sha-bang, special string starting a shell script

***** Asterisk

- *Wild card*, in globbing
- Any number of characters in a Regular Expression
- ** Exponentiation, arithmetic operator
- ** Extended *globbing* file-match operator

% Percent sign

- Modulo, division-remainder arithmetic operation
- Substring removal (pattern matching) operator

+ Plus sign

- *Character match*, in an extended Regular Expression
- Prefix to *alternate parameter*, in *parameter substitution*
- **++** *C-style* variable increment, within double parentheses

* * *

Shell Variables

$_ Last argument to previous command

$- Flags passed to script, using set

$! *Process ID* of last background job

- *${parameter+alt_value}*

  ${parameter:+alt_value}

  Alternate value of parameter, if set
- *${parameter-default}*

  ${parameter:-default}

  ${parameter=default}

  ${parameter:=default}

  Default parameters
- *${!varprefix*}*

  ${!varprefix@}

  Parameter *name* match
- *${parameter?err_msg}*

  Parameter-unset message
- *${parameter}*

  Value of *parameter*
- *Case modification* (version 4+ of Bash).
- *Script example*
- **Table** of *parameter substitution*

Parent / child process problem, a *child* process cannot export variables to a parent process

Parentheses

- Command group
- Enclose group of *Extended Regular Expressions*
- Double parentheses, in arithmetic expansion

`$PATH`, the *path* (location of system binaries)

- Appending directories to `$PATH` using the += operator.

Pathname, a `filename` that incorporates the complete *path* of a given file.

- Parsing *pathnames*

Perl, programming language

- Combined in the same file with a *Bash* script
- Embedded in a *Bash* script

Perquackey-type anagramming game (*Quackey* script)

Petals Around the Rose

PID, *Process ID*, an identification number assigned to a running process.

Pipe, | , a device for passing the output of a command to another command or to the shell

- Avoiding unnecessary commands in a *pipe*
- *Comments* embedded within
- Exit status of a pipe
- Pipefail, *set -o pipefail* option to indicate exit status within a *pipe*
- `$PIPESTATUS`, *exit status* of last executed pipe
- Piping output of a command to a script
- Redirecting `stdin`, rather than using cat in a *pipe*

Pitfalls

- - (dash) is *not* redirection operator
- // (double forward slash), behavior of cd command toward
- #!/bin/sh script header disables extended *Bash* features
- Abuse of *cat*
- *CGI* programming, using scripts for
- Closing *limit string* in a *here document*, indenting
- DOS-type newlines (\r\n) crash a script
- *Double-quoting* the *backslash* (\) character
- eval, risk of using
- Execute permission lacking for commands within a script
- *Exit status*, anomalous
- *Exit status* of arithmetic expression *not* equivalent to an *error code*
- *Export* problem, *child* process to *parent* process
- Extended *Bash* features not available
- Failing to *quote* variables within *test* brackets
- *GNU* command set, in cross-platform scripts
- *let* misuse: attempting to set string variables
- Multiple echo statements in a function whose output is captured
- *null* variable assignment
- Numerical and string comparison operators *not* equivalent

  = and **-eq** *not* interchangeable
- Omitting terminal *semicolon*, in a *curly-bracketed* code block
- Piping

  echo to a loop

  echo to *read* (however, this problem can be circumvented)

  tail `-f` to *grep*
- Preserving *whitespace* within a variable, unintended consequences
- *suid* commands inside a script
- Undocumented *Bash* features, danger of
- Updates to *Bash* breaking older scripts
- Uninitialized variables

- file descriptor`i` to *file descriptor j*

  `i>&j`
- `stdout` of a command to `stderr`

  `>&2`
- `stdout` *and* `stderr` of a command to a file

  `&>`
- tee, redirect to a file output of command(s) partway through a pipe

Reference Cards

- Miscellaneous constructs
- Parameter substitution/expansion
- Special shell variables
- String operations
- Test operators

  Binary comparison

  Files

Regular Expressions

- **^** (caret) Beginning-of-line
- **$** (dollar sign) *Anchor*
- **.** (dot) Match single character
- ***** (asterisk) Any number of characters
- **[ ]** (brackets) Enclose character set to match
- **** (backslash) Escape, interpret following character literally
- **\\< ... \\>** (angle brackets, *escaped*) Word boundary
- Extended REs

  + *Character match*

  \\{ \\} Escaped "curly" brackets

  [: :] POSIX character classes

`$REPLY`, Default value associated with read command

Restricted shell, shell (or script) with certain commands disabled

return, command that terminates a function

run-parts

- Running scripts in sequence, without user intervention

* * *

- **-z** is-zero-length (string comparison)
- **=** is-equal-to (string comparison)

  == is-equal-to (string comparison)
- **<** less-than (string comparison)
- **<** less-than, (integer comparison, within double parentheses)
- **<=** less-than-or-equal, (integer comparison, within *double parentheses*)
- **>** greater-than (string comparison)
- **>** greater-than, (integer comparison, within *double parentheses*)
- **>=** greater-than-or-equal, (integer comparison, within *double parentheses*)
- **||** Logical OR
- **&&** Logical AND
- **!** Negation operator, inverts exit status of a test

  != not-equal-to (string comparison)
- **Tables** of *test* operators

  Binary comparison

  File

Text and text file processing

Time / Date

Timed input

- Using *read -t*
- Using *stty*
- Using timing loop
- Using `$TMOUT`

Tips and hints for Bash scripts

- Array, as *return value* from a function

  Associative array more efficient than a numerically-indexed array
- Capturing the return value of a function, using *echo*
- *CGI* programming, using scripts for
- Comment blocks

  Using *anonymous here documents*

  Using *if-then* constructs
- Comment headers, special purpose
- *C*-style syntax , for manipulating variables
- Double-spacing a text file
- Filenames prefixed with a dash, removing
- Filter, feeding output back to *same* filter
- Function *return* value workarounds
- *if-grep* test fixup

Notes

[1] These are referred to as builtins, features internal to the shell.

[2]

Although recursion *is* possible in a shell script, it tends to be slow and its implementation is often an ugly kludge.

[3] An *acronym* is an *ersatz* word formed by pasting together the initial letters of the words into a tongue-tripping phrase. This morally corrupt and pernicious practice deserves appropriately severe punishment. Public flogging suggests itself.

[4] Many of the features of *ksh88*, and even a few from the updated *ksh93* have been merged into Bash.

[5] By convention, user-written shell scripts that are Bourne shell compliant generally take a name with a `.sh` extension. System scripts, such as those found in `/etc/rc.d`, do not necessarily conform to this nomenclature.

[6] More commonly seen in the literature as *she-bang* or *sh-bang*. This derives from the concatenation of the tokens *sharp* (#) and *bang* (!).

[7] Some flavors of UNIX (those based on 4.2 BSD) allegedly take a four-byte magic number, requiring a blank after the ! -- **`#! /bin/sh`**. According to Sven Mascheck this is probably a myth.

[8] The #! line in a shell script will be the first thing the command interpreter (**sh** or **bash**) sees. Since this line begins with a #, it will be correctly interpreted as a comment when the command interpreter finally executes the script. The line has already served its purpose - calling the command interpreter.

If, in fact, the script includes an *extra* #! line, then **bash** will interpret it as a comment.

```
#!/bin/bash

echo "Part 1 of script."
a=1

#!/bin/bash
# This does *not* launch a new script.

echo "Part 2 of script."
echo $a  # Value of $a stays at 1.
```

[9] This allows some cute tricks.

```
#!/bin/rm
# Self-deleting script.

# Nothing much seems to happen when you run this... except that the file disappears.

WHATEVER=85

echo "This line will never print (betcha!)."

exit $WHATEVER  # Doesn't matter. The script will not exit here.
                # Try an echo $? after script termination.
                # You'll get a 0, not a 85.
```

Also, try starting a `README` file with a **`#!/bin/more`**, and making it executable. The result is a self-listing documentation file. (A here document using cat is possibly a better alternative -- see Example 19-3).

[10] **P**ortable **O**perating **S**ystem *I*nterface, an attempt to standardize UNI**X**-like OSes. The POSIX specifications are listed on the Open Group site.

[11] To avoid this possibility, a script may begin with a #!/bin/env bash *sha-bang* line. This may be useful on UNIX machines where *bash* is not located in `/bin`

[12] If *Bash* is your default shell, then the #! isn't necessary at the beginning of a script. However, if launching a script from a different shell, such as *tcsh*, then you *will* need the #!.

[13] Caution: invoking a *Bash* script by **`sh scriptname`** turns off Bash-specific extensions, and the script may therefore fail to execute.

[14] A script needs *read*, as well as execute permission for it to run, since the shell needs to be able to read it.

[15] Why not simply invoke the script with **`scriptname`**? If the directory you are in ($PWD) is where `scriptname` is located, why doesn't this work? This fails because, for security reasons, the current directory (`./`) is not by default included in a user's $PATH. It is therefore necessary to explicitly invoke the script in the current directory with a **`./scriptname`**.

[16] An *operator* is an agent that carries out an *operation*. Some examples are the common arithmetic operators, **+ - * /**. In Bash, there is some overlap between the concepts of *operator* and keyword.

[17] This is more commonly known as the *ternary* operator. Unfortunately, *ternary* is an ugly word. It doesn't roll off the tongue, and it doesn't elucidate. It obfuscates. *Trinary* is by far the more elegant usage.

[18]

American **S**tandard **C**ode for **I**nformation **I**nterchange. This is a system for encoding text characters (alphabetic, numeric, and a limited set of symbols) as 7-bit numbers that can be stored and manipulated by computers. Many of the ASCII characters are represented on a standard keyboard.

[19]

A *PID*, or *process ID*, is a number assigned to a running process. The *PID*s of running processes may be viewed with a ps command.

`Definition:` A *process* is a currently executing command (or program), sometimes referred to as a *job*.

[20] The shell does the *brace expansion*. The command itself acts upon the *result* of the expansion.

[21] Exception: a code block in braces as part of a pipe *may* run as a subshell.

```
ls | { read firstline; read secondline; }
#  Error. The code block in braces runs as a subshell,
#+ so the output of "ls" cannot be passed to variables within the block.
echo "First line is $firstline; second line is $secondline"  # Won't work.

# Thanks, S.C.
```

[22] Even as in olden times a *philtre* denoted a potion alleged to have magical transformative powers, so does a UNIX *filter* transform its target in (roughly) analogous fashion. (The coder who comes up with a "love philtre" that runs on a Linux machine will likely win accolades and honors.)

[23] Bash stores a list of commands previously issued from the command-line in a *buffer*, or memory space, for recall with the builtin *history* commands.

[24] A linefeed (*newline*) is also a whitespace character. This explains why a *blank line*, consisting only of a linefeed, is considered whitespace.

[25] Technically, the *name* of a variable is called an *lvalue*, meaning that it appears on the *left* side of an assignment statment, as in **`VARIABLE=23`**. A variable's *value* is an *rvalue*, meaning that it appears on the *right* side of an assignment statement, as in **`VAR2=$VARIABLE`**.

A variable's *name* is, in fact, a *reference*, a *pointer* to the memory location(s) where the actual data associated with that variable is kept.

[26] Note that *functions* also take positional parameters.

[27]

The process calling the script sets the `$0` parameter. By convention, this parameter is the name of the script. See the manpage (manual page) for **execv**.

From the *command-line*, however, `$0` is the name of the shell.

```
bash$ echo $0
bash

tcsh% echo $0
tcsh
```

[28] If the the script is sourced or symlinked, then this will not work. It is safer to check $BASH_Source.

[29] Unless there is a file named `first` in the current working directory. Yet another reason to *quote*. (Thank you, Harald Koenig, for pointing this out.

[30]

Encapsulating "!" within double quotes gives an error when used *from the command line*. This is interpreted as a history command. Within a script, though, this problem does not occur, since the Bash history mechanism is disabled then.

Of more concern is the *apparently* inconsistent behavior of `\` within double quotes, and especially following an **echo -e** command.

```
bash$ echo hello\!
hello!
bash$ echo "hello\!"
hello\!


bash$ echo \
>
bash$ echo "\"
>
bash$ echo \a
a
bash$ echo "\a"
\a


bash$ echo x\ty
xty
bash$ echo "x\ty"
x\ty

bash$ echo -e x\ty
xty
bash$ echo -e "x\ty"
x       y
```

Double quotes following an *echo sometimes* escape `\`. Moreover, the `-e` option to *echo* causes the "\t" to be interpreted as a *tab*.

(Thank you, Wayne Pollock, for pointing this out, and Geoff Lee and Daniel Barclay for explaining it.)

[31] "Word splitting," in this context, means dividing a character string into separate and discrete arguments.

[32] In those instances when there is no return terminating the function.

[33] A *token* is a symbol or short string with a special meaning attached to it (a meta-meaning). In Bash, certain tokens, such as **[** and . (dot-command), may expand to *keywords* and commands.

[34] Per the 1913 edition of *Webster's Dictionary*:

```
Deprecate
...

To pray against, as an evil;
to seek to avert by prayer;
to desire the removal of;
to seek deliverance from;
to express deep regret for;
to disapprove of strongly.
```

[35] Be aware that *suid* binaries may open security holes. The *suid* flag has no effect on shell scripts.

[36] On Linux systems, the sticky bit is no longer used for files, only on directories.

[37] As S.C. points out, in a compound test, even quoting the string variable might not suffice. **`[ -n "$string" -o "$a" = "$b" ]`** may cause an error with some versions of Bash if `$string` is empty. The safe way is to append an extra character to possibly empty variables, **`[ "x$string" != x -o "x$a" = "x$b" ]`** (the "x's" cancel out).

[38] In a different context, **+=** can serve as a *string concatenation* operator. This can be useful for *modifying environmental variables*.

[39] *Side effects* are, of course, unintended -- and usually undesirable -- consequences.

[40] *Precedence*, in this context, has approximately the same meaning as *priority*

[41] A *stack register* is a set of consecutive memory locations, such that the values stored (*pushed*) are retrieved (*popped*) in *reverse* order. The last value stored is the first retrieved. This is sometimes called a *`LIFO`* (*last-in-first-out*) or *pushdown* stack.

[42] The PID of the currently running script is `$$`, of course.

[43] Somewhat analogous to recursion, in this context *nesting* refers to a pattern embedded within a larger pattern. One of the definitions of *nest*, according to the 1913 edition of *Webster's Dictionary*, illustrates this beautifully: "*A collection of boxes, cases, or the like, of graduated size, each put within the one next larger.*"

[44] The words "argument" and "parameter" are often used interchangeably. In the context of this document, they have the same precise meaning: *a variable passed to a script or function.*

[45] Within a script, inside a subshell, `$$` returns the PID of the script, not the subshell.

[46] In this context, *typing* a variable means to classify it and restrict its properties. For example, a variable *declared* or *typed* as an integer is no longer available for string operations.

```
declare -i intvar

intvar=23
echo "$intvar"   # 23
intvar=stringval
echo "$intvar"   # 0
```

[47] True "randomness," insofar as it exists at all, can only be found in certain incompletely understood natural phenomena, such as radioactive decay. Computers only *simulate* randomness, and computer-generated sequences of "random" numbers are therefore referred to as *pseudorandom*.

[48] The *seed* of a computer-generated pseudorandom number series can be considered an identification label. For example, think of the pseudorandom series with a seed of *23* as *`Series #23`*.

A property of a pseurandom number series is the length of the cycle before it starts repeating itself. A good pseurandom generator will produce series with very long cycles.

[49] This applies to either command-line arguments or parameters passed to a function.

[50] Note that `$substring` and `$replacement` may refer to either *literal strings* or *variables*, depending on context. See the first usage example.

[51] If $parameter is null in a non-interactive script, it will terminate with a 127 exit status (the Bash error code for "command not found").

[52] *Iteration*: Repeated execution of a command or group of commands, usually -- but not always, *while* a given condition holds, or *until* a given condition is met.

[53] These are shell builtins, whereas other loop commands, such as while and case, are keywords.

[54] Pattern-match lines may also *start* with a **(** left paren to give the layout a more structured appearance.

```
case $( arch ) in   # $( arch ) returns machine architecture.
  ( i386 ) echo "80386-based machine";;
# ^      ^
  ( i486 ) echo "80486-based machine";;
  ( i586 ) echo "Pentium-based machine";;
  ( i686 ) echo "Pentium2+-based machine";;
  (    * ) echo "Other type of machine";;
esac
```

[55] For purposes of *command substitution*, a **command** may be an external system command, an internal scripting builtin, or even a script function.

[56] In a more technically correct sense, *command substitution* extracts the `stdout` of a command, then assigns it to a variable using the = operator.

[57] In fact, nesting with backticks is also possible, but only by escaping the inner backticks, as John Default points out.

```
word_count=` wc -w \`echo * | awk '{print $8}'\` `
```

[58] As Nathan Coulter points out, "while forking a process is a low-cost operation, executing a new program in the newly-forked child process adds more overhead."

[59] An exception to this is the time command, listed in the official Bash documentation as a keyword ("reserved word").

[60] Note that *let* cannot be used for setting *string* variables.

[61] To *Export* information is to make it available in a more general context. See also scope.

[62] An *option* is an argument that acts as a flag, switching script behaviors on or off. The argument associated with a particular option indicates the behavior that the option (flag) switches on or off.

[63] Technically, an **exit** only terminates the process (or shell) in which it is running, *not* the *parent process*.

[64] Unless the **exec** is used to reassign file descriptors.

[65]

Hashing is a method of creating lookup keys for data stored in a table. The *data items themselves* are "scrambled" to create keys, using one of a number of simple mathematical *algorithms* (methods, or recipes).

An advantage of *hashing* is that it is fast. A disadvantage is that *collisions* -- where a single key maps to more than one data item -- are possible.

For examples of hashing see Example A-20 and Example A-21.

[66] The *readline* library is what Bash uses for reading input in an interactive shell.

[67] This only applies to *child processes*, of course.

[68] The C source for a number of loadable builtins is typically found in the `/usr/share/doc/bash-?.??/functions` directory.

Note that the `-f` option to **enable** is not portable to all systems.

[69] The same effect as **autoload** can be achieved with typeset -fu.

[70] The `-v` option also orders the sort by *upper- and lowercase prefixed* filenames.

[71]

Dotfiles are files whose names begin with a *dot*, such as `~/.Xdefaults`. Such filenames do not appear in a normal **ls** listing (although an **ls -a** will show them), and they cannot be deleted by an accidental **rm -rf ***. Dotfiles are generally used as setup and configuration files in a user's home directory.

[72] This particular feature may not yet be implemented in the version of the ext2/ext3 filesystem installed on your system. Check the documentation for your Linux distro.

[73] And even when *xargs* is not strictly necessary, it can speed up execution of a command involving batch-processing of multiple files.

[74] This is only true of the GNU version of **tr**, not the generic version often found on commercial UNIX systems.

[75] An *archive*, in the sense discussed here, is simply a set of related files stored in a single location.

[76] A *`tar czvf ArchiveName.tar.gz *`* *will* include dotfiles in subdirectories *below* the current working directory. This is an undocumented GNU **tar** "feature."

[77] The checksum may be expressed as a *hexadecimal* number, or to some other base.

[78] For even *better* security, use the *sha256sum*, *sha512*, and *sha1pass* commands.

[79] This is a symmetric block cipher, used to encrypt files on a single system or local network, as opposed to the *public key* cipher class, of which *pgp* is a well-known example.

[80] Creates a temporary *directory* when invoked with the `-d` option.

[81]

A *daemon* is a background process not attached to a terminal session. Daemons perform designated services either at specified times or explicitly triggered by certain events.

The word "daemon" means ghost in Greek, and there is certainly something mysterious, almost supernatural, about the way UNIX daemons wander about behind the scenes, silently carrying out their appointed tasks.

[82] This is actually a script adapted from the Debian Linux distribution.

[83] The *print queue* is the group of jobs "waiting in line" to be printed.

[84] Large mechanical *line printers* printed a single line of type at a time onto joined sheets of *greenbar* paper, to the accompaniment of a great deal of noise. The hardcopy thusly printed was referred to as a *printout*.

[85] For an excellent overview of this topic, see Andy Vaught's article, Introduction to Named Pipes, in the September, 1997 issue of *Linux Journal*.

[86] EBCDIC (pronounced "ebb-sid-ick") is an acronym for Extended Binary Coded Decimal Interchange Code, an obsolete IBM data format. A bizarre application of the `conv=ebcdic` option of **dd** is as a quick 'n easy, but not very secure text file encoder.

```
cat $file | dd conv=swab,ebcdic > $file_encrypted
# Encode (looks like gibberish).
# Might as well switch bytes (swab), too, for a little extra obscurity.
```

```
cat $file_encrypted | dd conv=swab,ascii > $file_plaintext
# Decode.
```

[87] A *macro* is a symbolic constant that expands into a command string or a set of operations on parameters. Simply put, it's a shortcut or abbreviation.

[88] This is the case on a Linux machine or a UNIX system with disk quotas.

[89] The **userdel** command will fail if the particular user being deleted is still logged on.

[90] For more detail on burning CDRs, see Alex Withers' article, Creating CDs, in the October, 1999 issue of *Linux Journal*.

[91] The `-c` option to mke2fs also invokes a check for bad blocks.

[92] Since only *root* has write permission in the `/var/lock` directory, a user script cannot set a lock file there.

[93] Operators of single-user Linux systems generally prefer something simpler for backups, such as **tar**.

[94] As of the version 4 update of Bash, the `-f` and `-c` options take a block size of 512 when in POSIX mode. Additionally, there are two new options: `-b` for socket buffer size, and `-T` for the limit on the number of *threads*.

[95] NAND is the logical *not-and* operator. Its effect is somewhat similar to subtraction.

[96] In Bash and other Bourne shell derivatives, it is possible to set variables in a single command's environment.

```
var1=value1 var2=value2 commandXXX
# $var1 and $var2 set in the environment of 'commandXXX' only.
```

[97] The *killall* system script should not be confused with the killall command in `/usr/bin`.

[98] A *meta-meaning* is the meaning of a term or expression on a higher level of abstraction. For example, the *literal* meaning of *regular expression* is an ordinary expression that conforms to accepted usage. The *meta-meaning* is drastically different, as discussed at length in this chapter.

[99] Since sed, awk, and grep process single lines, there will usually not be a newline to match. In those cases where there is a newline in a multiple line expression, the dot will match the newline.

```
#!/bin/bash

sed -e 'N;s/.*/[&]/' << EOF   # Here Document
line1
line2
EOF
# OUTPUT:
# [line1
# line2]



echo

awk '{ $0=$1 "\n" $2; if (/line.1/) {print}}' << EOF
line 1
line 2
EOF
# OUTPUT:
# line
# 1
```

```
# Thanks, S.C.

exit 0
```

[100] *Filename expansion* means expanding filename patterns or templates containing special characters. For example, `example.???` might expand to `example.001` and/or `example.txt`.

[101] A *wild card* character, analogous to a wild card in poker, can represent (almost) any other character.

[102] Filename expansion *can* match dotfiles, but only if the pattern explicitly includes the dot as a literal character.

```
~/[.]bashrc    #  Will not expand to ~/.bashrc
~/?bashrc      #  Neither will this.
               #  Wild cards and metacharacters will NOT
               #+ expand to a dot in globbing.

~/.[b]ashrc    #  Will expand to ~/.bashrc
~/.ba?hrc      #  Likewise.
~/.bashr*      #  Likewise.

# Setting the "dotglob" option turns this off.

# Thanks, S.C.
```

[103] Except, as Dennis Benzinger points out, if using **<<-** to suppress tabs.

[104] By convention in UNIX and Linux, data streams and peripherals (device files) are treated as files, in a fashion analogous to ordinary files.

[105] A *file descriptor* is simply a number that the operating system assigns to an open file to keep track of it. Consider it a simplified type of file pointer. It is analogous to a *file handle* in **C**.

[106] Using *`file descriptor 5`* might cause problems. When Bash creates a child process, as with exec, the child inherits fd 5 (see Chet Ramey's archived e-mail, SUBJECT: RE: File descriptor 5 is held open). Best leave this particular fd alone.

[107] An external command invoked with an exec does *not* (usually) fork off a subprocess / subshell.

[108] This has the same effect as a named pipe (temp file), and, in fact, named pipes were at one time used in process substitution.

[109] The **return** command is a Bash builtin.

[110] However, as Thomas Braunberger points out, a local variable declared in a function *is also visible to functions called by the parent function.*

```
#!/bin/bash

function1 ()
{
  local func1var=20

  echo "Within function1, \$func1var = $func1var."

  function2
}

function2 ()
{
  echo "Within function2, \$func1var = $func1var."
}

function1
```

```
exit 0


# Output of the script:

# Within function1, $func1var = 20.
# Within function2, $func1var = 20.
```

This is documented in the Bash manual:

"Local can only be used within a function; it makes the variable name have a visible scope restricted to that function *and its children*." [emphasis added] *The ABS Guide author considers this behavior to be a bug.*

[111] Otherwise known as *redundancy*.

[112] Otherwise known as *tautology*.

[113] Otherwise known as a *metaphor*.

[114] Otherwise known as a *recursive function*.

[115] Too many levels of recursion may crash a script with a segfault.

```
#!/bin/bash

#  Warning: Running this script could possibly lock up your system!
#  If you're lucky, it will segfault before using up all available memory.

recursive_function ()
{
echo "$1"     # Makes the function do something, and hastens the segfault.
(( $1 < $2 )) && recursive_function $(( $1 + 1 )) $2;
#  As long as 1st parameter is less than 2nd,
#+ increment 1st and recurse.
}

recursive_function 1 50000  # Recurse 50,000 levels!
#  Most likely segfaults (depending on stack size, set by ulimit -m).

#  Recursion this deep might cause even a C program to segfault,
#+ by using up all the memory allotted to the stack.


echo "This will probably not print."
exit 0  # This script will not exit normally.

#  Thanks, Stéphane Chazelas.
```

[116] ... as the first word of a command string. Obviously, an alias is only meaningful at the *beginning* of a command.

[117] However, aliases do seem to expand positional parameters.

[118] The entries in `/dev` provide mount points for physical and virtual devices. These entries use very little drive space.

Some devices, such as `/dev/null`, `/dev/zero`, and `/dev/urandom` are virtual. They are not actual physical devices and exist only in software.

[119] A *block device* reads and/or writes data in chunks, or *blocks*, in contrast to a *character device*, which acesses data in *character* units. Examples of block devices are hard drives, CDROM drives, and flash drives. Examples of character devices are keyboards, modems, sound cards.

[120]

Of course, the mount point `/mnt/flashdrive` must exist. If not, then, as *root*, **mkdir /mnt/flashdrive**.

To actually mount the drive, use the following command: **mount /mnt/flashdrive**

Newer Linux distros automount flash drives in the `/media` directory without user intervention.

[121] Certain system commands, such as procinfo, free, vmstat, lsdev, and uptime do this as well.

[122] By convention, *`signal 0`* is assigned to exit.

[123] Setting the suid permission on the script itself has no effect in Linux and most other UNIX flavors.

[124] In this context, "magic numbers" have an entirely different meaning than the magic numbers used to designate file types.

[125] Quite a number of Linux utilities are, in fact, shell wrappers. Some examples are `/usr/bin/pdf2ps`, `/usr/bin/batch`, and `/usr/bin/xmkmf`.

[126] ANSI is, of course, the acronym for the American National Standards Institute. This august body establishes and maintains various technical and industrial standards.

[127] This usually means liberal use of functions.

[128] See Marius van Oers' article, Unix Shell Scripting Malware, and also the *Denning* reference in the *bibliography*.

[129] Or, better yet, #!/bin/env sh.

[130] To be more specific, Bash 4+ has *limited* support for associative arrays. It's a bare-bones implementation, and it lacks the much of the functionality of such arrays in other programming languages. Note, however, that associative arrays in Bash seem to execute faster and more efficiently than numerically-indexed arrays.

[131] Copyright 1995-2009 by Chester Ramey.

[132] This only works with pipes and certain other *special* files.

[133] But only in conjunction with readline, i.e., from the command-line.

[134] And while you're at it, consider fixing the notorious piped read problem.

[135] This is the notorious *flog it to death* technique that works so well with slow learners, eccentrics, odd ducks, fools and geniuses.

[136] In fact, he has no credentials or special qualifications. He's a school dropout with no formal credentials or professional experience whatsoever. None. Zero. Nada. Aside from the *ABS Guide*, his major claim to fame is a First Place in the sack race at the Colfax Elementary School Field Day in June, 1958.

[137] Those who can, do. Those who can't . . . get an MCSE.

[138] Sometimes it seems as if he has spent his entire life flouting conventional wisdom and defying the sonorous Voice of Authority: "*Hey, you can't do that!*"

[139] Well, if you *absolutely* insist, you can try modifying Example A-44 to suit your purposes.

[140] It was hard to resist the obvious pun. No slight intended, since the book is a pretty decent introduction to the basic concepts of shell scripting.

[141] *Sed* executes without user intervention.

[142] If no address range is specified, the default is *all* lines.

[143] Its name derives from the initials of its authors, **A**ho, **W**einberg, and **K**ernighan.

[144] Out of range exit values can result in unexpected exit codes. An exit value greater than 255 returns an exit code modulo 256. For example, *exit 3809* gives an exit code of 225 (3809 % 256 = 225).

[145] An update of `/usr/include/sysexits.h` allocates previously unused exit codes from 64 - 78. It may be anticipated that the range of unallotted exit codes will be further restricted in the future. The

author of this document will *not* do fixups on the scripting examples to conform to the changing standard. This should not cause any problems, since there is no overlap or conflict in usage of exit codes between compiled C/C++ binaries and shell scripts.

[146] This does not apply to **csh**, **tcsh**, and other shells not related to or descended from the classic Bourne shell (**sh**).

[147] In older versions of UNIX, passwords *were* stored in `/etc/passwd`, and that explains the name of the file.

[148] Some early UNIX systems had a fast, small-capacity fixed disk (containing `/`, the root partition), and a second drive which was larger, but slower (containing `/usr` and other partitions). The most frequently used programs and utilities therefore resided on the small-but-fast drive, in `/bin`, and the others on the slower drive, in `/usr/bin`.

This likewise accounts for the split between `/sbin` and `/usr/sbin`, `/lib` and `/usr/lib`, etc.

[149] This works only from the *command line*, of course, and not within a script.

[150] Normally the default parameter completion files reside in either the `/etc/profile.d` directory or in `/etc/bash_completion`. These autoload on system startup. So, after writing a useful completion script, you might wish to move it (as *root*, of course) to one of these directories.

[151] It has been extensively documented that programmers are willing to put in long hours of effort in order to save ten minutes of "unnecessary" labor. This is known as *optimization*.

[152] Various readers have suggested modifications of the above batch file to prettify it and make it more compact and efficient. In the opinion of the *ABS Guide* author, this is wasted effort. A Bash script can access a DOS filesystem, or even an NTFS partition (with the help of ntfs-3g) to do batch or scripted operations.

[153] For all you clever types who failed intermediate algebra, a *determinant* is a numerical value associated with a multidimensional *matrix* (array of numbers).

```
For the simple case of a 2 x 2 determinant:

  |a  b|
  |b  a|

The solution is a*a - b*b, where "a" and "b" represent numbers.
```

Bash Reference Manual

Reference Documentation for Bash
Edition 5.0, for `Bash` Version 5.0.
May 2019

Chet Ramey, Case Western Reserve University
Brian Fox, Free Software Foundation

This text is a brief description of the features that are present in the Bash shell (version 5.0, 12 May 2019).

This is Edition 5.0, last updated 12 May 2019, of *The GNU Bash Reference Manual*, for `Bash`, Version 5.0.

Table of Contents

1 Introduction

1.1 What is Bash?

Bash is the shell, or command language interpreter, for the GNU operating system. The name is an acronym for the '`Bourne-Again SHell`', a pun on Stephen Bourne, the author of the direct ancestor of the current Unix shell `sh`, which appeared in the Seventh Edition Bell Labs Research version of Unix.

Bash is largely compatible with `sh` and incorporates useful features from the Korn shell `ksh` and the C shell `csh`. It is intended to be a conformant implementation of the IEEE POSIX Shell and Tools portion of the IEEE POSIX specification (IEEE Standard 1003.1). It offers functional improvements over `sh` for both interactive and programming use.

While the GNU operating system provides other shells, including a version of `csh`, Bash is the default shell. Like other GNU software, Bash is quite portable. It currently runs on nearly every version of Unix and a few other operating systems − independently-supported ports exist for MS-DOS, OS/2, and Windows platforms.

1.2 What is a shell?

At its base, a shell is simply a macro processor that executes commands. The term macro processor means functionality where text and symbols are expanded to create larger expressions.

A Unix shell is both a command interpreter and a programming language. As a command interpreter, the shell provides the user interface to the rich set of GNU utilities. The programming language features allow these utilities to be combined. Files containing commands can be created, and become commands themselves. These new commands have the same status as system commands in directories such as `/bin`, allowing users or groups to establish custom environments to automate their common tasks.

Shells may be used interactively or non-interactively. In interactive mode, they accept input typed from the keyboard. When executing non-interactively, shells execute commands read from a file.

A shell allows execution of GNU commands, both synchronously and asynchronously. The shell waits for synchronous commands to complete before accepting more input; asynchronous commands continue to execute in parallel with the shell while it reads and executes additional commands. The *redirection* constructs permit fine-grained control of the input and output of those commands. Moreover, the shell allows control over the contents of commands' environments.

Shells also provide a small set of built-in commands (*builtins*) implementing functionality impossible or inconvenient to obtain via separate utilities. For example, `cd`, `break`, `continue`, and `exec` cannot be implemented outside of the shell because they directly manipulate the shell itself. The `history`, `getopts`, `kill`, or `pwd` builtins, among others, could be implemented in separate utilities, but they are more convenient to use as builtin commands. All of the shell builtins are described in subsequent sections.

While executing commands is essential, most of the power (and complexity) of shells is due to their embedded programming languages. Like any high-level language, the shell provides variables, flow control constructs, quoting, and functions.

Shells offer features geared specifically for interactive use rather than to augment the programming language. These interactive features include job control, command line editing, command history and aliases. Each of these features is described in this manual.

2 Definitions

These definitions are used throughout the remainder of this manual.

`POSIX` A family of open system standards based on Unix. Bash is primarily concerned with the Shell and Utilities portion of the POSIX 1003.1 standard.

`blank` A space or tab character.

`builtin` A command that is implemented internally by the shell itself, rather than by an executable program somewhere in the file system.

`control operator`
: A `token` that performs a control function. It is a `newline` or one of the following: '`||`', '`&&`', '`&`', '`;`', '`;;`', '`;&`', '`;;&`', '`|`', '`|&`', '`(`', or '`)`'.

`exit status`
: The value returned by a command to its caller. The value is restricted to eight bits, so the maximum value is 255.

`field` A unit of text that is the result of one of the shell expansions. After expansion, when executing a command, the resulting fields are used as the command name and arguments.

`filename` A string of characters used to identify a file.

`job` A set of processes comprising a pipeline, and any processes descended from it, that are all in the same process group.

`job control`
: A mechanism by which users can selectively stop (suspend) and restart (resume) execution of processes.

`metacharacter`
: A character that, when unquoted, separates words. A metacharacter is a `space`, `tab`, `newline`, or one of the following characters: '`|`', '`&`', '`;`', '`(`', '`)`', '`<`', or '`>`'.

`name` A `word` consisting solely of letters, numbers, and underscores, and beginning with a letter or underscore. `Name`s are used as shell variable and function names. Also referred to as an `identifier`.

`operator` A `control operator` or a `redirection operator`. See Section 3.6 [Redirections], page 34, for a list of redirection operators. Operators contain at least one unquoted `metacharacter`.

`process group`
: A collection of related processes each having the same process group ID.

`process group ID`
: A unique identifier that represents a `process group` during its lifetime.

`reserved word`
: A `word` that has a special meaning to the shell. Most reserved words introduce shell flow control constructs, such as `for` and `while`.

`return status`
: A synonym for `exit status`.

`signal`
: A mechanism by which a process may be notified by the kernel of an event occurring in the system.

`special builtin`
: A shell builtin command that has been classified as special by the POSIX standard.

`token`
: A sequence of characters considered a single unit by the shell. It is either a `word` or an `operator`.

`word`
: A sequence of characters treated as a unit by the shell. Words may not include unquoted `metacharacters`.

3 Basic Shell Features

Bash is an acronym for '`Bourne-Again SHell`'. The Bourne shell is the traditional Unix shell originally written by Stephen Bourne. All of the Bourne shell builtin commands are available in Bash, The rules for evaluation and quoting are taken from the POSIX specification for the 'standard' Unix shell.

This chapter briefly summarizes the shell's 'building blocks': commands, control structures, shell functions, shell *parameters*, shell expansions, *redirections*, which are a way to direct input and output from and to named files, and how the shell executes commands.

3.1 Shell Syntax

When the shell reads input, it proceeds through a sequence of operations. If the input indicates the beginning of a comment, the shell ignores the comment symbol ('`#`'), and the rest of that line.

Otherwise, roughly speaking, the shell reads its input and divides the input into words and operators, employing the quoting rules to select which meanings to assign various words and characters.

The shell then parses these tokens into commands and other constructs, removes the special meaning of certain words or characters, expands others, redirects input and output as needed, executes the specified command, waits for the command's exit status, and makes that exit status available for further inspection or processing.

3.1.1 Shell Operation

The following is a brief description of the shell's operation when it reads and executes a command. Basically, the shell does the following:

1. Reads its input from a file (see Section 3.8 [Shell Scripts], page 42), from a string supplied as an argument to the `-c` invocation option (see Section 6.1 [Invoking Bash], page 85), or from the user's terminal.
2. Breaks the input into words and operators, obeying the quoting rules described in Section 3.1.2 [Quoting], page 6. These tokens are separated by `metacharacters`. Alias expansion is performed by this step (see Section 6.6 [Aliases], page 93).
3. Parses the tokens into simple and compound commands (see Section 3.2 [Shell Commands], page 8).
4. Performs the various shell expansions (see Section 3.5 [Shell Expansions], page 22), breaking the expanded tokens into lists of filenames (see Section 3.5.8 [Filename Expansion], page 32) and commands and arguments.
5. Performs any necessary redirections (see Section 3.6 [Redirections], page 34) and removes the redirection operators and their operands from the argument list.
6. Executes the command (see Section 3.7 [Executing Commands], page 38).
7. Optionally waits for the command to complete and collects its exit status (see Section 3.7.5 [Exit Status], page 41).

3.1.2 Quoting

Quoting is used to remove the special meaning of certain characters or words to the shell. Quoting can be used to disable special treatment for special characters, to prevent reserved words from being recognized as such, and to prevent parameter expansion.

Each of the shell metacharacters (see Chapter 2 [Definitions], page 3) has special meaning to the shell and must be quoted if it is to represent itself. When the command history expansion facilities are being used (see Section 9.3 [History Interaction], page 145), the *history expansion* character, usually '`!`', must be quoted to prevent history expansion. See Section 9.1 [Bash History Facilities], page 143, for more details concerning history expansion.

There are three quoting mechanisms: the *escape character*, single quotes, and double quotes.

3.1.2.1 Escape Character

A non-quoted backslash '\' is the Bash escape character. It preserves the literal value of the next character that follows, with the exception of `newline`. If a `\newline` pair appears, and the backslash itself is not quoted, the `\newline` is treated as a line continuation (that is, it is removed from the input stream and effectively ignored).

3.1.2.2 Single Quotes

Enclosing characters in single quotes ('`'`') preserves the literal value of each character within the quotes. A single quote may not occur between single quotes, even when preceded by a backslash.

3.1.2.3 Double Quotes

Enclosing characters in double quotes ('`"`') preserves the literal value of all characters within the quotes, with the exception of '`$`', '`` ` ``', '\', and, when history expansion is enabled, '`!`'. When the shell is in POSIX mode (see Section 6.11 [Bash POSIX Mode], page 99), the '`!`' has no special meaning within double quotes, even when history expansion is enabled. The characters '`$`' and '`` ` ``' retain their special meaning within double quotes (see Section 3.5 [Shell Expansions], page 22). The backslash retains its special meaning only when followed by one of the following characters: '`$`', '`` ` ``', '`"`', '\', or `newline`. Within double quotes, backslashes that are followed by one of these characters are removed. Backslashes preceding characters without a special meaning are left unmodified. A double quote may be quoted within double quotes by preceding it with a backslash. If enabled, history expansion will be performed unless an '`!`' appearing in double quotes is escaped using a backslash. The backslash preceding the '`!`' is not removed.

The special parameters '`*`' and '`@`' have special meaning when in double quotes (see Section 3.5.3 [Shell Parameter Expansion], page 24).

3.1.2.4 ANSI-C Quoting

Words of the form `$'string'` are treated specially. The word expands to *string*, with backslash-escaped characters replaced as specified by the ANSI C standard. Backslash escape sequences, if present, are decoded as follows:

`\a` alert (bell)

`\b` backspace

`\e`
`\E` an escape character (not ANSI C)

`\f` form feed

`\n` newline

`\r` carriage return

`\t` horizontal tab

`\v` vertical tab

`\\` backslash

`\'` single quote

`\"` double quote

`\?` question mark

`\nnn` the eight-bit character whose value is the octal value *nnn* (one to three octal digits)

`\xHH` the eight-bit character whose value is the hexadecimal value *HH* (one or two hex digits)

`\uHHHH` the Unicode (ISO/IEC 10646) character whose value is the hexadecimal value *HHHH* (one to four hex digits)

`\UHHHHHHHH`
the Unicode (ISO/IEC 10646) character whose value is the hexadecimal value *HHHHHHHH* (one to eight hex digits)

`\cx` a control-x character

The expanded result is single-quoted, as if the dollar sign had not been present.

3.1.2.5 Locale-Specific Translation

A double-quoted string preceded by a dollar sign (‘`$`’) will cause the string to be translated according to the current locale. If the current locale is `C` or `POSIX`, the dollar sign is ignored. If the string is translated and replaced, the replacement is double-quoted.

Some systems use the message catalog selected by the `LC_MESSAGES` shell variable. Others create the name of the message catalog from the value of the `TEXTDOMAIN` shell variable, possibly adding a suffix of ‘`.mo`’. If you use the `TEXTDOMAIN` variable, you may need to set the `TEXTDOMAINDIR` variable to the location of the message catalog files. Still others use both variables in this fashion: `TEXTDOMAINDIR`/`LC_MESSAGES`/LC_MESSAGES/`TEXTDOMAIN`.mo.

3.1.3 Comments

In a non-interactive shell, or an interactive shell in which the `interactive_comments` option to the `shopt` builtin is enabled (see Section 4.3.2 [The Shopt Builtin], page 65), a word beginning with ‘`#`’ causes that word and all remaining characters on that line to be ignored. An interactive shell without the `interactive_comments` option enabled does not allow comments. The `interactive_comments` option is on by default in interactive shells. See Section 6.3 [Interactive Shells], page 88, for a description of what makes a shell interactive.

3.2 Shell Commands

A simple shell command such as `echo a b c` consists of the command itself followed by arguments, separated by spaces.

More complex shell commands are composed of simple commands arranged together in a variety of ways: in a pipeline in which the output of one command becomes the input of a second, in a loop or conditional construct, or in some other grouping.

3.2.1 Simple Commands

A simple command is the kind of command encountered most often. It's just a sequence of words separated by `blank`s, terminated by one of the shell's control operators (see Chapter 2 [Definitions], page 3). The first word generally specifies a command to be executed, with the rest of the words being that command's arguments.

The return status (see Section 3.7.5 [Exit Status], page 41) of a simple command is its exit status as provided by the POSIX 1003.1 `waitpid` function, or 128+*n* if the command was terminated by signal *n*.

3.2.2 Pipelines

A `pipeline` is a sequence of one or more commands separated by one of the control operators '`|`' or '`|&`'.

The format for a pipeline is

```
[time [-p]] [!] command1 [ | or |& command2 ] ...
```

The output of each command in the pipeline is connected via a pipe to the input of the next command. That is, each command reads the previous command's output. This connection is performed before any redirections specified by the command.

If '`|&`' is used, *command1*'s standard error, in addition to its standard output, is connected to *command2*'s standard input through the pipe; it is shorthand for `2>&1 |`. This implicit redirection of the standard error to the standard output is performed after any redirections specified by the command.

The reserved word `time` causes timing statistics to be printed for the pipeline once it finishes. The statistics currently consist of elapsed (wall-clock) time and user and system time consumed by the command's execution. The `-p` option changes the output format to that specified by POSIX. When the shell is in POSIX mode (see Section 6.11 [Bash POSIX Mode], page 99), it does not recognize `time` as a reserved word if the next token begins with a '`-`'. The `TIMEFORMAT` variable may be set to a format string that specifies how the timing information should be displayed. See Section 5.2 [Bash Variables], page 73, for a description of the available formats. The use of `time` as a reserved word permits the timing of shell builtins, shell functions, and pipelines. An external `time` command cannot time these easily.

When the shell is in POSIX mode (see Section 6.11 [Bash POSIX Mode], page 99), `time` may be followed by a newline. In this case, the shell displays the total user and system time consumed by the shell and its children. The `TIMEFORMAT` variable may be used to specify the format of the time information.

If the pipeline is not executed asynchronously (see Section 3.2.3 [Lists], page 9), the shell waits for all commands in the pipeline to complete.

Each command in a pipeline is executed in its own subshell, which is a separate process (see Section 3.7.3 [Command Execution Environment], page 39). If the `lastpipe` option is enabled using the `shopt` builtin (see Section 4.3.2 [The Shopt Builtin], page 65), the last element of a pipeline may be run by the shell process.

The exit status of a pipeline is the exit status of the last command in the pipeline, unless the `pipefail` option is enabled (see Section 4.3.1 [The Set Builtin], page 61). If `pipefail` is enabled, the pipeline's return status is the value of the last (rightmost) command to exit with a non-zero status, or zero if all commands exit successfully. If the reserved word '`!`' precedes the pipeline, the exit status is the logical negation of the exit status as described above. The shell waits for all commands in the pipeline to terminate before returning a value.

3.2.3 Lists of Commands

A `list` is a sequence of one or more pipelines separated by one of the operators '`;`', '`&`', '`&&`', or '`||`', and optionally terminated by one of '`;`', '`&`', or a `newline`.

Of these list operators, '`&&`' and '`||`' have equal precedence, followed by '`;`' and '`&`', which have equal precedence.

A sequence of one or more newlines may appear in a `list` to delimit commands, equivalent to a semicolon.

If a command is terminated by the control operator '`&`', the shell executes the command asynchronously in a subshell. This is known as executing the command in the *background*, and these are referred to as *asynchronous* commands. The shell does not wait for the command to finish, and the return status is 0 (true). When job control is not active (see Chapter 7 [Job Control], page 104), the standard input for asynchronous commands, in the absence of any explicit redirections, is redirected from `/dev/null`.

Commands separated by a '`;`' are executed sequentially; the shell waits for each command to terminate in turn. The return status is the exit status of the last command executed.

AND and OR lists are sequences of one or more pipelines separated by the control operators '`&&`' and '`||`', respectively. AND and OR lists are executed with left associativity.

An AND list has the form

```
command1 && command2
```

command2 is executed if, and only if, *command1* returns an exit status of zero (success).

An OR list has the form

```
command1 || command2
```

command2 is executed if, and only if, *command1* returns a non-zero exit status.

The return status of AND and OR lists is the exit status of the last command executed in the list.

3.2.4 Compound Commands

Compound commands are the shell programming language constructs. Each construct begins with a reserved word or control operator and is terminated by a corresponding reserved word or operator. Any redirections (see Section 3.6 [Redirections], page 34) associated with a compound command apply to all commands within that compound command unless explicitly overridden.

In most cases a list of commands in a compound command's description may be separated from the rest of the command by one or more newlines, and may be followed by a newline in place of a semicolon.

Bash provides looping constructs, conditional commands, and mechanisms to group commands and execute them as a unit.

3.2.4.1 Looping Constructs

Bash supports the following looping constructs.

Note that wherever a '`;`' appears in the description of a command's syntax, it may be replaced with one or more newlines.

`until` The syntax of the `until` command is:

```
until test-commands; do consequent-commands; done
```

Execute *consequent-commands* as long as *test-commands* has an exit status which is not zero. The return status is the exit status of the last command executed in *consequent-commands*, or zero if none was executed.

`while` The syntax of the `while` command is:

```
while test-commands; do consequent-commands; done
```

Execute *consequent-commands* as long as *test-commands* has an exit status of zero. The return status is the exit status of the last command executed in *consequent-commands*, or zero if none was executed.

`for` The syntax of the `for` command is:

```
for name [ [in [words ...] ] ; ] do commands; done
```

Expand *words* (see Section 3.5 [Shell Expansions], page 22), and execute *commands* once for each member in the resultant list, with *name* bound to the current member. If '`in words`' is not present, the `for` command executes the *commands* once for each positional parameter that is set, as if '`in "$@"`' had been specified (see Section 3.4.2 [Special Parameters], page 21).

The return status is the exit status of the last command that executes. If there are no items in the expansion of *words*, no commands are executed, and the return status is zero.

An alternate form of the `for` command is also supported:

```
for (( expr1 ; expr2 ; expr3 )) ; do commands ; done
```

First, the arithmetic expression *expr1* is evaluated according to the rules described below (see Section 6.5 [Shell Arithmetic], page 92). The arithmetic expression *expr2* is then evaluated repeatedly until it evaluates to zero. Each time *expr2* evaluates to a non-zero value, *commands* are executed and the arithmetic expression *expr3* is evaluated. If any expression is omitted, it behaves as if it evaluates to 1. The return value is the exit status of the last command in *commands* that is executed, or false if any of the expressions is invalid.

The `break` and `continue` builtins (see Section 4.1 [Bourne Shell Builtins], page 43) may be used to control loop execution.

3.2.4.2 Conditional Constructs

`if` The syntax of the `if` command is:

```
if test-commands; then
  consequent-commands;
[elif more-test-commands; then
  more-consequents;]
[else alternate-consequents;]
fi
```

The *test-commands* list is executed, and if its return status is zero, the *consequent-commands* list is executed. If *test-commands* returns a non-zero status, each `elif` list is executed in turn, and if its exit status is zero, the corresponding *more-consequents* is executed and the command completes. If '`else alternate-consequents`' is present, and the final command in the final `if` or `elif` clause has a non-zero exit status, then *alternate-consequents* is executed. The return status is the exit status of the last command executed, or zero if no condition tested true.

`case` The syntax of the `case` command is:

```
case word in
    [ [(] pattern [| pattern]...) command-list ;;]...
esac
```

`case` will selectively execute the *command-list* corresponding to the first *pattern* that matches *word*. The match is performed according to the rules described below in Section 3.5.8.1 [Pattern Matching], page 33. If the `nocasematch` shell option (see the description of `shopt` in Section 4.3.2 [The Shopt Builtin], page 65) is enabled, the match is performed without regard to the case of alphabetic characters. The '`|`' is used to separate multiple patterns, and the '`)`' operator terminates a pattern list. A list of patterns and an associated command-list is known as a *clause*.

Each clause must be terminated with '`;;`', '`;&`', or '`;;&`'. The *word* undergoes tilde expansion, parameter expansion, command substitution, arithmetic expansion, and quote removal (see Section 3.5.3 [Shell Parameter Expansion], page 24) before matching is attempted. Each *pattern* undergoes tilde expansion, parameter expansion, command substitution, and arithmetic expansion.

There may be an arbitrary number of `case` clauses, each terminated by a '`;;`', '`;&`', or '`;;&`'. The first pattern that matches determines the command-list that is executed. It's a common idiom to use '`*`' as the final pattern to define the default case, since that pattern will always match.

Here is an example using `case` in a script that could be used to describe one interesting feature of an animal:

```
echo -n "Enter the name of an animal: "
read ANIMAL
echo -n "The $ANIMAL has "
case $ANIMAL in
  horse | dog | cat) echo -n "four";;
```

```
  man | kangaroo ) echo -n "two";;
  *) echo -n "an unknown number of";;
esac
echo " legs."
```

If the '`;;`' operator is used, no subsequent matches are attempted after the first pattern match. Using '`;&`' in place of '`;;`' causes execution to continue with the *command-list* associated with the next clause, if any. Using '`;;&`' in place of '`;;`' causes the shell to test the patterns in the next clause, if any, and execute any associated *command-list* on a successful match, continuing the case statement execution as if the pattern list had not matched.

The return status is zero if no *pattern* is matched. Otherwise, the return status is the exit status of the *command-list* executed.

`select`

The `select` construct allows the easy generation of menus. It has almost the same syntax as the `for` command:

```
select name [in words ...]; do commands; done
```

The list of words following `in` is expanded, generating a list of items. The set of expanded words is printed on the standard error output stream, each preceded by a number. If the '`in words`' is omitted, the positional parameters are printed, as if '`in "$@"`' had been specified. The `PS3` prompt is then displayed and a line is read from the standard input. If the line consists of a number corresponding to one of the displayed words, then the value of *name* is set to that word. If the line is empty, the words and prompt are displayed again. If `EOF` is read, the `select` command completes. Any other value read causes *name* to be set to null. The line read is saved in the variable `REPLY`.

The *commands* are executed after each selection until a `break` command is executed, at which point the `select` command completes.

Here is an example that allows the user to pick a filename from the current directory, and displays the name and index of the file selected.

```
select fname in *;
do
echo you picked $fname \($REPLY\)
break;
done
```

`((...))`

```
(( expression ))
```

The arithmetic *expression* is evaluated according to the rules described below (see Section 6.5 [Shell Arithmetic], page 92). If the value of the expression is non-zero, the return status is 0; otherwise the return status is 1. This is exactly equivalent to

```
let "expression"
```

See Section 4.2 [Bash Builtins], page 50, for a full description of the `let` builtin.

`[[...]]`

```
[[ expression ]]
```

Return a status of 0 or 1 depending on the evaluation of the conditional expression *expression*. Expressions are composed of the primaries described below in Section 6.4 [Bash Conditional Expressions], page 90. Word splitting and filename expansion are not performed on the words between the `[[` and `]]`; tilde expansion, parameter and variable expansion, arithmetic expansion, command substitution, process substitution, and quote removal are performed. Conditional operators such as '`-f`' must be unquoted to be recognized as primaries.

When used with `[[`, the '`<`' and '`>`' operators sort lexicographically using the current locale.

When the '`==`' and '`!=`' operators are used, the string to the right of the operator is considered a pattern and matched according to the rules described below in Section 3.5.8.1 [Pattern Matching], page 33, as if the `extglob` shell option were enabled. The '`=`' operator is identical to '`==`'. If the `nocasematch` shell option (see the description of `shopt` in Section 4.3.2 [The Shopt Builtin], page 65) is enabled, the match is performed without regard to the case of alphabetic characters. The return value is 0 if the string matches ('`==`') or does not match ('`!=`') the pattern, and 1 otherwise. Any part of the pattern may be quoted to force the quoted portion to be matched as a string.

An additional binary operator, '`=~`', is available, with the same precedence as '`==`' and '`!=`'. When it is used, the string to the right of the operator is considered a POSIX extended regular expression and matched accordingly (as in *regex*3)). The return value is 0 if the string matches the pattern, and 1 otherwise. If the regular expression is syntactically incorrect, the conditional expression's return value is 2. If the `nocasematch` shell option (see the description of `shopt` in Section 4.3.2 [The Shopt Builtin], page 65) is enabled, the match is performed without regard to the case of alphabetic characters. Any part of the pattern may be quoted to force the quoted portion to be matched as a string. Bracket expressions in regular expressions must be treated carefully, since normal quoting characters lose their meanings between brackets. If the pattern is stored in a shell variable, quoting the variable expansion forces the entire pattern to be matched as a string. Substrings matched by parenthesized subexpressions within the regular expression are saved in the array variable `BASH_REMATCH`. The element of `BASH_REMATCH` with index 0 is the portion of the string matching the entire regular expression. The element of `BASH_REMATCH` with index *n* is the portion of the string matching the *n*th parenthesized subexpression.

For example, the following will match a line (stored in the shell variable *line*) if there is a sequence of characters in the value consisting of any number, including zero, of space characters, zero or one instances of '`a`', then a '`b`':

```
[[ $line =~ [[:space:]]*?(a)b ]]
```

That means values like '`aab`' and '`  aaaaaab`' will match, as will a line containing a '`b`' anywhere in its value.

Storing the regular expression in a shell variable is often a useful way to avoid problems with quoting characters that are special to the shell. It is sometimes difficult to specify a regular expression literally without using quotes, or to keep track of the quoting used by regular expressions while paying attention to the shell's quote removal. Using a shell variable to store the pattern decreases these problems. For example, the following is equivalent to the above:

```
pattern='[[:space:]]*?(a)b'
[[ $line =~ $pattern ]]
```

If you want to match a character that's special to the regular expression grammar, it has to be quoted to remove its special meaning. This means that in the pattern '`xxx.txt`', the '`.`' matches any character in the string (its usual regular expression meaning), but in the pattern '`"xxx.txt"`' it can only match a literal '`.`'. Shell programmers should take special care with backslashes, since backslashes are used both by the shell and regular expressions to remove the special meaning from the following character. The following two sets of commands are *not* equivalent:

```
pattern='\.'

[[ . =~ $pattern ]]
[[ . =~ \. ]]

[[ . =~ "$pattern" ]]
[[ . =~ '\.' ]]
```

The first two matches will succeed, but the second two will not, because in the second two the backslash will be part of the pattern to be matched. In the first two examples, the backslash removes the special meaning from '`.`', so the literal '`.`' matches. If the string in the first examples were anything other than '`.`', say '`a`', the pattern would not match, because the quoted '`.`' in the pattern loses its special meaning of matching any single character.

Expressions may be combined using the following operators, listed in decreasing order of precedence:

`( expression )`
: Returns the value of *expression*. This may be used to override the normal precedence of operators.

`! expression`
: True if *expression* is false.

`expression1 && expression2`
: True if both *expression1* and *expression2* are true.

`expression1 || expression2`
: True if either *expression1* or *expression2* is true.

The `&&` and `||` operators do not evaluate *expression2* if the value of *expression1* is sufficient to determine the return value of the entire conditional expression.

3.2.4.3 Grouping Commands

Bash provides two ways to group a list of commands to be executed as a unit. When commands are grouped, redirections may be applied to the entire command list. For example, the output of all the commands in the list may be redirected to a single stream.

`()`

```
( list )
```

Placing a list of commands between parentheses causes a subshell environment to be created (see Section 3.7.3 [Command Execution Environment], page 39), and each of the commands in *list* to be executed in that subshell. Since the *list* is executed in a subshell, variable assignments do not remain in effect after the subshell completes.

`{}`

```
{ list; }
```

Placing a list of commands between curly braces causes the list to be executed in the current shell context. No subshell is created. The semicolon (or newline) following *list* is required.

In addition to the creation of a subshell, there is a subtle difference between these two constructs due to historical reasons. The braces are `reserved words`, so they must be separated from the *list* by `blank`s or other shell metacharacters. The parentheses are `operators`, and are recognized as separate tokens by the shell even if they are not separated from the *list* by whitespace.

The exit status of both of these constructs is the exit status of *list*.

3.2.5 Coprocesses

A `coprocess` is a shell command preceded by the `coproc` reserved word. A coprocess is executed asynchronously in a subshell, as if the command had been terminated with the '`&`' control operator, with a two-way pipe established between the executing shell and the coprocess.

The format for a coprocess is:

```
coproc [NAME] command [redirections]
```

This creates a coprocess named *NAME*. If *NAME* is not supplied, the default name is *COPROC*. *NAME* must not be supplied if *command* is a simple command (see Section 3.2.1 [Simple Commands], page 8); otherwise, it is interpreted as the first word of the simple command.

When the coprocess is executed, the shell creates an array variable (see Section 6.7 [Arrays], page 94) named `NAME` in the context of the executing shell. The standard output of *command* is connected via a pipe to a file descriptor in the executing shell, and that file descriptor is assigned to `NAME`[0]. The standard input of *command* is connected via a pipe to a file descriptor in the executing shell, and that file descriptor is assigned to `NAME`[1]. This pipe is established before any redirections specified by the command (see Section 3.6 [Redirections], page 34). The file descriptors can be utilized as arguments to shell commands and redirections using standard word expansions. Other than those created to execute command and process substitutions, the file descriptors are not available in subshells.

The process ID of the shell spawned to execute the coprocess is available as the value of the variable `NAME_PID`. The `wait` builtin command may be used to wait for the coprocess to terminate.

Since the coprocess is created as an asynchronous command, the `coproc` command always returns success. The return status of a coprocess is the exit status of *command*.

3.2.6 GNU Parallel

There are ways to run commands in parallel that are not built into Bash. GNU Parallel is a tool to do just that.

GNU Parallel, as its name suggests, can be used to build and run commands in parallel. You may run the same command with different arguments, whether they are filenames, usernames, hostnames, or lines read from files. GNU Parallel provides shorthand references to many of the most common operations (input lines, various portions of the input line, different ways to specify the input source, and so on). Parallel can replace `xargs` or feed commands from its input sources to several different instances of Bash.

For a complete description, refer to the GNU Parallel documentation. A few examples should provide a brief introduction to its use.

For example, it is easy to replace `xargs` to gzip all html files in the current directory and its subdirectories:

```
find . -type f -name '*.html' -print | parallel gzip
```

If you need to protect special characters such as newlines in file names, use find's `-print0` option and parallel's `-0` option.

You can use Parallel to move files from the current directory when the number of files is too large to process with one `mv` invocation:

```
ls | parallel mv {} destdir
```

As you can see, the {} is replaced with each line read from standard input. While using `ls` will work in most instances, it is not sufficient to deal with all filenames. If you need to accommodate special characters in filenames, you can use

```
find . -depth 1 \! -name '.*' -print0 | parallel -0 mv {} destdir
```

as alluded to above.

This will run as many `mv` commands as there are files in the current directory. You can emulate a parallel `xargs` by adding the `-X` option:

```
find . -depth 1 \! -name '.*' -print0 | parallel -0 -X mv {} destdir
```

GNU Parallel can replace certain common idioms that operate on lines read from a file (in this case, filenames listed one per line):

```
while IFS= read -r x; do
do-something1 "$x" "config-$x"
do-something2 < "$x"
done < file | process-output
```

with a more compact syntax reminiscent of lambdas:

```
cat list | parallel "do-something1 {} config-{} ; do-something2 < {}" |
```

```
                process-output
```

Parallel provides a built-in mechanism to remove filename extensions, which lends itself to batch file transformations or renaming:

```
ls *.gz | parallel -j+0 "zcat {} | bzip2 >{.}.bz2 && rm {}"
```

This will recompress all files in the current directory with names ending in .gz using bzip2, running one job per CPU (-j+0) in parallel. (We use `ls` for brevity here; using `find` as above is more robust in the face of filenames containing unexpected characters.) Parallel can take arguments from the command line; the above can also be written as

```
parallel "zcat {} | bzip2 >{.}.bz2 && rm {}" ::: *.gz
```

If a command generates output, you may want to preserve the input order in the output. For instance, the following command

```
{
    echo foss.org.my ;
    echo debian.org ;
    echo freenetproject.org ;
} | parallel traceroute
```

will display as output the traceroute invocation that finishes first. Adding the `-k` option

```
{
    echo foss.org.my ;
    echo debian.org ;
    echo freenetproject.org ;
} | parallel -k traceroute
```

will ensure that the output of `traceroute foss.org.my` is displayed first.

Finally, Parallel can be used to run a sequence of shell commands in parallel, similar to '`cat file | bash`'. It is not uncommon to take a list of filenames, create a series of shell commands to operate on them, and feed that list of commands to a shell. Parallel can speed this up. Assuming that `file` contains a list of shell commands, one per line,

```
parallel -j 10 < file
```

will evaluate the commands using the shell (since no explicit command is supplied as an argument), in blocks of ten shell jobs at a time.

3.3 Shell Functions

Shell functions are a way to group commands for later execution using a single name for the group. They are executed just like a "regular" command. When the name of a shell function is used as a simple command name, the list of commands associated with that function name is executed. Shell functions are executed in the current shell context; no new process is created to interpret them.

Functions are declared using this syntax:

name `()` *compound-command* `[` *redirections* `]`

or

`function` *name* `[()]` *compound-command* `[` *redirections* `]`

This defines a shell function named *name*. The reserved word `function` is optional. If the `function` reserved word is supplied, the parentheses are optional. The *body* of the function is the compound command *compound-command* (see Section 3.2.4 [Compound Commands], page 9). That command is usually a *list* enclosed between { and }, but may be any compound command listed above, with one exception: If the `function` reserved word is used, but the parentheses are not supplied, the braces are required. *compound-command* is executed whenever *name* is specified as the name of a command. When the shell is in POSIX mode (see Section 6.11 [Bash POSIX Mode], page 99), *name* may not be the same as one of the special builtins (see Section 4.4 [Special Builtins], page 71). Any redirections (see Section 3.6 [Redirections], page 34) associated with the shell function are performed when the function is executed.

A function definition may be deleted using the `-f` option to the `unset` builtin (see Section 4.1 [Bourne Shell Builtins], page 43).

The exit status of a function definition is zero unless a syntax error occurs or a readonly function with the same name already exists. When executed, the exit status of a function is the exit status of the last command executed in the body.

Note that for historical reasons, in the most common usage the curly braces that surround the body of the function must be separated from the body by `blank`s or newlines. This is because the braces are reserved words and are only recognized as such when they are separated from the command list by whitespace or another shell metacharacter. Also, when using the braces, the *list* must be terminated by a semicolon, a '`&`', or a newline.

When a function is executed, the arguments to the function become the positional parameters during its execution (see Section 3.4.1 [Positional Parameters], page 21). The special parameter '`#`' that expands to the number of positional parameters is updated to reflect the change. Special parameter `0` is unchanged. The first element of the `FUNCNAME` variable is set to the name of the function while the function is executing.

All other aspects of the shell execution environment are identical between a function and its caller with these exceptions: the `DEBUG` and `RETURN` traps are not inherited unless the function has been given the `trace` attribute using the `declare` builtin or the `-o functrace` option has been enabled with the `set` builtin, (in which case all functions inherit the `DEBUG` and `RETURN` traps), and the `ERR` trap is not inherited unless the `-o errtrace` shell option has been enabled. See Section 4.1 [Bourne Shell Builtins], page 43, for the description of the `trap` builtin.

The `FUNCNEST` variable, if set to a numeric value greater than 0, defines a maximum function nesting level. Function invocations that exceed the limit cause the entire command to abort.

If the builtin command `return` is executed in a function, the function completes and execution resumes with the next command after the function call. Any command associated with the `RETURN` trap is executed before execution resumes. When a function completes, the values of the positional parameters and the special parameter '`#`' are restored to the values they had prior to the function's execution. If a numeric argument is given to `return`, that is the function's return status; otherwise the function's return status is the exit status of the last command executed before the `return`.

Variables local to the function may be declared with the `local` builtin. These variables are visible only to the function and the commands it invokes. This is particularly important when a shell function calls other functions.

Local variables "shadow" variables with the same name declared at previous scopes. For instance, a local variable declared in a function hides a global variable of the same name: references and assignments refer to the local variable, leaving the global variable unmodified. When the function returns, the global variable is once again visible.

The shell uses *dynamic scoping* to control a variable's visibility within functions. With dynamic scoping, visible variables and their values are a result of the sequence of function calls that caused execution to reach the current function. The value of a variable that a function sees depends on its value within its caller, if any, whether that caller is the "global" scope or another shell function. This is also the value that a local variable declaration "shadows", and the value that is restored when the function returns.

For example, if a variable *var* is declared as local in function *func1*, and *func1* calls another function *func2*, references to *var* made from within *func2* will resolve to the local variable *var* from *func1*, shadowing any global variable named *var*.

The following script demonstrates this behavior. When executed, the script displays

```
In func2, var = func1 local
```

```
func1()
{
    local var='func1 local'
    func2
}

func2()
{
    echo "In func2, var = $var"
}

var=global
func1
```

The `unset` builtin also acts using the same dynamic scope: if a variable is local to the current scope, `unset` will unset it; otherwise the unset will refer to the variable found in any calling scope as described above. If a variable at the current local scope is unset, it will remain so until it is reset in that scope or until the function returns. Once the function returns, any instance of the variable at a previous scope will become visible. If the unset acts on a variable at a previous scope, any instance of a variable with that name that had been shadowed will become visible.

Function names and definitions may be listed with the `-f` option to the `declare` (`typeset`) builtin command (see Section 4.2 [Bash Builtins], page 50). The `-F` option to `declare` or `typeset` will list the function names only (and optionally the source file and line number, if the `extdebug` shell option is enabled). Functions may be exported so that subshells automatically have them defined with the `-f` option to the `export` builtin (see Section 4.1 [Bourne Shell Builtins], page 43).

Functions may be recursive. The `FUNCNEST` variable may be used to limit the depth of the function call stack and restrict the number of function invocations. By default, no limit is placed on the number of recursive calls.

3.4 Shell Parameters

A *parameter* is an entity that stores values. It can be a `name`, a number, or one of the special characters listed below. A *variable* is a parameter denoted by a `name`. A variable has a *value* and zero or more *attributes*. Attributes are assigned using the `declare` builtin command (see the description of the `declare` builtin in Section 4.2 [Bash Builtins], page 50).

A parameter is set if it has been assigned a value. The null string is a valid value. Once a variable is set, it may be unset only by using the `unset` builtin command.

A variable may be assigned to by a statement of the form

```
name=[value]
```

If *value* is not given, the variable is assigned the null string. All *value*s undergo tilde expansion, parameter and variable expansion, command substitution, arithmetic expansion, and quote removal (detailed below). If the variable has its `integer` attribute set, then *value* is evaluated as an arithmetic expression even if the `$((...))` expansion is not used (see Section 3.5.5 [Arithmetic Expansion], page 31). Word splitting is not performed, with the exception of `"$@"` as explained below. Filename expansion is not performed. Assignment statements may also appear as arguments to the `alias`, `declare`, `typeset`, `export`, `readonly`, and `local` builtin commands (*declaration* commands). When in POSIX mode (see Section 6.11 [Bash POSIX Mode], page 99), these builtins may appear in a command after one or more instances of the `command` builtin and retain these assignment statement properties.

In the context where an assignment statement is assigning a value to a shell variable or array index (see Section 6.7 [Arrays], page 94), the '`+=`' operator can be used to append to or add to the variable's previous value. This includes arguments to builtin commands such as `declare` that accept assignment statements (*declaration* commands). When '`+=`' is applied to a variable for which the *integer* attribute has been set, *value* is evaluated as an arithmetic expression and added to the variable's current value, which is also evaluated. When '`+=`' is applied to an array variable using compound assignment (see Section 6.7 [Arrays], page 94), the variable's value is not unset (as it is when using '`=`'), and new values are appended to the array beginning at one greater than the array's maximum index (for indexed arrays), or added as additional key-value pairs in an associative array. When applied to a string-valued variable, *value* is expanded and appended to the variable's value.

A variable can be assigned the *nameref* attribute using the `-n` option to the `declare` or `local` builtin commands (see Section 4.2 [Bash Builtins], page 50) to create a *nameref*, or a reference to another variable. This allows variables to be manipulated indirectly. Whenever the nameref variable is referenced, assigned to, unset, or has its attributes modified (other than using or changing the nameref attribute itself), the operation is actually performed on the variable specified by the nameref variable's value. A nameref is commonly used within shell functions to refer to a variable whose name is passed as an argument to the function. For instance, if a variable name is passed to a shell function as its first argument, running

```
declare -n ref=$1
```

inside the function creates a nameref variable *ref* whose value is the variable name passed as the first argument. References and assignments to *ref*, and changes to its attributes, are treated as references, assignments, and attribute modifications to the variable whose name was passed as `$1`.

If the control variable in a for loop has the nameref attribute, the list of words can be a list of shell variables, and a name reference will be established for each word in the list, in turn, when the loop is executed. Array variables cannot be given the nameref attribute. However, nameref variables can reference array variables and subscripted array variables. Namerefs can be unset using the -n option to the unset builtin (see Section 4.1 [Bourne Shell Builtins], page 43). Otherwise, if unset is executed with the name of a nameref variable as an argument, the variable referenced by the nameref variable will be unset.

3.4.1 Positional Parameters

A *positional parameter* is a parameter denoted by one or more digits, other than the single digit 0. Positional parameters are assigned from the shell's arguments when it is invoked, and may be reassigned using the set builtin command. Positional parameter N may be referenced as ${N}, or as $N when N consists of a single digit. Positional parameters may not be assigned to with assignment statements. The set and shift builtins are used to set and unset them (see Chapter 4 [Shell Builtin Commands], page 43). The positional parameters are temporarily replaced when a shell function is executed (see Section 3.3 [Shell Functions], page 17).

When a positional parameter consisting of more than a single digit is expanded, it must be enclosed in braces.

3.4.2 Special Parameters

The shell treats several parameters specially. These parameters may only be referenced; assignment to them is not allowed.

*
: ($*) Expands to the positional parameters, starting from one. When the expansion is not within double quotes, each positional parameter expands to a separate word. In contexts where it is performed, those words are subject to further word splitting and pathname expansion. When the expansion occurs within double quotes, it expands to a single word with the value of each parameter separated by the first character of the IFS special variable. That is, "$*" is equivalent to "$1*c*$2*c*...", where *c* is the first character of the value of the IFS variable. If IFS is unset, the parameters are separated by spaces. If IFS is null, the parameters are joined without intervening separators.

@
: ($@) Expands to the positional parameters, starting from one. In contexts where word splitting is performed, this expands each positional parameter to a separate word; if not within double quotes, these words are subject to word splitting. In contexts where word splitting is not performed, this expands to a single word with each positional parameter separated by a space. When the expansion occurs within double quotes, and word splitting is performed, each parameter expands to a separate word. That is, "$@" is equivalent to "$1" "$2" .... If the double-quoted expansion occurs within a word, the expansion of the first parameter is joined with the beginning part of the original word, and the expansion of the last parameter is joined with the last part of the original word. When there are no positional parameters, "$@" and $@ expand to nothing (i.e., they are removed).

#
: ($#) Expands to the number of positional parameters in decimal.

`?` ($?) Expands to the exit status of the most recently executed foreground pipeline.

`-` ($-, a hyphen.) Expands to the current option flags as specified upon invocation, by the `set` builtin command, or those set by the shell itself (such as the `-i` option).

`$` ($$) Expands to the process ID of the shell. In a `()` subshell, it expands to the process ID of the invoking shell, not the subshell.

`!` ($!) Expands to the process ID of the job most recently placed into the background, whether executed as an asynchronous command or using the `bg` builtin (see Section 7.2 [Job Control Builtins], page 105).

`0` ($0) Expands to the name of the shell or shell script. This is set at shell initialization. If Bash is invoked with a file of commands (see Section 3.8 [Shell Scripts], page 42), `$0` is set to the name of that file. If Bash is started with the `-c` option (see Section 6.1 [Invoking Bash], page 85), then `$0` is set to the first argument after the string to be executed, if one is present. Otherwise, it is set to the filename used to invoke Bash, as given by argument zero.

`_` ($_, an underscore.) At shell startup, set to the absolute pathname used to invoke the shell or shell script being executed as passed in the environment or argument list. Subsequently, expands to the last argument to the previous simple command executed in the foreground, after expansion. Also set to the full pathname used to invoke each command executed and placed in the environment exported to that command. When checking mail, this parameter holds the name of the mail file.

3.5 Shell Expansions

Expansion is performed on the command line after it has been split into `token`s. There are seven kinds of expansion performed:

- brace expansion
- tilde expansion
- parameter and variable expansion
- command substitution
- arithmetic expansion
- word splitting
- filename expansion

The order of expansions is: brace expansion; tilde expansion, parameter and variable expansion, arithmetic expansion, and command substitution (done in a left-to-right fashion); word splitting; and filename expansion.

On systems that can support it, there is an additional expansion available: *process substitution*. This is performed at the same time as tilde, parameter, variable, and arithmetic expansion and command substitution.

After these expansions are performed, quote characters present in the original word are removed unless they have been quoted themselves (*quote removal*).

Only brace expansion, word splitting, and filename expansion can increase the number of words of the expansion; other expansions expand a single word to a single word. The only exceptions to this are the expansions of `"$@"` and `$*` (see Section 3.4.2 [Special Parameters], page 21), and `"${name[@]}"` and `${name[*]}` (see Section 6.7 [Arrays], page 94).

After all expansions, `quote removal` (see Section 3.5.9 [Quote Removal], page 34) is performed.

3.5.1 Brace Expansion

Brace expansion is a mechanism by which arbitrary strings may be generated. This mechanism is similar to *filename expansion* (see Section 3.5.8 [Filename Expansion], page 32), but the filenames generated need not exist. Patterns to be brace expanded take the form of an optional *preamble*, followed by either a series of comma-separated strings or a sequence expression between a pair of braces, followed by an optional *postscript*. The preamble is prefixed to each string contained within the braces, and the postscript is then appended to each resulting string, expanding left to right.

Brace expansions may be nested. The results of each expanded string are not sorted; left to right order is preserved. For example,

```
bash$ echo a{d,c,b}e
ade ace abe
```

A sequence expression takes the form `{x..y[..incr]}`, where *x* and *y* are either integers or single characters, and *incr*, an optional increment, is an integer. When integers are supplied, the expression expands to each number between *x* and *y*, inclusive. Supplied integers may be prefixed with '`0`' to force each term to have the same width. When either *x* or *y* begins with a zero, the shell attempts to force all generated terms to contain the same number of digits, zero-padding where necessary. When characters are supplied, the expression expands to each character lexicographically between *x* and *y*, inclusive, using the default C locale. Note that both *x* and *y* must be of the same type. When the increment is supplied, it is used as the difference between each term. The default increment is 1 or -1 as appropriate.

Brace expansion is performed before any other expansions, and any characters special to other expansions are preserved in the result. It is strictly textual. Bash does not apply any syntactic interpretation to the context of the expansion or the text between the braces.

A correctly-formed brace expansion must contain unquoted opening and closing braces, and at least one unquoted comma or a valid sequence expression. Any incorrectly formed brace expansion is left unchanged.

A { or '`,`' may be quoted with a backslash to prevent its being considered part of a brace expression. To avoid conflicts with parameter expansion, the string '`${`' is not considered eligible for brace expansion, and inhibits brace expansion until the closing '`}`'.

This construct is typically used as shorthand when the common prefix of the strings to be generated is longer than in the above example:

```
mkdir /usr/local/src/bash/{old,new,dist,bugs}
```

or

```
chown root /usr/{ucb/{ex,edit},lib/{ex?.?*,how_ex}}
```

3.5.2 Tilde Expansion

If a word begins with an unquoted tilde character ('~'), all of the characters up to the first unquoted slash (or all characters, if there is no unquoted slash) are considered a *tilde-prefix*. If none of the characters in the tilde-prefix are quoted, the characters in the tilde-prefix following the tilde are treated as a possible *login name*. If this login name is the null string, the tilde is replaced with the value of the `HOME` shell variable. If `HOME` is unset, the home directory of the user executing the shell is substituted instead. Otherwise, the tilde-prefix is replaced with the home directory associated with the specified login name.

If the tilde-prefix is '`~+`', the value of the shell variable `PWD` replaces the tilde-prefix. If the tilde-prefix is '`~-`', the value of the shell variable `OLDPWD`, if it is set, is substituted.

If the characters following the tilde in the tilde-prefix consist of a number *N*, optionally prefixed by a '`+`' or a '`-`', the tilde-prefix is replaced with the corresponding element from the directory stack, as it would be displayed by the `dirs` builtin invoked with the characters following tilde in the tilde-prefix as an argument (see Section 6.8 [The Directory Stack], page 96). If the tilde-prefix, sans the tilde, consists of a number without a leading '`+`' or '`-`', '`+`' is assumed.

If the login name is invalid, or the tilde expansion fails, the word is left unchanged.

Each variable assignment is checked for unquoted tilde-prefixes immediately following a '`:`' or the first '`=`'. In these cases, tilde expansion is also performed. Consequently, one may use filenames with tildes in assignments to `PATH`, `MAILPATH`, and `CDPATH`, and the shell assigns the expanded value.

The following table shows how Bash treats unquoted tilde-prefixes:

`~` The value of `$HOME`

`~/foo` `$HOME/foo`

`~fred/foo`
The subdirectory `foo` of the home directory of the user `fred`

`~+/foo` `$PWD/foo`

`~-/foo` `${OLDPWD-'~-'}/foo`

`~N` The string that would be displayed by '`dirs +N`'

`~+N` The string that would be displayed by '`dirs +N`'

`~-N` The string that would be displayed by '`dirs -N`'

Bash also performs tilde expansion on words satisfying the conditions of variable assignments (see Section 3.4 [Shell Parameters], page 20) when they appear as arguments to simple commands. Bash does not do this, except for the *declaration* commands listed above, when in POSIX mode.

3.5.3 Shell Parameter Expansion

The '`$`' character introduces parameter expansion, command substitution, or arithmetic expansion. The parameter name or symbol to be expanded may be enclosed in braces, which are optional but serve to protect the variable to be expanded from characters immediately following it which could be interpreted as part of the name.

When braces are used, the matching ending brace is the first '`}`' not escaped by a backslash or within a quoted string, and not within an embedded arithmetic expansion, command substitution, or parameter expansion.

The basic form of parameter expansion is ${*parameter*}. The value of *parameter* is substituted. The *parameter* is a shell parameter as described above (see Section 3.4 [Shell Parameters], page 20) or an array reference (see Section 6.7 [Arrays], page 94). The braces are required when *parameter* is a positional parameter with more than one digit, or when *parameter* is followed by a character that is not to be interpreted as part of its name.

If the first character of *parameter* is an exclamation point (!), and *parameter* is not a *nameref*, it introduces a level of indirection. Bash uses the value formed by expanding the rest of *parameter* as the new *parameter*; this is then expanded and that value is used in the rest of the expansion, rather than the expansion of the original *parameter*. This is known as `indirect expansion`. The value is subject to tilde expansion, parameter expansion, command substitution, and arithmetic expansion. If *parameter* is a nameref, this expands to the name of the variable referenced by *parameter* instead of performing the complete indirect expansion. The exceptions to this are the expansions of ${!*prefix**} and ${!*name*[@]} described below. The exclamation point must immediately follow the left brace in order to introduce indirection.

In each of the cases below, *word* is subject to tilde expansion, parameter expansion, command substitution, and arithmetic expansion.

When not performing substring expansion, using the form described below (e.g., '`:-`'), Bash tests for a parameter that is unset or null. Omitting the colon results in a test only for a parameter that is unset. Put another way, if the colon is included, the operator tests for both *parameter*'s existence and that its value is not null; if the colon is omitted, the operator tests only for existence.

`${parameter:−word}`
: If *parameter* is unset or null, the expansion of *word* is substituted. Otherwise, the value of *parameter* is substituted.

`${parameter:=word}`
: If *parameter* is unset or null, the expansion of *word* is assigned to *parameter*. The value of *parameter* is then substituted. Positional parameters and special parameters may not be assigned to in this way.

`${parameter:?word}`
: If *parameter* is null or unset, the expansion of *word* (or a message to that effect if *word* is not present) is written to the standard error and the shell, if it is not interactive, exits. Otherwise, the value of *parameter* is substituted.

`${parameter:+word}`
: If *parameter* is null or unset, nothing is substituted, otherwise the expansion of *word* is substituted.

`${parameter:offset}`
`${parameter:offset:length}`
: This is referred to as Substring Expansion. It expands to up to *length* characters of the value of *parameter* starting at the character specified by *offset*. If

parameter is '@', an indexed array subscripted by '@' or '*', or an associative array name, the results differ as described below. If *length* is omitted, it expands to the substring of the value of *parameter* starting at the character specified by *offset* and extending to the end of the value. *length* and *offset* are arithmetic expressions (see Section 6.5 [Shell Arithmetic], page 92).

If *offset* evaluates to a number less than zero, the value is used as an offset in characters from the end of the value of *parameter*. If *length* evaluates to a number less than zero, it is interpreted as an offset in characters from the end of the value of *parameter* rather than a number of characters, and the expansion is the characters between *offset* and that result. Note that a negative offset must be separated from the colon by at least one space to avoid being confused with the ':-' expansion.

Here are some examples illustrating substring expansion on parameters and subscripted arrays:

```
$ string=01234567890abcdefgh
$ echo ${string:7}
7890abcdefgh
$ echo ${string:7:0}

$ echo ${string:7:2}
78
$ echo ${string:7:-2}
7890abcdef
$ echo ${string: -7}
bcdefgh
$ echo ${string: -7:0}

$ echo ${string: -7:2}
bc
$ echo ${string: -7:-2}
bcdef
$ set -- 01234567890abcdefgh
$ echo ${1:7}
7890abcdefgh
$ echo ${1:7:0}

$ echo ${1:7:2}
78
$ echo ${1:7:-2}
7890abcdef
$ echo ${1: -7}
bcdefgh
$ echo ${1: -7:0}

$ echo ${1: -7:2}
bc
```

```
$ echo ${1: -7:-2}
bcdef
$ array[0]=01234567890abcdefgh
$ echo ${array[0]:7}
7890abcdefgh
$ echo ${array[0]:7:0}

$ echo ${array[0]:7:2}
78
$ echo ${array[0]:7:-2}
7890abcdef
$ echo ${array[0]: -7}
bcdefgh
$ echo ${array[0]: -7:0}

$ echo ${array[0]: -7:2}
bc
$ echo ${array[0]: -7:-2}
bcdef
```

If *parameter* is '`@`', the result is *length* positional parameters beginning at *offset*. A negative *offset* is taken relative to one greater than the greatest positional parameter, so an offset of -1 evaluates to the last positional parameter. It is an expansion error if *length* evaluates to a number less than zero.

The following examples illustrate substring expansion using positional parameters:

```
$ set -- 1 2 3 4 5 6 7 8 9 0 a b c d e f g h
$ echo ${@:7}
7 8 9 0 a b c d e f g h
$ echo ${@:7:0}

$ echo ${@:7:2}
7 8
$ echo ${@:7:-2}
bash: -2: substring expression < 0
$ echo ${@: -7:2}
b c
$ echo ${@:0}
./bash 1 2 3 4 5 6 7 8 9 0 a b c d e f g h
$ echo ${@:0:2}
./bash 1
$ echo ${@: -7:0}
```

If *parameter* is an indexed array name subscripted by '`@`' or '`*`', the result is the *length* members of the array beginning with `${`*`parameter`*`[`*`offset`*`]}`. A negative *offset* is taken relative to one greater than the maximum index of the specified array. It is an expansion error if *length* evaluates to a number less than zero.

These examples show how you can use substring expansion with indexed arrays:

```
$ array=(0 1 2 3 4 5 6 7 8 9 0 a b c d e f g h)
$ echo ${array[@]:7}
7 8 9 0 a b c d e f g h
$ echo ${array[@]:7:2}
7 8
$ echo ${array[@]: -7:2}
b c
$ echo ${array[@]: -7:-2}
bash: -2: substring expression < 0
$ echo ${array[@]:0}
0 1 2 3 4 5 6 7 8 9 0 a b c d e f g h
$ echo ${array[@]:0:2}
0 1
$ echo ${array[@]: -7:0}
```

Substring expansion applied to an associative array produces undefined results.

Substring indexing is zero-based unless the positional parameters are used, in which case the indexing starts at 1 by default. If *offset* is 0, and the positional parameters are used, `$@` is prefixed to the list.

`${!`*`prefix`*`*}`
`${!`*`prefix`*`@}`

Expands to the names of variables whose names begin with *prefix*, separated by the first character of the `IFS` special variable. When '`@`' is used and the expansion appears within double quotes, each variable name expands to a separate word.

`${!`*`name`*`[@]}`
`${!`*`name`*`[*]}`

If *name* is an array variable, expands to the list of array indices (keys) assigned in *name*. If *name* is not an array, expands to 0 if *name* is set and null otherwise. When '`@`' is used and the expansion appears within double quotes, each key expands to a separate word.

`${#`*`parameter`*`}`

The length in characters of the expanded value of *parameter* is substituted. If *parameter* is '`*`' or '`@`', the value substituted is the number of positional parameters. If *parameter* is an array name subscripted by '`*`' or '`@`', the value substituted is the number of elements in the array. If *parameter* is an indexed array name subscripted by a negative number, that number is interpreted as relative to one greater than the maximum index of *parameter*, so negative indices count back from the end of the array, and an index of -1 references the last element.

`${parameter#word}`
`${parameter##word}`

The *word* is expanded to produce a pattern and matched according to the rules described below (see Section 3.5.8.1 [Pattern Matching], page 33). If the pattern matches the beginning of the expanded value of *parameter*, then the result of the expansion is the expanded value of *parameter* with the shortest matching pattern (the '`#`' case) or the longest matching pattern (the '`##`' case) deleted. If *parameter* is '`@`' or '`*`', the pattern removal operation is applied to each positional parameter in turn, and the expansion is the resultant list. If *parameter* is an array variable subscripted with '`@`' or '`*`', the pattern removal operation is applied to each member of the array in turn, and the expansion is the resultant list.

`${parameter%word}`
`${parameter%%word}`

The *word* is expanded to produce a pattern and matched according to the rules described below (see Section 3.5.8.1 [Pattern Matching], page 33). If the pattern matches a trailing portion of the expanded value of *parameter*, then the result of the expansion is the value of *parameter* with the shortest matching pattern (the '`%`' case) or the longest matching pattern (the '`%%`' case) deleted. If *parameter* is '`@`' or '`*`', the pattern removal operation is applied to each positional parameter in turn, and the expansion is the resultant list. If *parameter* is an array variable subscripted with '`@`' or '`*`', the pattern removal operation is applied to each member of the array in turn, and the expansion is the resultant list.

`${parameter/pattern/string}`

The *pattern* is expanded to produce a pattern just as in filename expansion. *Parameter* is expanded and the longest match of *pattern* against its value is replaced with *string*. The match is performed according to the rules described below (see Section 3.5.8.1 [Pattern Matching], page 33). If *pattern* begins with '`/`', all matches of *pattern* are replaced with *string*. Normally only the first match is replaced. If *pattern* begins with '`#`', it must match at the beginning of the expanded value of *parameter*. If *pattern* begins with '`%`', it must match at the end of the expanded value of *parameter*. If *string* is null, matches of *pattern* are deleted and the `/` following *pattern* may be omitted. If the `nocasematch` shell option (see the description of `shopt` in Section 4.3.2 [The Shopt Builtin], page 65) is enabled, the match is performed without regard to the case of alphabetic characters. If *parameter* is '`@`' or '`*`', the substitution operation is applied to each positional parameter in turn, and the expansion is the resultant list. If *parameter* is an array variable subscripted with '`@`' or '`*`', the substitution operation is applied to each member of the array in turn, and the expansion is the resultant list.

`${parameter^pattern}`
`${parameter^^pattern}`
`${parameter,pattern}`
`${parameter,,pattern}`

This expansion modifies the case of alphabetic characters in *parameter*. The *pattern* is expanded to produce a pattern just as in filename expansion. Each character in the expanded value of *parameter* is tested against *pattern*, and, if it matches the pattern, its case is converted. The pattern should not attempt to match more than one character. The '`^`' operator converts lowercase letters matching *pattern* to uppercase; the '`,`' operator converts matching uppercase letters to lowercase. The '`^^`' and '`,,`' expansions convert each matched character in the expanded value; the '`^`' and '`,`' expansions match and convert only the first character in the expanded value. If *pattern* is omitted, it is treated like a '`?`', which matches every character. If *parameter* is '`@`' or '`*`', the case modification operation is applied to each positional parameter in turn, and the expansion is the resultant list. If *parameter* is an array variable subscripted with '`@`' or '`*`', the case modification operation is applied to each member of the array in turn, and the expansion is the resultant list.

`${parameter@operator}`

The expansion is either a transformation of the value of *parameter* or information about *parameter* itself, depending on the value of *operator*. Each *operator* is a single letter:

`Q` The expansion is a string that is the value of *parameter* quoted in a format that can be reused as input.

`E` The expansion is a string that is the value of *parameter* with backslash escape sequences expanded as with the `$'...'` quoting mechanism.

`P` The expansion is a string that is the result of expanding the value of *parameter* as if it were a prompt string (see Section 6.9 [Controlling the Prompt], page 97).

`A` The expansion is a string in the form of an assignment statement or `declare` command that, if evaluated, will recreate *parameter* with its attributes and value.

`a` The expansion is a string consisting of flag values representing *parameter*'s attributes.

If *parameter* is '`@`' or '`*`', the operation is applied to each positional parameter in turn, and the expansion is the resultant list. If *parameter* is an array variable subscripted with '`@`' or '`*`', the operation is applied to each member of the array in turn, and the expansion is the resultant list.

The result of the expansion is subject to word splitting and pathname expansion as described below.

3.5.4 Command Substitution

Command substitution allows the output of a command to replace the command itself. Command substitution occurs when a command is enclosed as follows:

```
$(command)
```

or

```
`command`
```

Bash performs the expansion by executing *command* in a subshell environment and replacing the command substitution with the standard output of the command, with any trailing newlines deleted. Embedded newlines are not deleted, but they may be removed during word splitting. The command substitution `$(cat file)` can be replaced by the equivalent but faster `$(< file)`.

When the old-style backquote form of substitution is used, backslash retains its literal meaning except when followed by ‘`$`’, ‘`` ` ``’, or ‘`\`’. The first backquote not preceded by a backslash terminates the command substitution. When using the `$(command)` form, all characters between the parentheses make up the command; none are treated specially.

Command substitutions may be nested. To nest when using the backquoted form, escape the inner backquotes with backslashes.

If the substitution appears within double quotes, word splitting and filename expansion are not performed on the results.

3.5.5 Arithmetic Expansion

Arithmetic expansion allows the evaluation of an arithmetic expression and the substitution of the result. The format for arithmetic expansion is:

```
$(( expression ))
```

The expression is treated as if it were within double quotes, but a double quote inside the parentheses is not treated specially. All tokens in the expression undergo parameter and variable expansion, command substitution, and quote removal. The result is treated as the arithmetic expression to be evaluated. Arithmetic expansions may be nested.

The evaluation is performed according to the rules listed below (see Section 6.5 [Shell Arithmetic], page 92). If the expression is invalid, Bash prints a message indicating failure to the standard error and no substitution occurs.

3.5.6 Process Substitution

Process substitution allows a process's input or output to be referred to using a filename. It takes the form of

```
<(list)
```

or

```
>(list)
```

The process *list* is run asynchronously, and its input or output appears as a filename. This filename is passed as an argument to the current command as the result of the expansion. If the `>(list)` form is used, writing to the file will provide input for *list*. If the `<(list)` form is used, the file passed as an argument should be read to obtain the output of *list*. Note that no space may appear between the `<` or `>` and the left parenthesis, otherwise the construct would be interpreted as a redirection. Process substitution is supported on systems that support named pipes (FIFOs) or the `/dev/fd` method of naming open files.

When available, process substitution is performed simultaneously with parameter and variable expansion, command substitution, and arithmetic expansion.

3.5.7 Word Splitting

The shell scans the results of parameter expansion, command substitution, and arithmetic expansion that did not occur within double quotes for word splitting.

The shell treats each character of `$IFS` as a delimiter, and splits the results of the other expansions into words using these characters as field terminators. If `IFS` is unset, or its value is exactly `<space><tab><newline>`, the default, then sequences of `<space>`, `<tab>`, and `<newline>` at the beginning and end of the results of the previous expansions are ignored, and any sequence of `IFS` characters not at the beginning or end serves to delimit words. If `IFS` has a value other than the default, then sequences of the whitespace characters `space`, `tab`, and `newline` are ignored at the beginning and end of the word, as long as the whitespace character is in the value of `IFS` (an `IFS` whitespace character). Any character in `IFS` that is not `IFS` whitespace, along with any adjacent `IFS` whitespace characters, delimits a field. A sequence of `IFS` whitespace characters is also treated as a delimiter. If the value of `IFS` is null, no word splitting occurs.

Explicit null arguments (`""` or `''`) are retained and passed to commands as empty strings. Unquoted implicit null arguments, resulting from the expansion of parameters that have no values, are removed. If a parameter with no value is expanded within double quotes, a null argument results and is retained and passed to a command as an empty string. When a quoted null argument appears as part of a word whose expansion is non-null, the null argument is removed. That is, the word `-d''` becomes `-d` after word splitting and null argument removal.

Note that if no expansion occurs, no splitting is performed.

3.5.8 Filename Expansion

After word splitting, unless the `-f` option has been set (see Section 4.3.1 [The Set Builtin], page 61), Bash scans each word for the characters '`*`', '`?`', and '`[`'. If one of these characters appears, then the word is regarded as a *pattern*, and replaced with an alphabetically sorted list of filenames matching the pattern (see Section 3.5.8.1 [Pattern Matching], page 33). If no matching filenames are found, and the shell option `nullglob` is disabled, the word is left unchanged. If the `nullglob` option is set, and no matches are found, the word is removed. If the `failglob` shell option is set, and no matches are found, an error message is printed and the command is not executed. If the shell option `nocaseglob` is enabled, the match is performed without regard to the case of alphabetic characters.

When a pattern is used for filename expansion, the character '`.`' at the start of a filename or immediately following a slash must be matched explicitly, unless the shell option `dotglob` is set. The filenames '`.`' and '`..`' must always be matched explicitly, even if `dotglob` is set. In other cases, the '`.`' character is not treated specially.

When matching a filename, the slash character must always be matched explicitly by a slash in the pattern, but in other matching contexts it can be matched by a special pattern character as described below (see Section 3.5.8.1 [Pattern Matching], page 33).

See the description of `shopt` in Section 4.3.2 [The Shopt Builtin], page 65, for a description of the `nocaseglob`, `nullglob`, `failglob`, and `dotglob` options.

The `GLOBIGNORE` shell variable may be used to restrict the set of file names matching a pattern. If `GLOBIGNORE` is set, each matching file name that also matches one of the patterns in `GLOBIGNORE` is removed from the list of matches. If the `nocaseglob` option is set, the matching against the patterns in `GLOBIGNORE` is performed without regard to case. The filenames `.` and `..` are always ignored when `GLOBIGNORE` is set and not null. However, setting `GLOBIGNORE` to a non-null value has the effect of enabling the `dotglob` shell option, so all other filenames beginning with a '`.`' will match. To get the old behavior of ignoring filenames beginning with a '`.`', make '`.*`' one of the patterns in `GLOBIGNORE`. The `dotglob` option is disabled when `GLOBIGNORE` is unset.

3.5.8.1 Pattern Matching

Any character that appears in a pattern, other than the special pattern characters described below, matches itself. The NUL character may not occur in a pattern. A backslash escapes the following character; the escaping backslash is discarded when matching. The special pattern characters must be quoted if they are to be matched literally.

The special pattern characters have the following meanings:

`*` Matches any string, including the null string. When the `globstar` shell option is enabled, and '`*`' is used in a filename expansion context, two adjacent '`*`'s used as a single pattern will match all files and zero or more directories and subdirectories. If followed by a '`/`', two adjacent '`*`'s will match only directories and subdirectories.

`?` Matches any single character.

`[...]` Matches any one of the enclosed characters. A pair of characters separated by a hyphen denotes a *range expression*; any character that falls between those two characters, inclusive, using the current locale's collating sequence and character set, is matched. If the first character following the '`[`' is a '`!`' or a '`^`' then any character not enclosed is matched. A '−' may be matched by including it as the first or last character in the set. A '`]`' may be matched by including it as the first character in the set. The sorting order of characters in range expressions is determined by the current locale and the values of the `LC_COLLATE` and `LC_ALL` shell variables, if set.

For example, in the default C locale, '`[a-dx-z]`' is equivalent to '`[abcdxyz]`'. Many locales sort characters in dictionary order, and in these locales '`[a-dx-z]`' is typically not equivalent to '`[abcdxyz]`'; it might be equivalent to '`[aBbCcDdxXyYz]`', for example. To obtain the traditional interpretation of ranges in bracket expressions, you can force the use of the C locale by setting the `LC_COLLATE` or `LC_ALL` environment variable to the value '`C`', or enable the `globasciiranges` shell option.

Within '`[`' and '`]`', *character classes* can be specified using the syntax `[:`*class*`:]`, where *class* is one of the following classes defined in the POSIX standard:

```
alnum   alpha   ascii   blank   cntrl   digit   graph   lower
print   punct   space   upper   word    xdigit
```

A character class matches any character belonging to that class. The `word` character class matches letters, digits, and the character ‘_’.

Within ‘[’ and ‘]’, an *equivalence class* can be specified using the syntax `[=`*c*`=]`, which matches all characters with the same collation weight (as defined by the current locale) as the character *c*.

Within ‘[’ and ‘]’, the syntax `[.`*symbol*`.]` matches the collating symbol *symbol*.

If the `extglob` shell option is enabled using the `shopt` builtin, several extended pattern matching operators are recognized. In the following description, a *pattern-list* is a list of one or more patterns separated by a ‘|’. Composite patterns may be formed using one or more of the following sub-patterns:

`?(`*pattern-list*`)`
Matches zero or one occurrence of the given patterns.

`*(`*pattern-list*`)`
Matches zero or more occurrences of the given patterns.

`+(`*pattern-list*`)`
Matches one or more occurrences of the given patterns.

`@(`*pattern-list*`)`
Matches one of the given patterns.

`!(`*pattern-list*`)`
Matches anything except one of the given patterns.

Complicated extended pattern matching against long strings is slow, especially when the patterns contain alternations and the strings contain multiple matches. Using separate matches against shorter strings, or using arrays of strings instead of a single long string, may be faster.

3.5.9 Quote Removal

After the preceding expansions, all unquoted occurrences of the characters ‘\’, ‘'’, and ‘"’ that did not result from one of the above expansions are removed.

3.6 Redirections

Before a command is executed, its input and output may be *redirected* using a special notation interpreted by the shell. Redirection allows commands’ file handles to be duplicated, opened, closed, made to refer to different files, and can change the files the command reads from and writes to. Redirection may also be used to modify file handles in the current shell execution environment. The following redirection operators may precede or appear anywhere within a simple command or may follow a command. Redirections are processed in the order they appear, from left to right.

Each redirection that may be preceded by a file descriptor number may instead be preceded by a word of the form {*varname*}. In this case, for each redirection operator except >&- and <&-, the shell will allocate a file descriptor greater than 10 and assign it to {*varname*}. If >&- or <&- is preceded by {*varname*}, the value of *varname* defines the file descriptor to close. If {*varname*} is supplied, the redirection persists beyond the scope of the command, allowing the shell programmer to manage the file descriptor himself.

In the following descriptions, if the file descriptor number is omitted, and the first character of the redirection operator is '<', the redirection refers to the standard input (file descriptor 0). If the first character of the redirection operator is '>', the redirection refers to the standard output (file descriptor 1).

The word following the redirection operator in the following descriptions, unless otherwise noted, is subjected to brace expansion, tilde expansion, parameter expansion, command substitution, arithmetic expansion, quote removal, filename expansion, and word splitting. If it expands to more than one word, Bash reports an error.

Note that the order of redirections is significant. For example, the command

```
ls > dirlist 2>&1
```

directs both standard output (file descriptor 1) and standard error (file descriptor 2) to the file *dirlist*, while the command

```
ls 2>&1 > dirlist
```

directs only the standard output to file *dirlist*, because the standard error was made a copy of the standard output before the standard output was redirected to *dirlist*.

Bash handles several filenames specially when they are used in redirections, as described in the following table. If the operating system on which Bash is running provides these special files, bash will use them; otherwise it will emulate them internally with the behavior described below.

`/dev/fd/fd`
: If *fd* is a valid integer, file descriptor *fd* is duplicated.

`/dev/stdin`
: File descriptor 0 is duplicated.

`/dev/stdout`
: File descriptor 1 is duplicated.

`/dev/stderr`
: File descriptor 2 is duplicated.

`/dev/tcp/host/port`
: If *host* is a valid hostname or Internet address, and *port* is an integer port number or service name, Bash attempts to open the corresponding TCP socket.

`/dev/udp/host/port`
: If *host* is a valid hostname or Internet address, and *port* is an integer port number or service name, Bash attempts to open the corresponding UDP socket.

A failure to open or create a file causes the redirection to fail.

Redirections using file descriptors greater than 9 should be used with care, as they may conflict with file descriptors the shell uses internally.

3.6.1 Redirecting Input

Redirection of input causes the file whose name results from the expansion of *word* to be opened for reading on file descriptor `n`, or the standard input (file descriptor 0) if `n` is not specified.

The general format for redirecting input is:

```
[n]<word
```

3.6.2 Redirecting Output

Redirection of output causes the file whose name results from the expansion of *word* to be opened for writing on file descriptor *n*, or the standard output (file descriptor 1) if *n* is not specified. If the file does not exist it is created; if it does exist it is truncated to zero size.

The general format for redirecting output is:

```
[n]>[|]word
```

If the redirection operator is '`>`', and the `noclobber` option to the `set` builtin has been enabled, the redirection will fail if the file whose name results from the expansion of *word* exists and is a regular file. If the redirection operator is '`>|`', or the redirection operator is '`>`' and the `noclobber` option is not enabled, the redirection is attempted even if the file named by *word* exists.

3.6.3 Appending Redirected Output

Redirection of output in this fashion causes the file whose name results from the expansion of *word* to be opened for appending on file descriptor *n*, or the standard output (file descriptor 1) if *n* is not specified. If the file does not exist it is created.

The general format for appending output is:

```
[n]>>word
```

3.6.4 Redirecting Standard Output and Standard Error

This construct allows both the standard output (file descriptor 1) and the standard error output (file descriptor 2) to be redirected to the file whose name is the expansion of *word*.

There are two formats for redirecting standard output and standard error:

```
&>word
```

and

```
>&word
```

Of the two forms, the first is preferred. This is semantically equivalent to

```
>word 2>&1
```

When using the second form, *word* may not expand to a number or '`-`'. If it does, other redirection operators apply (see Duplicating File Descriptors below) for compatibility reasons.

3.6.5 Appending Standard Output and Standard Error

This construct allows both the standard output (file descriptor 1) and the standard error output (file descriptor 2) to be appended to the file whose name is the expansion of *word*.

The format for appending standard output and standard error is:

```
&>>word
```

This is semantically equivalent to

```
>>word 2>&1
```

(see Duplicating File Descriptors below).

3.6.6 Here Documents

This type of redirection instructs the shell to read input from the current source until a line containing only *word* (with no trailing blanks) is seen. All of the lines read up to that point are then used as the standard input (or file descriptor *n* if *n* is specified) for a command.

The format of here-documents is:

```
[n]<<[−]word
        here-document
delimiter
```

No parameter and variable expansion, command substitution, arithmetic expansion, or filename expansion is performed on *word*. If any part of *word* is quoted, the *delimiter* is the result of quote removal on *word*, and the lines in the here-document are not expanded. If *word* is unquoted, all lines of the here-document are subjected to parameter expansion, command substitution, and arithmetic expansion, the character sequence `\newline` is ignored, and ‘\’ must be used to quote the characters ‘\’, ‘$’, and ‘`’.

If the redirection operator is ‘`<<-`’, then all leading tab characters are stripped from input lines and the line containing *delimiter*. This allows here-documents within shell scripts to be indented in a natural fashion.

3.6.7 Here Strings

A variant of here documents, the format is:

```
[n]<<< word
```

The *word* undergoes tilde expansion, parameter and variable expansion, command substitution, arithmetic expansion, and quote removal. Pathname expansion and word splitting are not performed. The result is supplied as a single string, with a newline appended, to the command on its standard input (or file descriptor *n* if *n* is specified).

3.6.8 Duplicating File Descriptors

The redirection operator

```
[n]<&word
```

is used to duplicate input file descriptors. If *word* expands to one or more digits, the file descriptor denoted by *n* is made to be a copy of that file descriptor. If the digits in *word* do not specify a file descriptor open for input, a redirection error occurs. If *word* evaluates to ‘-’, file descriptor *n* is closed. If *n* is not specified, the standard input (file descriptor 0) is used.

The operator

```
[n]>&word
```

is used similarly to duplicate output file descriptors. If *n* is not specified, the standard output (file descriptor 1) is used. If the digits in *word* do not specify a file descriptor open for output, a redirection error occurs. If *word* evaluates to ‘-’, file descriptor *n* is closed. As a special case, if *n* is omitted, and *word* does not expand to one or more digits or ‘-’, the standard output and standard error are redirected as described previously.

3.6.9 Moving File Descriptors

The redirection operator

    [n]<&digit-

moves the file descriptor *digit* to file descriptor *n*, or the standard input (file descriptor 0) if *n* is not specified. *digit* is closed after being duplicated to *n*.

Similarly, the redirection operator

    [n]>&digit-

moves the file descriptor *digit* to file descriptor *n*, or the standard output (file descriptor 1) if *n* is not specified.

3.6.10 Opening File Descriptors for Reading and Writing

The redirection operator

    [n]<>word

causes the file whose name is the expansion of *word* to be opened for both reading and writing on file descriptor *n*, or on file descriptor 0 if *n* is not specified. If the file does not exist, it is created.

3.7 Executing Commands

3.7.1 Simple Command Expansion

When a simple command is executed, the shell performs the following expansions, assignments, and redirections, from left to right.

1. The words that the parser has marked as variable assignments (those preceding the command name) and redirections are saved for later processing.
2. The words that are not variable assignments or redirections are expanded (see Section 3.5 [Shell Expansions], page 22). If any words remain after expansion, the first word is taken to be the name of the command and the remaining words are the arguments.
3. Redirections are performed as described above (see Section 3.6 [Redirections], page 34).
4. The text after the '`=`' in each variable assignment undergoes tilde expansion, parameter expansion, command substitution, arithmetic expansion, and quote removal before being assigned to the variable.

If no command name results, the variable assignments affect the current shell environment. Otherwise, the variables are added to the environment of the executed command and do not affect the current shell environment. If any of the assignments attempts to assign a value to a readonly variable, an error occurs, and the command exits with a non-zero status.

If no command name results, redirections are performed, but do not affect the current shell environment. A redirection error causes the command to exit with a non-zero status.

If there is a command name left after expansion, execution proceeds as described below. Otherwise, the command exits. If one of the expansions contained a command substitution, the exit status of the command is the exit status of the last command substitution performed. If there were no command substitutions, the command exits with a status of zero.

3.7.2 Command Search and Execution

After a command has been split into words, if it results in a simple command and an optional list of arguments, the following actions are taken.

1. If the command name contains no slashes, the shell attempts to locate it. If there exists a shell function by that name, that function is invoked as described in Section 3.3 [Shell Functions], page 17.
2. If the name does not match a function, the shell searches for it in the list of shell builtins. If a match is found, that builtin is invoked.
3. If the name is neither a shell function nor a builtin, and contains no slashes, Bash searches each element of `$PATH` for a directory containing an executable file by that name. Bash uses a hash table to remember the full pathnames of executable files to avoid multiple `PATH` searches (see the description of `hash` in Section 4.1 [Bourne Shell Builtins], page 43). A full search of the directories in `$PATH` is performed only if the command is not found in the hash table. If the search is unsuccessful, the shell searches for a defined shell function named `command_not_found_handle`. If that function exists, it is invoked in a separate execution environment with the original command and the original command's arguments as its arguments, and the function's exit status becomes the exit status of that subshell. If that function is not defined, the shell prints an error message and returns an exit status of 127.
4. If the search is successful, or if the command name contains one or more slashes, the shell executes the named program in a separate execution environment. Argument 0 is set to the name given, and the remaining arguments to the command are set to the arguments supplied, if any.
5. If this execution fails because the file is not in executable format, and the file is not a directory, it is assumed to be a *shell script* and the shell executes it as described in Section 3.8 [Shell Scripts], page 42.
6. If the command was not begun asynchronously, the shell waits for the command to complete and collects its exit status.

3.7.3 Command Execution Environment

The shell has an *execution environment*, which consists of the following:

- open files inherited by the shell at invocation, as modified by redirections supplied to the `exec` builtin
- the current working directory as set by `cd`, `pushd`, or `popd`, or inherited by the shell at invocation
- the file creation mode mask as set by `umask` or inherited from the shell's parent
- current traps set by `trap`
- shell parameters that are set by variable assignment or with `set` or inherited from the shell's parent in the environment
- shell functions defined during execution or inherited from the shell's parent in the environment
- options enabled at invocation (either by default or with command-line arguments) or by `set`

- options enabled by `shopt` (see Section 4.3.2 [The Shopt Builtin], page 65)
- shell aliases defined with `alias` (see Section 6.6 [Aliases], page 93)
- various process IDs, including those of background jobs (see Section 3.2.3 [Lists], page 9), the value of `$$`, and the value of `$PPID`

When a simple command other than a builtin or shell function is to be executed, it is invoked in a separate execution environment that consists of the following. Unless otherwise noted, the values are inherited from the shell.

- the shell's open files, plus any modifications and additions specified by redirections to the command
- the current working directory
- the file creation mode mask
- shell variables and functions marked for export, along with variables exported for the command, passed in the environment (see Section 3.7.4 [Environment], page 40)
- traps caught by the shell are reset to the values inherited from the shell's parent, and traps ignored by the shell are ignored

A command invoked in this separate environment cannot affect the shell's execution environment.

Command substitution, commands grouped with parentheses, and asynchronous commands are invoked in a subshell environment that is a duplicate of the shell environment, except that traps caught by the shell are reset to the values that the shell inherited from its parent at invocation. Builtin commands that are invoked as part of a pipeline are also executed in a subshell environment. Changes made to the subshell environment cannot affect the shell's execution environment.

Subshells spawned to execute command substitutions inherit the value of the `-e` option from the parent shell. When not in POSIX mode, Bash clears the `-e` option in such subshells.

If a command is followed by a '`&`' and job control is not active, the default standard input for the command is the empty file `/dev/null`. Otherwise, the invoked command inherits the file descriptors of the calling shell as modified by redirections.

3.7.4 Environment

When a program is invoked it is given an array of strings called the *environment*. This is a list of name-value pairs, of the form `name=value`.

Bash provides several ways to manipulate the environment. On invocation, the shell scans its own environment and creates a parameter for each name found, automatically marking it for *export* to child processes. Executed commands inherit the environment. The `export` and '`declare -x`' commands allow parameters and functions to be added to and deleted from the environment. If the value of a parameter in the environment is modified, the new value becomes part of the environment, replacing the old. The environment inherited by any executed command consists of the shell's initial environment, whose values may be modified in the shell, less any pairs removed by the `unset` and '`export -n`' commands, plus any additions via the `export` and '`declare -x`' commands.

The environment for any simple command or function may be augmented temporarily by prefixing it with parameter assignments, as described in Section 3.4 [Shell Parameters], page 20. These assignment statements affect only the environment seen by that command.

If the `-k` option is set (see Section 4.3.1 [The Set Builtin], page 61), then all parameter assignments are placed in the environment for a command, not just those that precede the command name.

When Bash invokes an external command, the variable '`$_`' is set to the full pathname of the command and passed to that command in its environment.

3.7.5 Exit Status

The exit status of an executed command is the value returned by the *waitpid* system call or equivalent function. Exit statuses fall between 0 and 255, though, as explained below, the shell may use values above 125 specially. Exit statuses from shell builtins and compound commands are also limited to this range. Under certain circumstances, the shell will use special values to indicate specific failure modes.

For the shell's purposes, a command which exits with a zero exit status has succeeded. A non-zero exit status indicates failure. This seemingly counter-intuitive scheme is used so there is one well-defined way to indicate success and a variety of ways to indicate various failure modes. When a command terminates on a fatal signal whose number is *N*, Bash uses the value 128+*N* as the exit status.

If a command is not found, the child process created to execute it returns a status of 127. If a command is found but is not executable, the return status is 126.

If a command fails because of an error during expansion or redirection, the exit status is greater than zero.

The exit status is used by the Bash conditional commands (see Section 3.2.4.2 [Conditional Constructs], page 11) and some of the list constructs (see Section 3.2.3 [Lists], page 9).

All of the Bash builtins return an exit status of zero if they succeed and a non-zero status on failure, so they may be used by the conditional and list constructs. All builtins return an exit status of 2 to indicate incorrect usage, generally invalid options or missing arguments.

3.7.6 Signals

When Bash is interactive, in the absence of any traps, it ignores `SIGTERM` (so that '`kill 0`' does not kill an interactive shell), and `SIGINT` is caught and handled (so that the `wait` builtin is interruptible). When Bash receives a `SIGINT`, it breaks out of any executing loops. In all cases, Bash ignores `SIGQUIT`. If job control is in effect (see Chapter 7 [Job Control], page 104), Bash ignores `SIGTTIN`, `SIGTTOU`, and `SIGTSTP`.

Non-builtin commands started by Bash have signal handlers set to the values inherited by the shell from its parent. When job control is not in effect, asynchronous commands ignore `SIGINT` and `SIGQUIT` in addition to these inherited handlers. Commands run as a result of command substitution ignore the keyboard-generated job control signals `SIGTTIN`, `SIGTTOU`, and `SIGTSTP`.

The shell exits by default upon receipt of a `SIGHUP`. Before exiting, an interactive shell resends the `SIGHUP` to all jobs, running or stopped. Stopped jobs are sent `SIGCONT` to ensure that they receive the `SIGHUP`. To prevent the shell from sending the `SIGHUP` signal to a particular job, it should be removed from the jobs table with the `disown` builtin (see

Section 7.2 [Job Control Builtins], page 105) or marked to not receive `SIGHUP` using `disown -h`.

If the `huponexit` shell option has been set with `shopt` (see Section 4.3.2 [The Shopt Builtin], page 65), Bash sends a `SIGHUP` to all jobs when an interactive login shell exits.

If Bash is waiting for a command to complete and receives a signal for which a trap has been set, the trap will not be executed until the command completes. When Bash is waiting for an asynchronous command via the `wait` builtin, the reception of a signal for which a trap has been set will cause the `wait` builtin to return immediately with an exit status greater than 128, immediately after which the trap is executed.

3.8 Shell Scripts

A shell script is a text file containing shell commands. When such a file is used as the first non-option argument when invoking Bash, and neither the `-c` nor `-s` option is supplied (see Section 6.1 [Invoking Bash], page 85), Bash reads and executes commands from the file, then exits. This mode of operation creates a non-interactive shell. The shell first searches for the file in the current directory, and looks in the directories in `$PATH` if not found there.

When Bash runs a shell script, it sets the special parameter `0` to the name of the file, rather than the name of the shell, and the positional parameters are set to the remaining arguments, if any are given. If no additional arguments are supplied, the positional parameters are unset.

A shell script may be made executable by using the `chmod` command to turn on the execute bit. When Bash finds such a file while searching the `$PATH` for a command, it spawns a subshell to execute it. In other words, executing

```
filename arguments
```

is equivalent to executing

```
bash filename arguments
```

if `filename` is an executable shell script. This subshell reinitializes itself, so that the effect is as if a new shell had been invoked to interpret the script, with the exception that the locations of commands remembered by the parent (see the description of `hash` in Section 4.1 [Bourne Shell Builtins], page 43) are retained by the child.

Most versions of Unix make this a part of the operating system's command execution mechanism. If the first line of a script begins with the two characters '`#!`', the remainder of the line specifies an interpreter for the program. Thus, you can specify Bash, `awk`, Perl, or some other interpreter and write the rest of the script file in that language.

The arguments to the interpreter consist of a single optional argument following the interpreter name on the first line of the script file, followed by the name of the script file, followed by the rest of the arguments. Bash will perform this action on operating systems that do not handle it themselves. Note that some older versions of Unix limit the interpreter name and argument to a maximum of 32 characters.

Bash scripts often begin with `#! /bin/bash` (assuming that Bash has been installed in `/bin`), since this ensures that Bash will be used to interpret the script, even if it is executed under another shell.

4 Shell Builtin Commands

Builtin commands are contained within the shell itself. When the name of a builtin command is used as the first word of a simple command (see Section 3.2.1 [Simple Commands], page 8), the shell executes the command directly, without invoking another program. Builtin commands are necessary to implement functionality impossible or inconvenient to obtain with separate utilities.

This section briefly describes the builtins which Bash inherits from the Bourne Shell, as well as the builtin commands which are unique to or have been extended in Bash.

Several builtin commands are described in other chapters: builtin commands which provide the Bash interface to the job control facilities (see Section 7.2 [Job Control Builtins], page 105), the directory stack (see Section 6.8.1 [Directory Stack Builtins], page 96), the command history (see Section 9.2 [Bash History Builtins], page 143), and the programmable completion facilities (see Section 8.7 [Programmable Completion Builtins], page 136).

Many of the builtins have been extended by POSIX or Bash.

Unless otherwise noted, each builtin command documented as accepting options preceded by '`-`' accepts '`--`' to signify the end of the options. The `:`, `true`, `false`, and `test`/`[` builtins do not accept options and do not treat '`--`' specially. The `exit`, `logout`, `return`, `break`, `continue`, `let`, and `shift` builtins accept and process arguments beginning with '`-`' without requiring '`--`'. Other builtins that accept arguments but are not specified as accepting options interpret arguments beginning with '`-`' as invalid options and require '`--`' to prevent this interpretation.

4.1 Bourne Shell Builtins

The following shell builtin commands are inherited from the Bourne Shell. These commands are implemented as specified by the POSIX standard.

`:` (a colon)

```
: [arguments]
```

Do nothing beyond expanding *arguments* and performing redirections. The return status is zero.

`.` (a period)

```
. filename [arguments]
```

Read and execute commands from the *filename* argument in the current shell context. If *filename* does not contain a slash, the `PATH` variable is used to find *filename*. When Bash is not in POSIX mode, the current directory is searched if *filename* is not found in `$PATH`. If any *arguments* are supplied, they become the positional parameters when *filename* is executed. Otherwise the positional parameters are unchanged. If the `-T` option is enabled, `source` inherits any trap on `DEBUG`; if it is not, any `DEBUG` trap string is saved and restored around the call to `source`, and `source` unsets the `DEBUG` trap while it executes. If `-T` is not set, and the sourced file changes the `DEBUG` trap, the new value is retained when `source` completes. The return status is the exit status of the last command executed, or zero if no commands are executed. If *filename* is not found, or cannot be read, the return status is non-zero. This builtin is equivalent to `source`.

`break`

```
break [n]
```

Exit from a `for`, `while`, `until`, or `select` loop. If *n* is supplied, the *n*th enclosing loop is exited. *n* must be greater than or equal to 1. The return status is zero unless *n* is not greater than or equal to 1.

`cd`

```
cd [-L|[-P [-e]] [-@] [directory]
```

Change the current working directory to *directory*. If *directory* is not supplied, the value of the `HOME` shell variable is used. Any additional arguments following *directory* are ignored. If the shell variable `CDPATH` exists, it is used as a search path: each directory name in `CDPATH` is searched for *directory*, with alternative directory names in `CDPATH` separated by a colon (':'). If *directory* begins with a slash, `CDPATH` is not used.

The `-P` option means to not follow symbolic links: symbolic links are resolved while `cd` is traversing *directory* and before processing an instance of '`..`' in *directory*.

By default, or when the `-L` option is supplied, symbolic links in *directory* are resolved after `cd` processes an instance of '`..`' in *directory*.

If '`..`' appears in *directory*, it is processed by removing the immediately preceding pathname component, back to a slash or the beginning of *directory*.

If the `-e` option is supplied with `-P` and the current working directory cannot be successfully determined after a successful directory change, `cd` will return an unsuccessful status.

On systems that support it, the `-@` option presents the extended attributes associated with a file as a directory.

If *directory* is '`-`', it is converted to `$OLDPWD` before the directory change is attempted.

If a non-empty directory name from `CDPATH` is used, or if '`-`' is the first argument, and the directory change is successful, the absolute pathname of the new working directory is written to the standard output.

The return status is zero if the directory is successfully changed, non-zero otherwise.

`continue`

```
continue [n]
```

Resume the next iteration of an enclosing `for`, `while`, `until`, or `select` loop. If *n* is supplied, the execution of the *n*th enclosing loop is resumed. *n* must be greater than or equal to 1. The return status is zero unless *n* is not greater than or equal to 1.

`eval`

```
eval [arguments]
```

The arguments are concatenated together into a single command, which is then read and executed, and its exit status returned as the exit status of `eval`. If there are no arguments or only empty arguments, the return status is zero.

`exec`

```
exec [-cl] [-a name] [command [arguments]]
```

If *command* is supplied, it replaces the shell without creating a new process. If the `-l` option is supplied, the shell places a dash at the beginning of the zeroth argument passed to *command*. This is what the `login` program does. The `-c` option causes *command* to be executed with an empty environment. If `-a` is supplied, the shell passes *name* as the zeroth argument to *command*. If *command* cannot be executed for some reason, a non-interactive shell exits, unless the `execfail` shell option is enabled. In that case, it returns failure. An interactive shell returns failure if the file cannot be executed. A subshell exits unconditionally if `exec` fails. If no *command* is specified, redirections may be used to affect the current shell environment. If there are no redirection errors, the return status is zero; otherwise the return status is non-zero.

`exit`

```
exit [n]
```

Exit the shell, returning a status of *n* to the shell's parent. If *n* is omitted, the exit status is that of the last command executed. Any trap on `EXIT` is executed before the shell terminates.

`export`

```
export [-fn] [-p] [name[=value]]
```

Mark each *name* to be passed to child processes in the environment. If the `-f` option is supplied, the *name*s refer to shell functions; otherwise the names refer to shell variables. The `-n` option means to no longer mark each *name* for export. If no *names* are supplied, or if the `-p` option is given, a list of names of all exported variables is displayed. The `-p` option displays output in a form that may be reused as input. If a variable name is followed by =*value*, the value of the variable is set to *value*.

The return status is zero unless an invalid option is supplied, one of the names is not a valid shell variable name, or `-f` is supplied with a name that is not a shell function.

`getopts`

```
getopts optstring name [args]
```

`getopts` is used by shell scripts to parse positional parameters. *optstring* contains the option characters to be recognized; if a character is followed by a colon, the option is expected to have an argument, which should be separated from it by whitespace. The colon (‘`:`’) and question mark (‘`?`’) may not be used as option characters. Each time it is invoked, `getopts` places the next option in the shell variable *name*, initializing *name* if it does not exist, and the index of the next argument to be processed into the variable `OPTIND`. `OPTIND` is initialized to 1 each time the shell or a shell script is invoked. When an option requires an argument, `getopts` places that argument into the variable `OPTARG`. The shell does not reset `OPTIND` automatically; it must be manually reset between multiple calls to `getopts` within the same shell invocation if a new set of parameters is to be used.

When the end of options is encountered, `getopts` exits with a return value greater than zero. `OPTIND` is set to the index of the first non-option argument, and *name* is set to '`?`'.

`getopts` normally parses the positional parameters, but if more arguments are given in *args*, `getopts` parses those instead.

`getopts` can report errors in two ways. If the first character of *optstring* is a colon, *silent* error reporting is used. In normal operation, diagnostic messages are printed when invalid options or missing option arguments are encountered. If the variable `OPTERR` is set to 0, no error messages will be displayed, even if the first character of `optstring` is not a colon.

If an invalid option is seen, `getopts` places '`?`' into *name* and, if not silent, prints an error message and unsets `OPTARG`. If `getopts` is silent, the option character found is placed in `OPTARG` and no diagnostic message is printed.

If a required argument is not found, and `getopts` is not silent, a question mark ('`?`') is placed in *name*, `OPTARG` is unset, and a diagnostic message is printed. If `getopts` is silent, then a colon ('`:`') is placed in *name* and `OPTARG` is set to the option character found.

`hash`

```
hash [-r] [-p filename] [-dt] [name]
```

Each time `hash` is invoked, it remembers the full pathnames of the commands specified as *name* arguments, so they need not be searched for on subsequent invocations. The commands are found by searching through the directories listed in `$PATH`. Any previously-remembered pathname is discarded. The `-p` option inhibits the path search, and *filename* is used as the location of *name*. The `-r` option causes the shell to forget all remembered locations. The `-d` option causes the shell to forget the remembered location of each *name*. If the `-t` option is supplied, the full pathname to which each *name* corresponds is printed. If multiple *name* arguments are supplied with `-t`, the *name* is printed before the hashed full pathname. The `-l` option causes output to be displayed in a format that may be reused as input. If no arguments are given, or if only `-l` is supplied, information about remembered commands is printed. The return status is zero unless a *name* is not found or an invalid option is supplied.

`pwd`

```
pwd [-LP]
```

Print the absolute pathname of the current working directory. If the `-P` option is supplied, the pathname printed will not contain symbolic links. If the `-L` option is supplied, the pathname printed may contain symbolic links. The return status is zero unless an error is encountered while determining the name of the current directory or an invalid option is supplied.

`readonly`

```
readonly [-aAf] [-p] [name[=value]] ...
```

Mark each *name* as readonly. The values of these names may not be changed by subsequent assignment. If the `-f` option is supplied, each *name* refers to a shell function. The `-a` option means each *name* refers to an indexed array variable; the `-A` option means each *name* refers to an associative array variable. If both options are supplied, `-A` takes precedence. If no *name* arguments are given, or if the `-p` option is supplied, a list of all readonly names is printed. The other options may be used to restrict the output to a subset of the set of readonly names. The `-p` option causes output to be displayed in a format that may be reused as input. If a variable name is followed by =*value*, the value of the variable is set to *value*. The return status is zero unless an invalid option is supplied, one of the *name* arguments is not a valid shell variable or function name, or the `-f` option is supplied with a name that is not a shell function.

`return`

```
return [n]
```

Cause a shell function to stop executing and return the value *n* to its caller. If *n* is not supplied, the return value is the exit status of the last command executed in the function. If `return` is executed by a trap handler, the last command used to determine the status is the last command executed before the trap handler. If `return` is executed during a `DEBUG` trap, the last command used to determine the status is the last command executed by the trap handler before `return` was invoked. `return` may also be used to terminate execution of a script being executed with the `.` (`source`) builtin, returning either *n* or the exit status of the last command executed within the script as the exit status of the script. If *n* is supplied, the return value is its least significant 8 bits. Any command associated with the `RETURN` trap is executed before execution resumes after the function or script. The return status is non-zero if `return` is supplied a non-numeric argument or is used outside a function and not during the execution of a script by `.` or `source`.

`shift`

```
shift [n]
```

Shift the positional parameters to the left by *n*. The positional parameters from *n*+1 . . . `$#` are renamed to `$1` . . . `$#`-*n*. Parameters represented by the numbers `$#` to `$#`-*n*+1 are unset. *n* must be a non-negative number less than or equal to `$#`. If *n* is zero or greater than `$#`, the positional parameters are not changed. If *n* is not supplied, it is assumed to be 1. The return status is zero unless *n* is greater than `$#` or less than zero, non-zero otherwise.

`test`
`[`

```
test expr
```

Evaluate a conditional expression *expr* and return a status of 0 (true) or 1 (false). Each operator and operand must be a separate argument. Expressions are composed of the primaries described below in Section 6.4 [Bash Conditional Expressions], page 90. `test` does not accept any options, nor does it accept and ignore an argument of `--` as signifying the end of options.

When the `[` form is used, the last argument to the command must be a `]`.

Expressions may be combined using the following operators, listed in decreasing order of precedence. The evaluation depends on the number of arguments; see below. Operator precedence is used when there are five or more arguments.

`! expr`
: True if *expr* is false.

`( expr )`
: Returns the value of *expr*. This may be used to override the normal precedence of operators.

`expr1 -a expr2`
: True if both *expr1* and *expr2* are true.

`expr1 -o expr2`
: True if either *expr1* or *expr2* is true.

The `test` and `[` builtins evaluate conditional expressions using a set of rules based on the number of arguments.

0 arguments
: The expression is false.

1 argument
: The expression is true if, and only if, the argument is not null.

2 arguments
: If the first argument is '`!`', the expression is true if and only if the second argument is null. If the first argument is one of the unary conditional operators (see Section 6.4 [Bash Conditional Expressions], page 90), the expression is true if the unary test is true. If the first argument is not a valid unary operator, the expression is false.

3 arguments
: The following conditions are applied in the order listed.

1. If the second argument is one of the binary conditional operators (see Section 6.4 [Bash Conditional Expressions], page 90), the result of the expression is the result of the binary test using the first and third arguments as operands. The '`-a`' and '`-o`' operators are considered binary operators when there are three arguments.
2. If the first argument is '`!`', the value is the negation of the two-argument test using the second and third arguments.
3. If the first argument is exactly '`(`' and the third argument is exactly '`)`', the result is the one-argument test of the second argument.
4. Otherwise, the expression is false.

4 arguments
: If the first argument is '`!`', the result is the negation of the three-argument expression composed of the remaining arguments. Otherwise, the expression is parsed and evaluated according to precedence using the rules listed above.

5 or more arguments
: The expression is parsed and evaluated according to precedence using the rules listed above.

When used with `test` or '`[`', the '`<`' and '`>`' operators sort lexicographically using ASCII ordering.

`times`

```
times
```

Print out the user and system times used by the shell and its children. The return status is zero.

`trap`

```
trap [-lp] [arg] [sigspec ...]
```

The commands in *arg* are to be read and executed when the shell receives signal *sigspec*. If *arg* is absent (and there is a single *sigspec*) or equal to '`-`', each specified signal's disposition is reset to the value it had when the shell was started. If *arg* is the null string, then the signal specified by each *sigspec* is ignored by the shell and commands it invokes. If *arg* is not present and `-p` has been supplied, the shell displays the trap commands associated with each *sigspec*. If no arguments are supplied, or only `-p` is given, `trap` prints the list of commands associated with each signal number in a form that may be reused as shell input. The `-l` option causes the shell to print a list of signal names and their corresponding numbers. Each *sigspec* is either a signal name or a signal number. Signal names are case insensitive and the `SIG` prefix is optional.

If a *sigspec* is `0` or `EXIT`, *arg* is executed when the shell exits. If a *sigspec* is `DEBUG`, the command *arg* is executed before every simple command, `for` command, `case` command, `select` command, every arithmetic `for` command, and before the first command executes in a shell function. Refer to the description of the `extdebug` option to the `shopt` builtin (see Section 4.3.2 [The Shopt Builtin], page 65) for details of its effect on the `DEBUG` trap. If a *sigspec* is `RETURN`, the command *arg* is executed each time a shell function or a script executed with the `.` or `source` builtins finishes executing.

If a *sigspec* is `ERR`, the command *arg* is executed whenever a pipeline (which may consist of a single simple command), a list, or a compound command returns a non-zero exit status, subject to the following conditions. The `ERR` trap is not executed if the failed command is part of the command list immediately following an `until` or `while` keyword, part of the test following the `if` or `elif` reserved words, part of a command executed in a `&&` or `||` list except the command following the final `&&` or `||`, any command in a pipeline but the last, or if the command's return status is being inverted using `!`. These are the same conditions obeyed by the `errexit` (`-e`) option.

Signals ignored upon entry to the shell cannot be trapped or reset. Trapped signals that are not being ignored are reset to their original values in a subshell or subshell environment when one is created.

The return status is zero unless a *sigspec* does not specify a valid signal.

`umask`

```
umask [-p] [-S] [mode]
```

Set the shell process's file creation mask to *mode*. If *mode* begins with a digit, it is interpreted as an octal number; if not, it is interpreted as a symbolic mode mask similar to that accepted by the `chmod` command. If *mode* is omitted, the current value of the mask is printed. If the `-S` option is supplied without a *mode* argument, the mask is printed in a symbolic format. If the `-p` option is supplied, and *mode* is omitted, the output is in a form that may be reused as input. The return status is zero if the mode is successfully changed or if no *mode* argument is supplied, and non-zero otherwise.

Note that when the mode is interpreted as an octal number, each number of the umask is subtracted from `7`. Thus, a umask of `022` results in permissions of `755`.

`unset`

```
unset [-fnv] [name]
```

Remove each variable or function *name*. If the `-v` option is given, each *name* refers to a shell variable and that variable is removed. If the `-f` option is given, the *name*s refer to shell functions, and the function definition is removed. If the `-n` option is supplied, and *name* is a variable with the *nameref* attribute, *name* will be unset rather than the variable it references. `-n` has no effect if the `-f` option is supplied. If no options are supplied, each *name* refers to a variable; if there is no variable by that name, any function with that name is unset. Readonly variables and functions may not be unset. The return status is zero unless a *name* is readonly.

4.2 Bash Builtin Commands

This section describes builtin commands which are unique to or have been extended in Bash. Some of these commands are specified in the POSIX standard.

`alias`

```
alias [-p] [name[=value] ...]
```

Without arguments or with the `-p` option, `alias` prints the list of aliases on the standard output in a form that allows them to be reused as input. If arguments are supplied, an alias is defined for each *name* whose *value* is given. If no *value* is given, the name and value of the alias is printed. Aliases are described in Section 6.6 [Aliases], page 93.

`bind`

```
bind [-m keymap] [-lpsvPSVX]
bind [-m keymap] [-q function] [-u function] [-r keyseq]
bind [-m keymap] -f filename
bind [-m keymap] -x keyseq:shell-command
bind [-m keymap] keyseq:function-name
bind [-m keymap] keyseq:readline-command
```

Display current Readline (see Chapter 8 [Command Line Editing], page 108) key and function bindings, bind a key sequence to a Readline function or macro, or set a Readline variable. Each non-option argument is a command as it would appear in a Readline initialization file (see Section 8.3 [Readline Init File], page 111), but each binding or command must be passed as a separate argument; e.g., '`"\C-x\C-r":re-read-init-file`'.

Options, if supplied, have the following meanings:

`-m` *keymap* Use *keymap* as the keymap to be affected by the subsequent bindings. Acceptable *keymap* names are `emacs`, `emacs-standard`, `emacs-meta`, `emacs-ctlx`, `vi`, `vi-move`, `vi-command`, and `vi-insert`. `vi` is equivalent to `vi-command` (`vi-move` is also a synonym); `emacs` is equivalent to `emacs-standard`.

`-l` List the names of all Readline functions.

`-p` Display Readline function names and bindings in such a way that they can be used as input or in a Readline initialization file.

`-P` List current Readline function names and bindings.

`-v` Display Readline variable names and values in such a way that they can be used as input or in a Readline initialization file.

`-V` List current Readline variable names and values.

`-s` Display Readline key sequences bound to macros and the strings they output in such a way that they can be used as input or in a Readline initialization file.

`-S` Display Readline key sequences bound to macros and the strings they output.

`-f` *filename*
Read key bindings from *filename*.

`-q` *function*
Query about which keys invoke the named *function*.

`-u` *function*
Unbind all keys bound to the named *function*.

`-r` *keyseq* Remove any current binding for *keyseq*.

`-x` *keyseq:shell-command*
Cause *shell-command* to be executed whenever *keyseq* is entered. When *shell-command* is executed, the shell sets the `READLINE_LINE` variable to the contents of the Readline line buffer and the `READLINE_POINT` variable to the current location of the insertion point. If the executed command changes the value of `READLINE_LINE` or `READLINE_POINT`, those new values will be reflected in the editing state.

`-X` List all key sequences bound to shell commands and the associated commands in a format that can be reused as input.

The return status is zero unless an invalid option is supplied or an error occurs.

`builtin`

```
builtin [shell-builtin [args]]
```

Run a shell builtin, passing it *args*, and return its exit status. This is useful when defining a shell function with the same name as a shell builtin, retaining the functionality of the builtin within the function. The return status is non-zero if *shell-builtin* is not a shell builtin command.

`caller`

```
caller [expr]
```

Returns the context of any active subroutine call (a shell function or a script executed with the `.` or `source` builtins).

Without *expr*, `caller` displays the line number and source filename of the current subroutine call. If a non-negative integer is supplied as *expr*, `caller` displays the line number, subroutine name, and source file corresponding to that position in the current execution call stack. This extra information may be used, for example, to print a stack trace. The current frame is frame 0.

The return value is 0 unless the shell is not executing a subroutine call or *expr* does not correspond to a valid position in the call stack.

`command`

```
command [-pVv] command [arguments ...]
```

Runs *command* with *arguments* ignoring any shell function named *command*. Only shell builtin commands or commands found by searching the `PATH` are executed. If there is a shell function named `ls`, running '`command ls`' within the function will execute the external command `ls` instead of calling the function recursively. The `-p` option means to use a default value for `PATH` that is guaranteed to find all of the standard utilities. The return status in this case is 127 if *command* cannot be found or an error occurred, and the exit status of *command* otherwise.

If either the `-V` or `-v` option is supplied, a description of *command* is printed. The `-v` option causes a single word indicating the command or file name used to invoke *command* to be displayed; the `-V` option produces a more verbose description. In this case, the return status is zero if *command* is found, and non-zero if not.

`declare`

```
declare [-aAfFgilnrtux] [-p] [name[=value] ...]
```

Declare variables and give them attributes. If no *names* are given, then display the values of variables instead.

The `-p` option will display the attributes and values of each *name*. When `-p` is used with *name* arguments, additional options, other than `-f` and `-F`, are ignored.

When `-p` is supplied without *name* arguments, `declare` will display the attributes and values of all variables having the attributes specified by the additional options. If no other options are supplied with `-p`, `declare` will display the attributes and values of all shell variables. The `-f` option will restrict the display to shell functions.

The `-F` option inhibits the display of function definitions; only the function name and attributes are printed. If the `extdebug` shell option is enabled using `shopt` (see Section 4.3.2 [The Shopt Builtin], page 65), the source file name and line number where each *name* is defined are displayed as well. `-F` implies `-f`.

The `-g` option forces variables to be created or modified at the global scope, even when `declare` is executed in a shell function. It is ignored in all other cases.

The following options can be used to restrict output to variables with the specified attributes or to give variables attributes:

`-a` Each *name* is an indexed array variable (see Section 6.7 [Arrays], page 94).

`-A` Each *name* is an associative array variable (see Section 6.7 [Arrays], page 94).

`-f` Use function names only.

`-i` The variable is to be treated as an integer; arithmetic evaluation (see Section 6.5 [Shell Arithmetic], page 92) is performed when the variable is assigned a value.

`-l` When the variable is assigned a value, all upper-case characters are converted to lower-case. The upper-case attribute is disabled.

`-n` Give each *name* the *nameref* attribute, making it a name reference to another variable. That other variable is defined by the value of *name*. All references, assignments, and attribute modifications to *name*, except for those using or changing the `-n` attribute itself, are performed on the variable referenced by *name*'s value. The nameref attribute cannot be applied to array variables.

`-r` Make *name*s readonly. These names cannot then be assigned values by subsequent assignment statements or unset.

`-t` Give each *name* the `trace` attribute. Traced functions inherit the `DEBUG` and `RETURN` traps from the calling shell. The trace attribute has no special meaning for variables.

`-u` When the variable is assigned a value, all lower-case characters are converted to upper-case. The lower-case attribute is disabled.

`-x` Mark each *name* for export to subsequent commands via the environment.

Using '`+`' instead of '`-`' turns off the attribute instead, with the exceptions that '`+a`' and '`+A`' may not be used to destroy array variables and '`+r`' will not remove the readonly attribute. When used in a function, `declare` makes each *name* local, as with the `local` command, unless the `-g` option is used. If a variable name is followed by =*value*, the value of the variable is set to *value*.

When using `-a` or `-A` and the compound assignment syntax to create array variables, additional attributes do not take effect until subsequent assignments.

The return status is zero unless an invalid option is encountered, an attempt is made to define a function using '`-f foo=bar`', an attempt is made to assign a value to a readonly variable, an attempt is made to assign a value to an array variable without using the compound assignment syntax (see Section 6.7 [Arrays], page 94), one of the *names* is not a valid shell variable name, an attempt is made to turn off readonly status for a readonly variable, an attempt is made to turn off array status for an array variable, or an attempt is made to display a non-existent function with `-f`.

`echo`

`echo [-neE] [`*arg*` ...]`

Output the *arg*s, separated by spaces, terminated with a newline. The return status is 0 unless a write error occurs. If `-n` is specified, the trailing newline is suppressed. If the `-e` option is given, interpretation of the following backslash-escaped characters is enabled. The `-E` option disables the interpretation of these escape characters, even on systems where they are interpreted by default. The `xpg_echo` shell option may be used to dynamically determine whether or not `echo` expands these escape characters by default. `echo` does not interpret `--` to mean the end of options.

`echo` interprets the following escape sequences:

`\a` alert (bell)

`\b` backspace

`\c` suppress further output

`\e`
`\E` escape

`\f` form feed

`\n` new line

`\r` carriage return

`\t` horizontal tab

`\v` vertical tab

`\\` backslash

`\0`*nnn* the eight-bit character whose value is the octal value *nnn* (zero to three octal digits)

`\x`*HH* the eight-bit character whose value is the hexadecimal value *HH* (one or two hex digits)

`\u`*HHHH* the Unicode (ISO/IEC 10646) character whose value is the hexadecimal value *HHHH* (one to four hex digits)

`\U`*HHHHHHHH*
the Unicode (ISO/IEC 10646) character whose value is the hexadecimal value *HHHHHHHH* (one to eight hex digits)

`enable`

```
enable [-a] [-dnps] [-f filename] [name ...]
```

Enable and disable builtin shell commands. Disabling a builtin allows a disk command which has the same name as a shell builtin to be executed without specifying a full pathname, even though the shell normally searches for builtins before disk commands. If `-n` is used, the *name*s become disabled. Otherwise *name*s are enabled. For example, to use the `test` binary found via `$PATH` instead of the shell builtin version, type '`enable -n test`'.

If the `-p` option is supplied, or no *name* arguments appear, a list of shell builtins is printed. With no other arguments, the list consists of all enabled shell builtins. The `-a` option means to list each builtin with an indication of whether or not it is enabled.

The `-f` option means to load the new builtin command *name* from shared object *filename*, on systems that support dynamic loading. The `-d` option will delete a builtin loaded with `-f`.

If there are no options, a list of the shell builtins is displayed. The `-s` option restricts `enable` to the POSIX special builtins. If `-s` is used with `-f`, the new builtin becomes a special builtin (see Section 4.4 [Special Builtins], page 71).

The return status is zero unless a *name* is not a shell builtin or there is an error loading a new builtin from a shared object.

`help`

```
help [-dms] [pattern]
```

Display helpful information about builtin commands. If *pattern* is specified, `help` gives detailed help on all commands matching *pattern*, otherwise a list of the builtins is printed.

Options, if supplied, have the following meanings:

`-d` Display a short description of each *pattern*

`-m` Display the description of each *pattern* in a manpage-like format

`-s` Display only a short usage synopsis for each *pattern*

The return status is zero unless no command matches *pattern*.

`let`

```
let expression [expression ...]
```

The `let` builtin allows arithmetic to be performed on shell variables. Each *expression* is evaluated according to the rules given below in Section 6.5 [Shell Arithmetic], page 92. If the last *expression* evaluates to 0, `let` returns 1; otherwise 0 is returned.

`local`

```
local [option] name[=value] ...
```

For each argument, a local variable named *name* is created, and assigned *value*. The *option* can be any of the options accepted by `declare`. `local` can only be used within a function; it makes the variable *name* have a visible scope restricted to that function and its children. If *name* is '`-`', the set of shell options is made local to the function in which `local` is invoked: shell options changed using the `set` builtin inside the function are restored to their original values when the function returns. The return status is zero unless `local` is used outside a function, an invalid *name* is supplied, or *name* is a readonly variable.

`logout`

```
logout [n]
```

Exit a login shell, returning a status of *n* to the shell's parent.

`mapfile`

```
mapfile [-d delim] [-n count] [-O origin] [-s count]
    [-t] [-u fd] [-C callback] [-c quantum] [array]
```

Read lines from the standard input into the indexed array variable *array*, or from file descriptor *fd* if the `-u` option is supplied. The variable `MAPFILE` is the default *array*. Options, if supplied, have the following meanings:

`-d` The first character of *delim* is used to terminate each input line, rather than newline. If *delim* is the empty string, `mapfile` will terminate a line when it reads a NUL character.

`-n` Copy at most *count* lines. If *count* is 0, all lines are copied.

`-O` Begin assigning to *array* at index *origin*. The default index is 0.

`-s` Discard the first *count* lines read.

`-t` Remove a trailing *delim* (default newline) from each line read.

`-u` Read lines from file descriptor *fd* instead of the standard input.

`-C` Evaluate *callback* each time *quantum* lines are read. The `-c` option specifies *quantum*.

`-c` Specify the number of lines read between each call to *callback*.

If `-C` is specified without `-c`, the default quantum is 5000. When *callback* is evaluated, it is supplied the index of the next array element to be assigned and the line to be assigned to that element as additional arguments. *callback* is evaluated after the line is read but before the array element is assigned.

If not supplied with an explicit origin, `mapfile` will clear *array* before assigning to it.

`mapfile` returns successfully unless an invalid option or option argument is supplied, *array* is invalid or unassignable, or *array* is not an indexed array.

`printf`

```
printf [-v var] format [arguments]
```

Write the formatted *arguments* to the standard output under the control of the *format*. The `-v` option causes the output to be assigned to the variable *var* rather than being printed to the standard output.

The *format* is a character string which contains three types of objects: plain characters, which are simply copied to standard output, character escape sequences, which are converted and copied to the standard output, and format specifications, each of which causes printing of the next successive *argument*. In addition to the standard `printf(1)` formats, `printf` interprets the following extensions:

`%b` Causes `printf` to expand backslash escape sequences in the corresponding *argument* in the same way as `echo -e` (see Section 4.2 [Bash Builtins], page 50).

`%q` Causes `printf` to output the corresponding *argument* in a format that can be reused as shell input.

`%(datefmt)T`
Causes `printf` to output the date-time string resulting from using *datefmt* as a format string for `strftime`(3). The corresponding *argument* is an integer representing the number of seconds since the epoch. Two special argument values may be used: -1 represents the current time, and -2 represents the time the shell was invoked. If no argument is specified, conversion behaves as if -1 had been given. This is an exception to the usual `printf` behavior.

Arguments to non-string format specifiers are treated as C language constants, except that a leading plus or minus sign is allowed, and if the leading character is a single or double quote, the value is the ASCII value of the following character.

The *format* is reused as necessary to consume all of the *arguments*. If the *format* requires more *arguments* than are supplied, the extra format specifications behave as if a zero value or null string, as appropriate, had been supplied. The return value is zero on success, non-zero on failure.

`read`

```
read [-ers] [-a aname] [-d delim] [-i text] [-n nchars]
    [-N nchars] [-p prompt] [-t timeout] [-u fd] [name ...]
```

One line is read from the standard input, or from the file descriptor *fd* supplied as an argument to the `-u` option, split into words as described above in Section 3.5.7 [Word Splitting], page 32, and the first word is assigned to the first *name*, the second word to the second *name*, and so on. If there are more words than names, the remaining words and their intervening delimiters are assigned to the last *name*. If there are fewer words read from the input stream than names, the remaining names are assigned empty values. The characters in the value of the `IFS` variable are used to split the line into words using the same rules the shell uses for expansion (described above in Section 3.5.7 [Word Splitting], page 32). The backslash character '\' may be used to remove any special meaning for the next character read and for line continuation. If no names are supplied, the line read is assigned to the variable `REPLY`. The exit status is zero, unless end-of-file is encountered, `read` times out (in which case the status is greater than 128), a variable assignment error (such as assigning to a readonly variable) occurs, or an invalid file descriptor is supplied as the argument to `-u`.

Options, if supplied, have the following meanings:

`-a` *aname* The words are assigned to sequential indices of the array variable *aname*, starting at 0. All elements are removed from *aname* before the assignment. Other *name* arguments are ignored.

`-d` *delim* The first character of *delim* is used to terminate the input line, rather than newline. If *delim* is the empty string, `read` will terminate a line when it reads a NUL character.

`-e` Readline (see Chapter 8 [Command Line Editing], page 108) is used to obtain the line. Readline uses the current (or default, if line editing was not previously active) editing settings, but uses Readline's default filename completion.

`-i` *text* If Readline is being used to read the line, *text* is placed into the editing buffer before editing begins.

`-n` *nchars* `read` returns after reading *nchars* characters rather than waiting for a complete line of input, but honors a delimiter if fewer than *nchars* characters are read before the delimiter.

`-N` *nchars* `read` returns after reading exactly *nchars* characters rather than waiting for a complete line of input, unless EOF is encountered or `read` times out. Delimiter characters encountered in the input are not treated specially and do not cause `read` to return until *nchars* characters are read. The result is not split on the characters in `IFS`; the intent is that the variable is assigned exactly the characters read (with the exception of backslash; see the `-r` option below).

`-p` *prompt* Display *prompt*, without a trailing newline, before attempting to read any input. The prompt is displayed only if input is coming from a terminal.

`-r` If this option is given, backslash does not act as an escape character. The backslash is considered to be part of the line. In particular, a backslash-newline pair may not then be used as a line continuation.

`-s` Silent mode. If input is coming from a terminal, characters are not echoed.

`-t` *timeout*

Cause `read` to time out and return failure if a complete line of input (or a specified number of characters) is not read within *timeout* seconds. *timeout* may be a decimal number with a fractional portion following the decimal point. This option is only effective if `read` is reading input from a terminal, pipe, or other special file; it has no effect when reading from regular files. If `read` times out, `read` saves any partial input read into the specified variable *name*. If *timeout* is 0, `read` returns immediately, without trying to read and data. The exit status is 0 if input is available on the specified file descriptor, non-zero otherwise. The exit status is greater than 128 if the timeout is exceeded.

`-u` *fd* Read input from file descriptor *fd*.

`readarray`

```
readarray [-d delim] [-n count] [-O origin] [-s count]
    [-t] [-u fd] [-C callback] [-c quantum] [array]
```

Read lines from the standard input into the indexed array variable *array*, or from file descriptor *fd* if the `-u` option is supplied.

A synonym for `mapfile`.

`source`

```
source filename
```

A synonym for `.` (see Section 4.1 [Bourne Shell Builtins], page 43).

`type`

```
type [-afptP] [name ...]
```

For each *name*, indicate how it would be interpreted if used as a command name.

If the `-t` option is used, `type` prints a single word which is one of '`alias`', '`function`', '`builtin`', '`file`' or '`keyword`', if *name* is an alias, shell function, shell builtin, disk file, or shell reserved word, respectively. If the *name* is not found, then nothing is printed, and `type` returns a failure status.

If the `-p` option is used, `type` either returns the name of the disk file that would be executed, or nothing if `-t` would not return '`file`'.

The `-P` option forces a path search for each *name*, even if `-t` would not return '`file`'.

If a command is hashed, `-p` and `-P` print the hashed value, which is not necessarily the file that appears first in `$PATH`.

If the `-a` option is used, `type` returns all of the places that contain an executable named *file*. This includes aliases and functions, if and only if the `-p` option is not also used.

If the `-f` option is used, `type` does not attempt to find shell functions, as with the `command` builtin.

The return status is zero if all of the *names* are found, non-zero if any are not found.

`typeset`

```
typeset [-afFgrxilnrtux] [-p] [name[=value] ...]
```

The `typeset` command is supplied for compatibility with the Korn shell. It is a synonym for the `declare` builtin command.

`ulimit`

```
ulimit [-HSabcdefiklmnpqrstuvxPT] [limit]
```

`ulimit` provides control over the resources available to processes started by the shell, on systems that allow such control. If an option is given, it is interpreted as follows:

`-S` Change and report the soft limit associated with a resource.

`-H` Change and report the hard limit associated with a resource.

`-a` All current limits are reported.

`-b` The maximum socket buffer size.

`-c` The maximum size of core files created.

`-d` The maximum size of a process's data segment.

`-e` The maximum scheduling priority ("nice").

`-f` The maximum size of files written by the shell and its children.

`-i` The maximum number of pending signals.

`-k` The maximum number of kqueues that may be allocated.

`-l` The maximum size that may be locked into memory.

`-m` The maximum resident set size (many systems do not honor this limit).

`-n` The maximum number of open file descriptors (most systems do not allow this value to be set).

`-p` The pipe buffer size.

`-q` The maximum number of bytes in POSIX message queues.

`-r` The maximum real-time scheduling priority.

`-s` The maximum stack size.

`-t` The maximum amount of cpu time in seconds.

`-u` The maximum number of processes available to a single user.

`-v` The maximum amount of virtual memory available to the shell, and, on some systems, to its children.

`-x` The maximum number of file locks.

`-P` The maximum number of pseudoterminals.

`-T` The maximum number of threads.

If *limit* is given, and the `-a` option is not used, *limit* is the new value of the specified resource. The special *limit* values `hard`, `soft`, and `unlimited` stand for the current hard limit, the current soft limit, and no limit, respectively. A hard limit cannot be increased by a non-root user once it is set; a soft limit may be increased up to the value of the hard limit. Otherwise, the current value of the soft limit for the specified resource is printed, unless the `-H` option is supplied. When setting new limits, if neither `-H` nor `-S` is supplied, both the hard and soft limits are set. If no option is given, then `-f` is assumed. Values are in 1024-byte increments, except for `-t`, which is in seconds; `-p`, which is in units of 512-byte blocks; `-P`, `-T`, `-b`, `-k`, `-n` and `-u`, which are unscaled values; and, when in POSIX Mode (see Section 6.11 [Bash POSIX Mode], page 99), `-c` and `-f`, which are in 512-byte increments.

The return status is zero unless an invalid option or argument is supplied, or an error occurs while setting a new limit.

`unalias`

    unalias [-a] [name ...]

Remove each *name* from the list of aliases. If `-a` is supplied, all aliases are removed. Aliases are described in Section 6.6 [Aliases], page 93.

4.3 Modifying Shell Behavior

4.3.1 The Set Builtin

This builtin is so complicated that it deserves its own section. `set` allows you to change the values of shell options and set the positional parameters, or to display the names and values of shell variables.

`set`

    set [--abefhkmnptuvxBCEHPT] [-o option-name] [argument ...]
    set [+abefhkmnptuvxBCEHPT] [+o option-name] [argument ...]

If no options or arguments are supplied, `set` displays the names and values of all shell variables and functions, sorted according to the current locale, in a format that may be reused as input for setting or resetting the currently-set variables. Read-only variables cannot be reset. In POSIX mode, only shell variables are listed.

When options are supplied, they set or unset shell attributes. Options, if specified, have the following meanings:

`-a` Each variable or function that is created or modified is given the export attribute and marked for export to the environment of subsequent commands.

`-b` Cause the status of terminated background jobs to be reported immediately, rather than before printing the next primary prompt.

`-e` Exit immediately if a pipeline (see Section 3.2.2 [Pipelines], page 8), which may consist of a single simple command (see Section 3.2.1 [Simple Commands], page 8), a list (see Section 3.2.3 [Lists], page 9), or a compound command (see Section 3.2.4 [Compound Commands], page 9) returns a non-zero status. The shell does not exit if the command that fails is part of the command list immediately following a `while` or `until` keyword, part of the test in an `if` statement, part of any command executed in a `&&` or `||` list except the command following the final `&&` or `||`, any command in a pipeline but the last, or if the command's return status is being inverted with `!`. If a compound command other than a subshell returns a non-zero status because a command failed while `-e` was being ignored, the shell does not exit. A trap on `ERR`, if set, is executed before the shell exits.

This option applies to the shell environment and each subshell environment separately (see Section 3.7.3 [Command Execution Environment], page 39), and may cause subshells to exit before executing all the commands in the subshell.

If a compound command or shell function executes in a context where `-e` is being ignored, none of the commands executed within the compound command or function body will be affected by the `-e` setting, even if `-e` is set and a command returns a failure status. If a compound command or shell function sets `-e` while executing in a context where `-e` is ignored, that setting will not have any effect until the compound command or the command containing the function call completes.

`-f` Disable filename expansion (globbing).

`-h` Locate and remember (hash) commands as they are looked up for execution. This option is enabled by default.

`-k` All arguments in the form of assignment statements are placed in the environment for a command, not just those that precede the command name.

`-m` Job control is enabled (see Chapter 7 [Job Control], page 104). All processes run in a separate process group. When a background job completes, the shell prints a line containing its exit status.

`-n` Read commands but do not execute them. This may be used to check a script for syntax errors. This option is ignored by interactive shells.

`-o` *`option-name`*
Set the option corresponding to *option-name*:

`allexport`
Same as `-a`.

`braceexpand`
Same as `-B`.

`emacs` Use an `emacs`-style line editing interface (see Chapter 8 [Command Line Editing], page 108). This also affects the editing interface used for `read -e`.

`errexit` Same as `-e`.

`errtrace` Same as `-E`.

`functrace`
Same as `-T`.

`hashall` Same as `-h`.

`histexpand`
Same as `-H`.

`history` Enable command history, as described in Section 9.1 [Bash History Facilities], page 143. This option is on by default in interactive shells.

`ignoreeof`
An interactive shell will not exit upon reading EOF.

`keyword` Same as `-k`.

`monitor` Same as `-m`.

`noclobber`
Same as `-C`.

`noexec` Same as `-n`.

`noglob` Same as `-f`.

`nolog` Currently ignored.

`notify` Same as `-b`.

`nounset` Same as `-u`.

`onecmd` Same as `-t`.

`physical` Same as `-P`.

`pipefail` If set, the return value of a pipeline is the value of the last (rightmost) command to exit with a non-zero status, or zero if all commands in the pipeline exit successfully. This option is disabled by default.

`posix` Change the behavior of Bash where the default operation differs from the POSIX standard to match the standard (see Section 6.11 [Bash POSIX Mode], page 99). This is intended to make Bash behave as a strict superset of that standard.

`privileged`
Same as `-p`.

`verbose` Same as `-v`.

`vi` Use a `vi`-style line editing interface. This also affects the editing interface used for `read -e`.

`xtrace` Same as `-x`.

`-p` Turn on privileged mode. In this mode, the `$BASH_ENV` and `$ENV` files are not processed, shell functions are not inherited from the environment, and the `SHELLOPTS`, `BASHOPTS`, `CDPATH` and `GLOBIGNORE` variables, if they appear in the environment, are ignored. If the shell is started with the effective user (group) id not equal to the real user (group) id, and the `-p` option is not supplied, these actions are taken and the effective user id is set to the real user id. If the `-p` option is supplied at startup, the effective user id is not reset. Turning this option off causes the effective user and group ids to be set to the real user and group ids.

`-t` Exit after reading and executing one command.

`-u` Treat unset variables and parameters other than the special parameters '`@`' or '`*`' as an error when performing parameter expansion.

An error message will be written to the standard error, and a non-interactive shell will exit.

`-v` Print shell input lines as they are read.

`-x` Print a trace of simple commands, `for` commands, `case` commands, `select` commands, and arithmetic `for` commands and their arguments or associated word lists after they are expanded and before they are executed. The value of the `PS4` variable is expanded and the resultant value is printed before the command and its expanded arguments.

`-B` The shell will perform brace expansion (see Section 3.5.1 [Brace Expansion], page 23). This option is on by default.

`-C` Prevent output redirection using '`>`', '`>&`', and '`<>`' from overwriting existing files.

`-E` If set, any trap on `ERR` is inherited by shell functions, command substitutions, and commands executed in a subshell environment. The `ERR` trap is normally not inherited in such cases.

`-H` Enable '`!`' style history substitution (see Section 9.3 [History Interaction], page 145). This option is on by default for interactive shells.

`-P` If set, do not resolve symbolic links when performing commands such as `cd` which change the current directory. The physical directory is used instead. By default, Bash follows the logical chain of directories when performing commands which change the current directory.

For example, if `/usr/sys` is a symbolic link to `/usr/local/sys` then:

```
$ cd /usr/sys; echo $PWD
/usr/sys
$ cd ..; pwd
/usr
```

If `set -P` is on, then:

```
$ cd /usr/sys; echo $PWD
/usr/local/sys
$ cd ..; pwd
/usr/local
```

`-T` If set, any trap on `DEBUG` and `RETURN` are inherited by shell functions, command substitutions, and commands executed in a subshell environment. The `DEBUG` and `RETURN` traps are normally not inherited in such cases.

`--` If no arguments follow this option, then the positional parameters are unset. Otherwise, the positional parameters are set to the *arguments*, even if some of them begin with a '`-`'.

`-` Signal the end of options, cause all remaining *arguments* to be assigned to the positional parameters. The `-x` and `-v` options are turned off. If there are no arguments, the positional parameters remain unchanged.

Using '`+`' rather than '`-`' causes these options to be turned off. The options can also be used upon invocation of the shell. The current set of options may be found in `$-`.

The remaining N *arguments* are positional parameters and are assigned, in order, to `$1`, `$2`, ... `$N`. The special parameter `#` is set to N.

The return status is always zero unless an invalid option is supplied.

4.3.2 The Shopt Builtin

This builtin allows you to change additional shell optional behavior.

`shopt`

    shopt [-pqsu] [-o] [optname ...]

Toggle the values of settings controlling optional shell behavior. The settings can be either those listed below, or, if the `-o` option is used, those available with the `-o` option to the `set` builtin command (see Section 4.3.1 [The Set Builtin], page 61). With no options, or with the `-p` option, a list of all settable options is displayed, with an indication of whether or not each is set; if *optnames* are supplied, the output is restricted to those options. The `-p` option causes output to be displayed in a form that may be reused as input. Other options have the following meanings:

`-s` Enable (set) each *optname*.

`-u` Disable (unset) each *optname*.

`-q` Suppresses normal output; the return status indicates whether the *optname* is set or unset. If multiple *optname* arguments are given with `-q`, the return status is zero if all *optnames* are enabled; non-zero otherwise.

`-o` Restricts the values of *optname* to be those defined for the `-o` option to the `set` builtin (see Section 4.3.1 [The Set Builtin], page 61).

If either `-s` or `-u` is used with no *optname* arguments, `shopt` shows only those options which are set or unset, respectively.

Unless otherwise noted, the `shopt` options are disabled (off) by default.

The return status when listing options is zero if all *optnames* are enabled, non-zero otherwise. When setting or unsetting options, the return status is zero unless an *optname* is not a valid shell option.

The list of `shopt` options is:

`assoc_expand_once`
If set, the shell suppresses multiple evaluation of associative array subscripts during arithmetic expression evaluation, while executing builtins that can perform variable assignments, and while executing builtins that perform array dereferencing.

`autocd` If set, a command name that is the name of a directory is executed as if it were the argument to the `cd` command. This option is only used by interactive shells.

`cdable_vars`
If this is set, an argument to the `cd` builtin command that is not a directory is assumed to be the name of a variable whose value is the directory to change to.

`cdspell` If set, minor errors in the spelling of a directory component in a `cd` command will be corrected. The errors checked for are transposed characters, a missing character, and a character too many. If a correction is found, the corrected path is printed, and the command proceeds. This option is only used by interactive shells.

`checkhash`
If this is set, Bash checks that a command found in the hash table exists before trying to execute it. If a hashed command no longer exists, a normal path search is performed.

`checkjobs`
If set, Bash lists the status of any stopped and running jobs before exiting an interactive shell. If any jobs are running, this causes the exit to be deferred until a second exit is attempted without an intervening command (see Chapter 7 [Job Control], page 104). The shell always postpones exiting if any jobs are stopped.

`checkwinsize`
If set, Bash checks the window size after each external (non-builtin) command and, if necessary, updates the values of `LINES` and `COLUMNS`. This option is enabled by default.

`cmdhist` If set, Bash attempts to save all lines of a multiple-line command in the same history entry. This allows easy re-editing of multi-line commands. This option is enabled by default, but only has an effect if command history is enabled (see Section 9.1 [Bash History Facilities], page 143).

`compat31` If set, Bash changes its behavior to that of version 3.1 with respect to quoted arguments to the conditional command's '`=~`' operator and with respect to locale-specific string comparison when using the `[[` conditional command's '`<`' and '`>`' operators. Bash versions prior to bash-4.1 use ASCII collation and strcmp(3); bash-4.1 and later use the current locale's collation sequence and strcoll(3).

`compat32` If set, Bash changes its behavior to that of version 3.2 with respect to locale-specific string comparison when using the `[[` conditional command's '`<`' and '`>`' operators (see previous item) and the effect of interrupting a command list. Bash versions 3.2 and earlier continue with the next command in the list after one terminates due to an interrupt.

`compat40` If set, Bash changes its behavior to that of version 4.0 with respect to locale-specific string comparison when using the `[[` conditional command's '`<`' and '`>`' operators (see description of `compat31`) and the effect of interrupting a command list. Bash versions 4.0 and later interrupt the list as if the shell received the interrupt; previous versions continue with the next command in the list.

`compat41` If set, Bash, when in POSIX mode, treats a single quote in a double-quoted parameter expansion as a special character. The single quotes must match (an even number) and the characters between the single quotes are considered quoted. This is the behavior of POSIX mode through version 4.1. The default Bash behavior remains as in previous versions.

`compat42` If set, Bash does not process the replacement string in the pattern substitution word expansion using quote removal.

`compat43` If set, Bash does not print a warning message if an attempt is made to use a quoted compound array assignment as an argument to `declare`, makes word expansion errors non-fatal errors that cause the current command to fail (the default behavior is to make them fatal errors that cause the shell to exit), and does not reset the loop state when a shell function is executed (this allows `break` or `continue` in a shell function to affect loops in the caller's context).

`compat44` If set, Bash saves the positional parameters to BASH_ARGV and BASH_ARGC before they are used, regardless of whether or not extended debugging mode is enabled.

`complete_fullquote`
If set, Bash quotes all shell metacharacters in filenames and directory names when performing completion. If not set, Bash removes metacharacters such as the dollar sign from the set of characters that will be quoted in completed filenames when these metacharacters appear in shell variable references in words to be completed. This means that dollar signs in variable names that expand to directories will not be quoted; however, any dollar signs appearing in filenames will not be quoted, either. This is active only when bash is using backslashes to quote completed filenames. This variable is set by default, which is the default Bash behavior in versions through 4.2.

`direxpand`
If set, Bash replaces directory names with the results of word expansion when performing filename completion. This changes the contents of the readline editing buffer. If not set, Bash attempts to preserve what the user typed.

`dirspell` If set, Bash attempts spelling correction on directory names during word completion if the directory name initially supplied does not exist.

`dotglob` If set, Bash includes filenames beginning with a '.' in the results of filename expansion. The filenames '`.`' and '`..`' must always be matched explicitly, even if `dotglob` is set.

`execfail` If this is set, a non-interactive shell will not exit if it cannot execute the file specified as an argument to the `exec` builtin command. An interactive shell does not exit if `exec` fails.

`expand_aliases`
If set, aliases are expanded as described below under Aliases, Section 6.6 [Aliases], page 93. This option is enabled by default for interactive shells.

`extdebug` If set at shell invocation, or in a shell startup file, arrange to execute the debugger profile before the shell starts, identical to the `--debugger` option. If set after invocation, behavior intended for use by debuggers is enabled:

1. The `-F` option to the `declare` builtin (see Section 4.2 [Bash Builtins], page 50) displays the source file name and line number corresponding to each function name supplied as an argument.
2. If the command run by the `DEBUG` trap returns a non-zero value, the next command is skipped and not executed.
3. If the command run by the `DEBUG` trap returns a value of 2, and the shell is executing in a subroutine (a shell function or a shell script executed by the `.` or `source` builtins), the shell simulates a call to `return`.
4. `BASH_ARGC` and `BASH_ARGV` are updated as described in their descriptions (see Section 5.2 [Bash Variables], page 73).
5. Function tracing is enabled: command substitution, shell functions, and subshells invoked with `(` *command* `)` inherit the `DEBUG` and `RETURN` traps.
6. Error tracing is enabled: command substitution, shell functions, and subshells invoked with `(` *command* `)` inherit the `ERR` trap.

`extglob` If set, the extended pattern matching features described above (see Section 3.5.8.1 [Pattern Matching], page 33) are enabled.

`extquote` If set, `$'`*string*`'` and `$"`*string*`"` quoting is performed within `${`*parameter*`}` expansions enclosed in double quotes. This option is enabled by default.

`failglob` If set, patterns which fail to match filenames during filename expansion result in an expansion error.

`force_fignore`
If set, the suffixes specified by the `FIGNORE` shell variable cause words to be ignored when performing word completion even if the

ignored words are the only possible completions. See Section 5.2 [Bash Variables], page 73, for a description of FIGNORE. This option is enabled by default.

globasciiranges
: If set, range expressions used in pattern matching bracket expressions (see Section 3.5.8.1 [Pattern Matching], page 33) behave as if in the traditional C locale when performing comparisons. That is, the current locale's collating sequence is not taken into account, so 'b' will not collate between 'A' and 'B', and upper-case and lower-case ASCII characters will collate together.

globstar
: If set, the pattern '**' used in a filename expansion context will match all files and zero or more directories and subdirectories. If the pattern is followed by a '/', only directories and subdirectories match.

gnu_errfmt
: If set, shell error messages are written in the standard GNU error message format.

histappend
: If set, the history list is appended to the file named by the value of the HISTFILE variable when the shell exits, rather than overwriting the file.

histreedit
: If set, and Readline is being used, a user is given the opportunity to re-edit a failed history substitution.

histverify
: If set, and Readline is being used, the results of history substitution are not immediately passed to the shell parser. Instead, the resulting line is loaded into the Readline editing buffer, allowing further modification.

hostcomplete
: If set, and Readline is being used, Bash will attempt to perform hostname completion when a word containing a '@' is being completed (see Section 8.4.6 [Commands For Completion], page 129). This option is enabled by default.

huponexit
: If set, Bash will send SIGHUP to all jobs when an interactive login shell exits (see Section 3.7.6 [Signals], page 41).

inherit_errexit
: If set, command substitution inherits the value of the errexit option, instead of unsetting it in the subshell environment. This option is enabled when POSIX mode is enabled.

`interactive_comments`
: Allow a word beginning with '`#`' to cause that word and all remaining characters on that line to be ignored in an interactive shell. This option is enabled by default.

`lastpipe`
: If set, and job control is not active, the shell runs the last command of a pipeline not executed in the background in the current shell environment.

`lithist`
: If enabled, and the `cmdhist` option is enabled, multi-line commands are saved to the history with embedded newlines rather than using semicolon separators where possible.

`localvar_inherit`
: If set, local variables inherit the value and attributes of a variable of the same name that exists at a previous scope before any new value is assigned. The *nameref* attribute is not inherited.

`localvar_unset`
: If set, calling `unset` on local variables in previous function scopes marks them so subsequent lookups find them unset until that function returns. This is identical to the behavior of unsetting local variables at the current function scope.

`login_shell`
: The shell sets this option if it is started as a login shell (see Section 6.1 [Invoking Bash], page 85). The value may not be changed.

`mailwarn`
: If set, and a file that Bash is checking for mail has been accessed since the last time it was checked, the message `"The mail in *mailfile* has been read"` is displayed.

`no_empty_cmd_completion`
: If set, and Readline is being used, Bash will not attempt to search the `PATH` for possible completions when completion is attempted on an empty line.

`nocaseglob`
: If set, Bash matches filenames in a case-insensitive fashion when performing filename expansion.

`nocasematch`
: If set, Bash matches patterns in a case-insensitive fashion when performing matching while executing `case` or `[[` conditional commands, when performing pattern substitution word expansions, or when filtering possible completions as part of programmable completion.

`nullglob`
: If set, Bash allows filename patterns which match no files to expand to a null string, rather than themselves.

`progcomp` If set, the programmable completion facilities (see Section 8.6 [Programmable Completion], page 134) are enabled. This option is enabled by default.

`progcomp_alias`
If set, and programmable completion is enabled, Bash treats a command name that doesn't have any completions as a possible alias and attempts alias expansion. If it has an alias, Bash attempts programmable completion using the command word resulting from the expanded alias.

`promptvars`
If set, prompt strings undergo parameter expansion, command substitution, arithmetic expansion, and quote removal after being expanded as described below (see Section 6.9 [Controlling the Prompt], page 97). This option is enabled by default.

`restricted_shell`
The shell sets this option if it is started in restricted mode (see Section 6.10 [The Restricted Shell], page 99). The value may not be changed. This is not reset when the startup files are executed, allowing the startup files to discover whether or not a shell is restricted.

`shift_verbose`
If this is set, the `shift` builtin prints an error message when the shift count exceeds the number of positional parameters.

`sourcepath`
If set, the `source` builtin uses the value of `PATH` to find the directory containing the file supplied as an argument. This option is enabled by default.

`xpg_echo` If set, the `echo` builtin expands backslash-escape sequences by default.

The return status when listing options is zero if all *optnames* are enabled, non-zero otherwise. When setting or unsetting options, the return status is zero unless an *optname* is not a valid shell option.

4.4 Special Builtins

For historical reasons, the POSIX standard has classified several builtin commands as *special*. When Bash is executing in POSIX mode, the special builtins differ from other builtin commands in three respects:

1. Special builtins are found before shell functions during command lookup.
2. If a special builtin returns an error status, a non-interactive shell exits.
3. Assignment statements preceding the command stay in effect in the shell environment after the command completes.

When Bash is not executing in POSIX mode, these builtins behave no differently than the rest of the Bash builtin commands. The Bash POSIX mode is described in Section 6.11 [Bash POSIX Mode], page 99.

These are the POSIX special builtins:

```
break : . continue eval exec exit export readonly return set
shift trap unset
```

5 Shell Variables

This chapter describes the shell variables that Bash uses. Bash automatically assigns default values to a number of variables.

5.1 Bourne Shell Variables

Bash uses certain shell variables in the same way as the Bourne shell. In some cases, Bash assigns a default value to the variable.

`CDPATH` A colon-separated list of directories used as a search path for the `cd` builtin command.

`HOME` The current user's home directory; the default for the `cd` builtin command. The value of this variable is also used by tilde expansion (see Section 3.5.2 [Tilde Expansion], page 24).

`IFS` A list of characters that separate fields; used when the shell splits words as part of expansion.

`MAIL` If this parameter is set to a filename or directory name and the `MAILPATH` variable is not set, Bash informs the user of the arrival of mail in the specified file or Maildir-format directory.

`MAILPATH` A colon-separated list of filenames which the shell periodically checks for new mail. Each list entry can specify the message that is printed when new mail arrives in the mail file by separating the filename from the message with a '`?`'. When used in the text of the message, `$_` expands to the name of the current mail file.

`OPTARG` The value of the last option argument processed by the `getopts` builtin.

`OPTIND` The index of the last option argument processed by the `getopts` builtin.

`PATH` A colon-separated list of directories in which the shell looks for commands. A zero-length (null) directory name in the value of `PATH` indicates the current directory. A null directory name may appear as two adjacent colons, or as an initial or trailing colon.

`PS1` The primary prompt string. The default value is '`\s-\v\$ `'. See Section 6.9 [Controlling the Prompt], page 97, for the complete list of escape sequences that are expanded before `PS1` is displayed.

`PS2` The secondary prompt string. The default value is '`> `'. `PS2` is expanded in the same way as `PS1` before being displayed.

5.2 Bash Variables

These variables are set or used by Bash, but other shells do not normally treat them specially.

A few variables used by Bash are described in different chapters: variables for controlling the job control facilities (see Section 7.3 [Job Control Variables], page 107).

`BASH` The full pathname used to execute the current instance of Bash.

BASHOPTS A colon-separated list of enabled shell options. Each word in the list is a valid argument for the -s option to the shopt builtin command (see Section 4.3.2 [The Shopt Builtin], page 65). The options appearing in BASHOPTS are those reported as 'on' by 'shopt'. If this variable is in the environment when Bash starts up, each shell option in the list will be enabled before reading any startup files. This variable is readonly.

BASHPID Expands to the process ID of the current Bash process. This differs from $$ under certain circumstances, such as subshells that do not require Bash to be re-initialized. Assignments to BASHPID have no effect. If BASHPID is unset, it loses its special properties, even if it is subsequently reset.

BASH_ALIASES
An associative array variable whose members correspond to the internal list of aliases as maintained by the alias builtin. (see Section 4.1 [Bourne Shell Builtins], page 43). Elements added to this array appear in the alias list; however, unsetting array elements currently does not cause aliases to be removed from the alias list. If BASH_ALIASES is unset, it loses its special properties, even if it is subsequently reset.

BASH_ARGC
An array variable whose values are the number of parameters in each frame of the current bash execution call stack. The number of parameters to the current subroutine (shell function or script executed with . or source) is at the top of the stack. When a subroutine is executed, the number of parameters passed is pushed onto BASH_ARGC. The shell sets BASH_ARGC only when in extended debugging mode (see Section 4.3.2 [The Shopt Builtin], page 65, for a description of the extdebug option to the shopt builtin). Setting extdebug after the shell has started to execute a script, or referencing this variable when extdebug is not set, may result in inconsistent values.

BASH_ARGV
An array variable containing all of the parameters in the current bash execution call stack. The final parameter of the last subroutine call is at the top of the stack; the first parameter of the initial call is at the bottom. When a subroutine is executed, the parameters supplied are pushed onto BASH_ARGV. The shell sets BASH_ARGV only when in extended debugging mode (see Section 4.3.2 [The Shopt Builtin], page 65, for a description of the extdebug option to the shopt builtin). Setting extdebug after the shell has started to execute a script, or referencing this variable when extdebug is not set, may result in inconsistent values.

BASH_ARGV0
When referenced, this variable expands to the name of the shell or shell script (identical to $0; See Section 3.4.2 [Special Parameters], page 21, for the description of special parameter 0). Assignment to BASH_ARGV0 causes the value assigned to also be assigned to $0. If BASH_ARGV0 is unset, it loses its special properties, even if it is subsequently reset.

`BASH_CMDS`
: An associative array variable whose members correspond to the internal hash table of commands as maintained by the `hash` builtin (see Section 4.1 [Bourne Shell Builtins], page 43). Elements added to this array appear in the hash table; however, unsetting array elements currently does not cause command names to be removed from the hash table. If `BASH_CMDS` is unset, it loses its special properties, even if it is subsequently reset.

`BASH_COMMAND`
: The command currently being executed or about to be executed, unless the shell is executing a command as the result of a trap, in which case it is the command executing at the time of the trap.

`BASH_COMPAT`
: The value is used to set the shell's compatibility level. See Section 4.3.2 [The Shopt Builtin], page 65, for a description of the various compatibility levels and their effects. The value may be a decimal number (e.g., 4.2) or an integer (e.g., 42) corresponding to the desired compatibility level. If `BASH_COMPAT` is unset or set to the empty string, the compatibility level is set to the default for the current version. If `BASH_COMPAT` is set to a value that is not one of the valid compatibility levels, the shell prints an error message and sets the compatibility level to the default for the current version. The valid compatibility levels correspond to the compatibility options accepted by the `shopt` builtin described above (for example, *compat42* means that 4.2 and 42 are valid values). The current version is also a valid value.

`BASH_ENV`
: If this variable is set when Bash is invoked to execute a shell script, its value is expanded and used as the name of a startup file to read before executing the script. See Section 6.2 [Bash Startup Files], page 87.

`BASH_EXECUTION_STRING`
: The command argument to the `-c` invocation option.

`BASH_LINENO`
: An array variable whose members are the line numbers in source files where each corresponding member of *FUNCNAME* was invoked. `${BASH_LINENO[$i]}` is the line number in the source file (`${BASH_SOURCE[$i+1]}`) where `${FUNCNAME[$i]}` was called (or `${BASH_LINENO[$i-1]}` if referenced within another shell function). Use `LINENO` to obtain the current line number.

`BASH_LOADABLES_PATH`
: A colon-separated list of directories in which the shell looks for dynamically loadable builtins specified by the `enable` command.

`BASH_REMATCH`
: An array variable whose members are assigned by the '`=~`' binary operator to the `[[` conditional command (see Section 3.2.4.2 [Conditional Constructs], page 11). The element with index 0 is the portion of the string matching the entire regular expression. The element with index *n* is the portion of the string matching the *n*th parenthesized subexpression. This variable is read-only.

`BASH_SOURCE`
An array variable whose members are the source filenames where the corresponding shell function names in the `FUNCNAME` array variable are defined. The shell function `${FUNCNAME[$i]}` is defined in the file `${BASH_SOURCE[$i]}` and called from `${BASH_SOURCE[$i+1]}`

`BASH_SUBSHELL`
Incremented by one within each subshell or subshell environment when the shell begins executing in that environment. The initial value is 0.

`BASH_VERSINFO`
A readonly array variable (see Section 6.7 [Arrays], page 94) whose members hold version information for this instance of Bash. The values assigned to the array members are as follows:

`BASH_VERSINFO[0]`
The major version number (the *release*).

`BASH_VERSINFO[1]`
The minor version number (the *version*).

`BASH_VERSINFO[2]`
The patch level.

`BASH_VERSINFO[3]`
The build version.

`BASH_VERSINFO[4]`
The release status (e.g., *beta1*).

`BASH_VERSINFO[5]`
The value of `MACHTYPE`.

`BASH_VERSION`
The version number of the current instance of Bash.

`BASH_XTRACEFD`
If set to an integer corresponding to a valid file descriptor, Bash will write the trace output generated when '`set -x`' is enabled to that file descriptor. This allows tracing output to be separated from diagnostic and error messages. The file descriptor is closed when `BASH_XTRACEFD` is unset or assigned a new value. Unsetting `BASH_XTRACEFD` or assigning it the empty string causes the trace output to be sent to the standard error. Note that setting `BASH_XTRACEFD` to 2 (the standard error file descriptor) and then unsetting it will result in the standard error being closed.

`CHILD_MAX`
Set the number of exited child status values for the shell to remember. Bash will not allow this value to be decreased below a POSIX-mandated minimum, and there is a maximum value (currently 8192) that this may not exceed. The minimum value is system-dependent.

`COLUMNS` Used by the `select` command to determine the terminal width when printing selection lists. Automatically set if the `checkwinsize` option is enabled (see

Section 4.3.2 [The Shopt Builtin], page 65), or in an interactive shell upon receipt of a SIGWINCH.

COMP_CWORD
: An index into ${COMP_WORDS} of the word containing the current cursor position. This variable is available only in shell functions invoked by the programmable completion facilities (see Section 8.6 [Programmable Completion], page 134).

COMP_LINE
: The current command line. This variable is available only in shell functions and external commands invoked by the programmable completion facilities (see Section 8.6 [Programmable Completion], page 134).

COMP_POINT
: The index of the current cursor position relative to the beginning of the current command. If the current cursor position is at the end of the current command, the value of this variable is equal to ${#COMP_LINE}. This variable is available only in shell functions and external commands invoked by the programmable completion facilities (see Section 8.6 [Programmable Completion], page 134).

COMP_TYPE
: Set to an integer value corresponding to the type of completion attempted that caused a completion function to be called: *TAB*, for normal completion, '?', for listing completions after successive tabs, '!', for listing alternatives on partial word completion, '@', to list completions if the word is not unmodified, or '%', for menu completion. This variable is available only in shell functions and external commands invoked by the programmable completion facilities (see Section 8.6 [Programmable Completion], page 134).

COMP_KEY
: The key (or final key of a key sequence) used to invoke the current completion function.

COMP_WORDBREAKS
: The set of characters that the Readline library treats as word separators when performing word completion. If COMP_WORDBREAKS is unset, it loses its special properties, even if it is subsequently reset.

COMP_WORDS
: An array variable consisting of the individual words in the current command line. The line is split into words as Readline would split it, using COMP_WORDBREAKS as described above. This variable is available only in shell functions invoked by the programmable completion facilities (see Section 8.6 [Programmable Completion], page 134).

COMPREPLY
: An array variable from which Bash reads the possible completions generated by a shell function invoked by the programmable completion facility (see Section 8.6 [Programmable Completion], page 134). Each array element contains one possible completion.

COPROC
: An array variable created to hold the file descriptors for output from and input to an unnamed coprocess (see Section 3.2.5 [Coprocesses], page 15).

`DIRSTACK` An array variable containing the current contents of the directory stack. Directories appear in the stack in the order they are displayed by the `dirs` builtin. Assigning to members of this array variable may be used to modify directories already in the stack, but the `pushd` and `popd` builtins must be used to add and remove directories. Assignment to this variable will not change the current directory. If `DIRSTACK` is unset, it loses its special properties, even if it is subsequently reset.

`EMACS` If Bash finds this variable in the environment when the shell starts with value '`t`', it assumes that the shell is running in an Emacs shell buffer and disables line editing.

`ENV` Similar to `BASH_ENV`; used when the shell is invoked in POSIX Mode (see Section 6.11 [Bash POSIX Mode], page 99).

`EPOCHREALTIME`
Each time this parameter is referenced, it expands to the number of seconds since the Unix Epoch as a floating point value with micro-second granularity (see the documentation for the C library function *time* for the definition of Epoch). Assignments to `EPOCHREALTIME` are ignored. If `EPOCHREALTIME` is unset, it loses its special properties, even if it is subsequently reset.

`EPOCHSECONDS`
Each time this parameter is referenced, it expands to the number of seconds since the Unix Epoch (see the documentation for the C library function *time* for the definition of Epoch). Assignments to `EPOCHSECONDS` are ignored. If `EPOCHSECONDS` is unset, it loses its special properties, even if it is subsequently reset.

`EUID` The numeric effective user id of the current user. This variable is readonly.

`EXECIGNORE`
A colon-separated list of shell patterns (see Section 3.5.8.1 [Pattern Matching], page 33) defining the list of filenames to be ignored by command search using `PATH`. Files whose full pathnames match one of these patterns are not considered executable files for the purposes of completion and command execution via `PATH` lookup. This does not affect the behavior of the `[`, `test`, and `[[` commands. Full pathnames in the command hash table are not subject to `EXECIGNORE`. Use this variable to ignore shared library files that have the executable bit set, but are not executable files. The pattern matching honors the setting of the `extglob` shell option.

`FCEDIT` The editor used as a default by the `-e` option to the `fc` builtin command.

`FIGNORE` A colon-separated list of suffixes to ignore when performing filename completion. A filename whose suffix matches one of the entries in `FIGNORE` is excluded from the list of matched filenames. A sample value is '`.o:~`'

`FUNCNAME` An array variable containing the names of all shell functions currently in the execution call stack. The element with index 0 is the name of any currently-executing shell function. The bottom-most element (the one with the highest index) is `"main"`. This variable exists only when a shell function is executing.

Assignments to `FUNCNAME` have no effect. If `FUNCNAME` is unset, it loses its special properties, even if it is subsequently reset.

This variable can be used with `BASH_LINENO` and `BASH_SOURCE`. Each element of `FUNCNAME` has corresponding elements in `BASH_LINENO` and `BASH_SOURCE` to describe the call stack. For instance, `${FUNCNAME[$i]}` was called from the file `${BASH_SOURCE[$i+1]}` at line number `${BASH_LINENO[$i]}`. The `caller` builtin displays the current call stack using this information.

`FUNCNEST` If set to a numeric value greater than 0, defines a maximum function nesting level. Function invocations that exceed this nesting level will cause the current command to abort.

`GLOBIGNORE`
A colon-separated list of patterns defining the set of file names to be ignored by filename expansion. If a file name matched by a filename expansion pattern also matches one of the patterns in `GLOBIGNORE`, it is removed from the list of matches. The pattern matching honors the setting of the `extglob` shell option.

`GROUPS` An array variable containing the list of groups of which the current user is a member. Assignments to `GROUPS` have no effect. If `GROUPS` is unset, it loses its special properties, even if it is subsequently reset.

`histchars`
Up to three characters which control history expansion, quick substitution, and tokenization (see Section 9.3 [History Interaction], page 145). The first character is the *history expansion* character, that is, the character which signifies the start of a history expansion, normally '`!`'. The second character is the character which signifies 'quick substitution' when seen as the first character on a line, normally '`^`'. The optional third character is the character which indicates that the remainder of the line is a comment when found as the first character of a word, usually '`#`'. The history comment character causes history substitution to be skipped for the remaining words on the line. It does not necessarily cause the shell parser to treat the rest of the line as a comment.

`HISTCMD` The history number, or index in the history list, of the current command. If `HISTCMD` is unset, it loses its special properties, even if it is subsequently reset.

`HISTCONTROL`
A colon-separated list of values controlling how commands are saved on the history list. If the list of values includes '`ignorespace`', lines which begin with a space character are not saved in the history list. A value of '`ignoredups`' causes lines which match the previous history entry to not be saved. A value of '`ignoreboth`' is shorthand for '`ignorespace`' and '`ignoredups`'. A value of '`erasedups`' causes all previous lines matching the current line to be removed from the history list before that line is saved. Any value not in the above list is ignored. If `HISTCONTROL` is unset, or does not include a valid value, all lines read by the shell parser are saved on the history list, subject to the value of `HISTIGNORE`. The second and subsequent lines of a multi-line compound command are not tested, and are added to the history regardless of the value of `HISTCONTROL`.

HISTFILE The name of the file to which the command history is saved. The default value is ~/.bash_history.

HISTFILESIZE
The maximum number of lines contained in the history file. When this variable is assigned a value, the history file is truncated, if necessary, to contain no more than that number of lines by removing the oldest entries. The history file is also truncated to this size after writing it when a shell exits. If the value is 0, the history file is truncated to zero size. Non-numeric values and numeric values less than zero inhibit truncation. The shell sets the default value to the value of HISTSIZE after reading any startup files.

HISTIGNORE
A colon-separated list of patterns used to decide which command lines should be saved on the history list. Each pattern is anchored at the beginning of the line and must match the complete line (no implicit '*' is appended). Each pattern is tested against the line after the checks specified by HISTCONTROL are applied. In addition to the normal shell pattern matching characters, '&' matches the previous history line. '&' may be escaped using a backslash; the backslash is removed before attempting a match. The second and subsequent lines of a multi-line compound command are not tested, and are added to the history regardless of the value of HISTIGNORE. The pattern matching honors the setting of the extglob shell option.

HISTIGNORE subsumes the function of HISTCONTROL. A pattern of '&' is identical to ignoredups, and a pattern of '[ ]*' is identical to ignorespace. Combining these two patterns, separating them with a colon, provides the functionality of ignoreboth.

HISTSIZE The maximum number of commands to remember on the history list. If the value is 0, commands are not saved in the history list. Numeric values less than zero result in every command being saved on the history list (there is no limit). The shell sets the default value to 500 after reading any startup files.

HISTTIMEFORMAT
If this variable is set and not null, its value is used as a format string for *strftime* to print the time stamp associated with each history entry displayed by the history builtin. If this variable is set, time stamps are written to the history file so they may be preserved across shell sessions. This uses the history comment character to distinguish timestamps from other history lines.

HOSTFILE Contains the name of a file in the same format as /etc/hosts that should be read when the shell needs to complete a hostname. The list of possible hostname completions may be changed while the shell is running; the next time hostname completion is attempted after the value is changed, Bash adds the contents of the new file to the existing list. If HOSTFILE is set, but has no value, or does not name a readable file, Bash attempts to read /etc/hosts to obtain the list of possible hostname completions. When HOSTFILE is unset, the hostname list is cleared.

HOSTNAME The name of the current host.

`HOSTTYPE` A string describing the machine Bash is running on.

`IGNOREEOF`
: Controls the action of the shell on receipt of an `EOF` character as the sole input. If set, the value denotes the number of consecutive `EOF` characters that can be read as the first character on an input line before the shell will exit. If the variable exists but does not have a numeric value, or has no value, then the default is 10. If the variable does not exist, then `EOF` signifies the end of input to the shell. This is only in effect for interactive shells.

`INPUTRC` The name of the Readline initialization file, overriding the default of `~/.inputrc`.

`INSIDE_EMACS`
: If Bash finds this variable in the environment when the shell starts, it assumes that the shell is running in an Emacs shell buffer and may disable line editing depending on the value of `TERM`.

`LANG` Used to determine the locale category for any category not specifically selected with a variable starting with `LC_`.

`LC_ALL` This variable overrides the value of `LANG` and any other `LC_` variable specifying a locale category.

`LC_COLLATE`
: This variable determines the collation order used when sorting the results of filename expansion, and determines the behavior of range expressions, equivalence classes, and collating sequences within filename expansion and pattern matching (see Section 3.5.8 [Filename Expansion], page 32).

`LC_CTYPE` This variable determines the interpretation of characters and the behavior of character classes within filename expansion and pattern matching (see Section 3.5.8 [Filename Expansion], page 32).

`LC_MESSAGES`
: This variable determines the locale used to translate double-quoted strings preceded by a ‘`$`’ (see Section 3.1.2.5 [Locale Translation], page 7).

`LC_NUMERIC`
: This variable determines the locale category used for number formatting.

`LC_TIME` This variable determines the locale category used for data and time formatting.

`LINENO` The line number in the script or shell function currently executing.

`LINES` Used by the `select` command to determine the column length for printing selection lists. Automatically set if the `checkwinsize` option is enabled (see Section 4.3.2 [The Shopt Builtin], page 65), or in an interactive shell upon receipt of a `SIGWINCH`.

`MACHTYPE` A string that fully describes the system type on which Bash is executing, in the standard GNU *cpu-company-system* format.

`MAILCHECK`
: How often (in seconds) that the shell should check for mail in the files specified in the `MAILPATH` or `MAIL` variables. The default is 60 seconds. When it is time

to check for mail, the shell does so before displaying the primary prompt. If this variable is unset, or set to a value that is not a number greater than or equal to zero, the shell disables mail checking.

MAPFILE
: An array variable created to hold the text read by the `mapfile` builtin when no variable name is supplied.

OLDPWD
: The previous working directory as set by the `cd` builtin.

OPTERR
: If set to the value 1, Bash displays error messages generated by the `getopts` builtin command.

OSTYPE
: A string describing the operating system Bash is running on.

PIPESTATUS
: An array variable (see Section 6.7 [Arrays], page 94) containing a list of exit status values from the processes in the most-recently-executed foreground pipeline (which may contain only a single command).

POSIXLY_CORRECT
: If this variable is in the environment when Bash starts, the shell enters POSIX mode (see Section 6.11 [Bash POSIX Mode], page 99) before reading the startup files, as if the `--posix` invocation option had been supplied. If it is set while the shell is running, Bash enables POSIX mode, as if the command

```
set -o posix
```

had been executed. When the shell enters POSIX mode, it sets this variable if it was not already set.

PPID
: The process ID of the shell's parent process. This variable is readonly.

PROMPT_COMMAND
: If set, the value is interpreted as a command to execute before the printing of each primary prompt (`$PS1`).

PROMPT_DIRTRIM
: If set to a number greater than zero, the value is used as the number of trailing directory components to retain when expanding the `\w` and `\W` prompt string escapes (see Section 6.9 [Controlling the Prompt], page 97). Characters removed are replaced with an ellipsis.

PS0
: The value of this parameter is expanded like *PS1* and displayed by interactive shells after reading a command and before the command is executed.

PS3
: The value of this variable is used as the prompt for the `select` command. If this variable is not set, the `select` command prompts with '`#? `'

PS4
: The value of this parameter is expanded like *PS1* and the expanded value is the prompt printed before the command line is echoed when the `-x` option is set (see Section 4.3.1 [The Set Builtin], page 61). The first character of the expanded value is replicated multiple times, as necessary, to indicate multiple levels of indirection. The default is '`+ `'.

PWD
: The current working directory as set by the `cd` builtin.

`RANDOM` Each time this parameter is referenced, a random integer between 0 and 32767 is generated. Assigning a value to this variable seeds the random number generator.

`READLINE_LINE`
The contents of the Readline line buffer, for use with '`bind -x`' (see Section 4.2 [Bash Builtins], page 50).

`READLINE_POINT`
The position of the insertion point in the Readline line buffer, for use with '`bind -x`' (see Section 4.2 [Bash Builtins], page 50).

`REPLY` The default variable for the `read` builtin.

`SECONDS` This variable expands to the number of seconds since the shell was started. Assignment to this variable resets the count to the value assigned, and the expanded value becomes the value assigned plus the number of seconds since the assignment.

`SHELL` The full pathname to the shell is kept in this environment variable. If it is not set when the shell starts, Bash assigns to it the full pathname of the current user's login shell.

`SHELLOPTS`
A colon-separated list of enabled shell options. Each word in the list is a valid argument for the `-o` option to the `set` builtin command (see Section 4.3.1 [The Set Builtin], page 61). The options appearing in `SHELLOPTS` are those reported as '`on`' by '`set -o`'. If this variable is in the environment when Bash starts up, each shell option in the list will be enabled before reading any startup files. This variable is readonly.

`SHLVL` Incremented by one each time a new instance of Bash is started. This is intended to be a count of how deeply your Bash shells are nested.

`TIMEFORMAT`
The value of this parameter is used as a format string specifying how the timing information for pipelines prefixed with the `time` reserved word should be displayed. The '`%`' character introduces an escape sequence that is expanded to a time value or other information. The escape sequences and their meanings are as follows; the braces denote optional portions.

`%%` A literal '`%`'.

`%[p][l]R` The elapsed time in seconds.

`%[p][l]U` The number of CPU seconds spent in user mode.

`%[p][l]S` The number of CPU seconds spent in system mode.

`%P` The CPU percentage, computed as (%U + %S) / %R.

The optional *p* is a digit specifying the precision, the number of fractional digits after a decimal point. A value of 0 causes no decimal point or fraction to be output. At most three places after the decimal point may be specified; values of *p* greater than 3 are changed to 3. If *p* is not specified, the value 3 is used.

The optional `l` specifies a longer format, including minutes, of the form *MM*m*SS*.*FF*s. The value of *p* determines whether or not the fraction is included.

If this variable is not set, Bash acts as if it had the value

```
$'\nreal\t%3lR\nuser\t%3lU\nsys\t%3lS'
```

If the value is null, no timing information is displayed. A trailing newline is added when the format string is displayed.

`TMOUT` If set to a value greater than zero, `TMOUT` is treated as the default timeout for the `read` builtin (see Section 4.2 [Bash Builtins], page 50). The `select` command (see Section 3.2.4.2 [Conditional Constructs], page 11) terminates if input does not arrive after `TMOUT` seconds when input is coming from a terminal.

In an interactive shell, the value is interpreted as the number of seconds to wait for a line of input after issuing the primary prompt. Bash terminates after waiting for that number of seconds if a complete line of input does not arrive.

`TMPDIR` If set, Bash uses its value as the name of a directory in which Bash creates temporary files for the shell's use.

`UID` The numeric real user id of the current user. This variable is readonly.

6 Bash Features

This chapter describes features unique to Bash.

6.1 Invoking Bash

```
bash [long-opt] [-ir] [-abefhkmnptuvxdBCDHP] [-o option]
    [-O shopt_option] [argument ...]
bash [long-opt] [-abefhkmnptuvxdBCDHP] [-o option]
    [-O shopt_option] -c string [argument ...]
bash [long-opt] -s [-abefhkmnptuvxdBCDHP] [-o option]
    [-O shopt_option] [argument ...]
```

All of the single-character options used with the `set` builtin (see Section 4.3.1 [The Set Builtin], page 61) can be used as options when the shell is invoked. In addition, there are several multi-character options that you can use. These options must appear on the command line before the single-character options to be recognized.

`--debugger`
: Arrange for the debugger profile to be executed before the shell starts. Turns on extended debugging mode (see Section 4.3.2 [The Shopt Builtin], page 65, for a description of the `extdebug` option to the `shopt` builtin).

`--dump-po-strings`
: A list of all double-quoted strings preceded by '`$`' is printed on the standard output in the GNU `gettext` PO (portable object) file format. Equivalent to `-D` except for the output format.

`--dump-strings`
: Equivalent to `-D`.

`--help`
: Display a usage message on standard output and exit successfully.

`--init-file` *filename*
`--rcfile` *filename*
: Execute commands from *filename* (instead of `~/.bashrc`) in an interactive shell.

`--login`
: Equivalent to `-l`.

`--noediting`
: Do not use the GNU Readline library (see Chapter 8 [Command Line Editing], page 108) to read command lines when the shell is interactive.

`--noprofile`
: Don't load the system-wide startup file `/etc/profile` or any of the personal initialization files `~/.bash_profile`, `~/.bash_login`, or `~/.profile` when Bash is invoked as a login shell.

`--norc`
: Don't read the `~/.bashrc` initialization file in an interactive shell. This is on by default if the shell is invoked as `sh`.

`--posix`
: Change the behavior of Bash where the default operation differs from the POSIX standard to match the standard. This is intended to make Bash behave as a strict superset of that standard. See Section 6.11 [Bash POSIX Mode], page 99, for a description of the Bash POSIX mode.

`--restricted`
: Make the shell a restricted shell (see Section 6.10 [The Restricted Shell], page 99).

`--verbose`
: Equivalent to `-v`. Print shell input lines as they're read.

`--version`
: Show version information for this instance of Bash on the standard output and exit successfully.

There are several single-character options that may be supplied at invocation which are not available with the `set` builtin.

`-c`
: Read and execute commands from the first non-option argument *command_string*, then exit. If there are arguments after the *command_string*, the first argument is assigned to `$0` and any remaining arguments are assigned to the positional parameters. The assignment to `$0` sets the name of the shell, which is used in warning and error messages.

`-i`
: Force the shell to run interactively. Interactive shells are described in Section 6.3 [Interactive Shells], page 88.

`-l`
: Make this shell act as if it had been directly invoked by login. When the shell is interactive, this is equivalent to starting a login shell with '`exec -l bash`'. When the shell is not interactive, the login shell startup files will be executed. '`exec bash -l`' or '`exec bash --login`' will replace the current shell with a Bash login shell. See Section 6.2 [Bash Startup Files], page 87, for a description of the special behavior of a login shell.

`-r`
: Make the shell a restricted shell (see Section 6.10 [The Restricted Shell], page 99).

`-s`
: If this option is present, or if no arguments remain after option processing, then commands are read from the standard input. This option allows the positional parameters to be set when invoking an interactive shell or when reading input through a pipe.

`-D`
: A list of all double-quoted strings preceded by '`$`' is printed on the standard output. These are the strings that are subject to language translation when the current locale is not `C` or `POSIX` (see Section 3.1.2.5 [Locale Translation], page 7). This implies the `-n` option; no commands will be executed.

`[-+]O [`*`shopt_option`*`]`
: *shopt_option* is one of the shell options accepted by the `shopt` builtin (see Section 4.3.2 [The Shopt Builtin], page 65). If *shopt_option* is present, `-O` sets the value of that option; `+O` unsets it. If *shopt_option* is not supplied, the names and values of the shell options accepted by `shopt` are printed on the standard output. If the invocation option is `+O`, the output is displayed in a format that may be reused as input.

`--`
: A `--` signals the end of options and disables further option processing. Any arguments after the `--` are treated as filenames and arguments.

A *login* shell is one whose first character of argument zero is '-', or one invoked with the --login option.

An *interactive* shell is one started without non-option arguments, unless -s is specified, without specifying the -c option, and whose input and output are both connected to terminals (as determined by isatty(3)), or one started with the -i option. See Section 6.3 [Interactive Shells], page 88, for more information.

If arguments remain after option processing, and neither the -c nor the -s option has been supplied, the first argument is assumed to be the name of a file containing shell commands (see Section 3.8 [Shell Scripts], page 42). When Bash is invoked in this fashion, $0 is set to the name of the file, and the positional parameters are set to the remaining arguments. Bash reads and executes commands from this file, then exits. Bash's exit status is the exit status of the last command executed in the script. If no commands are executed, the exit status is 0.

6.2 Bash Startup Files

This section describes how Bash executes its startup files. If any of the files exist but cannot be read, Bash reports an error. Tildes are expanded in filenames as described above under Tilde Expansion (see Section 3.5.2 [Tilde Expansion], page 24).

Interactive shells are described in Section 6.3 [Interactive Shells], page 88.

Invoked as an interactive login shell, or with --login

When Bash is invoked as an interactive login shell, or as a non-interactive shell with the --login option, it first reads and executes commands from the file /etc/profile, if that file exists. After reading that file, it looks for ~/.bash_profile, ~/.bash_login, and ~/.profile, in that order, and reads and executes commands from the first one that exists and is readable. The --noprofile option may be used when the shell is started to inhibit this behavior.

When an interactive login shell exits, or a non-interactive login shell executes the exit builtin command, Bash reads and executes commands from the file ~/.bash_logout, if it exists.

Invoked as an interactive non-login shell

When an interactive shell that is not a login shell is started, Bash reads and executes commands from ~/.bashrc, if that file exists. This may be inhibited by using the --norc option. The --rcfile *file* option will force Bash to read and execute commands from *file* instead of ~/.bashrc.

So, typically, your ~/.bash_profile contains the line

```
if [ -f ~/.bashrc ]; then . ~/.bashrc; fi
```

after (or before) any login-specific initializations.

Invoked non-interactively

When Bash is started non-interactively, to run a shell script, for example, it looks for the variable BASH_ENV in the environment, expands its value if it appears there, and uses the expanded value as the name of a file to read and execute. Bash behaves as if the following command were executed:

```
if [ -n "$BASH_ENV" ]; then . "$BASH_ENV"; fi
```

but the value of the `PATH` variable is not used to search for the filename.

As noted above, if a non-interactive shell is invoked with the `--login` option, Bash attempts to read and execute commands from the login shell startup files.

Invoked with name `sh`

If Bash is invoked with the name `sh`, it tries to mimic the startup behavior of historical versions of `sh` as closely as possible, while conforming to the POSIX standard as well.

When invoked as an interactive login shell, or as a non-interactive shell with the `--login` option, it first attempts to read and execute commands from `/etc/profile` and `~/.profile`, in that order. The `--noprofile` option may be used to inhibit this behavior. When invoked as an interactive shell with the name `sh`, Bash looks for the variable `ENV`, expands its value if it is defined, and uses the expanded value as the name of a file to read and execute. Since a shell invoked as `sh` does not attempt to read and execute commands from any other startup files, the `--rcfile` option has no effect. A non-interactive shell invoked with the name `sh` does not attempt to read any other startup files.

When invoked as `sh`, Bash enters POSIX mode after the startup files are read.

Invoked in POSIX mode

When Bash is started in POSIX mode, as with the `--posix` command line option, it follows the POSIX standard for startup files. In this mode, interactive shells expand the `ENV` variable and commands are read and executed from the file whose name is the expanded value. No other startup files are read.

Invoked by remote shell daemon

Bash attempts to determine when it is being run with its standard input connected to a network connection, as when executed by the remote shell daemon, usually `rshd`, or the secure shell daemon `sshd`. If Bash determines it is being run in this fashion, it reads and executes commands from `~/.bashrc`, if that file exists and is readable. It will not do this if invoked as `sh`. The `--norc` option may be used to inhibit this behavior, and the `--rcfile` option may be used to force another file to be read, but neither `rshd` nor `sshd` generally invoke the shell with those options or allow them to be specified.

Invoked with unequal effective and real UID/GIDs

If Bash is started with the effective user (group) id not equal to the real user (group) id, and the `-p` option is not supplied, no startup files are read, shell functions are not inherited from the environment, the `SHELLOPTS`, `BASHOPTS`, `CDPATH`, and `GLOBIGNORE` variables, if they appear in the environment, are ignored, and the effective user id is set to the real user id. If the `-p` option is supplied at invocation, the startup behavior is the same, but the effective user id is not reset.

6.3 Interactive Shells

6.3.1 What is an Interactive Shell?

An interactive shell is one started without non-option arguments, unless -s is specified, without specifying the -c option, and whose input and error output are both connected to terminals (as determined by isatty(3)), or one started with the -i option.

An interactive shell generally reads from and writes to a user's terminal.

The -s invocation option may be used to set the positional parameters when an interactive shell is started.

6.3.2 Is this Shell Interactive?

To determine within a startup script whether or not Bash is running interactively, test the value of the '-' special parameter. It contains i when the shell is interactive. For example:

```
case "$-" in
*i*) echo This shell is interactive ;;
*) echo This shell is not interactive ;;
esac
```

Alternatively, startup scripts may examine the variable PS1; it is unset in non-interactive shells, and set in interactive shells. Thus:

```
if [ -z "$PS1" ]; then
        echo This shell is not interactive
else
        echo This shell is interactive
fi
```

6.3.3 Interactive Shell Behavior

When the shell is running interactively, it changes its behavior in several ways.

1. Startup files are read and executed as described in Section 6.2 [Bash Startup Files], page 87.
2. Job Control (see Chapter 7 [Job Control], page 104) is enabled by default. When job control is in effect, Bash ignores the keyboard-generated job control signals SIGTTIN, SIGTTOU, and SIGTSTP.
3. Bash expands and displays PS1 before reading the first line of a command, and expands and displays PS2 before reading the second and subsequent lines of a multi-line command. Bash expands and displays PS0 after it reads a command but before executing it. See Section 6.9 [Controlling the Prompt], page 97, for a complete list of prompt string escape sequences.
4. Bash executes the value of the PROMPT_COMMAND variable as a command before printing the primary prompt, $PS1 (see Section 5.2 [Bash Variables], page 73).
5. Readline (see Chapter 8 [Command Line Editing], page 108) is used to read commands from the user's terminal.
6. Bash inspects the value of the ignoreeof option to set -o instead of exiting immediately when it receives an EOF on its standard input when reading a command (see Section 4.3.1 [The Set Builtin], page 61).
7. Command history (see Section 9.1 [Bash History Facilities], page 143) and history expansion (see Section 9.3 [History Interaction], page 145) are enabled by default.

Bash will save the command history to the file named by `$HISTFILE` when a shell with history enabled exits.

8. Alias expansion (see Section 6.6 [Aliases], page 93) is performed by default.
9. In the absence of any traps, Bash ignores `SIGTERM` (see Section 3.7.6 [Signals], page 41).
10. In the absence of any traps, `SIGINT` is caught and handled (see Section 3.7.6 [Signals], page 41). `SIGINT` will interrupt some shell builtins.
11. An interactive login shell sends a `SIGHUP` to all jobs on exit if the `huponexit` shell option has been enabled (see Section 3.7.6 [Signals], page 41).
12. The `-n` invocation option is ignored, and '`set -n`' has no effect (see Section 4.3.1 [The Set Builtin], page 61).
13. Bash will check for mail periodically, depending on the values of the `MAIL`, `MAILPATH`, and `MAILCHECK` shell variables (see Section 5.2 [Bash Variables], page 73).
14. Expansion errors due to references to unbound shell variables after '`set -u`' has been enabled will not cause the shell to exit (see Section 4.3.1 [The Set Builtin], page 61).
15. The shell will not exit on expansion errors caused by *var* being unset or null in `${var:?word}` expansions (see Section 3.5.3 [Shell Parameter Expansion], page 24).
16. Redirection errors encountered by shell builtins will not cause the shell to exit.
17. When running in POSIX mode, a special builtin returning an error status will not cause the shell to exit (see Section 6.11 [Bash POSIX Mode], page 99).
18. A failed `exec` will not cause the shell to exit (see Section 4.1 [Bourne Shell Builtins], page 43).
19. Parser syntax errors will not cause the shell to exit.
20. Simple spelling correction for directory arguments to the `cd` builtin is enabled by default (see the description of the `cdspell` option to the `shopt` builtin in Section 4.3.2 [The Shopt Builtin], page 65).
21. The shell will check the value of the `TMOUT` variable and exit if a command is not read within the specified number of seconds after printing `$PS1` (see Section 5.2 [Bash Variables], page 73).

6.4 Bash Conditional Expressions

Conditional expressions are used by the `[[` compound command and the `test` and `[` builtin commands. The `test` and `[` commands determine their behavior based on the number of arguments; see the descriptions of those commands for any other command-specific actions.

Expressions may be unary or binary, and are formed from the following primaries. Unary expressions are often used to examine the status of a file. There are string operators and numeric comparison operators as well. Bash handles several filenames specially when they are used in expressions. If the operating system on which Bash is running provides these special files, Bash will use them; otherwise it will emulate them internally with this behavior: If the *file* argument to one of the primaries is of the form `/dev/fd/N`, then file descriptor *N* is checked. If the *file* argument to one of the primaries is one of `/dev/stdin`, `/dev/stdout`, or `/dev/stderr`, file descriptor 0, 1, or 2, respectively, is checked.

When used with `[[`, the '`<`' and '`>`' operators sort lexicographically using the current locale. The `test` command uses ASCII ordering.

Unless otherwise specified, primaries that operate on files follow symbolic links and operate on the target of the link, rather than the link itself.

`-a` *file* True if *file* exists.

`-b` *file* True if *file* exists and is a block special file.

`-c` *file* True if *file* exists and is a character special file.

`-d` *file* True if *file* exists and is a directory.

`-e` *file* True if *file* exists.

`-f` *file* True if *file* exists and is a regular file.

`-g` *file* True if *file* exists and its set-group-id bit is set.

`-h` *file* True if *file* exists and is a symbolic link.

`-k` *file* True if *file* exists and its "sticky" bit is set.

`-p` *file* True if *file* exists and is a named pipe (FIFO).

`-r` *file* True if *file* exists and is readable.

`-s` *file* True if *file* exists and has a size greater than zero.

`-t` *fd* True if file descriptor *fd* is open and refers to a terminal.

`-u` *file* True if *file* exists and its set-user-id bit is set.

`-w` *file* True if *file* exists and is writable.

`-x` *file* True if *file* exists and is executable.

`-G` *file* True if *file* exists and is owned by the effective group id.

`-L` *file* True if *file* exists and is a symbolic link.

`-N` *file* True if *file* exists and has been modified since it was last read.

`-O` *file* True if *file* exists and is owned by the effective user id.

`-S` *file* True if *file* exists and is a socket.

file1 `-ef` *file2*
True if *file1* and *file2* refer to the same device and inode numbers.

file1 `-nt` *file2*
True if *file1* is newer (according to modification date) than *file2*, or if *file1* exists and *file2* does not.

file1 `-ot` *file2*
True if *file1* is older than *file2*, or if *file2* exists and *file1* does not.

`-o` *optname*
True if the shell option *optname* is enabled. The list of options appears in the description of the `-o` option to the `set` builtin (see Section 4.3.1 [The Set Builtin], page 61).

`-v` *varname*
True if the shell variable *varname* is set (has been assigned a value).

`-R` *varname*
: True if the shell variable *varname* is set and is a name reference.

`-z` *string* True if the length of *string* is zero.

`-n` *string*
string True if the length of *string* is non-zero.

string1 `==` *string2*
string1 `=` *string2*
: True if the strings are equal. When used with the `[[` command, this performs pattern matching as described above (see Section 3.2.4.2 [Conditional Constructs], page 11).

  '`=`' should be used with the `test` command for POSIX conformance.

string1 `!=` *string2*
: True if the strings are not equal.

string1 `<` *string2*
: True if *string1* sorts before *string2* lexicographically.

string1 `>` *string2*
: True if *string1* sorts after *string2* lexicographically.

arg1 `OP` *arg2*
: `OP` is one of '`-eq`', '`-ne`', '`-lt`', '`-le`', '`-gt`', or '`-ge`'. These arithmetic binary operators return true if *arg1* is equal to, not equal to, less than, less than or equal to, greater than, or greater than or equal to *arg2*, respectively. *Arg1* and *arg2* may be positive or negative integers. When used with the `[[` command, *Arg1* and *Arg2* are evaluated as arithmetic expressions (see Section 6.5 [Shell Arithmetic], page 92).

6.5 Shell Arithmetic

The shell allows arithmetic expressions to be evaluated, as one of the shell expansions or by using the `((` compound command, the `let` builtin, or the `-i` option to the `declare` builtin.

Evaluation is done in fixed-width integers with no check for overflow, though division by 0 is trapped and flagged as an error. The operators and their precedence, associativity, and values are the same as in the C language. The following list of operators is grouped into levels of equal-precedence operators. The levels are listed in order of decreasing precedence.

id`++` *id*`--` variable post-increment and post-decrement

`++`*id* `--`*id* variable pre-increment and pre-decrement

`- +` unary minus and plus

`! ~` logical and bitwise negation

`**` exponentiation

`* / %` multiplication, division, remainder

`+ -` addition, subtraction

`<< >>` left and right bitwise shifts

`<= >= < >` comparison

`== !=` equality and inequality

`&` bitwise AND

`^` bitwise exclusive OR

`|` bitwise OR

`&&` logical AND

`||` logical OR

`expr ? expr : expr`
conditional operator

`= *= /= %= += -= <<= >>= &= ^= |=`
assignment

`expr1 , expr2`
comma

Shell variables are allowed as operands; parameter expansion is performed before the expression is evaluated. Within an expression, shell variables may also be referenced by name without using the parameter expansion syntax. A shell variable that is null or unset evaluates to 0 when referenced by name without using the parameter expansion syntax. The value of a variable is evaluated as an arithmetic expression when it is referenced, or when a variable which has been given the *integer* attribute using ‘`declare -i`’ is assigned a value. A null value evaluates to 0. A shell variable need not have its *integer* attribute turned on to be used in an expression.

Constants with a leading 0 are interpreted as octal numbers. A leading ‘`0x`’ or ‘`0X`’ denotes hexadecimal. Otherwise, numbers take the form [*base*`#`]*n*, where the optional *base* is a decimal number between 2 and 64 representing the arithmetic base, and *n* is a number in that base. If *base*`#` is omitted, then base 10 is used. When specifying *n*, the digits greater than 9 are represented by the lowercase letters, the uppercase letters, ‘`@`’, and ‘`_`’, in that order. If *base* is less than or equal to 36, lowercase and uppercase letters may be used interchangeably to represent numbers between 10 and 35.

Operators are evaluated in order of precedence. Sub-expressions in parentheses are evaluated first and may override the precedence rules above.

6.6 Aliases

Aliases allow a string to be substituted for a word when it is used as the first word of a simple command. The shell maintains a list of aliases that may be set and unset with the `alias` and `unalias` builtin commands.

The first word of each simple command, if unquoted, is checked to see if it has an alias. If so, that word is replaced by the text of the alias. The characters ‘`/`’, ‘`$`’, ‘`` ` ``’, ‘`=`’ and any of the shell metacharacters or quoting characters listed above may not appear in an alias name. The replacement text may contain any valid shell input, including shell metacharacters. The first word of the replacement text is tested for aliases, but a word that is identical to an alias being expanded is not expanded a second time. This means that one may alias `ls` to `"ls -F"`, for instance, and Bash does not try to recursively expand the replacement text. If the last character of the alias value is a *blank*, then the next command word following the alias is also checked for alias expansion.

Aliases are created and listed with the `alias` command, and removed with the `unalias` command.

There is no mechanism for using arguments in the replacement text, as in `csh`. If arguments are needed, a shell function should be used (see Section 3.3 [Shell Functions], page 17).

Aliases are not expanded when the shell is not interactive, unless the `expand_aliases` shell option is set using `shopt` (see Section 4.3.2 [The Shopt Builtin], page 65).

The rules concerning the definition and use of aliases are somewhat confusing. Bash always reads at least one complete line of input, and all lines that make up a compound command, before executing any of the commands on that line or the compound command. Aliases are expanded when a command is read, not when it is executed. Therefore, an alias definition appearing on the same line as another command does not take effect until the next line of input is read. The commands following the alias definition on that line are not affected by the new alias. This behavior is also an issue when functions are executed. Aliases are expanded when a function definition is read, not when the function is executed, because a function definition is itself a command. As a consequence, aliases defined in a function are not available until after that function is executed. To be safe, always put alias definitions on a separate line, and do not use `alias` in compound commands.

For almost every purpose, shell functions are preferred over aliases.

6.7 Arrays

Bash provides one-dimensional indexed and associative array variables. Any variable may be used as an indexed array; the `declare` builtin will explicitly declare an array. There is no maximum limit on the size of an array, nor any requirement that members be indexed or assigned contiguously. Indexed arrays are referenced using integers (including arithmetic expressions (see Section 6.5 [Shell Arithmetic], page 92)) and are zero-based; associative arrays use arbitrary strings. Unless otherwise noted, indexed array indices must be non-negative integers.

An indexed array is created automatically if any variable is assigned to using the syntax

```
name[subscript]=value
```

The *subscript* is treated as an arithmetic expression that must evaluate to a number. To explicitly declare an array, use

```
declare -a name
```

The syntax

```
declare -a name[subscript]
```

is also accepted; the *subscript* is ignored.

Associative arrays are created using

```
declare -A name
```

Attributes may be specified for an array variable using the `declare` and `readonly` builtins. Each attribute applies to all members of an array.

Arrays are assigned to using compound assignments of the form

```
name=(value1 value2 ... )
```

where each *value* is of the form `[subscript]=`*string*. Indexed array assignments do not require anything but *string*. When assigning to indexed arrays, if the optional subscript is supplied, that index is assigned to; otherwise the index of the element assigned is the last index assigned to by the statement plus one. Indexing starts at zero.

When assigning to an associative array, the subscript is required.

This syntax is also accepted by the `declare` builtin. Individual array elements may be assigned to using the `name[subscript]=value` syntax introduced above.

When assigning to an indexed array, if *name* is subscripted by a negative number, that number is interpreted as relative to one greater than the maximum index of *name*, so negative indices count back from the end of the array, and an index of -1 references the last element.

Any element of an array may be referenced using `${name[subscript]}`. The braces are required to avoid conflicts with the shell's filename expansion operators. If the *subscript* is '`@`' or '`*`', the word expands to all members of the array *name*. These subscripts differ only when the word appears within double quotes. If the word is double-quoted, `${name[*]}` expands to a single word with the value of each array member separated by the first character of the `IFS` variable, and `${name[@]}` expands each element of *name* to a separate word. When there are no array members, `${name[@]}` expands to nothing. If the double-quoted expansion occurs within a word, the expansion of the first parameter is joined with the beginning part of the original word, and the expansion of the last parameter is joined with the last part of the original word. This is analogous to the expansion of the special parameters '`@`' and '`*`'. `${#name[subscript]}` expands to the length of `${name[subscript]}`. If *subscript* is '`@`' or '`*`', the expansion is the number of elements in the array. If the *subscript* used to reference an element of an indexed array evaluates to a number less than zero, it is interpreted as relative to one greater than the maximum index of the array, so negative indices count back from the end of the array, and an index of -1 refers to the last element.

Referencing an array variable without a subscript is equivalent to referencing with a subscript of 0. Any reference to a variable using a valid subscript is legal, and `bash` will create an array if necessary.

An array variable is considered set if a subscript has been assigned a value. The null string is a valid value.

It is possible to obtain the keys (indices) of an array as well as the values. ${!*name*[@]} and ${!*name*[*]} expand to the indices assigned in array variable *name*. The treatment when in double quotes is similar to the expansion of the special parameters '`@`' and '`*`' within double quotes.

The `unset` builtin is used to destroy arrays. `unset name[subscript]` destroys the array element at index *subscript*. Negative subscripts to indexed arrays are interpreted as described above. Unsetting the last element of an array variable does not unset the variable. `unset name`, where *name* is an array, removes the entire array. A subscript of '`*`' or '`@`' also removes the entire array.

When using a variable name with a subscript as an argument to a command, such as with `unset`, without using the word expansion syntax described above, the argument is subject to the shell's filename expansion. If filename expansion is not desired, the argument should be quoted.

The `declare`, `local`, and `readonly` builtins each accept a `-a` option to specify an indexed array and a `-A` option to specify an associative array. If both options are supplied, `-A` takes precedence. The `read` builtin accepts a `-a` option to assign a list of words read from the standard input to an array, and can read values from the standard input into individual array elements. The `set` and `declare` builtins display array values in a way that allows them to be reused as input.

6.8 The Directory Stack

The directory stack is a list of recently-visited directories. The `pushd` builtin adds directories to the stack as it changes the current directory, and the `popd` builtin removes specified directories from the stack and changes the current directory to the directory removed. The `dirs` builtin displays the contents of the directory stack. The current directory is always the "top" of the directory stack.

The contents of the directory stack are also visible as the value of the `DIRSTACK` shell variable.

6.8.1 Directory Stack Builtins

`dirs`

```
dirs [-clpv] [+N | -N]
```

Display the list of currently remembered directories. Directories are added to the list with the `pushd` command; the `popd` command removes directories from the list. The current directory is always the first directory in the stack.

`-c` Clears the directory stack by deleting all of the elements.

`-l` Produces a listing using full pathnames; the default listing format uses a tilde to denote the home directory.

`-p` Causes `dirs` to print the directory stack with one entry per line.

`-v` Causes `dirs` to print the directory stack with one entry per line, prefixing each entry with its index in the stack.

`+N` Displays the *N*th directory (counting from the left of the list printed by `dirs` when invoked without options), starting with zero.

`-N` Displays the *N*th directory (counting from the right of the list printed by `dirs` when invoked without options), starting with zero.

`popd`

```
popd [-n] [+N | -N]
```

When no arguments are given, `popd` removes the top directory from the stack and performs a `cd` to the new top directory. The elements are numbered from

0 starting at the first directory listed with `dirs`; that is, `popd` is equivalent to `popd +0`.

`-n` Suppresses the normal change of directory when removing directories from the stack, so that only the stack is manipulated.

`+N` Removes the *N*th directory (counting from the left of the list printed by `dirs`), starting with zero.

`-N` Removes the *N*th directory (counting from the right of the list printed by `dirs`), starting with zero.

`pushd`

```
pushd [-n] [+N | -N | dir]
```

Save the current directory on the top of the directory stack and then `cd` to *dir*. With no arguments, `pushd` exchanges the top two directories and makes the new top the current directory.

`-n` Suppresses the normal change of directory when rotating or adding directories to the stack, so that only the stack is manipulated.

`+N` Brings the *N*th directory (counting from the left of the list printed by `dirs`, starting with zero) to the top of the list by rotating the stack.

`-N` Brings the *N*th directory (counting from the right of the list printed by `dirs`, starting with zero) to the top of the list by rotating the stack.

`dir` Makes *dir* be the top of the stack, making it the new current directory as if it had been supplied as an argument to the `cd` builtin.

6.9 Controlling the Prompt

The value of the variable `PROMPT_COMMAND` is examined just before Bash prints each primary prompt. If `PROMPT_COMMAND` is set and has a non-null value, then the value is executed just as if it had been typed on the command line.

In addition, the following table describes the special characters which can appear in the prompt variables `PS0`, `PS1`, `PS2`, and `PS4`:

`\a` A bell character.

`\d` The date, in "Weekday Month Date" format (e.g., "Tue May 26").

`\D{format}`
The *format* is passed to `strftime`(3) and the result is inserted into the prompt string; an empty *format* results in a locale-specific time representation. The braces are required.

`\e` An escape character.

`\h` The hostname, up to the first '.'.

`\H` The hostname.

`\j` The number of jobs currently managed by the shell.

`\l` The basename of the shell's terminal device name.

`\n` A newline.

`\r` A carriage return.

`\s` The name of the shell, the basename of `$0` (the portion following the final slash).

`\t` The time, in 24-hour HH:MM:SS format.

`\T` The time, in 12-hour HH:MM:SS format.

`\@` The time, in 12-hour am/pm format.

`\A` The time, in 24-hour HH:MM format.

`\u` The username of the current user.

`\v` The version of Bash (e.g., 2.00)

`\V` The release of Bash, version + patchlevel (e.g., 2.00.0)

`\w` The current working directory, with `$HOME` abbreviated with a tilde (uses the `$PROMPT_DIRTRIM` variable).

`\W` The basename of `$PWD`, with `$HOME` abbreviated with a tilde.

`\!` The history number of this command.

`\#` The command number of this command.

`\$` If the effective uid is 0, `#`, otherwise `$`.

`\`*nnn* The character whose ASCII code is the octal value *nnn*.

`\\` A backslash.

`\[` Begin a sequence of non-printing characters. This could be used to embed a terminal control sequence into the prompt.

`\]` End a sequence of non-printing characters.

The command number and the history number are usually different: the history number of a command is its position in the history list, which may include commands restored from the history file (see Section 9.1 [Bash History Facilities], page 143), while the command number is the position in the sequence of commands executed during the current shell session.

After the string is decoded, it is expanded via parameter expansion, command substitution, arithmetic expansion, and quote removal, subject to the value of the `promptvars` shell option (see Section 4.3.2 [The Shopt Builtin], page 65). This can have unwanted side effects if escaped portions of the string appear within command substitution or contain characters special to word expansion.

6.10 The Restricted Shell

If Bash is started with the name `rbash`, or the `--restricted` or `-r` option is supplied at invocation, the shell becomes restricted. A restricted shell is used to set up an environment more controlled than the standard shell. A restricted shell behaves identically to `bash` with the exception that the following are disallowed or not performed:

- Changing directories with the `cd` builtin.
- Setting or unsetting the values of the `SHELL`, `PATH`, `ENV`, or `BASH_ENV` variables.
- Specifying command names containing slashes.
- Specifying a filename containing a slash as an argument to the `.` builtin command.
- Specifying a filename containing a slash as an argument to the `-p` option to the `hash` builtin command.
- Importing function definitions from the shell environment at startup.
- Parsing the value of `SHELLOPTS` from the shell environment at startup.
- Redirecting output using the '`>`', '`>|`', '`<>`', '`>&`', '`&>`', and '`>>`' redirection operators.
- Using the `exec` builtin to replace the shell with another command.
- Adding or deleting builtin commands with the `-f` and `-d` options to the `enable` builtin.
- Using the `enable` builtin command to enable disabled shell builtins.
- Specifying the `-p` option to the `command` builtin.
- Turning off restricted mode with '`set +r`' or '`set +o restricted`'.

These restrictions are enforced after any startup files are read.

When a command that is found to be a shell script is executed (see Section 3.8 [Shell Scripts], page 42), `rbash` turns off any restrictions in the shell spawned to execute the script.

The restricted shell mode is only one component of a useful restricted environment. It should be accompanied by setting `PATH` to a value that allows execution of only a few verified commands (commands that allow shell escapes are particularly vulnerable), leaving the user in a non-writable directory other than his home directory after login, not allowing the restricted shell to execute shell scripts, and cleaning the environment of variables that cause some commands to modify their behavior (e.g., `VISUAL` or `PAGER`).

Modern systems provide more secure ways to implement a restricted environment, such as `jails`, `zones`, or `containers`.

6.11 Bash POSIX Mode

Starting Bash with the `--posix` command-line option or executing '`set -o posix`' while Bash is running will cause Bash to conform more closely to the POSIX standard by changing the behavior to match that specified by POSIX in areas where the Bash default differs.

When invoked as `sh`, Bash enters POSIX mode after reading the startup files.

The following list is what's changed when 'POSIX mode' is in effect:

1. Bash ensures that the `POSIXLY_CORRECT` variable is set.
2. When a command in the hash table no longer exists, Bash will re-search `$PATH` to find the new location. This is also available with '`shopt -s checkhash`'.

3. The message printed by the job control code and builtins when a job exits with a non-zero status is 'Done(status)'.
4. The message printed by the job control code and builtins when a job is stopped is 'Stopped(*signame*)', where *signame* is, for example, `SIGTSTP`.
5. Alias expansion is always enabled, even in non-interactive shells.
6. Reserved words appearing in a context where reserved words are recognized do not undergo alias expansion.
7. The POSIX `PS1` and `PS2` expansions of '`!`' to the history number and '`!!`' to '`!`' are enabled, and parameter expansion is performed on the values of `PS1` and `PS2` regardless of the setting of the `promptvars` option.
8. The POSIX startup files are executed (`$ENV`) rather than the normal Bash files.
9. Tilde expansion is only performed on assignments preceding a command name, rather than on all assignment statements on the line.
10. The default history file is `~/.sh_history` (this is the default value of `$HISTFILE`).
11. Redirection operators do not perform filename expansion on the word in the redirection unless the shell is interactive.
12. Redirection operators do not perform word splitting on the word in the redirection.
13. Function names must be valid shell `name`s. That is, they may not contain characters other than letters, digits, and underscores, and may not start with a digit. Declaring a function with an invalid name causes a fatal syntax error in non-interactive shells.
14. Function names may not be the same as one of the POSIX special builtins.
15. POSIX special builtins are found before shell functions during command lookup.
16. When printing shell function definitions (e.g., by `type`), Bash does not print the `function` keyword.
17. Literal tildes that appear as the first character in elements of the `PATH` variable are not expanded as described above under Section 3.5.2 [Tilde Expansion], page 24.
18. The `time` reserved word may be used by itself as a command. When used in this way, it displays timing statistics for the shell and its completed children. The `TIMEFORMAT` variable controls the format of the timing information.
19. When parsing and expanding a ${...} expansion that appears within double quotes, single quotes are no longer special and cannot be used to quote a closing brace or other special character, unless the operator is one of those defined to perform pattern removal. In this case, they do not have to appear as matched pairs.
20. The parser does not recognize `time` as a reserved word if the next token begins with a '`-`'.
21. The '`!`' character does not introduce history expansion within a double-quoted string, even if the `histexpand` option is enabled.
22. If a POSIX special builtin returns an error status, a non-interactive shell exits. The fatal errors are those listed in the POSIX standard, and include things like passing incorrect options, redirection errors, variable assignment errors for assignments preceding the command name, and so on.

23. A non-interactive shell exits with an error status if a variable assignment error occurs when no command name follows the assignment statements. A variable assignment error occurs, for example, when trying to assign a value to a readonly variable.
24. A non-interactive shell exits with an error status if a variable assignment error occurs in an assignment statement preceding a special builtin, but not with any other simple command.
25. A non-interactive shell exits with an error status if the iteration variable in a `for` statement or the selection variable in a `select` statement is a readonly variable.
26. Non-interactive shells exit if *filename* in `.` *filename* is not found.
27. Non-interactive shells exit if a syntax error in an arithmetic expansion results in an invalid expression.
28. Non-interactive shells exit if a parameter expansion error occurs.
29. Non-interactive shells exit if there is a syntax error in a script read with the `.` or `source` builtins, or in a string processed by the `eval` builtin.
30. Process substitution is not available.
31. While variable indirection is available, it may not be applied to the '`#`' and '`?`' special parameters.
32. When expanding the '`*`' special parameter in a pattern context where the expansion is double-quoted does not treat the `$*` as if it were double-quoted.
33. Assignment statements preceding POSIX special builtins persist in the shell environment after the builtin completes.
34. Assignment statements preceding shell function calls persist in the shell environment after the function returns, as if a POSIX special builtin command had been executed.
35. The `command` builtin does not prevent builtins that take assignment statements as arguments from expanding them as assignment statements; when not in POSIX mode, assignment builtins lose their assignment statement expansion properties when preceded by `command`.
36. The `bg` builtin uses the required format to describe each job placed in the background, which does not include an indication of whether the job is the current or previous job.
37. The output of '`kill -l`' prints all the signal names on a single line, separated by spaces, without the '`SIG`' prefix.
38. The `kill` builtin does not accept signal names with a '`SIG`' prefix.
39. The `export` and `readonly` builtin commands display their output in the format required by POSIX.
40. The `trap` builtin displays signal names without the leading `SIG`.
41. The `trap` builtin doesn't check the first argument for a possible signal specification and revert the signal handling to the original disposition if it is, unless that argument consists solely of digits and is a valid signal number. If users want to reset the handler for a given signal to the original disposition, they should use '`-`' as the first argument.
42. The `.` and `source` builtins do not search the current directory for the filename argument if it is not found by searching `PATH`.
43. Enabling POSIX mode has the effect of setting the `inherit_errexit` option, so subshells spawned to execute command substitutions inherit the value of the `-e` option from the

parent shell. When the `inherit_errexit` option is not enabled, Bash clears the `-e` option in such subshells.

44. Enabling POSIX mode has the effect of setting the `shift_verbose` option, so numeric arguments to `shift` that exceed the number of positional parameters will result in an error message.
45. When the `alias` builtin displays alias definitions, it does not display them with a leading '`alias` ' unless the `-p` option is supplied.
46. When the `set` builtin is invoked without options, it does not display shell function names and definitions.
47. When the `set` builtin is invoked without options, it displays variable values without quotes, unless they contain shell metacharacters, even if the result contains nonprinting characters.
48. When the `cd` builtin is invoked in *logical* mode, and the pathname constructed from `$PWD` and the directory name supplied as an argument does not refer to an existing directory, `cd` will fail instead of falling back to *physical* mode.
49. When the `cd` builtin cannot change a directory because the length of the pathname constructed from `$PWD` and the directory name supplied as an argument exceeds *PATH_MAX* when all symbolic links are expanded, `cd` will fail instead of attempting to use only the supplied directory name.
50. The `pwd` builtin verifies that the value it prints is the same as the current directory, even if it is not asked to check the file system with the `-P` option.
51. When listing the history, the `fc` builtin does not include an indication of whether or not a history entry has been modified.
52. The default editor used by `fc` is `ed`.
53. The `type` and `command` builtins will not report a non-executable file as having been found, though the shell will attempt to execute such a file if it is the only so-named file found in `$PATH`.
54. The `vi` editing mode will invoke the `vi` editor directly when the '`v`' command is run, instead of checking `$VISUAL` and `$EDITOR`.
55. When the `xpg_echo` option is enabled, Bash does not attempt to interpret any arguments to `echo` as options. Each argument is displayed, after escape characters are converted.
56. The `ulimit` builtin uses a block size of 512 bytes for the `-c` and `-f` options.
57. The arrival of `SIGCHLD` when a trap is set on `SIGCHLD` does not interrupt the `wait` builtin and cause it to return immediately. The trap command is run once for each child that exits.
58. The `read` builtin may be interrupted by a signal for which a trap has been set. If Bash receives a trapped signal while executing `read`, the trap handler executes and `read` returns an exit status greater than 128.
59. Bash removes an exited background process's status from the list of such statuses after the `wait` builtin is used to obtain it.

There is other POSIX behavior that Bash does not implement by default even when in POSIX mode. Specifically:

1. The fc builtin checks $EDITOR as a program to edit history entries if FCEDIT is unset, rather than defaulting directly to ed. fc uses ed if EDITOR is unset.
2. As noted above, Bash requires the xpg_echo option to be enabled for the echo builtin to be fully conformant.

Bash can be configured to be POSIX-conformant by default, by specifying the --enable-strict-posix-default to configure when building (see Section 10.8 [Optional Features], page 152).

7 Job Control

This chapter discusses what job control is, how it works, and how Bash allows you to access its facilities.

7.1 Job Control Basics

Job control refers to the ability to selectively stop (suspend) the execution of processes and continue (resume) their execution at a later point. A user typically employs this facility via an interactive interface supplied jointly by the operating system kernel's terminal driver and Bash.

The shell associates a *job* with each pipeline. It keeps a table of currently executing jobs, which may be listed with the `jobs` command. When Bash starts a job asynchronously, it prints a line that looks like:

```
[1] 25647
```

indicating that this job is job number 1 and that the process ID of the last process in the pipeline associated with this job is 25647. All of the processes in a single pipeline are members of the same job. Bash uses the *job* abstraction as the basis for job control.

To facilitate the implementation of the user interface to job control, the operating system maintains the notion of a current terminal process group ID. Members of this process group (processes whose process group ID is equal to the current terminal process group ID) receive keyboard-generated signals such as `SIGINT`. These processes are said to be in the foreground. Background processes are those whose process group ID differs from the terminal's; such processes are immune to keyboard-generated signals. Only foreground processes are allowed to read from or, if the user so specifies with `stty tostop`, write to the terminal. Background processes which attempt to read from (write to when `stty tostop` is in effect) the terminal are sent a `SIGTTIN` (`SIGTTOU`) signal by the kernel's terminal driver, which, unless caught, suspends the process.

If the operating system on which Bash is running supports job control, Bash contains facilities to use it. Typing the *suspend* character (typically '`^Z`', Control-Z) while a process is running causes that process to be stopped and returns control to Bash. Typing the *delayed suspend* character (typically '`^Y`', Control-Y) causes the process to be stopped when it attempts to read input from the terminal, and control to be returned to Bash. The user then manipulates the state of this job, using the `bg` command to continue it in the background, the `fg` command to continue it in the foreground, or the `kill` command to kill it. A '`^Z`' takes effect immediately, and has the additional side effect of causing pending output and typeahead to be discarded.

There are a number of ways to refer to a job in the shell. The character '`%`' introduces a job specification (*jobspec*).

Job number `n` may be referred to as '`%n`'. The symbols '`%%`' and '`%+`' refer to the shell's notion of the current job, which is the last job stopped while it was in the foreground or started in the background. A single '`%`' (with no accompanying job specification) also refers to the current job. The previous job may be referenced using '`%-`'. If there is only a single job, '`%+`' and '`%-`' can both be used to refer to that job. In output pertaining to jobs (e.g., the output of the `jobs` command), the current job is always flagged with a '`+`', and the previous job with a '`-`'.

A job may also be referred to using a prefix of the name used to start it, or using a substring that appears in its command line. For example, '`%ce`' refers to a stopped `ce` job. Using '`%?ce`', on the other hand, refers to any job containing the string '`ce`' in its command line. If the prefix or substring matches more than one job, Bash reports an error.

Simply naming a job can be used to bring it into the foreground: '`%1`' is a synonym for '`fg %1`', bringing job 1 from the background into the foreground. Similarly, '`%1 &`' resumes job 1 in the background, equivalent to '`bg %1`'

The shell learns immediately whenever a job changes state. Normally, Bash waits until it is about to print a prompt before reporting changes in a job's status so as to not interrupt any other output. If the `-b` option to the `set` builtin is enabled, Bash reports such changes immediately (see Section 4.3.1 [The Set Builtin], page 61). Any trap on `SIGCHLD` is executed for each child process that exits.

If an attempt to exit Bash is made while jobs are stopped, (or running, if the `checkjobs` option is enabled – see Section 4.3.2 [The Shopt Builtin], page 65), the shell prints a warning message, and if the `checkjobs` option is enabled, lists the jobs and their statuses. The `jobs` command may then be used to inspect their status. If a second attempt to exit is made without an intervening command, Bash does not print another warning, and any stopped jobs are terminated.

When the shell is waiting for a job or process using the `wait` builtin, and job control is enabled, `wait` will return when the job changes state. The `-f` option causes `wait` to wait until the job or process terminates before returning.

7.2 Job Control Builtins

`bg`

```
bg [jobspec ...]
```

Resume each suspended job *jobspec* in the background, as if it had been started with '`&`'. If *jobspec* is not supplied, the current job is used. The return status is zero unless it is run when job control is not enabled, or, when run with job control enabled, any *jobspec* was not found or specifies a job that was started without job control.

`fg`

```
fg [jobspec]
```

Resume the job *jobspec* in the foreground and make it the current job. If *jobspec* is not supplied, the current job is used. The return status is that of the command placed into the foreground, or non-zero if run when job control is disabled or, when run with job control enabled, *jobspec* does not specify a valid job or *jobspec* specifies a job that was started without job control.

`jobs`

```
jobs [-lnprs] [jobspec]
jobs -x command [arguments]
```

The first form lists the active jobs. The options have the following meanings:

`-l` List process IDs in addition to the normal information.

`-n` Display information only about jobs that have changed status since the user was last notified of their status.

`-p` List only the process ID of the job's process group leader.

`-r` Display only running jobs.

`-s` Display only stopped jobs.

If *jobspec* is given, output is restricted to information about that job. If *jobspec* is not supplied, the status of all jobs is listed.

If the `-x` option is supplied, `jobs` replaces any *jobspec* found in *command* or *arguments* with the corresponding process group ID, and executes *command*, passing it *arguments*, returning its exit status.

`kill`

```
kill [-s sigspec] [-n signum] [-sigspec] jobspec or pid
kill -l|-L [exit_status]
```

Send a signal specified by *sigspec* or *signum* to the process named by job specification *jobspec* or process ID *pid*. *sigspec* is either a case-insensitive signal name such as `SIGINT` (with or without the `SIG` prefix) or a signal number; *signum* is a signal number. If *sigspec* and *signum* are not present, `SIGTERM` is used. The `-l` option lists the signal names. If any arguments are supplied when `-l` is given, the names of the signals corresponding to the arguments are listed, and the return status is zero. *exit_status* is a number specifying a signal number or the exit status of a process terminated by a signal. The `-L` option is equivalent to `-l`. The return status is zero if at least one signal was successfully sent, or non-zero if an error occurs or an invalid option is encountered.

`wait`

```
wait [-fn] [jobspec or pid ...]
```

Wait until the child process specified by each process ID *pid* or job specification *jobspec* exits and return the exit status of the last command waited for. If a job spec is given, all processes in the job are waited for. If no arguments are given, all currently active child processes are waited for, and the return status is zero. If the `-n` option is supplied, `wait` waits for a single job to terminate and returns its exit status. Supplying the `-f` option, when job control is enabled, forces `wait` to wait for each *pid* or *jobspec* to terminate before returning its status, intead of returning when it changes status. If neither *jobspec* nor *pid* specifies an active child process of the shell, the return status is 127.

`disown`

```
disown [-ar] [-h] [jobspec ... | pid ... ]
```

Without options, remove each *jobspec* from the table of active jobs. If the `-h` option is given, the job is not removed from the table, but is marked so that `SIGHUP` is not sent to the job if the shell receives a `SIGHUP`. If *jobspec* is not present, and neither the `-a` nor the `-r` option is supplied, the current job is used. If no *jobspec* is supplied, the `-a` option means to remove or mark all jobs; the `-r` option without a *jobspec* argument restricts operation to running jobs.

`suspend`

    suspend [-f]

Suspend the execution of this shell until it receives a `SIGCONT` signal. A login shell cannot be suspended; the `-f` option can be used to override this and force the suspension.

When job control is not active, the `kill` and `wait` builtins do not accept *jobspec* arguments. They must be supplied process IDs.

7.3 Job Control Variables

`auto_resume`

This variable controls how the shell interacts with the user and job control. If this variable exists then single word simple commands without redirections are treated as candidates for resumption of an existing job. There is no ambiguity allowed; if there is more than one job beginning with the string typed, then the most recently accessed job will be selected. The name of a stopped job, in this context, is the command line used to start it. If this variable is set to the value '`exact`', the string supplied must match the name of a stopped job exactly; if set to '`substring`', the string supplied needs to match a substring of the name of a stopped job. The '`substring`' value provides functionality analogous to the '`%?`' job ID (see Section 7.1 [Job Control Basics], page 104). If set to any other value, the supplied string must be a prefix of a stopped job's name; this provides functionality analogous to the '`%`' job ID.

8 Command Line Editing

This chapter describes the basic features of the GNU command line editing interface. Command line editing is provided by the Readline library, which is used by several different programs, including Bash. Command line editing is enabled by default when using an interactive shell, unless the `--noediting` option is supplied at shell invocation. Line editing is also used when using the `-e` option to the `read` builtin command (see Section 4.2 [Bash Builtins], page 50). By default, the line editing commands are similar to those of Emacs. A vi-style line editing interface is also available. Line editing can be enabled at any time using the `-o emacs` or `-o vi` options to the `set` builtin command (see Section 4.3.1 [The Set Builtin], page 61), or disabled using the `+o emacs` or `+o vi` options to `set`.

8.1 Introduction to Line Editing

The following paragraphs describe the notation used to represent keystrokes.

The text *C-k* is read as 'Control-K' and describes the character produced when the `k` key is pressed while the Control key is depressed.

The text *M-k* is read as 'Meta-K' and describes the character produced when the Meta key (if you have one) is depressed, and the `k` key is pressed. The Meta key is labeled `ALT` on many keyboards. On keyboards with two keys labeled `ALT` (usually to either side of the space bar), the `ALT` on the left side is generally set to work as a Meta key. The `ALT` key on the right may also be configured to work as a Meta key or may be configured as some other modifier, such as a Compose key for typing accented characters.

If you do not have a Meta or `ALT` key, or another key working as a Meta key, the identical keystroke can be generated by typing `ESC` *first*, and then typing `k`. Either process is known as *metafying* the `k` key.

The text *M-C-k* is read as 'Meta-Control-k' and describes the character produced by *metafying* *C-k*.

In addition, several keys have their own names. Specifically, `DEL`, `ESC`, `LFD`, `SPC`, `RET`, and `TAB` all stand for themselves when seen in this text, or in an init file (see Section 8.3 [Readline Init File], page 111). If your keyboard lacks a `LFD` key, typing `C-j` will produce the desired character. The `RET` key may be labeled `Return` or `Enter` on some keyboards.

8.2 Readline Interaction

Often during an interactive session you type in a long line of text, only to notice that the first word on the line is misspelled. The Readline library gives you a set of commands for manipulating the text as you type it in, allowing you to just fix your typo, and not forcing you to retype the majority of the line. Using these editing commands, you move the cursor to the place that needs correction, and delete or insert the text of the corrections. Then, when you are satisfied with the line, you simply press `RET`. You do not have to be at the end of the line to press `RET`; the entire line is accepted regardless of the location of the cursor within the line.

8.2.1 Readline Bare Essentials

In order to enter characters into the line, simply type them. The typed character appears where the cursor was, and then the cursor moves one space to the right. If you mistype a character, you can use your erase character to back up and delete the mistyped character.

Sometimes you may mistype a character, and not notice the error until you have typed several other characters. In that case, you can type `C-b` to move the cursor to the left, and then correct your mistake. Afterwards, you can move the cursor to the right with `C-f`.

When you add text in the middle of a line, you will notice that characters to the right of the cursor are 'pushed over' to make room for the text that you have inserted. Likewise, when you delete text behind the cursor, characters to the right of the cursor are 'pulled back' to fill in the blank space created by the removal of the text. A list of the bare essentials for editing the text of an input line follows.

`C-b` Move back one character.

`C-f` Move forward one character.

`DEL` or `Backspace`
Delete the character to the left of the cursor.

`C-d` Delete the character underneath the cursor.

Printing characters
Insert the character into the line at the cursor.

`C-_` or `C-x C-u`
Undo the last editing command. You can undo all the way back to an empty line.

(Depending on your configuration, the `Backspace` key be set to delete the character to the left of the cursor and the `DEL` key set to delete the character underneath the cursor, like `C-d`, rather than the character to the left of the cursor.)

8.2.2 Readline Movement Commands

The above table describes the most basic keystrokes that you need in order to do editing of the input line. For your convenience, many other commands have been added in addition to `C-b`, `C-f`, `C-d`, and `DEL`. Here are some commands for moving more rapidly about the line.

`C-a` Move to the start of the line.

`C-e` Move to the end of the line.

`M-f` Move forward a word, where a word is composed of letters and digits.

`M-b` Move backward a word.

`C-l` Clear the screen, reprinting the current line at the top.

Notice how `C-f` moves forward a character, while `M-f` moves forward a word. It is a loose convention that control keystrokes operate on characters while meta keystrokes operate on words.

8.2.3 Readline Killing Commands

Killing text means to delete the text from the line, but to save it away for later use, usually by *yanking* (re-inserting) it back into the line. ('Cut' and 'paste' are more recent jargon for 'kill' and 'yank'.)

If the description for a command says that it 'kills' text, then you can be sure that you can get the text back in a different (or the same) place later.

When you use a kill command, the text is saved in a *kill-ring*. Any number of consecutive kills save all of the killed text together, so that when you yank it back, you get it all. The kill ring is not line specific; the text that you killed on a previously typed line is available to be yanked back later, when you are typing another line.

Here is the list of commands for killing text.

C-k Kill the text from the current cursor position to the end of the line.

M-d Kill from the cursor to the end of the current word, or, if between words, to the end of the next word. Word boundaries are the same as those used by *M-f*.

M-DEL Kill from the cursor the start of the current word, or, if between words, to the start of the previous word. Word boundaries are the same as those used by *M-b*.

C-w Kill from the cursor to the previous whitespace. This is different than *M-DEL* because the word boundaries differ.

Here is how to *yank* the text back into the line. Yanking means to copy the most-recently-killed text from the kill buffer.

C-y Yank the most recently killed text back into the buffer at the cursor.

M-y Rotate the kill-ring, and yank the new top. You can only do this if the prior command is *C-y* or *M-y*.

8.2.4 Readline Arguments

You can pass numeric arguments to Readline commands. Sometimes the argument acts as a repeat count, other times it is the *sign* of the argument that is significant. If you pass a negative argument to a command which normally acts in a forward direction, that command will act in a backward direction. For example, to kill text back to the start of the line, you might type '`M-- C-k`'.

The general way to pass numeric arguments to a command is to type meta digits before the command. If the first 'digit' typed is a minus sign ('`-`'), then the sign of the argument will be negative. Once you have typed one meta digit to get the argument started, you can type the remainder of the digits, and then the command. For example, to give the *C-d* command an argument of 10, you could type '`M-1 0 C-d`', which will delete the next ten characters on the input line.

8.2.5 Searching for Commands in the History

Readline provides commands for searching through the command history (see Section 9.1 [Bash History Facilities], page 143) for lines containing a specified string. There are two search modes: *incremental* and *non-incremental*.

Incremental searches begin before the user has finished typing the search string. As each character of the search string is typed, Readline displays the next entry from the history matching the string typed so far. An incremental search requires only as many characters as needed to find the desired history entry. To search backward in the history for a particular string, type `C-r`. Typing `C-s` searches forward through the history. The characters present in the value of the `isearch-terminators` variable are used to terminate an incremental search. If that variable has not been assigned a value, the `ESC` and `C-J` characters will terminate an incremental search. `C-g` will abort an incremental search and restore the original line. When the search is terminated, the history entry containing the search string becomes the current line.

To find other matching entries in the history list, type `C-r` or `C-s` as appropriate. This will search backward or forward in the history for the next entry matching the search string typed so far. Any other key sequence bound to a Readline command will terminate the search and execute that command. For instance, a `RET` will terminate the search and accept the line, thereby executing the command from the history list. A movement command will terminate the search, make the last line found the current line, and begin editing.

Readline remembers the last incremental search string. If two `C-r`s are typed without any intervening characters defining a new search string, any remembered search string is used.

Non-incremental searches read the entire search string before starting to search for matching history lines. The search string may be typed by the user or be part of the contents of the current line.

8.3 Readline Init File

Although the Readline library comes with a set of Emacs-like keybindings installed by default, it is possible to use a different set of keybindings. Any user can customize programs that use Readline by putting commands in an *inputrc* file, conventionally in his home directory. The name of this file is taken from the value of the shell variable `INPUTRC`. If that variable is unset, the default is `~/.inputrc`. If that file does not exist or cannot be read, the ultimate default is `/etc/inputrc`. The `bind` builtin command can also be used to set Readline keybindings and variables. See Section 4.2 [Bash Builtins], page 50.

When a program which uses the Readline library starts up, the init file is read, and the key bindings are set.

In addition, the `C-x C-r` command re-reads this init file, thus incorporating any changes that you might have made to it.

8.3.1 Readline Init File Syntax

There are only a few basic constructs allowed in the Readline init file. Blank lines are ignored. Lines beginning with a '`#`' are comments. Lines beginning with a '`$`' indicate conditional constructs (see Section 8.3.2 [Conditional Init Constructs], page 119). Other lines denote variable settings and key bindings.

Variable Settings

You can modify the run-time behavior of Readline by altering the values of variables in Readline using the `set` command within the init file. The syntax is simple:

```
set variable value
```

Here, for example, is how to change from the default Emacs-like key binding to use `vi` line editing commands:

```
set editing-mode vi
```

Variable names and values, where appropriate, are recognized without regard to case. Unrecognized variable names are ignored.

Boolean variables (those that can be set to on or off) are set to on if the value is null or empty, *on* (case-insensitive), or 1. Any other value results in the variable being set to off.

The `bind -V` command lists the current Readline variable names and values. See Section 4.2 [Bash Builtins], page 50.

A great deal of run-time behavior is changeable with the following variables.

`bell-style`
: Controls what happens when Readline wants to ring the terminal bell. If set to '`none`', Readline never rings the bell. If set to '`visible`', Readline uses a visible bell if one is available. If set to '`audible`' (the default), Readline attempts to ring the terminal's bell.

`bind-tty-special-chars`
: If set to '`on`' (the default), Readline attempts to bind the control characters treated specially by the kernel's terminal driver to their Readline equivalents.

`blink-matching-paren`
: If set to '`on`', Readline attempts to briefly move the cursor to an opening parenthesis when a closing parenthesis is inserted. The default is '`off`'.

`colored-completion-prefix`
: If set to '`on`', when listing completions, Readline displays the common prefix of the set of possible completions using a different color. The color definitions are taken from the value of the `LS_COLORS` environment variable. The default is '`off`'.

`colored-stats`
: If set to '`on`', Readline displays possible completions using different colors to indicate their file type. The color definitions are taken from the value of the `LS_COLORS` environment variable. The default is '`off`'.

`comment-begin`
: The string to insert at the beginning of the line when the `insert-comment` command is executed. The default value is `"#"`.

`completion-display-width`
: The number of screen columns used to display possible matches when performing completion. The value is ignored if it is less than

0 or greater than the terminal screen width. A value of 0 will cause matches to be displayed one per line. The default value is -1.

`completion-ignore-case`
: If set to '`on`', Readline performs filename matching and completion in a case-insensitive fashion. The default value is '`off`'.

`completion-map-case`
: If set to '`on`', and *completion-ignore-case* is enabled, Readline treats hyphens ('`-`') and underscores ('`_`') as equivalent when performing case-insensitive filename matching and completion. The default value is '`off`'.

`completion-prefix-display-length`
: The length in characters of the common prefix of a list of possible completions that is displayed without modification. When set to a value greater than zero, common prefixes longer than this value are replaced with an ellipsis when displaying possible completions.

`completion-query-items`
: The number of possible completions that determines when the user is asked whether the list of possibilities should be displayed. If the number of possible completions is greater than this value, Readline will ask the user whether or not he wishes to view them; otherwise, they are simply listed. This variable must be set to an integer value greater than or equal to 0. A negative value means Readline should never ask. The default limit is `100`.

`convert-meta`
: If set to '`on`', Readline will convert characters with the eighth bit set to an ASCII key sequence by stripping the eighth bit and prefixing an `ESC` character, converting them to a meta-prefixed key sequence. The default value is '`on`', but will be set to '`off`' if the locale is one that contains eight-bit characters.

`disable-completion`
: If set to '`On`', Readline will inhibit word completion. Completion characters will be inserted into the line as if they had been mapped to `self-insert`. The default is '`off`'.

`echo-control-characters`
: When set to '`on`', on operating systems that indicate they support it, readline echoes a character corresponding to a signal generated from the keyboard. The default is '`on`'.

`editing-mode`
: The `editing-mode` variable controls which default set of key bindings is used. By default, Readline starts up in Emacs editing mode, where the keystrokes are most similar to Emacs. This variable can be set to either '`emacs`' or '`vi`'.

`emacs-mode-string`
: If the *show-mode-in-prompt* variable is enabled, this string is displayed immediately before the last line of the primary prompt when emacs editing mode is active. The value is expanded like a key binding, so the standard set of meta- and control prefixes and backslash escape sequences is available. Use the '`\1`' and '`\2`' escapes to begin and end sequences of non-printing characters, which can be used to embed a terminal control sequence into the mode string. The default is '`@`'.

`enable-bracketed-paste`
: When set to '`On`', Readline will configure the terminal in a way that will enable it to insert each paste into the editing buffer as a single string of characters, instead of treating each character as if it had been read from the keyboard. This can prevent pasted characters from being interpreted as editing commands. The default is '`off`'.

`enable-keypad`
: When set to '`on`', Readline will try to enable the application keypad when it is called. Some systems need this to enable the arrow keys. The default is '`off`'.

`enable-meta-key`
: When set to '`on`', Readline will try to enable any meta modifier key the terminal claims to support when it is called. On many terminals, the meta key is used to send eight-bit characters. The default is '`on`'.

`expand-tilde`
: If set to '`on`', tilde expansion is performed when Readline attempts word completion. The default is '`off`'.

`history-preserve-point`
: If set to '`on`', the history code attempts to place the point (the current cursor position) at the same location on each history line retrieved with `previous-history` or `next-history`. The default is '`off`'.

`history-size`
: Set the maximum number of history entries saved in the history list. If set to zero, any existing history entries are deleted and no new entries are saved. If set to a value less than zero, the number of history entries is not limited. By default, the number of history entries is not limited. If an attempt is made to set *history-size* to a non-numeric value, the maximum number of history entries will be set to 500.

`horizontal-scroll-mode`
: This variable can be set to either '`on`' or '`off`'. Setting it to '`on`' means that the text of the lines being edited will scroll horizontally on a single screen line when they are longer than the width of the

screen, instead of wrapping onto a new screen line. By default, this variable is set to 'off'.

`input-meta`
: If set to 'on', Readline will enable eight-bit input (it will not clear the eighth bit in the characters it reads), regardless of what the terminal claims it can support. The default value is 'off', but Readline will set it to 'on' if the locale contains eight-bit characters. The name `meta-flag` is a synonym for this variable.

`isearch-terminators`
: The string of characters that should terminate an incremental search without subsequently executing the character as a command (see Section 8.2.5 [Searching], page 110). If this variable has not been given a value, the characters ESC and *C-J* will terminate an incremental search.

`keymap`
: Sets Readline's idea of the current keymap for key binding commands. Built-in `keymap` names are `emacs`, `emacs-standard`, `emacs-meta`, `emacs-ctlx`, `vi`, `vi-move`, `vi-command`, and `vi-insert`. `vi` is equivalent to `vi-command` (`vi-move` is also a synonym); `emacs` is equivalent to `emacs-standard`. Applications may add additional names. The default value is `emacs`. The value of the `editing-mode` variable also affects the default keymap.

`keyseq-timeout`
: Specifies the duration Readline will wait for a character when reading an ambiguous key sequence (one that can form a complete key sequence using the input read so far, or can take additional input to complete a longer key sequence). If no input is received within the timeout, Readline will use the shorter but complete key sequence. Readline uses this value to determine whether or not input is available on the current input source (`rl_instream` by default). The value is specified in milliseconds, so a value of 1000 means that Readline will wait one second for additional input. If this variable is set to a value less than or equal to zero, or to a non-numeric value, Readline will wait until another key is pressed to decide which key sequence to complete. The default value is `500`.

`mark-directories`
: If set to 'on', completed directory names have a slash appended. The default is 'on'.

`mark-modified-lines`
: This variable, when set to 'on', causes Readline to display an asterisk ('*') at the start of history lines which have been modified. This variable is 'off' by default.

`mark-symlinked-directories`
: If set to '`on`', completed names which are symbolic links to directories have a slash appended (subject to the value of `mark-directories`). The default is '`off`'.

`match-hidden-files`
: This variable, when set to '`on`', causes Readline to match files whose names begin with a '`.`' (hidden files) when performing filename completion. If set to '`off`', the leading '`.`' must be supplied by the user in the filename to be completed. This variable is '`on`' by default.

`menu-complete-display-prefix`
: If set to '`on`', menu completion displays the common prefix of the list of possible completions (which may be empty) before cycling through the list. The default is '`off`'.

`output-meta`
: If set to '`on`', Readline will display characters with the eighth bit set directly rather than as a meta-prefixed escape sequence. The default is '`off`', but Readline will set it to '`on`' if the locale contains eight-bit characters.

`page-completions`
: If set to '`on`', Readline uses an internal `more`-like pager to display a screenful of possible completions at a time. This variable is '`on`' by default.

`print-completions-horizontally`
: If set to '`on`', Readline will display completions with matches sorted horizontally in alphabetical order, rather than down the screen. The default is '`off`'.

`revert-all-at-newline`
: If set to '`on`', Readline will undo all changes to history lines before returning when `accept-line` is executed. By default, history lines may be modified and retain individual undo lists across calls to `readline`. The default is '`off`'.

`show-all-if-ambiguous`
: This alters the default behavior of the completion functions. If set to '`on`', words which have more than one possible completion cause the matches to be listed immediately instead of ringing the bell. The default value is '`off`'.

`show-all-if-unmodified`
: This alters the default behavior of the completion functions in a fashion similar to *show-all-if-ambiguous*. If set to '`on`', words which have more than one possible completion without any possible partial completion (the possible completions don't share a common prefix) cause the matches to be listed immediately instead of ringing the bell. The default value is '`off`'.

`show-mode-in-prompt`
: If set to '`on`', add a string to the beginning of the prompt indicating the editing mode: emacs, vi command, or vi insertion. The mode strings are user-settable (e.g., *emacs-mode-string*). The default value is '`off`'.

`skip-completed-text`
: If set to '`on`', this alters the default completion behavior when inserting a single match into the line. It's only active when performing completion in the middle of a word. If enabled, readline does not insert characters from the completion that match characters after point in the word being completed, so portions of the word following the cursor are not duplicated. For instance, if this is enabled, attempting completion when the cursor is after the '`e`' in '`Makefile`' will result in '`Makefile`' rather than '`Makefilefile`', assuming there is a single possible completion. The default value is '`off`'.

`vi-cmd-mode-string`
: If the *show-mode-in-prompt* variable is enabled, this string is displayed immediately before the last line of the primary prompt when vi editing mode is active and in command mode. The value is expanded like a key binding, so the standard set of meta- and control prefixes and backslash escape sequences is available. Use the '`\1`' and '`\2`' escapes to begin and end sequences of non-printing characters, which can be used to embed a terminal control sequence into the mode string. The default is '`(cmd)`'.

`vi-ins-mode-string`
: If the *show-mode-in-prompt* variable is enabled, this string is displayed immediately before the last line of the primary prompt when vi editing mode is active and in insertion mode. The value is expanded like a key binding, so the standard set of meta- and control prefixes and backslash escape sequences is available. Use the '`\1`' and '`\2`' escapes to begin and end sequences of non-printing characters, which can be used to embed a terminal control sequence into the mode string. The default is '`(ins)`'.

`visible-stats`
: If set to '`on`', a character denoting a file's type is appended to the filename when listing possible completions. The default is '`off`'.

Key Bindings

The syntax for controlling key bindings in the init file is simple. First you need to find the name of the command that you want to change. The following sections contain tables of the command name, the default keybinding, if any, and a short description of what the command does.

Once you know the name of the command, simply place on a line in the init file the name of the key you wish to bind the command to, a colon, and then the name of the command. There can be no space between the key name and the colon – that will be interpreted as part of the key name. The name of the key can be expressed in different ways, depending on what you find most comfortable.

In addition to command names, readline allows keys to be bound to a string that is inserted when the key is pressed (a *macro*).

The `bind -p` command displays Readline function names and bindings in a format that can put directly into an initialization file. See Section 4.2 [Bash Builtins], page 50.

keyname: *function-name* or *macro*

keyname is the name of a key spelled out in English. For example:

```
Control-u: universal-argument
Meta-Rubout: backward-kill-word
Control-o: "> output"
```

In the example above, *C-u* is bound to the function `universal-argument`, *M-DEL* is bound to the function `backward-kill-word`, and *C-o* is bound to run the macro expressed on the right hand side (that is, to insert the text '`> output`' into the line).

A number of symbolic character names are recognized while processing this key binding syntax: *DEL*, *ESC*, *ESCAPE*, *LFD*, *NEWLINE*, *RET*, *RETURN*, *RUBOUT*, *SPACE*, *SPC*, and *TAB*.

"*keyseq*": *function-name* or *macro*

keyseq differs from *keyname* above in that strings denoting an entire key sequence can be specified, by placing the key sequence in double quotes. Some GNU Emacs style key escapes can be used, as in the following example, but the special character names are not recognized.

```
"\C-u": universal-argument
"\C-x\C-r": re-read-init-file
"\e[11~": "Function Key 1"
```

In the above example, *C-u* is again bound to the function `universal-argument` (just as it was in the first example), '*C-x C-r*' is bound to the function `re-read-init-file`, and '`ESC [ 1 1 ~`' is bound to insert the text '`Function Key 1`'.

The following GNU Emacs style escape sequences are available when specifying key sequences:

`\C-` control prefix

`\M-` meta prefix

`\e` an escape character

`\\` backslash

`\"` ", a double quotation mark

`\'` ', a single quote or apostrophe

In addition to the GNU Emacs style escape sequences, a second set of backslash escapes is available:

`\a` alert (bell)

`\b` backspace

`\d` delete

`\f` form feed

`\n` newline

`\r` carriage return

`\t` horizontal tab

`\v` vertical tab

`\nnn` the eight-bit character whose value is the octal value *nnn* (one to three digits)

`\xHH` the eight-bit character whose value is the hexadecimal value *HH* (one or two hex digits)

When entering the text of a macro, single or double quotes must be used to indicate a macro definition. Unquoted text is assumed to be a function name. In the macro body, the backslash escapes described above are expanded. Backslash will quote any other character in the macro text, including '"' and '''. For example, the following binding will make '`C-x \`' insert a single '\' into the line:

```
"\C-x\\": "\\"
```

8.3.2 Conditional Init Constructs

Readline implements a facility similar in spirit to the conditional compilation features of the C preprocessor which allows key bindings and variable settings to be performed as the result of tests. There are four parser directives used.

`$if` The `$if` construct allows bindings to be made based on the editing mode, the terminal being used, or the application using Readline. The text of the test, after any comparison operator, extends to the end of the line; unless otherwise noted, no characters are required to isolate it.

`mode` The `mode=` form of the `$if` directive is used to test whether Readline is in `emacs` or `vi` mode. This may be used in conjunction with the '`set keymap`' command, for instance, to set bindings in the `emacs-standard` and `emacs-ctlx` keymaps only if Readline is starting out in `emacs` mode.

`term` The `term=` form may be used to include terminal-specific key bindings, perhaps to bind the key sequences output by the terminal's function keys. The word on the right side of the '`=`' is tested against both the full name of the terminal and the portion of the terminal name before the first '`-`'. This allows `sun` to match both `sun` and `sun-cmd`, for instance.

`version` The `version` test may be used to perform comparisons against specific Readline versions. The `version` expands to the current Readline version. The set of comparison operators includes '`=`' (and '`==`'), '`!=`', '`<=`', '`>=`', '`<`', and '`>`'. The version number supplied on the right side of the operator consists of a major version number, an optional decimal point, and an optional minor version (e.g., '`7.1`'). If the minor version is omitted, it is assumed to be '`0`'. The operator may be separated from the string `version` and from the version number argument by whitespace. The following example sets a variable if the Readline version being used is 7.0 or newer:

```
$if version >= 7.0
set show-mode-in-prompt on
$endif
```

`application`
The *application* construct is used to include application-specific settings. Each program using the Readline library sets the *application name*, and you can test for a particular value. This could be used to bind key sequences to functions useful for a specific program. For instance, the following command adds a key sequence that quotes the current or previous word in Bash:

```
$if Bash
# Quote the current or previous word
"\C-xq": "\eb\"\ef\""
$endif
```

`variable` The *variable* construct provides simple equality tests for Readline variables and values. The permitted comparison operators are '`=`', '`==`', and '`!=`'. The variable name must be separated from the comparison operator by whitespace; the operator may be separated from the value on the right hand side by whitespace. Both string and boolean variables may be tested. Boolean variables must be tested against the values *on* and *off*. The following example is equivalent to the `mode=emacs` test described above:

```
$if editing-mode == emacs
set show-mode-in-prompt on
$endif
```

`$endif` This command, as seen in the previous example, terminates an `$if` command.

`$else` Commands in this branch of the `$if` directive are executed if the test fails.

`$include` This directive takes a single filename as an argument and reads commands and bindings from that file. For example, the following directive reads from `/etc/inputrc`:

```
$include /etc/inputrc
```

8.3.3 Sample Init File

Here is an example of an *inputrc* file. This illustrates key binding, variable assignment, and conditional syntax.

```
# This file controls the behaviour of line input editing for
# programs that use the GNU Readline library.  Existing
# programs include FTP, Bash, and GDB.
#
# You can re-read the inputrc file with C-x C-r.
# Lines beginning with '#' are comments.
#
# First, include any system-wide bindings and variable
# assignments from /etc/Inputrc
$include /etc/Inputrc

#
# Set various bindings for emacs mode.

set editing-mode emacs

$if mode=emacs

Meta-Control-h:  backward-kill-word Text after the function name is ignored

#
# Arrow keys in keypad mode
#
#"\M-OD":        backward-char
#"\M-OC":        forward-char
#"\M-OA":        previous-history
#"\M-OB":        next-history
#
# Arrow keys in ANSI mode
#
"\M-[D":        backward-char
"\M-[C":        forward-char
"\M-[A":        previous-history
"\M-[B":        next-history
#
# Arrow keys in 8 bit keypad mode
#
#"\M-\C-OD":       backward-char
#"\M-\C-OC":       forward-char
#"\M-\C-OA":       previous-history
#"\M-\C-OB":       next-history
#
# Arrow keys in 8 bit ANSI mode
#
#"\M-\C-[D":       backward-char
#"\M-\C-[C":       forward-char
```

```
#"\M-\C-[A":       previous-history
#"\M-\C-[B":       next-history

C-q: quoted-insert

$endif

# An old-style binding.  This happens to be the default.
TAB: complete

# Macros that are convenient for shell interaction
$if Bash
# edit the path
"\C-xp": "PATH=${PATH}\e\C-e\C-a\ef\C-f"
# prepare to type a quoted word --
# insert open and close double quotes
# and move to just after the open quote
"\C-x\"": "\"\"\C-b"
# insert a backslash (testing backslash escapes
# in sequences and macros)
"\C-x\\": "\\"
# Quote the current or previous word
"\C-xq": "\eb\"\ef\""
# Add a binding to refresh the line, which is unbound
"\C-xr": redraw-current-line
# Edit variable on current line.
"\M-\C-v": "\C-a\C-k$\C-y\M-\C-e\C-a\C-y="
$endif

# use a visible bell if one is available
set bell-style visible

# don't strip characters to 7 bits when reading
set input-meta on

# allow iso-latin1 characters to be inserted rather
# than converted to prefix-meta sequences
set convert-meta off

# display characters with the eighth bit set directly
# rather than as meta-prefixed characters
set output-meta on

# if there are more than 150 possible completions for
# a word, ask the user if he wants to see all of them
set completion-query-items 150
```

```
# For FTP
$if Ftp
"\C-xg": "get \M-?"
"\C-xt": "put \M-?"
"\M-.": yank-last-arg
$endif
```

8.4 Bindable Readline Commands

This section describes Readline commands that may be bound to key sequences. You can list your key bindings by executing `bind -P` or, for a more terse format, suitable for an *inputrc* file, `bind -p`. (See Section 4.2 [Bash Builtins], page 50.) Command names without an accompanying key sequence are unbound by default.

In the following descriptions, *point* refers to the current cursor position, and *mark* refers to a cursor position saved by the `set-mark` command. The text between the point and mark is referred to as the *region*.

8.4.1 Commands For Moving

`beginning-of-line (C-a)`
: Move to the start of the current line.

`end-of-line (C-e)`
: Move to the end of the line.

`forward-char (C-f)`
: Move forward a character.

`backward-char (C-b)`
: Move back a character.

`forward-word (M-f)`
: Move forward to the end of the next word. Words are composed of letters and digits.

`backward-word (M-b)`
: Move back to the start of the current or previous word. Words are composed of letters and digits.

`shell-forward-word ()`
: Move forward to the end of the next word. Words are delimited by non-quoted shell metacharacters.

`shell-backward-word ()`
: Move back to the start of the current or previous word. Words are delimited by non-quoted shell metacharacters.

`previous-screen-line ()`
: Attempt to move point to the same physical screen column on the previous physical screen line. This will not have the desired effect if the current Readline line does not take up more than one physical line or if point is not greater than the length of the prompt plus the screen width.

`next-screen-line ()`
: Attempt to move point to the same physical screen column on the next physical screen line. This will not have the desired effect if the current Readline line does not take up more than one physical line or if the length of the current Readline line is not greater than the length of the prompt plus the screen width.

`clear-screen (C-l)`
: Clear the screen and redraw the current line, leaving the current line at the top of the screen.

`redraw-current-line ()`
: Refresh the current line. By default, this is unbound.

8.4.2 Commands For Manipulating The History

`accept-line (Newline or Return)`
: Accept the line regardless of where the cursor is. If this line is non-empty, add it to the history list according to the setting of the `HISTCONTROL` and `HISTIGNORE` variables. If this line is a modified history line, then restore the history line to its original state.

`previous-history (C-p)`
: Move 'back' through the history list, fetching the previous command.

`next-history (C-n)`
: Move 'forward' through the history list, fetching the next command.

`beginning-of-history (M-<)`
: Move to the first line in the history.

`end-of-history (M->)`
: Move to the end of the input history, i.e., the line currently being entered.

`reverse-search-history (C-r)`
: Search backward starting at the current line and moving 'up' through the history as necessary. This is an incremental search.

`forward-search-history (C-s)`
: Search forward starting at the current line and moving 'down' through the history as necessary. This is an incremental search.

`non-incremental-reverse-search-history (M-p)`
: Search backward starting at the current line and moving 'up' through the history as necessary using a non-incremental search for a string supplied by the user. The search string may match anywhere in a history line.

`non-incremental-forward-search-history (M-n)`
: Search forward starting at the current line and moving 'down' through the history as necessary using a non-incremental search for a string supplied by the user. The search string may match anywhere in a history line.

`history-search-forward ()`
: Search forward through the history for the string of characters between the start of the current line and the point. The search string must match at the

beginning of a history line. This is a non-incremental search. By default, this command is unbound.

`history-search-backward ()`
: Search backward through the history for the string of characters between the start of the current line and the point. The search string must match at the beginning of a history line. This is a non-incremental search. By default, this command is unbound.

`history-substring-search-forward ()`
: Search forward through the history for the string of characters between the start of the current line and the point. The search string may match anywhere in a history line. This is a non-incremental search. By default, this command is unbound.

`history-substring-search-backward ()`
: Search backward through the history for the string of characters between the start of the current line and the point. The search string may match anywhere in a history line. This is a non-incremental search. By default, this command is unbound.

`yank-nth-arg (M-C-y)`
: Insert the first argument to the previous command (usually the second word on the previous line) at point. With an argument *n*, insert the *n*th word from the previous command (the words in the previous command begin with word 0). A negative argument inserts the *n*th word from the end of the previous command. Once the argument *n* is computed, the argument is extracted as if the '!*n*' history expansion had been specified.

`yank-last-arg (M-. or M-_)`
: Insert last argument to the previous command (the last word of the previous history entry). With a numeric argument, behave exactly like `yank-nth-arg`. Successive calls to `yank-last-arg` move back through the history list, inserting the last word (or the word specified by the argument to the first call) of each line in turn. Any numeric argument supplied to these successive calls determines the direction to move through the history. A negative argument switches the direction through the history (back or forward). The history expansion facilities are used to extract the last argument, as if the '!$' history expansion had been specified.

8.4.3 Commands For Changing Text

end-of-file `(usually C-d)`
: The character indicating end-of-file as set, for example, by `stty`. If this character is read when there are no characters on the line, and point is at the beginning of the line, Readline interprets it as the end of input and returns EOF.

`delete-char (C-d)`
: Delete the character at point. If this function is bound to the same character as the tty EOF character, as *C-d* commonly is, see above for the effects.

`backward-delete-char (Rubout)`
: Delete the character behind the cursor. A numeric argument means to kill the characters instead of deleting them.

`forward-backward-delete-char ()`
: Delete the character under the cursor, unless the cursor is at the end of the line, in which case the character behind the cursor is deleted. By default, this is not bound to a key.

`quoted-insert (C-q or C-v)`
: Add the next character typed to the line verbatim. This is how to insert key sequences like *C-q*, for example.

`self-insert (a, b, A, 1, !, ...)`
: Insert yourself.

`bracketed-paste-begin ()`
: This function is intended to be bound to the "bracketed paste" escape sequence sent by some terminals, and such a binding is assigned by default. It allows Readline to insert the pasted text as a single unit without treating each character as if it had been read from the keyboard. The characters are inserted as if each one was bound to `self-insert` instead of executing any editing commands.

`transpose-chars (C-t)`
: Drag the character before the cursor forward over the character at the cursor, moving the cursor forward as well. If the insertion point is at the end of the line, then this transposes the last two characters of the line. Negative arguments have no effect.

`transpose-words (M-t)`
: Drag the word before point past the word after point, moving point past that word as well. If the insertion point is at the end of the line, this transposes the last two words on the line.

`upcase-word (M-u)`
: Uppercase the current (or following) word. With a negative argument, uppercase the previous word, but do not move the cursor.

`downcase-word (M-l)`
: Lowercase the current (or following) word. With a negative argument, lowercase the previous word, but do not move the cursor.

`capitalize-word (M-c)`
: Capitalize the current (or following) word. With a negative argument, capitalize the previous word, but do not move the cursor.

`overwrite-mode ()`
: Toggle overwrite mode. With an explicit positive numeric argument, switches to overwrite mode. With an explicit non-positive numeric argument, switches to insert mode. This command affects only `emacs` mode; `vi` mode does overwrite differently. Each call to `readline()` starts in insert mode.

In overwrite mode, characters bound to `self-insert` replace the text at point rather than pushing the text to the right. Characters bound to `backward-delete-char` replace the character before point with a space.

By default, this command is unbound.

8.4.4 Killing And Yanking

`kill-line (C-k)`
: Kill the text from point to the end of the line.

`backward-kill-line (C-x Rubout)`
: Kill backward from the cursor to the beginning of the current line.

`unix-line-discard (C-u)`
: Kill backward from the cursor to the beginning of the current line.

`kill-whole-line ()`
: Kill all characters on the current line, no matter where point is. By default, this is unbound.

`kill-word (M-d)`
: Kill from point to the end of the current word, or if between words, to the end of the next word. Word boundaries are the same as `forward-word`.

`backward-kill-word (M-DEL)`
: Kill the word behind point. Word boundaries are the same as `backward-word`.

`shell-kill-word ()`
: Kill from point to the end of the current word, or if between words, to the end of the next word. Word boundaries are the same as `shell-forward-word`.

`shell-backward-kill-word ()`
: Kill the word behind point. Word boundaries are the same as `shell-backward-word`.

`unix-word-rubout (C-w)`
: Kill the word behind point, using white space as a word boundary. The killed text is saved on the kill-ring.

`unix-filename-rubout ()`
: Kill the word behind point, using white space and the slash character as the word boundaries. The killed text is saved on the kill-ring.

`delete-horizontal-space ()`
: Delete all spaces and tabs around point. By default, this is unbound.

`kill-region ()`
: Kill the text in the current region. By default, this command is unbound.

`copy-region-as-kill ()`
: Copy the text in the region to the kill buffer, so it can be yanked right away. By default, this command is unbound.

`copy-backward-word ()`
: Copy the word before point to the kill buffer. The word boundaries are the same as `backward-word`. By default, this command is unbound.

`copy-forward-word ()`
: Copy the word following point to the kill buffer. The word boundaries are the same as `forward-word`. By default, this command is unbound.

`yank (C-y)`
: Yank the top of the kill ring into the buffer at point.

`yank-pop (M-y)`
: Rotate the kill-ring, and yank the new top. You can only do this if the prior command is `yank` or `yank-pop`.

8.4.5 Specifying Numeric Arguments

`digit-argument (`*`M-0`*`, `*`M-1`*`, ... `*`M--`*`)`
: Add this digit to the argument already accumulating, or start a new argument. *`M--`* starts a negative argument.

`universal-argument ()`
: This is another way to specify an argument. If this command is followed by one or more digits, optionally with a leading minus sign, those digits define the argument. If the command is followed by digits, executing `universal-argument` again ends the numeric argument, but is otherwise ignored. As a special case, if this command is immediately followed by a character that is neither a digit nor minus sign, the argument count for the next command is multiplied by four. The argument count is initially one, so executing this function the first time makes the argument count four, a second time makes the argument count sixteen, and so on. By default, this is not bound to a key.

8.4.6 Letting Readline Type For You

`complete (TAB)`
: Attempt to perform completion on the text before point. The actual completion performed is application-specific. Bash attempts completion treating the text as a variable (if the text begins with '`$`'), username (if the text begins with '`~`'), hostname (if the text begins with '`@`'), or command (including aliases and functions) in turn. If none of these produces a match, filename completion is attempted.

`possible-completions (M-?)`
: List the possible completions of the text before point. When displaying completions, Readline sets the number of columns used for display to the value of `completion-display-width`, the value of the environment variable `COLUMNS`, or the screen width, in that order.

`insert-completions (M-*)`
: Insert all completions of the text before point that would have been generated by `possible-completions`.

`menu-complete ()`
: Similar to `complete`, but replaces the word to be completed with a single match from the list of possible completions. Repeated execution of `menu-complete` steps through the list of possible completions, inserting each match in turn.

At the end of the list of completions, the bell is rung (subject to the setting of `bell-style`) and the original text is restored. An argument of *n* moves *n* positions forward in the list of matches; a negative argument may be used to move backward through the list. This command is intended to be bound to `TAB`, but is unbound by default.

`menu-complete-backward ()`
: Identical to `menu-complete`, but moves backward through the list of possible completions, as if `menu-complete` had been given a negative argument.

`delete-char-or-list ()`
: Deletes the character under the cursor if not at the beginning or end of the line (like `delete-char`). If at the end of the line, behaves identically to `possible-completions`. This command is unbound by default.

`complete-filename (M-/)`
: Attempt filename completion on the text before point.

`possible-filename-completions (C-x /)`
: List the possible completions of the text before point, treating it as a filename.

`complete-username (M-~)`
: Attempt completion on the text before point, treating it as a username.

`possible-username-completions (C-x ~)`
: List the possible completions of the text before point, treating it as a username.

`complete-variable (M-$)`
: Attempt completion on the text before point, treating it as a shell variable.

`possible-variable-completions (C-x $)`
: List the possible completions of the text before point, treating it as a shell variable.

`complete-hostname (M-@)`
: Attempt completion on the text before point, treating it as a hostname.

`possible-hostname-completions (C-x @)`
: List the possible completions of the text before point, treating it as a hostname.

`complete-command (M-!)`
: Attempt completion on the text before point, treating it as a command name. Command completion attempts to match the text against aliases, reserved words, shell functions, shell builtins, and finally executable filenames, in that order.

`possible-command-completions (C-x !)`
: List the possible completions of the text before point, treating it as a command name.

`dynamic-complete-history (M-TAB)`
: Attempt completion on the text before point, comparing the text against lines from the history list for possible completion matches.

`dabbrev-expand ()`
: Attempt menu completion on the text before point, comparing the text against lines from the history list for possible completion matches.

`complete-into-braces (M-{)`
: Perform filename completion and insert the list of possible completions enclosed within braces so the list is available to the shell (see Section 3.5.1 [Brace Expansion], page 23).

8.4.7 Keyboard Macros

`start-kbd-macro (C-x ()`
: Begin saving the characters typed into the current keyboard macro.

`end-kbd-macro (C-x ))`
: Stop saving the characters typed into the current keyboard macro and save the definition.

`call-last-kbd-macro (C-x e)`
: Re-execute the last keyboard macro defined, by making the characters in the macro appear as if typed at the keyboard.

`print-last-kbd-macro ()`
: Print the last keboard macro defined in a format suitable for the *inputrc* file.

8.4.8 Some Miscellaneous Commands

`re-read-init-file (C-x C-r)`
: Read in the contents of the *inputrc* file, and incorporate any bindings or variable assignments found there.

`abort (C-g)`
: Abort the current editing command and ring the terminal's bell (subject to the setting of `bell-style`).

`do-lowercase-version (M-A, M-B, M-x, ...)`
: If the metafied character *x* is upper case, run the command that is bound to the corresponding metafied lower case character. The behavior is undefined if *x* is already lower case.

`prefix-meta (ESC)`
: Metafy the next character typed. This is for keyboards without a meta key. Typing '`ESC f`' is equivalent to typing *`M-f`*.

`undo (C-_ or C-x C-u)`
: Incremental undo, separately remembered for each line.

`revert-line (M-r)`
: Undo all changes made to this line. This is like executing the `undo` command enough times to get back to the beginning.

`tilde-expand (M-&)`
: Perform tilde expansion on the current word.

`set-mark (C-@)`
: Set the mark to the point. If a numeric argument is supplied, the mark is set to that position.

`exchange-point-and-mark (C-x C-x)`
: Swap the point with the mark. The current cursor position is set to the saved position, and the old cursor position is saved as the mark.

`character-search (C-])`
: A character is read and point is moved to the next occurrence of that character. A negative count searches for previous occurrences.

`character-search-backward (M-C-])`
: A character is read and point is moved to the previous occurrence of that character. A negative count searches for subsequent occurrences.

`skip-csi-sequence ()`
: Read enough characters to consume a multi-key sequence such as those defined for keys like Home and End. Such sequences begin with a Control Sequence Indicator (CSI), usually ESC-[. If this sequence is bound to "\e[", keys producing such sequences will have no effect unless explicitly bound to a readline command, instead of inserting stray characters into the editing buffer. This is unbound by default, but usually bound to ESC-[.

`insert-comment (M-#)`
: Without a numeric argument, the value of the `comment-begin` variable is inserted at the beginning of the current line. If a numeric argument is supplied, this command acts as a toggle: if the characters at the beginning of the line do not match the value of `comment-begin`, the value is inserted, otherwise the characters in `comment-begin` are deleted from the beginning of the line. In either case, the line is accepted as if a newline had been typed. The default value of `comment-begin` causes this command to make the current line a shell comment. If a numeric argument causes the comment character to be removed, the line will be executed by the shell.

`dump-functions ()`
: Print all of the functions and their key bindings to the Readline output stream. If a numeric argument is supplied, the output is formatted in such a way that it can be made part of an *inputrc* file. This command is unbound by default.

`dump-variables ()`
: Print all of the settable variables and their values to the Readline output stream. If a numeric argument is supplied, the output is formatted in such a way that it can be made part of an *inputrc* file. This command is unbound by default.

`dump-macros ()`
: Print all of the Readline key sequences bound to macros and the strings they output. If a numeric argument is supplied, the output is formatted in such a way that it can be made part of an *inputrc* file. This command is unbound by default.

`glob-complete-word (M-g)`
: The word before point is treated as a pattern for pathname expansion, with an asterisk implicitly appended. This pattern is used to generate a list of matching file names for possible completions.

`glob-expand-word (C-x *)`
: The word before point is treated as a pattern for pathname expansion, and the list of matching file names is inserted, replacing the word. If a numeric argument is supplied, a '`*`' is appended before pathname expansion.

`glob-list-expansions (C-x g)`
: The list of expansions that would have been generated by `glob-expand-word` is displayed, and the line is redrawn. If a numeric argument is supplied, a '`*`' is appended before pathname expansion.

`display-shell-version (C-x C-v)`
: Display version information about the current instance of Bash.

`shell-expand-line (M-C-e)`
: Expand the line as the shell does. This performs alias and history expansion as well as all of the shell word expansions (see Section 3.5 [Shell Expansions], page 22).

`history-expand-line (M-^)`
: Perform history expansion on the current line.

`magic-space ()`
: Perform history expansion on the current line and insert a space (see Section 9.3 [History Interaction], page 145).

`alias-expand-line ()`
: Perform alias expansion on the current line (see Section 6.6 [Aliases], page 93).

`history-and-alias-expand-line ()`
: Perform history and alias expansion on the current line.

`insert-last-argument (M-. or M-_)`
: A synonym for `yank-last-arg`.

`operate-and-get-next (C-o)`
: Accept the current line for execution and fetch the next line relative to the current line from the history for editing. A numeric argument, if supplied, specifies the history entry to use instead of the current line.

`edit-and-execute-command (C-x C-e)`
: Invoke an editor on the current command line, and execute the result as shell commands. Bash attempts to invoke `$VISUAL`, `$EDITOR`, and `emacs` as the editor, in that order.

8.5 Readline vi Mode

While the Readline library does not have a full set of `vi` editing functions, it does contain enough to allow simple editing of the line. The Readline `vi` mode behaves as specified in the POSIX standard.

In order to switch interactively between `emacs` and `vi` editing modes, use the '`set -o emacs`' and '`set -o vi`' commands (see Section 4.3.1 [The Set Builtin], page 61). The Readline default is `emacs` mode.

When you enter a line in `vi` mode, you are already placed in 'insertion' mode, as if you had typed an '`i`'. Pressing `ESC` switches you into 'command' mode, where you can edit the text of the line with the standard `vi` movement keys, move to previous history lines with '`k`' and subsequent lines with '`j`', and so forth.

8.6 Programmable Completion

When word completion is attempted for an argument to a command for which a completion specification (a *compspec*) has been defined using the `complete` builtin (see Section 8.7 [Programmable Completion Builtins], page 136), the programmable completion facilities are invoked.

First, the command name is identified. If a compspec has been defined for that command, the compspec is used to generate the list of possible completions for the word. If the command word is the empty string (completion attempted at the beginning of an empty line), any compspec defined with the `-E` option to `complete` is used. If the command word is a full pathname, a compspec for the full pathname is searched for first. If no compspec is found for the full pathname, an attempt is made to find a compspec for the portion following the final slash. If those searches do not result in a compspec, any compspec defined with the `-D` option to `complete` is used as the default. If there is no default compspec, Bash attempts alias expansion on the command word as a final resort, and attempts to find a compspec for the command word from any successful expansion

Once a compspec has been found, it is used to generate the list of matching words. If a compspec is not found, the default Bash completion described above (see Section 8.4.6 [Commands For Completion], page 129) is performed.

First, the actions specified by the compspec are used. Only matches which are prefixed by the word being completed are returned. When the `-f` or `-d` option is used for filename or directory name completion, the shell variable `FIGNORE` is used to filter the matches. See Section 5.2 [Bash Variables], page 73, for a description of `FIGNORE`.

Any completions specified by a filename expansion pattern to the `-G` option are generated next. The words generated by the pattern need not match the word being completed. The `GLOBIGNORE` shell variable is not used to filter the matches, but the `FIGNORE` shell variable is used.

Next, the string specified as the argument to the `-W` option is considered. The string is first split using the characters in the `IFS` special variable as delimiters. Shell quoting is honored within the string, in order to provide a mechanism for the words to contain shell metacharacters or characters in the value of `IFS`. Each word is then expanded using brace expansion, tilde expansion, parameter and variable expansion, command substitution, and arithmetic expansion, as described above (see Section 3.5 [Shell Expansions], page 22). The results are split using the rules described above (see Section 3.5.7 [Word Splitting], page 32). The results of the expansion are prefix-matched against the word being completed, and the matching words become the possible completions.

After these matches have been generated, any shell function or command specified with the `-F` and `-C` options is invoked. When the command or function is invoked, the `COMP_`

LINE, COMP_POINT, COMP_KEY, and COMP_TYPE variables are assigned values as described above (see Section 5.2 [Bash Variables], page 73). If a shell function is being invoked, the COMP_WORDS and COMP_CWORD variables are also set. When the function or command is invoked, the first argument ($1) is the name of the command whose arguments are being completed, the second argument ($2) is the word being completed, and the third argument ($3) is the word preceding the word being completed on the current command line. No filtering of the generated completions against the word being completed is performed; the function or command has complete freedom in generating the matches.

Any function specified with -F is invoked first. The function may use any of the shell facilities, including the compgen and compopt builtins described below (see Section 8.7 [Programmable Completion Builtins], page 136), to generate the matches. It must put the possible completions in the COMPREPLY array variable, one per array element.

Next, any command specified with the -C option is invoked in an environment equivalent to command substitution. It should print a list of completions, one per line, to the standard output. Backslash may be used to escape a newline, if necessary.

After all of the possible completions are generated, any filter specified with the -X option is applied to the list. The filter is a pattern as used for pathname expansion; a '&' in the pattern is replaced with the text of the word being completed. A literal '&' may be escaped with a backslash; the backslash is removed before attempting a match. Any completion that matches the pattern will be removed from the list. A leading '!' negates the pattern; in this case any completion not matching the pattern will be removed. If the nocasematch shell option (see the description of shopt in Section 4.3.2 [The Shopt Builtin], page 65) is enabled, the match is performed without regard to the case of alphabetic characters.

Finally, any prefix and suffix specified with the -P and -S options are added to each member of the completion list, and the result is returned to the Readline completion code as the list of possible completions.

If the previously-applied actions do not generate any matches, and the -o dirnames option was supplied to complete when the compspec was defined, directory name completion is attempted.

If the -o plusdirs option was supplied to complete when the compspec was defined, directory name completion is attempted and any matches are added to the results of the other actions.

By default, if a compspec is found, whatever it generates is returned to the completion code as the full set of possible completions. The default Bash completions are not attempted, and the Readline default of filename completion is disabled. If the -o bashdefault option was supplied to complete when the compspec was defined, the default Bash completions are attempted if the compspec generates no matches. If the -o default option was supplied to complete when the compspec was defined, Readline's default completion will be performed if the compspec (and, if attempted, the default Bash completions) generate no matches.

When a compspec indicates that directory name completion is desired, the programmable completion functions force Readline to append a slash to completed names which are symbolic links to directories, subject to the value of the *mark-directories* Readline variable, regardless of the setting of the *mark-symlinked-directories* Readline variable.

There is some support for dynamically modifying completions. This is most useful when used in combination with a default completion specified with -D. It's possible for shell functions executed as completion handlers to indicate that completion should be retried by returning an exit status of 124. If a shell function returns 124, and changes the compspec associated with the command on which completion is being attempted (supplied as the first argument when the function is executed), programmable completion restarts from the beginning, with an attempt to find a new compspec for that command. This allows a set of completions to be built dynamically as completion is attempted, rather than being loaded all at once.

For instance, assuming that there is a library of compspecs, each kept in a file corresponding to the name of the command, the following default completion function would load completions dynamically:

```
_completion_loader()
{
    . "/etc/bash_completion.d/$1.sh" >/dev/null 2>&1 && return 124
}
complete -D -F _completion_loader -o bashdefault -o default
```

8.7 Programmable Completion Builtins

Three builtin commands are available to manipulate the programmable completion facilities: one to specify how the arguments to a particular command are to be completed, and two to modify the completion as it is happening.

`compgen`

: `compgen [`*option*`] [`*word*`]`

Generate possible completion matches for *word* according to the *option*s, which may be any option accepted by the `complete` builtin with the exception of `-p` and `-r`, and write the matches to the standard output. When using the `-F` or `-C` options, the various shell variables set by the programmable completion facilities, while available, will not have useful values.

The matches will be generated in the same way as if the programmable completion code had generated them directly from a completion specification with the same flags. If *word* is specified, only those completions matching *word* will be displayed.

The return value is true unless an invalid option is supplied, or no matches were generated.

`complete`

: `complete [-abcdefgjksuv] [-o` *comp-option*`] [-DEI] [-A` *action*`] [-`
`G` *globpat*`]`
`[-W` *wordlist*`] [-F` *function*`] [-C` *command*`] [-X` *filterpat*`]`
`[-P` *prefix*`] [-S` *suffix*`]` *name* `[`*name* `...]`
`complete -pr [-DEI] [`*name* `...]`

Specify how arguments to each *name* should be completed. If the `-p` option is supplied, or if no options are supplied, existing completion specifications are printed in a way that allows them to be reused as input. The `-r` option removes a completion specification for each *name*, or, if no *name*s are supplied, all completion specifications. The `-D` option indicates that other supplied options and actions should apply to the "default" command completion; that is, completion attempted on a command for which no completion has previously been defined. The `-E` option indicates that other supplied options and actions should apply to "empty" command completion; that is, completion attempted on a blank line. The `-I` option indicates that other supplied options and actions should apply to completion on the initial non-assignment word on the line, or after a command delimiter such as '`;`' or '`|`', which is usually command name completion. If multiple options are supplied, the `-D` option takes precedence over `-E`, and both take precedence over `-I`. If any of `-D`, `-E`, or `-I` are supplied, any other *name* arguments are ignored; these completions only apply to the case specified by the option.

The process of applying these completion specifications when word completion is attempted is described above (see Section 8.6 [Programmable Completion], page 134).

Other options, if specified, have the following meanings. The arguments to the `-G`, `-W`, and `-X` options (and, if necessary, the `-P` and `-S` options) should be quoted to protect them from expansion before the `complete` builtin is invoked.

`-o` *`comp-option`*

The *comp-option* controls several aspects of the compspec's behavior beyond the simple generation of completions. *comp-option* may be one of:

`bashdefault`

Perform the rest of the default Bash completions if the compspec generates no matches.

`default` Use Readline's default filename completion if the compspec generates no matches.

`dirnames` Perform directory name completion if the compspec generates no matches.

`filenames`

Tell Readline that the compspec generates filenames, so it can perform any filename-specific processing (like adding a slash to directory names, quoting special characters, or suppressing trailing spaces). This option is intended to be used with shell functions specified with `-F`.

`noquote` Tell Readline not to quote the completed words if they are filenames (quoting filenames is the default).

`nosort` Tell Readline not to sort the list of possible completions alphabetically.

`nospace` Tell Readline not to append a space (the default) to words completed at the end of the line.

`plusdirs` After any matches defined by the compspec are generated, directory name completion is attempted and any matches are added to the results of the other actions.

`-A` *action* The *action* may be one of the following to generate a list of possible completions:

`alias` Alias names. May also be specified as `-a`.

`arrayvar` Array variable names.

`binding` Readline key binding names (see Section 8.4 [Bindable Readline Commands], page 124).

`builtin` Names of shell builtin commands. May also be specified as `-b`.

`command` Command names. May also be specified as `-c`.

`directory`
Directory names. May also be specified as `-d`.

`disabled` Names of disabled shell builtins.

`enabled` Names of enabled shell builtins.

`export` Names of exported shell variables. May also be specified as `-e`.

`file` File names. May also be specified as `-f`.

`function` Names of shell functions.

`group` Group names. May also be specified as `-g`.

`helptopic`
Help topics as accepted by the `help` builtin (see Section 4.2 [Bash Builtins], page 50).

`hostname` Hostnames, as taken from the file specified by the `HOSTFILE` shell variable (see Section 5.2 [Bash Variables], page 73).

`job` Job names, if job control is active. May also be specified as `-j`.

`keyword` Shell reserved words. May also be specified as `-k`.

`running` Names of running jobs, if job control is active.

`service` Service names. May also be specified as `-s`.

`setopt` Valid arguments for the `-o` option to the `set` builtin (see Section 4.3.1 [The Set Builtin], page 61).

`shopt` Shell option names as accepted by the `shopt` builtin (see Section 4.2 [Bash Builtins], page 50).

`signal` Signal names.

`stopped` Names of stopped jobs, if job control is active.

`user` User names. May also be specified as `-u`.

`variable` Names of all shell variables. May also be specified as `-v`.

`-C` *command*
: *command* is executed in a subshell environment, and its output is used as the possible completions.

`-F` *function*
: The shell function *function* is executed in the current shell environment. When it is executed, $1 is the name of the command whose arguments are being completed, $2 is the word being completed, and $3 is the word preceding the word being completed, as described above (see Section 8.6 [Programmable Completion], page 134). When it finishes, the possible completions are retrieved from the value of the `COMPREPLY` array variable.

`-G` *globpat*
: The filename expansion pattern *globpat* is expanded to generate the possible completions.

`-P` *prefix* *prefix* is added at the beginning of each possible completion after all other options have been applied.

`-S` *suffix* *suffix* is appended to each possible completion after all other options have been applied.

`-W` *wordlist*
: The *wordlist* is split using the characters in the `IFS` special variable as delimiters, and each resultant word is expanded. The possible completions are the members of the resultant list which match the word being completed.

`-X` *filterpat*
: *filterpat* is a pattern as used for filename expansion. It is applied to the list of possible completions generated by the preceding options and arguments, and each completion matching *filterpat* is removed from the list. A leading '`!`' in *filterpat* negates the pattern; in this case, any completion not matching *filterpat* is removed.

The return value is true unless an invalid option is supplied, an option other than `-p` or `-r` is supplied without a *name* argument, an attempt is made to remove a completion specification for a *name* for which no specification exists, or an error occurs adding a completion specification.

`compopt`

```
compopt [-o option] [-DEI] [+o option] [name]
```

Modify completion options for each *name* according to the *options*, or for the currently-executing completion if no *names* are supplied. If no *options* are given, display the completion options for each *name* or the current completion.

The possible values of *option* are those valid for the `complete` builtin described above. The `-D` option indicates that other supplied options should apply to the "default" command completion; that is, completion attempted on a command for which no completion has previously been defined. The `-E` option indicates that other supplied options should apply to "empty" command completion; that is, completion attempted on a blank line. The `-I` option indicates that other supplied options should apply to completion on the initial non-assignment word on the line, or after a command delimiter such as '`;`' or '`|`', which is usually command name completion.

If multiple options are supplied, the `-D` option takes precedence over `-E`, and both take precedence over `-I`

The return value is true unless an invalid option is supplied, an attempt is made to modify the options for a *name* for which no completion specification exists, or an output error occurs.

8.8 A Programmable Completion Example

The most common way to obtain additional completion functionality beyond the default actions `complete` and `compgen` provide is to use a shell function and bind it to a particular command using `complete -F`.

The following function provides completions for the `cd` builtin. It is a reasonably good example of what shell functions must do when used for completion. This function uses the word passed as `$2` to determine the directory name to complete. You can also use the `COMP_WORDS` array variable; the current word is indexed by the `COMP_CWORD` variable.

The function relies on the `complete` and `compgen` builtins to do much of the work, adding only the things that the Bash `cd` does beyond accepting basic directory names: tilde expansion (see Section 3.5.2 [Tilde Expansion], page 24), searching directories in *$CDPATH*, which is described above (see Section 4.1 [Bourne Shell Builtins], page 43), and basic support for the `cdable_vars` shell option (see Section 4.3.2 [The Shopt Builtin], page 65). `_comp_cd` modifies the value of *IFS* so that it contains only a newline to accommodate file names containing spaces and tabs – `compgen` prints the possible completions it generates one per line.

Possible completions go into the *COMPREPLY* array variable, one completion per array element. The programmable completion system retrieves the completions from there when the function returns.

```
# A completion function for the cd builtin
# based on the cd completion function from the bash_completion package
_comp_cd()
{
    local IFS=$' \t\n'    # normalize IFS
    local cur _skipdot _cdpath
    local i j k

    # Tilde expansion, which also expands tilde to full pathname
    case "$2" in
    \~*)    eval cur="$2" ;;
```

```
        *)      cur=$2 ;;
        esac

        # no cdpath or absolute pathname -- straight directory completion
        if [[ -z "${CDPATH:-}" ]] || [[ "$cur" == @(./*|../*|/*) ]]; then
            # compgen prints paths one per line; could also use while loop
            IFS=$'\n'
            COMPREPLY=( $(compgen -d -- "$cur") )
            IFS=$' \t\n'
        # CDPATH+directories in the current directory if not in CDPATH
        else
            IFS=$'\n'
            _skipdot=false
            # preprocess CDPATH to convert null directory names to .
            _cdpath=${CDPATH/#:/.:}
            _cdpath=${_cdpath//::/:.:}
            _cdpath=${_cdpath/%:/:.}
            for i in ${_cdpath//:/$'\n'}; do
                if [[ $i -ef . ]]; then _skipdot=true; fi
                k="${#COMPREPLY[@]}"
                for j in $( compgen -d -- "$i/$cur" ); do
                    COMPREPLY[k++]=${j#$i/}        # cut off directory
                done
            done
            $_skipdot || COMPREPLY+=( $(compgen -d -- "$cur") )
            IFS=$' \t\n'
        fi

        # variable names if appropriate shell option set and no completions
        if shopt -q cdable_vars && [[ ${#COMPREPLY[@]} -eq 0 ]]; then
            COMPREPLY=( $(compgen -v -- "$cur") )
        fi

        return 0
    }
```

We install the completion function using the `-F` option to `complete`:

```
# Tell readline to quote appropriate and append slashes to directories;
# use the bash default completion for other arguments
complete -o filenames -o nospace -o bashdefault -F _comp_cd cd
```

Since we'd like Bash and Readline to take care of some of the other details for us, we use several other options to tell Bash and Readline what to do. The `-o filenames` option tells Readline that the possible completions should be treated as filenames, and quoted appropriately. That option will also cause Readline to append a slash to filenames it can determine are directories (which is why we might want to extend `_comp_cd` to append a slash if we're using directories found via *CDPATH*: Readline can't tell those completions are directories). The `-o nospace` option tells Readline to not append a space character to the directory name, in case we want to append to it. The `-o bashdefault` option brings in the rest of the "Bash default" completions – possible completion that Bash adds to the default Readline set. These include things like command name completion, variable completion for words beginning with '`$`' or '`${`', completions containing pathname expansion patterns (see Section 3.5.8 [Filename Expansion], page 32), and so on.

Once installed using `complete`, `_comp_cd` will be called every time we attempt word completion for a `cd` command.

Many more examples – an extensive collection of completions for most of the common GNU, Unix, and Linux commands – are available as part of the bash_completion project. This is installed by default on many GNU/Linux distributions. Originally written by Ian Macdonald, the project now lives at `https://github.com/scop/bash-completion/`. There are ports for other systems such as Solaris and Mac OS X.

An older version of the bash_completion package is distributed with bash in the `examples/complete` subdirectory.

9 Using History Interactively

This chapter describes how to use the GNU History Library interactively, from a user's standpoint. It should be considered a user's guide. For information on using the GNU History Library in other programs, see the GNU Readline Library Manual.

9.1 Bash History Facilities

When the `-o history` option to the `set` builtin is enabled (see Section 4.3.1 [The Set Builtin], page 61), the shell provides access to the *command history*, the list of commands previously typed. The value of the `HISTSIZE` shell variable is used as the number of commands to save in a history list. The text of the last `$HISTSIZE` commands (default 500) is saved. The shell stores each command in the history list prior to parameter and variable expansion but after history expansion is performed, subject to the values of the shell variables `HISTIGNORE` and `HISTCONTROL`.

When the shell starts up, the history is initialized from the file named by the `HISTFILE` variable (default `~/.bash_history`). The file named by the value of `HISTFILE` is truncated, if necessary, to contain no more than the number of lines specified by the value of the `HISTFILESIZE` variable. When a shell with history enabled exits, the last `$HISTSIZE` lines are copied from the history list to the file named by `$HISTFILE`. If the `histappend` shell option is set (see Section 4.2 [Bash Builtins], page 50), the lines are appended to the history file, otherwise the history file is overwritten. If `HISTFILE` is unset, or if the history file is unwritable, the history is not saved. After saving the history, the history file is truncated to contain no more than `$HISTFILESIZE` lines. If `HISTFILESIZE` is unset, or set to null, a non-numeric value, or a numeric value less than zero, the history file is not truncated.

If the `HISTTIMEFORMAT` is set, the time stamp information associated with each history entry is written to the history file, marked with the history comment character. When the history file is read, lines beginning with the history comment character followed immediately by a digit are interpreted as timestamps for the following history entry.

The builtin command `fc` may be used to list or edit and re-execute a portion of the history list. The `history` builtin may be used to display or modify the history list and manipulate the history file. When using command-line editing, search commands are available in each editing mode that provide access to the history list (see Section 8.4.2 [Commands For History], page 125).

The shell allows control over which commands are saved on the history list. The `HISTCONTROL` and `HISTIGNORE` variables may be set to cause the shell to save only a subset of the commands entered. The `cmdhist` shell option, if enabled, causes the shell to attempt to save each line of a multi-line command in the same history entry, adding semicolons where necessary to preserve syntactic correctness. The `lithist` shell option causes the shell to save the command with embedded newlines instead of semicolons. The `shopt` builtin is used to set these options. See Section 4.3.2 [The Shopt Builtin], page 65, for a description of `shopt`.

9.2 Bash History Builtins

Bash provides two builtin commands which manipulate the history list and history file.

`fc`

```
fc [-e ename] [-lnr] [first] [last]
fc -s [pat=rep] [command]
```

The first form selects a range of commands from *first* to *last* from the history list and displays or edits and re-executes them. Both *first* and *last* may be specified as a string (to locate the most recent command beginning with that string) or as a number (an index into the history list, where a negative number is used as an offset from the current command number). If *last* is not specified, it is set to *first*. If *first* is not specified, it is set to the previous command for editing and −16 for listing. If the `-l` flag is given, the commands are listed on standard output. The `-n` flag suppresses the command numbers when listing. The `-r` flag reverses the order of the listing. Otherwise, the editor given by *ename* is invoked on a file containing those commands. If *ename* is not given, the value of the following variable expansion is used: `${FCEDIT:-${EDITOR:-vi}}`. This says to use the value of the `FCEDIT` variable if set, or the value of the `EDITOR` variable if that is set, or `vi` if neither is set. When editing is complete, the edited commands are echoed and executed.

In the second form, *command* is re-executed after each instance of *pat* in the selected command is replaced by *rep*. *command* is intepreted the same as *first* above.

A useful alias to use with the `fc` command is `r='fc -s'`, so that typing '`r cc`' runs the last command beginning with `cc` and typing '`r`' re-executes the last command (see Section 6.6 [Aliases], page 93).

`history`

```
history [n]
history -c
history -d offset
history -d start-end
history [-anrw] [filename]
history -ps arg
```

With no options, display the history list with line numbers. Lines prefixed with a '`*`' have been modified. An argument of *n* lists only the last *n* lines. If the shell variable `HISTTIMEFORMAT` is set and not null, it is used as a format string for *strftime* to display the time stamp associated with each displayed history entry. No intervening blank is printed between the formatted time stamp and the history line.

Options, if supplied, have the following meanings:

`-c` Clear the history list. This may be combined with the other options to replace the history list completely.

`-d` *offset* Delete the history entry at position *offset*. If *offset* is positive, it should be specified as it appears when the history is displayed. If *offset* is negative, it is interpreted as relative to one greater than the last history position, so negative indices count back from the end of the history, and an index of '`-1`' refers to the current `history -d` command.

`-d start-end`
: Delete the history entries between positions *start* and *end*, inclusive. Positive and negative values for *start* and *end* are interpreted as described above.

`-a`
: Append the new history lines to the history file. These are history lines entered since the beginning of the current Bash session, but not already appended to the history file.

`-n`
: Append the history lines not already read from the history file to the current history list. These are lines appended to the history file since the beginning of the current Bash session.

`-r`
: Read the history file and append its contents to the history list.

`-w`
: Write out the current history list to the history file.

`-p`
: Perform history substitution on the *args* and display the result on the standard output, without storing the results in the history list.

`-s`
: The *args* are added to the end of the history list as a single entry.

When any of the `-w`, `-r`, `-a`, or `-n` options is used, if *filename* is given, then it is used as the history file. If not, then the value of the `HISTFILE` variable is used.

9.3 History Expansion

The History library provides a history expansion feature that is similar to the history expansion provided by `csh`. This section describes the syntax used to manipulate the history information.

History expansions introduce words from the history list into the input stream, making it easy to repeat commands, insert the arguments to a previous command into the current input line, or fix errors in previous commands quickly.

History expansion is performed immediately after a complete line is read, before the shell breaks it into words, and is performed on each line individually. Bash attempts to inform the history expansion functions about quoting still in effect from previous lines.

History expansion takes place in two parts. The first is to determine which line from the history list should be used during substitution. The second is to select portions of that line for inclusion into the current one. The line selected from the history is called the *event*, and the portions of that line that are acted upon are called *words*. Various *modifiers* are available to manipulate the selected words. The line is broken into words in the same fashion that Bash does, so that several words surrounded by quotes are considered one word. History expansions are introduced by the appearance of the history expansion character, which is '`!`' by default.

History expansion implements shell-like quoting conventions: a backslash can be used to remove the special handling for the next character; single quotes enclose verbatim sequences of characters, and can be used to inhibit history expansion; and characters enclosed within double quotes may be subject to history expansion, since backslash can escape the history expansion character, but single quotes may not, since they are not treated specially within double quotes.

When using the shell, only '\' and ''' may be used to escape the history expansion character, but the history expansion character is also treated as quoted if it immediately precedes the closing double quote in a double-quoted string.

Several shell options settable with the `shopt` builtin (see Section 4.3.2 [The Shopt Builtin], page 65) may be used to tailor the behavior of history expansion. If the `histverify` shell option is enabled, and Readline is being used, history substitutions are not immediately passed to the shell parser. Instead, the expanded line is reloaded into the Readline editing buffer for further modification. If Readline is being used, and the `histreedit` shell option is enabled, a failed history expansion will be reloaded into the Readline editing buffer for correction. The `-p` option to the `history` builtin command may be used to see what a history expansion will do before using it. The `-s` option to the `history` builtin may be used to add commands to the end of the history list without actually executing them, so that they are available for subsequent recall. This is most useful in conjunction with Readline.

The shell allows control of the various characters used by the history expansion mechanism with the `histchars` variable, as explained above (see Section 5.2 [Bash Variables], page 73). The shell uses the history comment character to mark history timestamps when writing the history file.

9.3.1 Event Designators

An event designator is a reference to a command line entry in the history list. Unless the reference is absolute, events are relative to the current position in the history list.

`!`
: Start a history substitution, except when followed by a space, tab, the end of the line, '=' or '(' (when the `extglob` shell option is enabled using the `shopt` builtin).

`!n`
: Refer to command line *n*.

`!-n`
: Refer to the command *n* lines back.

`!!`
: Refer to the previous command. This is a synonym for '`!-1`'.

`!string`
: Refer to the most recent command preceding the current position in the history list starting with *string*.

`!?string[?]`
: Refer to the most recent command preceding the current position in the history list containing *string*. The trailing '`?`' may be omitted if the *string* is followed immediately by a newline.

`^string1^string2^`
: Quick Substitution. Repeat the last command, replacing *string1* with *string2*. Equivalent to `!!:s/string1/string2/`.

`!#`
: The entire command line typed so far.

9.3.2 Word Designators

Word designators are used to select desired words from the event. A '`:`' separates the event specification from the word designator. It may be omitted if the word designator begins with a '`^`', '`$`', '`*`', '`-`', or '`%`'. Words are numbered from the beginning of the line, with the first word being denoted by 0 (zero). Words are inserted into the current line separated by single spaces.

For example,

`!!` designates the preceding command. When you type this, the preceding command is repeated in toto.

`!!:$` designates the last argument of the preceding command. This may be shortened to `!$`.

`!fi:2` designates the second argument of the most recent command starting with the letters `fi`.

Here are the word designators:

`0 (zero)` The `0`th word. For many applications, this is the command word.

n The *n*th word.

`^` The first argument; that is, word 1.

`$` The last argument.

`%` The word matched by the most recent '`?`*string*`?`' search.

x`-`*y* A range of words; '`-`*y*' abbreviates '`0-`*y*'.

`*` All of the words, except the `0`th. This is a synonym for '`1-$`'. It is not an error to use '`*`' if there is just one word in the event; the empty string is returned in that case.

x`*` Abbreviates '*x*`-$`'

x`-` Abbreviates '*x*`-$`' like '*x*`*`', but omits the last word.

If a word designator is supplied without an event specification, the previous command is used as the event.

9.3.3 Modifiers

After the optional word designator, you can add a sequence of one or more of the following modifiers, each preceded by a '`:`'.

`h` Remove a trailing pathname component, leaving only the head.

`t` Remove all leading pathname components, leaving the tail.

`r` Remove a trailing suffix of the form '`.`*suffix*', leaving the basename.

`e` Remove all but the trailing suffix.

`p` Print the new command but do not execute it.

`q` Quote the substituted words, escaping further substitutions.

`x` Quote the substituted words as with '`q`', but break into words at spaces, tabs, and newlines.

`s/old/new/`
: Substitute *new* for the first occurrence of *old* in the event line. Any delimiter may be used in place of '`/`'. The delimiter may be quoted in *old* and *new* with a single backslash. If '`&`' appears in *new*, it is replaced by *old*. A single backslash will quote the '`&`'. The final delimiter is optional if it is the last character on the input line.

`&`
: Repeat the previous substitution.

`g`
`a`
: Cause changes to be applied over the entire event line. Used in conjunction with '`s`', as in `gs/old/new/`, or with '`&`'.

`G`
: Apply the following '`s`' modifier once to each word in the event.

10 Installing Bash

This chapter provides basic instructions for installing Bash on the various supported platforms. The distribution supports the GNU operating systems, nearly every version of Unix, and several non-Unix systems such as BeOS and Interix. Other independent ports exist for MS-DOS, OS/2, and Windows platforms.

10.1 Basic Installation

These are installation instructions for Bash.

The simplest way to compile Bash is:

1. `cd` to the directory containing the source code and type '`./configure`' to configure Bash for your system. If you're using `csh` on an old version of System V, you might need to type '`sh ./configure`' instead to prevent `csh` from trying to execute `configure` itself.

   Running `configure` takes some time. While running, it prints messages telling which features it is checking for.
2. Type '`make`' to compile Bash and build the `bashbug` bug reporting script.
3. Optionally, type '`make tests`' to run the Bash test suite.
4. Type '`make install`' to install `bash` and `bashbug`. This will also install the manual pages and Info file.

The `configure` shell script attempts to guess correct values for various system-dependent variables used during compilation. It uses those values to create a `Makefile` in each directory of the package (the top directory, the `builtins`, `doc`, and `support` directories, each directory under `lib`, and several others). It also creates a `config.h` file containing system-dependent definitions. Finally, it creates a shell script named `config.status` that you can run in the future to recreate the current configuration, a file `config.cache` that saves the results of its tests to speed up reconfiguring, and a file `config.log` containing compiler output (useful mainly for debugging `configure`). If at some point `config.cache` contains results you don't want to keep, you may remove or edit it.

To find out more about the options and arguments that the `configure` script understands, type

```
bash-4.2$ ./configure --help
```

at the Bash prompt in your Bash source directory.

If you want to build Bash in a directory separate from the source directory – to build for multiple architectures, for example – just use the full path to the configure script. The following commands will build bash in a directory under `/usr/local/build` from the source code in `/usr/local/src/bash-4.4`:

```
mkdir /usr/local/build/bash-4.4
cd /usr/local/build/bash-4.4
bash /usr/local/src/bash-4.4/configure
make
```

See Section 10.3 [Compiling For Multiple Architectures], page 150, for more information about building in a directory separate from the source.

If you need to do unusual things to compile Bash, please try to figure out how `configure` could check whether or not to do them, and mail diffs or instructions to `bash-maintainers@gnu.org` so they can be considered for the next release.

The file `configure.ac` is used to create `configure` by a program called Autoconf. You only need `configure.ac` if you want to change it or regenerate `configure` using a newer version of Autoconf. If you do this, make sure you are using Autoconf version 2.50 or newer.

You can remove the program binaries and object files from the source code directory by typing '`make clean`'. To also remove the files that `configure` created (so you can compile Bash for a different kind of computer), type '`make distclean`'.

10.2 Compilers and Options

Some systems require unusual options for compilation or linking that the `configure` script does not know about. You can give `configure` initial values for variables by setting them in the environment. Using a Bourne-compatible shell, you can do that on the command line like this:

```
CC=c89 CFLAGS=-O2 LIBS=-lposix ./configure
```

On systems that have the `env` program, you can do it like this:

```
env CPPFLAGS=-I/usr/local/include LDFLAGS=-s ./configure
```

The configuration process uses GCC to build Bash if it is available.

10.3 Compiling For Multiple Architectures

You can compile Bash for more than one kind of computer at the same time, by placing the object files for each architecture in their own directory. To do this, you must use a version of `make` that supports the `VPATH` variable, such as GNU `make`. `cd` to the directory where you want the object files and executables to go and run the `configure` script from the source directory (see Section 10.1 [Basic Installation], page 149). You may need to supply the `--srcdir=PATH` argument to tell `configure` where the source files are. `configure` automatically checks for the source code in the directory that `configure` is in and in '..'.

If you have to use a `make` that does not supports the `VPATH` variable, you can compile Bash for one architecture at a time in the source code directory. After you have installed Bash for one architecture, use '`make distclean`' before reconfiguring for another architecture.

Alternatively, if your system supports symbolic links, you can use the `support/mkclone` script to create a build tree which has symbolic links back to each file in the source directory. Here's an example that creates a build directory in the current directory from a source directory `/usr/gnu/src/bash-2.0`:

```
bash /usr/gnu/src/bash-2.0/support/mkclone -s /usr/gnu/src/bash-2.0 .
```

The `mkclone` script requires Bash, so you must have already built Bash for at least one architecture before you can create build directories for other architectures.

10.4 Installation Names

By default, '`make install`' will install into `/usr/local/bin`, `/usr/local/man`, etc. You can specify an installation prefix other than `/usr/local` by giving `configure` the option

`--prefix=PATH`, or by specifying a value for the `DESTDIR` '`make`' variable when running '`make install`'.

You can specify separate installation prefixes for architecture-specific files and architecture-independent files. If you give `configure` the option `--exec-prefix=PATH`, '`make install`' will use *PATH* as the prefix for installing programs and libraries. Documentation and other data files will still use the regular prefix.

10.5 Specifying the System Type

There may be some features `configure` can not figure out automatically, but need to determine by the type of host Bash will run on. Usually `configure` can figure that out, but if it prints a message saying it can not guess the host type, give it the `--host=TYPE` option. '`TYPE`' can either be a short name for the system type, such as '`sun4`', or a canonical name with three fields: '`CPU-COMPANY-SYSTEM`' (e.g., '`i386-unknown-freebsd4.2`').

See the file `support/config.sub` for the possible values of each field.

10.6 Sharing Defaults

If you want to set default values for `configure` scripts to share, you can create a site shell script called `config.site` that gives default values for variables like `CC`, `cache_file`, and `prefix`. `configure` looks for `PREFIX/share/config.site` if it exists, then `PREFIX/etc/config.site` if it exists. Or, you can set the `CONFIG_SITE` environment variable to the location of the site script. A warning: the Bash `configure` looks for a site script, but not all `configure` scripts do.

10.7 Operation Controls

`configure` recognizes the following options to control how it operates.

`--cache-file=file`
: Use and save the results of the tests in *file* instead of `./config.cache`. Set *file* to `/dev/null` to disable caching, for debugging `configure`.

`--help`
: Print a summary of the options to `configure`, and exit.

`--quiet`
`--silent`
`-q`
: Do not print messages saying which checks are being made.

`--srcdir=dir`
: Look for the Bash source code in directory *dir*. Usually `configure` can determine that directory automatically.

`--version`
: Print the version of Autoconf used to generate the `configure` script, and exit.

`configure` also accepts some other, not widely used, boilerplate options. '`configure --help`' prints the complete list.

10.8 Optional Features

The Bash `configure` has a number of `--enable-feature` options, where *feature* indicates an optional part of Bash. There are also several `--with-package` options, where *package* is something like '`bash-malloc`' or '`purify`'. To turn off the default use of a package, use `--without-package`. To configure Bash without a feature that is enabled by default, use `--disable-feature`.

Here is a complete list of the `--enable-` and `--with-` options that the Bash `configure` recognizes.

`--with-afs`
: Define if you are using the Andrew File System from Transarc.

`--with-bash-malloc`
: Use the Bash version of `malloc` in the directory `lib/malloc`. This is not the same `malloc` that appears in GNU libc, but an older version originally derived from the 4.2 BSD `malloc`. This `malloc` is very fast, but wastes some space on each allocation. This option is enabled by default. The `NOTES` file contains a list of systems for which this should be turned off, and `configure` disables this option automatically for a number of systems.

`--with-curses`
: Use the curses library instead of the termcap library. This should be supplied if your system has an inadequate or incomplete termcap database.

`--with-gnu-malloc`
: A synonym for `--with-bash-malloc`.

`--with-installed-readline[=PREFIX]`
: Define this to make Bash link with a locally-installed version of Readline rather than the version in `lib/readline`. This works only with Readline 5.0 and later versions. If *PREFIX* is `yes` or not supplied, `configure` uses the values of the make variables `includedir` and `libdir`, which are subdirectories of `prefix` by default, to find the installed version of Readline if it is not in the standard system include and library directories. If *PREFIX* is `no`, Bash links with the version in `lib/readline`. If *PREFIX* is set to any other value, `configure` treats it as a directory pathname and looks for the installed version of Readline in subdirectories of that directory (include files in *PREFIX*/`include` and the library in *PREFIX*/`lib`).

`--with-purify`
: Define this to use the Purify memory allocation checker from Rational Software.

`--enable-minimal-config`
: This produces a shell with minimal features, close to the historical Bourne shell.

There are several `--enable-` options that alter how Bash is compiled and linked, rather than changing run-time features.

`--enable-largefile`
: Enable support for large files (`http://www.unix.org/version2/whatsnew/lfs20mar.html`) if the operating system requires special compiler options to

build programs which can access large files. This is enabled by default, if the operating system provides large file support.

`--enable-profiling`
: This builds a Bash binary that produces profiling information to be processed by `gprof` each time it is executed.

`--enable-static-link`
: This causes Bash to be linked statically, if `gcc` is being used. This could be used to build a version to use as root's shell.

The '`minimal-config`' option can be used to disable all of the following options, but it is processed first, so individual options may be enabled using '`enable-`*`feature`*'.

All of the following options except for '`disabled-builtins`', '`direxpand-default`', and '`xpg-echo-default`' are enabled by default, unless the operating system does not provide the necessary support.

`--enable-alias`
: Allow alias expansion and include the `alias` and `unalias` builtins (see Section 6.6 [Aliases], page 93).

`--enable-arith-for-command`
: Include support for the alternate form of the `for` command that behaves like the C language `for` statement (see Section 3.2.4.1 [Looping Constructs], page 10).

`--enable-array-variables`
: Include support for one-dimensional array shell variables (see Section 6.7 [Arrays], page 94).

`--enable-bang-history`
: Include support for `csh`-like history substitution (see Section 9.3 [History Interaction], page 145).

`--enable-brace-expansion`
: Include `csh`-like brace expansion (`b{a,b}c` ↦ `bac bbc`). See Section 3.5.1 [Brace Expansion], page 23, for a complete description.

`--enable-casemod-attributes`
: Include support for case-modifying attributes in the `declare` builtin and assignment statements. Variables with the *uppercase* attribute, for example, will have their values converted to uppercase upon assignment.

`--enable-casemod-expansion`
: Include support for case-modifying word expansions.

`--enable-command-timing`
: Include support for recognizing `time` as a reserved word and for displaying timing statistics for the pipeline following `time` (see Section 3.2.2 [Pipelines], page 8). This allows pipelines as well as shell builtins and functions to be timed.

`--enable-cond-command`
: Include support for the `[[` conditional command. (see Section 3.2.4.2 [Conditional Constructs], page 11).

`--enable-cond-regexp`
: Include support for matching POSIX regular expressions using the '`=~`' binary operator in the `[[` conditional command. (see Section 3.2.4.2 [Conditional Constructs], page 11).

`--enable-coprocesses`
: Include support for coprocesses and the `coproc` reserved word (see Section 3.2.2 [Pipelines], page 8).

`--enable-debugger`
: Include support for the bash debugger (distributed separately).

`--enable-dev-fd-stat-broken`
: If calling `stat` on /dev/fd/*N* returns different results than calling `fstat` on file descriptor *N*, supply this option to enable a workaround. This has implications for conditional commands that test file attributes.

`--enable-direxpand-default`
: Cause the `direxpand` shell option (see Section 4.3.2 [The Shopt Builtin], page 65) to be enabled by default when the shell starts. It is normally disabled by default.

`--enable-directory-stack`
: Include support for a `csh`-like directory stack and the `pushd`, `popd`, and `dirs` builtins (see Section 6.8 [The Directory Stack], page 96).

`--enable-disabled-builtins`
: Allow builtin commands to be invoked via '`builtin xxx`' even after `xxx` has been disabled using '`enable -n xxx`'. See Section 4.2 [Bash Builtins], page 50, for details of the `builtin` and `enable` builtin commands.

`--enable-dparen-arithmetic`
: Include support for the `((...))` command (see Section 3.2.4.2 [Conditional Constructs], page 11).

`--enable-extended-glob`
: Include support for the extended pattern matching features described above under Section 3.5.8.1 [Pattern Matching], page 33.

`--enable-extended-glob-default`
: Set the default value of the *extglob* shell option described above under Section 4.3.2 [The Shopt Builtin], page 65, to be enabled.

`--enable-function-import`
: Include support for importing function definitions exported by another instance of the shell from the environment. This option is enabled by default.

`--enable-glob-asciirange-default`
: Set the default value of the *globasciiranges* shell option described above under Section 4.3.2 [The Shopt Builtin], page 65, to be enabled. This controls the behavior of character ranges when used in pattern matching bracket expressions.

`--enable-help-builtin`
: Include the `help` builtin, which displays help on shell builtins and variables (see Section 4.2 [Bash Builtins], page 50).

`--enable-history`
: Include command history and the `fc` and `history` builtin commands (see Section 9.1 [Bash History Facilities], page 143).

`--enable-job-control`
: This enables the job control features (see Chapter 7 [Job Control], page 104), if the operating system supports them.

`--enable-multibyte`
: This enables support for multibyte characters if the operating system provides the necessary support.

`--enable-net-redirections`
: This enables the special handling of filenames of the form `/dev/tcp/`*host*`/`*port* and `/dev/udp/`*host*`/`*port* when used in redirections (see Section 3.6 [Redirections], page 34).

`--enable-process-substitution`
: This enables process substitution (see Section 3.5.6 [Process Substitution], page 31) if the operating system provides the necessary support.

`--enable-progcomp`
: Enable the programmable completion facilities (see Section 8.6 [Programmable Completion], page 134). If Readline is not enabled, this option has no effect.

`--enable-prompt-string-decoding`
: Turn on the interpretation of a number of backslash-escaped characters in the `$PS0`, `$PS1`, `$PS2`, and `$PS4` prompt strings. See Section 6.9 [Controlling the Prompt], page 97, for a complete list of prompt string escape sequences.

`--enable-readline`
: Include support for command-line editing and history with the Bash version of the Readline library (see Chapter 8 [Command Line Editing], page 108).

`--enable-restricted`
: Include support for a *restricted shell*. If this is enabled, Bash, when called as `rbash`, enters a restricted mode. See Section 6.10 [The Restricted Shell], page 99, for a description of restricted mode.

`--enable-select`
: Include the `select` compound command, which allows the generation of simple menus (see Section 3.2.4.2 [Conditional Constructs], page 11).

`--enable-separate-helpfiles`
: Use external files for the documentation displayed by the `help` builtin instead of storing the text internally.

`--enable-single-help-strings`
: Store the text displayed by the `help` builtin as a single string for each help topic. This aids in translating the text to different languages. You may need to disable this if your compiler cannot handle very long string literals.

`--enable-strict-posix-default`
: Make Bash POSIX-conformant by default (see Section 6.11 [Bash POSIX Mode], page 99).

`--enable-usg-echo-default`
: A synonym for `--enable-xpg-echo-default`.

`--enable-xpg-echo-default`
: Make the `echo` builtin expand backslash-escaped characters by default, without requiring the `-e` option. This sets the default value of the `xpg_echo` shell option to `on`, which makes the Bash `echo` behave more like the version specified in the Single Unix Specification, version 3. See Section 4.2 [Bash Builtins], page 50, for a description of the escape sequences that `echo` recognizes.

The file `config-top.h` contains C Preprocessor '`#define`' statements for options which are not settable from `configure`. Some of these are not meant to be changed; beware of the consequences if you do. Read the comments associated with each definition for more information about its effect.

Appendix A Reporting Bugs

Please report all bugs you find in Bash. But first, you should make sure that it really is a bug, and that it appears in the latest version of Bash. The latest version of Bash is always available for FTP from `ftp://ftp.gnu.org/pub/gnu/bash/`.

Once you have determined that a bug actually exists, use the `bashbug` command to submit a bug report. If you have a fix, you are encouraged to mail that as well! Suggestions and 'philosophical' bug reports may be mailed to `bug-bash@gnu.org` or posted to the Usenet newsgroup `gnu.bash.bug`.

All bug reports should include:

- The version number of Bash.
- The hardware and operating system.
- The compiler used to compile Bash.
- A description of the bug behaviour.
- A short script or 'recipe' which exercises the bug and may be used to reproduce it.

`bashbug` inserts the first three items automatically into the template it provides for filing a bug report.

Please send all reports concerning this manual to `bug-bash@gnu.org`.

Appendix B Major Differences From The Bourne Shell

Bash implements essentially the same grammar, parameter and variable expansion, redirection, and quoting as the Bourne Shell. Bash uses the POSIX standard as the specification of how these features are to be implemented. There are some differences between the traditional Bourne shell and Bash; this section quickly details the differences of significance. A number of these differences are explained in greater depth in previous sections. This section uses the version of `sh` included in SVR4.2 (the last version of the historical Bourne shell) as the baseline reference.

- Bash is POSIX-conformant, even where the POSIX specification differs from traditional `sh` behavior (see Section 6.11 [Bash POSIX Mode], page 99).
- Bash has multi-character invocation options (see Section 6.1 [Invoking Bash], page 85).
- Bash has command-line editing (see Chapter 8 [Command Line Editing], page 108) and the `bind` builtin.
- Bash provides a programmable word completion mechanism (see Section 8.6 [Programmable Completion], page 134), and builtin commands `complete`, `compgen`, and `compopt`, to manipulate it.
- Bash has command history (see Section 9.1 [Bash History Facilities], page 143) and the `history` and `fc` builtins to manipulate it. The Bash history list maintains timestamp information and uses the value of the `HISTTIMEFORMAT` variable to display it.
- Bash implements `csh`-like history expansion (see Section 9.3 [History Interaction], page 145).
- Bash has one-dimensional array variables (see Section 6.7 [Arrays], page 94), and the appropriate variable expansions and assignment syntax to use them. Several of the Bash builtins take options to act on arrays. Bash provides a number of built-in array variables.
- The `$'...'` quoting syntax, which expands ANSI-C backslash-escaped characters in the text between the single quotes, is supported (see Section 3.1.2.4 [ANSI-C Quoting], page 6).
- Bash supports the `$"..."` quoting syntax to do locale-specific translation of the characters between the double quotes. The `-D`, `--dump-strings`, and `--dump-po-strings` invocation options list the translatable strings found in a script (see Section 3.1.2.5 [Locale Translation], page 7).
- Bash implements the `!` keyword to negate the return value of a pipeline (see Section 3.2.2 [Pipelines], page 8). Very useful when an `if` statement needs to act only if a test fails. The Bash '`-o pipefail`' option to `set` will cause a pipeline to return a failure status if any command fails.
- Bash has the `time` reserved word and command timing (see Section 3.2.2 [Pipelines], page 8). The display of the timing statistics may be controlled with the `TIMEFORMAT` variable.
- Bash implements the `for (( expr1 ; expr2 ; expr3 ))` arithmetic for command, similar to the C language (see Section 3.2.4.1 [Looping Constructs], page 10).
- Bash includes the `select` compound command, which allows the generation of simple menus (see Section 3.2.4.2 [Conditional Constructs], page 11).

- Bash includes the `[[` compound command, which makes conditional testing part of the shell grammar (see Section 3.2.4.2 [Conditional Constructs], page 11), including optional regular expression matching.
- Bash provides optional case-insensitive matching for the `case` and `[[` constructs.
- Bash includes brace expansion (see Section 3.5.1 [Brace Expansion], page 23) and tilde expansion (see Section 3.5.2 [Tilde Expansion], page 24).
- Bash implements command aliases and the `alias` and `unalias` builtins (see Section 6.6 [Aliases], page 93).
- Bash provides shell arithmetic, the `((` compound command (see Section 3.2.4.2 [Conditional Constructs], page 11), and arithmetic expansion (see Section 6.5 [Shell Arithmetic], page 92).
- Variables present in the shell's initial environment are automatically exported to child processes. The Bourne shell does not normally do this unless the variables are explicitly marked using the `export` command.
- Bash supports the '`+=`' assignment operator, which appends to the value of the variable named on the left hand side.
- Bash includes the POSIX pattern removal '`%`', '`#`', '`%%`' and '`##`' expansions to remove leading or trailing substrings from variable values (see Section 3.5.3 [Shell Parameter Expansion], page 24).
- The expansion `${#xx}`, which returns the length of `${xx}`, is supported (see Section 3.5.3 [Shell Parameter Expansion], page 24).
- The expansion `${var:`*offset*`[:`*length*`]}`, which expands to the substring of `var`'s value of length *length*, beginning at *offset*, is present (see Section 3.5.3 [Shell Parameter Expansion], page 24).
- The expansion `${var/[/]`*pattern*`[/`*replacement*`]}`, which matches *pattern* and replaces it with *replacement* in the value of `var`, is available (see Section 3.5.3 [Shell Parameter Expansion], page 24).
- The expansion `${!`*prefix*`*}` expansion, which expands to the names of all shell variables whose names begin with *prefix*, is available (see Section 3.5.3 [Shell Parameter Expansion], page 24).
- Bash has *indirect* variable expansion using `${!word}` (see Section 3.5.3 [Shell Parameter Expansion], page 24).
- Bash can expand positional parameters beyond `$9` using `${`*num*`}`.
- The POSIX `$()` form of command substitution is implemented (see Section 3.5.4 [Command Substitution], page 31), and preferred to the Bourne shell's ```` `` ```` (which is also implemented for backwards compatibility).
- Bash has process substitution (see Section 3.5.6 [Process Substitution], page 31).
- Bash automatically assigns variables that provide information about the current user (`UID`, `EUID`, and `GROUPS`), the current host (`HOSTTYPE`, `OSTYPE`, `MACHTYPE`, and `HOSTNAME`), and the instance of Bash that is running (`BASH`, `BASH_VERSION`, and `BASH_VERSINFO`). See Section 5.2 [Bash Variables], page 73, for details.
- The `IFS` variable is used to split only the results of expansion, not all words (see Section 3.5.7 [Word Splitting], page 32). This closes a longstanding shell security hole.

- The filename expansion bracket expression code uses '`!`' and '`^`' to negate the set of characters between the brackets. The Bourne shell uses only '`!`'.
- Bash implements the full set of POSIX filename expansion operators, including *character classes*, *equivalence classes*, and *collating symbols* (see Section 3.5.8 [Filename Expansion], page 32).
- Bash implements extended pattern matching features when the `extglob` shell option is enabled (see Section 3.5.8.1 [Pattern Matching], page 33).
- It is possible to have a variable and a function with the same name; `sh` does not separate the two name spaces.
- Bash functions are permitted to have local variables using the `local` builtin, and thus useful recursive functions may be written (see Section 4.2 [Bash Builtins], page 50).
- Variable assignments preceding commands affect only that command, even builtins and functions (see Section 3.7.4 [Environment], page 40). In `sh`, all variable assignments preceding commands are global unless the command is executed from the file system.
- Bash performs filename expansion on filenames specified as operands to input and output redirection operators (see Section 3.6 [Redirections], page 34).
- Bash contains the '`<>`' redirection operator, allowing a file to be opened for both reading and writing, and the '`&>`' redirection operator, for directing standard output and standard error to the same file (see Section 3.6 [Redirections], page 34).
- Bash includes the '`<<<`' redirection operator, allowing a string to be used as the standard input to a command.
- Bash implements the '`[n]<&`*`word`*' and '`[n]>&`*`word`*' redirection operators, which move one file descriptor to another.
- Bash treats a number of filenames specially when they are used in redirection operators (see Section 3.6 [Redirections], page 34).
- Bash can open network connections to arbitrary machines and services with the redirection operators (see Section 3.6 [Redirections], page 34).
- The `noclobber` option is available to avoid overwriting existing files with output redirection (see Section 4.3.1 [The Set Builtin], page 61). The '`>|`' redirection operator may be used to override `noclobber`.
- The Bash `cd` and `pwd` builtins (see Section 4.1 [Bourne Shell Builtins], page 43) each take `-L` and `-P` options to switch between logical and physical modes.
- Bash allows a function to override a builtin with the same name, and provides access to that builtin's functionality within the function via the `builtin` and `command` builtins (see Section 4.2 [Bash Builtins], page 50).
- The `command` builtin allows selective disabling of functions when command lookup is performed (see Section 4.2 [Bash Builtins], page 50).
- Individual builtins may be enabled or disabled using the `enable` builtin (see Section 4.2 [Bash Builtins], page 50).
- The Bash `exec` builtin takes additional options that allow users to control the contents of the environment passed to the executed command, and what the zeroth argument to the command is to be (see Section 4.1 [Bourne Shell Builtins], page 43).
- Shell functions may be exported to children via the environment using `export -f` (see Section 3.3 [Shell Functions], page 17).

- The Bash `export`, `readonly`, and `declare` builtins can take a `-f` option to act on shell functions, a `-p` option to display variables with various attributes set in a format that can be used as shell input, a `-n` option to remove various variable attributes, and '`name=value`' arguments to set variable attributes and values simultaneously.
- The Bash `hash` builtin allows a name to be associated with an arbitrary filename, even when that filename cannot be found by searching the `$PATH`, using '`hash -p`' (see Section 4.1 [Bourne Shell Builtins], page 43).
- Bash includes a `help` builtin for quick reference to shell facilities (see Section 4.2 [Bash Builtins], page 50).
- The `printf` builtin is available to display formatted output (see Section 4.2 [Bash Builtins], page 50).
- The Bash `read` builtin (see Section 4.2 [Bash Builtins], page 50) will read a line ending in '\' with the `-r` option, and will use the `REPLY` variable as a default if no non-option arguments are supplied. The Bash `read` builtin also accepts a prompt string with the `-p` option and will use Readline to obtain the line when given the `-e` option. The `read` builtin also has additional options to control input: the `-s` option will turn off echoing of input characters as they are read, the `-t` option will allow `read` to time out if input does not arrive within a specified number of seconds, the `-n` option will allow reading only a specified number of characters rather than a full line, and the `-d` option will read until a particular character rather than newline.
- The `return` builtin may be used to abort execution of scripts executed with the `.` or `source` builtins (see Section 4.1 [Bourne Shell Builtins], page 43).
- Bash includes the `shopt` builtin, for finer control of shell optional capabilities (see Section 4.3.2 [The Shopt Builtin], page 65), and allows these options to be set and unset at shell invocation (see Section 6.1 [Invoking Bash], page 85).
- Bash has much more optional behavior controllable with the `set` builtin (see Section 4.3.1 [The Set Builtin], page 61).
- The '`-x`' (`xtrace`) option displays commands other than simple commands when performing an execution trace (see Section 4.3.1 [The Set Builtin], page 61).
- The `test` builtin (see Section 4.1 [Bourne Shell Builtins], page 43) is slightly different, as it implements the POSIX algorithm, which specifies the behavior based on the number of arguments.
- Bash includes the `caller` builtin, which displays the context of any active subroutine call (a shell function or a script executed with the `.` or `source` builtins). This supports the bash debugger.
- The `trap` builtin (see Section 4.1 [Bourne Shell Builtins], page 43) allows a `DEBUG` pseudo-signal specification, similar to `EXIT`. Commands specified with a `DEBUG` trap are executed before every simple command, `for` command, `case` command, `select` command, every arithmetic `for` command, and before the first command executes in a shell function. The `DEBUG` trap is not inherited by shell functions unless the function has been given the `trace` attribute or the `functrace` option has been enabled using the `shopt` builtin. The `extdebug` shell option has additional effects on the `DEBUG` trap.

  The `trap` builtin (see Section 4.1 [Bourne Shell Builtins], page 43) allows an `ERR` pseudo-signal specification, similar to `EXIT` and `DEBUG`. Commands specified with an `ERR` trap are executed after a simple command fails, with a few exceptions. The `ERR` trap is not inherited by shell functions unless the `-o errtrace` option to the `set` builtin is enabled.

The `trap` builtin (see Section 4.1 [Bourne Shell Builtins], page 43) allows a `RETURN` pseudo-signal specification, similar to `EXIT` and `DEBUG`. Commands specified with an `RETURN` trap are executed before execution resumes after a shell function or a shell script executed with `.` or `source` returns. The `RETURN` trap is not inherited by shell functions unless the function has been given the `trace` attribute or the `functrace` option has been enabled using the `shopt` builtin.

- The Bash `type` builtin is more extensive and gives more information about the names it finds (see Section 4.2 [Bash Builtins], page 50).
- The Bash `umask` builtin permits a `-p` option to cause the output to be displayed in the form of a `umask` command that may be reused as input (see Section 4.1 [Bourne Shell Builtins], page 43).
- Bash implements a `csh`-like directory stack, and provides the `pushd`, `popd`, and `dirs` builtins to manipulate it (see Section 6.8 [The Directory Stack], page 96). Bash also makes the directory stack visible as the value of the `DIRSTACK` shell variable.
- Bash interprets special backslash-escaped characters in the prompt strings when interactive (see Section 6.9 [Controlling the Prompt], page 97).
- The Bash restricted mode is more useful (see Section 6.10 [The Restricted Shell], page 99); the SVR4.2 shell restricted mode is too limited.
- The `disown` builtin can remove a job from the internal shell job table (see Section 7.2 [Job Control Builtins], page 105) or suppress the sending of `SIGHUP` to a job when the shell exits as the result of a `SIGHUP`.
- Bash includes a number of features to support a separate debugger for shell scripts.
- The SVR4.2 shell has two privilege-related builtins (`mldmode` and `priv`) not present in Bash.
- Bash does not have the `stop` or `newgrp` builtins.
- Bash does not use the `SHACCT` variable or perform shell accounting.
- The SVR4.2 `sh` uses a `TIMEOUT` variable like Bash uses `TMOUT`.

More features unique to Bash may be found in Chapter 6 [Bash Features], page 85.

B.1 Implementation Differences From The SVR4.2 Shell

Since Bash is a completely new implementation, it does not suffer from many of the limitations of the SVR4.2 shell. For instance:

- Bash does not fork a subshell when redirecting into or out of a shell control structure such as an `if` or `while` statement.
- Bash does not allow unbalanced quotes. The SVR4.2 shell will silently insert a needed closing quote at `EOF` under certain circumstances. This can be the cause of some hard-to-find errors.
- The SVR4.2 shell uses a baroque memory management scheme based on trapping `SIGSEGV`. If the shell is started from a process with `SIGSEGV` blocked (e.g., by using the `system()` C library function call), it misbehaves badly.

- In a questionable attempt at security, the SVR4.2 shell, when invoked without the `-p` option, will alter its real and effective UID and GID if they are less than some magic threshold value, commonly 100. This can lead to unexpected results.
- The SVR4.2 shell does not allow users to trap `SIGSEGV`, `SIGALRM`, or `SIGCHLD`.
- The SVR4.2 shell does not allow the `IFS`, `MAILCHECK`, `PATH`, `PS1`, or `PS2` variables to be unset.
- The SVR4.2 shell treats '`^`' as the undocumented equivalent of '`|`'.
- Bash allows multiple option arguments when it is invoked (`-x -v`); the SVR4.2 shell allows only one option argument (`-xv`). In fact, some versions of the shell dump core if the second argument begins with a '`-`'.
- The SVR4.2 shell exits a script if any builtin fails; Bash exits a script only if one of the POSIX special builtins fails, and only for certain failures, as enumerated in the POSIX standard.
- The SVR4.2 shell behaves differently when invoked as `jsh` (it turns on job control).

Appendix C GNU Free Documentation License

Version 1.3, 3 November 2008

0. PREAMBLE

   The purpose of this License is to make a manual, textbook, or other functional and useful document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

   This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

   We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

   This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

   A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

   A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

   The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

   You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

   The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

   In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

   You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

   You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

   A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

   If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

   Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

   If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

   You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

   However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

   Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

   Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

    The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See `http://www.gnu.org/copyleft/`.

    Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

    "Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

    "CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

    "Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

    An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

    The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C)  year  your name.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.3
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover
Texts.  A copy of the license is included in the section entitled ``GNU
Free Documentation License''.
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with. . . Texts." line with this:

```
with the Invariant Sections being list their titles, with
the Front-Cover Texts being list, and with the Back-Cover Texts
being list.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Appendix D Indexes

D.1 Index of Shell Builtin Commands

D.2 Index of Shell Reserved Words

D.3 Parameter and Variable Index

D

E

F

G

H

I

K

L

M

O

P

R

S

T

U

V

D.4 Function Index

A

B

C

D

E

F

G

H

I

K

M

N

O

P

Q

R

S

T

U

Y

D.5 Concept Index

A

B

C

D

E

F

H

I

J

K

L

M

N

O

P

Q

R

S

T

V

W

Y