Advanced Bash-Scripting Guide

An in-depth exploration of the art of shell scripting

Mendel Cooper

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

Mendel Cooper

10

Publication date 10 Mar 2014

Abstract

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

List of Tables

List of Examples

Part Part 1. Introduction

Script: *A writing; a written document. [Obs.]*

--*Webster's Dictionary*, 1913 ed.

The shell is a command interpreter. More than just the insulating layer between the operating system kernel and the user, it's also a fairly powerful programming language. A shell program, called a *script*, is an easy-to-use tool for building applications by “gluing together” system calls, tools, utilities, and compiled binaries. Virtually the entire repertoire of UNIX commands, utilities, and tools is available for invocation by a shell script. If that were not enough, internal shell commands, such as testing and loop constructs, lend additional power and flexibility to scripts. Shell scripts are especially well suited for administrative system tasks and other routine repetitive tasks not requiring the bells and whistles of a full-blown tightly structured programming language.

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 [http://www.raspberrypi.org/]. 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 aesthetically 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. . . .

[1]These are referred to as builtins, features internal to the shell.

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 [http://www.faqs.org/faqs/unix-faq/shell/csh-whynot/] 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

[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.

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 [http://bash.deta.in/abs-guide-latest.tar.bz2] not be available, then cut-and-paste from the HTML [http://www.tldp.org/LDP/abs/abs-guide.html.tar.gz] or pdf [http://bash.deta.in/abs-guide.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 [mailto:thegrendel.abs@gmail.com] of this book wrote the example scripts that follow.

His countenance was bold and bashed not.

--Edmund Spenser

[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.

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* (#!) [1] 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 [2] *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. [3]

[1]More commonly seen in the literature as *she-bang* or *sh-bang*. This derives from the concatenation of the tokens *sharp* (#) and *bang* (!).

[2]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 [http://www.in-ulm.de/~mascheck/various/shebang/#details] this is probably a myth.

[3]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/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. [4] 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 [5] **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. [6]

#! 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. [7] Note again that **`#!/bin/sh`** invokes the default shell interpreter, which defaults to `/bin/bash` on a Linux machine.

Tip

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.

```
#!/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.
```
[4]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, "Multi-line message using *cat*").

[5]**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 [http://www.opengroup.org/onlinepubs/007904975/toc.htm].

[6]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`

[7]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 #!.

```
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.

Invoking the script

Having written the script, you can invoke it by **`sh scriptname`**, [8] 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) [9]

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`**. [10] 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.

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.

[8] Caution: invoking a *Bash* script by **`sh scriptname`** turns off Bash-specific extensions, and the script may therefore fail to execute.

[9] A script needs *read*, as well as execute permission for it to run, since the shell needs to be able to read it.

[10] 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`**.

Part 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

#

;

;;

;;&, ;&

.

.

.

"

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

'

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

,

comma operator. The *comma operator* [1] 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.

[1] 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.

\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, *Quoting* 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.

:

!

*

*

?

?

$

$

${}

$' ... '

$*, $@

$?

$$

()
: **command group.**

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

Important

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,...}

{a..z}

{} \; **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*.

Note

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

[ ]

[[ ]]

[ ]

[ ]

$[...]

(( ))

> &> >& >> < <>

<< **redirection used in a here document.**

<<< **redirection used in a here string.**

<, >

\<, \>

|

>|

||

& **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
```

```
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.
```

Caution

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.

&&

-

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.

Tip

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, “Reassigning the positional parameters”)

-

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

Caution

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.

^

^, ^^ **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 expec
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 En
           # Of course, '0d' is the hex equivalent of Control
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, linefe
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 th
#  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 ret
#  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

exit 0
```

- **Ctl-N**

  Erases a line of text recalled from *history buffer* [8] (on the command-line).

- **Ctl-O**

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

- **Ctl-P**

[8] 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.

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. [9] 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:].

[9] A linefeed (*newline*) is also a whitespace character. This explains why a *blank line*, consisting only of a linefeed, is considered whitespace.

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.

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*.

$

Variable Assignment

=

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.

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

Note

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.

Caution

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&=ZZZZZ

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.

Note

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 [2] : `$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. [3] 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 examp
# ./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

# ...
```

[2] Note that *functions* also take positional parameters.
[3] 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
```

```
if [ -n "${10}" ]  # Parameters > $9 must be enclosed in 
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-lin
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 Mons
# 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. [4] There must also exist symbolic links to all the alternate names of the script. See Example 16.2, “Hello or Good-bye”.

Tip

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 paramete
```

[4]If the the script is sourced or symlinked, then this will not work. It is safer to check $BASH_Source.

```
critical_argument01=$variable1_

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

#  A more straightforward way of dealing with this is
#+ to simply test whether expected positional parameters h
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
#                     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 positiona

#  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 int
#+ 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 p
```

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
#                    ^^^^^^^^
```

Note

The **shift** command works in a similar fashion on parameters passed to a function. See Example 36.18, “Return value trickery”.

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. [1]

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


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
```

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). [2] Keeping $ as a special character within double quotes permits referencing a quoted variable (*`"$vari-`*

[1]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.
[2]

`able"`), that is, replacing the variable with its value (see Example 4.1, “Variable assignment and substitution”, above).

Use double quotes to prevent word splitting. [3] 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 #     ^     ^
```

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.)

[3] “Word splitting,” in this context, means dividing a character string into separate and discrete arguments.

```
  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.
```

Tip

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”).

Note

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.
```

Escaping

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

Caution

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

\r

\t

\v

\b

\a

\0xx

\"

\$

\\

Note

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.
```

```
   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.)
```

Note

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).

Note

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." [1]

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, "Using *cmp* to compare two files within a script." and Example 16.20, "Checking words in a list for validity").

Note

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.
```

[1] In those instances when there is no return terminating the function.

```
#    !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.
```

Caution

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.

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.
```

Caution

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, “What is truth?”, 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
```

Note

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* [1] 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.

[1]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.

Note

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!
```

Note

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
```

File test operators

Returns true if...

-e	file exists
-a	file exists
	This is identical in effect to -e. It has been “deprecated,” [2] and its use is discouraged.
-f	file is a *regular* file (not a directory or device file)
-s	file is not zero size

[2] 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.
```

-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. [3] 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. [4] 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
```

[3]Be aware that *suid* binaries may open security holes. The *suid* flag has no effect on shell scripts.
[4]On Linux systems, the sticky bit is no longer used for files, only on directories.

```
# 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, “Hiding the cookie jar”, Example 11.8, “A *grep* replacement for binary files”, Example 11.3, “*Fileinfo:* operating on a file list contained in a variable”, Example 31.3, “Creating a ramdisk”, and Example A.1, “*mailformat*: Formatting an e-mail message” also illustrate uses of the file test operators.

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" ]`

Caution

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 =.

Note

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, “The Bubble Sort” 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.*

Caution

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, “Testing whether a string is *null*”) normally works, however, this is an unsafe practice. *Always* quote a tested string. [5]

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.
```

[5]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).

```
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
```

Caution

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, “Compound Condition Tests Using && and ||”, Example 27.17, “Simulating a two-dimensional array, then tilting it”, and Example A.29, “Spammer Hunt” to see compound comparison operators in action.

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, “Using arrays and other miscellaneous trickery to deal four random hands from a deck of cards” and Example 17.11, “*Backlight*: changes the brightness of the (laptop) screen backlight” demonstrate nested *`if/then`* condition tests.

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

Operators

assignment

variable assignment Initializing or changing the value of a variable

=

arithmetic operators

+

-

*

/

**

%

+=

-=

*=

/=

%=

Note

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.

```
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 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, “Base64 encoding/decoding”) script.

bitwise operators

<<

<<=

>>

>>=

&

&=

|

|=

~

^

^=

logical (boolean) operators

!

&&

||

miscellaneous operators

,

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).

```
# 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.
```

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, “A C-style *for* loop” and Example 8.4, “Representation of numerical constants”.

Operator Precedence

In a script, operations execute in order of *precedence*: the higher precedence operations execute *before* the lower precedence ones. [3]

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

[3] *Precedence*, in this context, has approximately the same meaning as *priority*

Operator	**Meaning**	**Comments**
`-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.
```

Tip

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 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.

Internal Variables

Builtin variables: variables affecting bash script behavior

$BASH

$BASH_ENV

$BASH_SUBSHELL

$BASHPID

$BASH_VERSINFO[n]

$BASH_VERSION

$CDPATH

$DIRSTACK

$EDITOR

$EUID

$FUNCNAME

$GLOBIGNORE

$GROUPS

$HOME

$HOSTNAME

$HOSTTYPE

$IFS

$IGNOREEOF

$LC_COLLATE

$LC_CTYPE

$LINENO

$MACHTYPE

$OLDPWD

$OSTYPE

$PATH

$PIPESTATUS

Other Special Parameters

`$-`

`$!`

`$_`

`$?`

`$$`

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* [6] 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

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
```

[6] 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
```

-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.
```

Caution

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.
```

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")
```

$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* [7] 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
```

[7]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*.

```
#  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 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. [8] (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.
```

[8]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.

```
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
```

Note

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, “Generating 10-digit random numbers”, and Example A.36, “Insertion sort”), or even piping to md5sum (see Example 36.16, “A “horserace” game”).

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

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

expr "$string" : '.*'

These are the equivalent of *strlen()* in *C*.

```
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 positio

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 `$string` parameter is “*” or “@”, then this extracts the positional parameters,[1] starting at `$position`.

${string:position:length}

Extracts *`$length`* characters of substring from *`$string`* at *`$position`*.

```
stringZ=abcABC123ABCabc
#       0123456789.....
#       0-based indexing.

echo ${stringZ:0}                            # abcABC123
```

[1]This applies to either command-line arguments or parameters passed to a function.

```
echo ${stringZ:1}                            # bcABC123AB
echo ${stringZ:7}                            # 23ABCabc

echo ${stringZ:7:3}                          # 23A
                                             # Three char



# Is it possible to index from the right end of the strin

echo ${stringZ:-4}                           # abcABC123A
# 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 parame

# 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-strin
  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 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 positi
echo ${@:2}          # Same as above.

echo ${*:2:3}        # Echoes three positional parameters
```

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" '\(.[b-c]*[A-Z]..[0-9]\)'`
echo `expr "$stringZ" : '\(.[b-c]*[A-Z]..[0-9]\)'`
echo `expr "$stringZ" : '\(.......\)'`
# 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" '.*\([A-C][A-C][A-C][a-c]*\)
```

```
   echo `expr "$stringZ" : '.*\(......\)'`
```

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" s
# 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
  #+ starting from the right-hand-side of the variable $i .
done ### This could be condensed into a "one-liner" if desire

# 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

echo ${stringZ%%b*c}     # a
# Strip out longest match between 'b' and 'c', from back of S
```

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 "p

#  Uses the "macptopbm" binary from the "netpbm" package,
#+ which is maintained by Brian Henderson (bryanh@giraffe-dat
#  Netpbm is a standard part of most Linux distros.

OPERATION=macptopbm
SUFFIX=pbm          # New filename suffix.

if [ -n "$1" ]
then
  directory=$1      # If directory name given as a script arg
else
  directory=$PWD    # Otherwise use current working directory
fi

#  Assumes all files in the target directory are MacPaint ima
#+ with a ".mac" filename suffix.

for file in $directory/*    # Filename globbing.
do
  filename=${file%.*c}      #  Strip ".mac" suffix off filena
                            #+ ('.*c' matches everything
       #+ between '.' and 'c', inclusive).
  $OPERATION $file > "$filename.$SUFFIX"
                            # Redirect conversion to new file
  rm -f $file               # Delete original files after con
  echo "$filename.$SUFFIX"  # Log what is happening to stdout
done

exit 0

# Exercise:
# --------
#  As it stands, this script converts *all* the files in the
#+ 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 distr

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
# 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 add
##################################################################

rm "$OUTFILE"      # Delete intermediate *.wav file.
                   # If you want to keep it, comment out abov
```

```
exit $?

#  Note:
#  ----
#  On a Website, simply clicking on a *.ram streaming audio f
#+ usually only downloads the URL of the actual *.ra audio fi
#  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 filen
#
#  If you're really ambitious, expand the script
#+ to do automatic downloads and conversions of streaming aud
#  Given a URL, batch download streaming audio files (using '
#+ 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

---

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`*.[2]

${string//substring/replacement}

Replace all matches of *`$substring`* with *`$replacement`*.

```
stringZ=abcABC123ABCabc

echo ${stringZ/abc/xyz}       # xyzABC123ABCabc
                              # Replaces first match of

echo ${stringZ//abc/xyz}      # xyzABC123ABCxyz
                              # Replaces all matches of

echo  ---------------
echo "$stringZ"               # abcABC123ABCabc
echo  ---------------
                              # The string itself is not

# 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.
```

[2]Note that *`$substring`* and *`$replacement`* may refer to either *literal strings* or *variables*, depending on context. See the first usage example.

${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 ma

echo ${stringZ/%abc/XYZ}          # abcABC123ABCXYZ
                                  # Replaces back-end mat
```

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
```

Further Reference

For more on string manipulation in scripts, refer to the section called “Parameter Substitution” and the relevant section of the expr command listing.

Script examples:

1. Example 16.9, “Using *expr*”
2. Example 10.9, “Length of a variable”
3. Example 10.10, “Pattern matching in parameter substitution”
4. Example 10.11, “Renaming file extensions:”
5. Example 10.13, “Matching patterns at prefix or suffix of string”
6. Example A.36, “Insertion sort”
7. Example A.41, “Quacky: a Perquackey-type word game”

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 durati
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
```

Note

${parameter-default}* and *${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 conditio
# 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 bloc
#+ 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, “Backup of all files changed in last day”, Example 31.2, “Setting up a swapfile using `/dev/zero`”, and Example A.6, “Collatz series”.

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, [3] 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
```

[3]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”).

```
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, “Using *expr*”). 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.

Note

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 a
  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, "*days-between*: Days between two dates":

```
# 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" -- s
```

Manfred Schwarb's more elaborate variation of the above:

```
strip_leading_zero2 () # Strip possible leading zero(s), since ot
{                      # Bash will interpret such numbers as octa
  shopt -s extglob     # Turn on extended globbing.
  local val=${1##+(0)} # Use local variable, longest matching ser
  shopt -u extglob     # Turn off extended globbing.
  _strip_leading_zero2=${val:-0}
                       # If input was 0, return 0 instead of "".
}
```

Another usage illustration:

```
echo `basename $PWD`        # Basename of current working directo
echo "${PWD##*/}"           # Basename of current working directo
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, "*password*: Generating random 8-character passwords" 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=${var1#*-*}
echo "var1 (with everything, up to and including first -
#  t=${var1#*-}  works just the same,
#+ since # matches the shortest string,
#+ and * matches everything preceding, including an empty
# (Thanks, Stephane Chazelas, for pointing this out.)

t=${var1##*-*}
echo "If var1 contains a \"-\", returns empty string...
```

```
t=${var1%*-*}
echo "var1 (with everything from the last - on stripped o

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 $path_name` in this partic
#  t=${path_name%/}; t=${t##*/}   is a more general solut
#+ but still fails sometimes.
#  If $path_name ends with a newline, then `basename $pat
#+ 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////
#  Removing suffixes, especially when the basename has no
#+ 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, lengt

echo

t=${path_name/bozo/clown}
echo "$path_name with \"bozo\" replaced  by \"clown\" = $
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 strin

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 
# Bash, version 2.04, adds this feature.

xyz23=whatever
xyz24=

a=${!xyz*}         #  Expands to *names* of declared vari
# ^ ^   ^           + 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 ind
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.

Loops

A *loop* is a block of code that *iterates* [1] a list of commands as long as the *loop control condition* is true.

for loops

for *arg* in *[list]*

while

until

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.

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.

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

[1] *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.

```
  # 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, “The Bubble Sort” for an illustration of nested while loops, and Example 27.13, “The Sieve of Eratosthenes” to see a while loop nested inside an until loop.

Loop Control

Tournez cent tours, tournez mille tours,

Tournez souvent et tournez toujours . . .

--Verlaine, “Chevaux de bois”

Commands affecting loop behavior

break,
continue

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

select

Chapter 12. Command Substitution

Command substitution reassigns the output of a command [1] or even multiple commands; it literally plugs the command output into another context. [2]

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.
```

Note

Command substitution invokes a subshell.

Caution

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
```

[1] For purposes of *command substitution*, a **command** may be an external system command, an internal scripting builtin, or even a script function.
[2] In a more technically correct sense, *command substitution* extracts the `stdout` of a command, then assigns it to a variable using the = operator.

```
# 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.
```

Caution

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 vers

#  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=0
    kbdoutput=`cat /proc/bus/usb/devices 2>/dev/null|grep -E "^I.*Cls=03.*Prot=01"
...
fi
```

Caution

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.
```

Note

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. [3]

```
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.
```

[3] 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}'\` `
```

```
# See also the agram.sh script for an exercise in anagram finding.

exit $?
```

Examples of command substitution in shell scripts:

1. Example 11.8, “A *grep* replacement for binary files”
2. Example 11.27, “Using *command substitution* to generate the *case* variable”
3. Example 9.16, “Reseeding RANDOM”
4. Example 16.3, “*Badname*, eliminate file names in current directory containing bad characters and white-space.”
5. Example 16.22, “*lowercase*: Changes all filenames in working directory to lowercase.”
6. Example 16.17, “Emulating *grep* in a script”
7. Example 16.54, “Using *seq* to generate loop arguments”
8. Example 11.14, “Using *efax* in batch mode”
9. Example 11.11, “Listing the *symbolic links* in a directory”
10. Example 16.32, “Stripping comments from C program files”
11. Example 20.8, “Redirected *for* loop”
12. Example A.16, “*tree*: Displaying a directory tree”
13. Example 29.3, “Finding the process associated with a PID”
14. Example 16.47, “Monthly Payment on a Mortgage”
15. Example 16.48, “Base Conversion”
16. Example 16.49, “Invoking *bc* using a *here document*”

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)

Arithmetic expansion with double parentheses, and using let

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 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
- lastlog

- ldd
- less
- let
- lex
- lid
- ln
- locate
- lockfile
- logger
- logname
- logout
- logrotate
- look
- losetup
- lp
- ls
- lsdev
- lsmod
- lsof
- lspci
- lsusb
- ltrace
- lynx
- lzcat
- lzma
- m4
- mail
- mailstats
- mailto
- make

- MAKEDEV
- man
- mapfile
- mcookie
- md5sum
- merge
- mesg
- mimencode
- mkbootdisk
- mkdir
- mkdosfs
- mke2fs
- mkfifo
- mkisofs
- mknod
- mkswap
- mktemp
- mmencode
- modinfo
- modprobe
- 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
- rlogin
- rm
- rmdir
- rmmod
- route
- rpm
- rpm2cpio
- rsh
- rsync
- runlevel
- run-parts
- rx
- rz
- sar
- scp
- script
- sdiff
- sed
- seq

- service
- set
- setfacl
- setquota
- setserial
- setterm
- sha1sum
- shar
- shopt
- shred
- shutdown
- size
- skill
- sleep
- slocate
- snice
- sort
- source
- sox
- split
- sq
- ssh
- stat
- strace
- strings
- strip
- stty
- su
- sudo
- sum

- suspend
- swapoff
- swapon
- sx
- sync
- sz
- tac
- tail
- tar
- tbl
- tcpdump
- tee
- telinit
- telnet
- Tex
- texexec
- time
- times
- tmpwatch
- top
- touch
- tput
- tr
- traceroute
- true
- tset
- tsort
- tty
- tune2fs
- type

- typeset
- ulimit
- umask
- umount
- uname
- unarc
- unarj
- uncompress
- unexpand
- uniq
- units
- unlzma
- unrar
- unset
- unsq
- unzip
- uptime
- usbmodules
- useradd
- userdel
- usermod
- users
- usleep
- 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* [1] a separate process -- or because a particular builtin needs direct access to the shell internals.

[1]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."

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*. [2]

I/O

echo

printf

read

Filesystem

cd

pwd

pushd,
popd,
dirs

Variables

let

eval

set

unset

export

declare,
typeset

readonly

getopts

Script Behavior

source, . (dot command)

exit

exec

shopt

caller

[2]An exception to this is the time command, listed in the official Bash documentation as a keyword ("reserved word").

Commands

true

times

kill

killall

command

builtin

enable

autoload

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.

Basic Commands

The first commands a novice learns

ls

cat, **tac**

rev

cp

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, "Renaming file extensions:" and Example A.2, "*rn*: A simple-minded file renaming utility".

Note

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

rmdir

mkdir

chmod

chattr

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, "*wh, whois* domain name lookup").

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.

Caution

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**

Complex Commands

Commands for more advanced users

find

xargs

expr

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 the section called “Shell Wrappers”) is an attractive alternative to **expr**.

See the section called “Manipulating Strings” for more on using **expr** in string operations.

Time / Date Commands

Time/date and timing

date

zdump

time

touch

at

batch

cal

sleep

usleep

hwclock,
clock

Text Processing Commands

Commands affecting text and text files

sort

tsort

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

tr

fold

fmt

col

column

colrm

nl

pr

gettext

msgfmt

iconv

recode

TeX, **gs**

texexec

enscript

groff, **tbl**, **eqn**

lex, **yacc**

File and Archiving Commands

Archiving

tar

shar

ar

rpm

cpio

rpm2cpio

pax

Compression

gzip

diff3,
merge

sdiff

cmp

comm

Utilities

basename

dirname

split, **csplit**

Encoding and Encryption

sum, **cksum**,
md5sum,
sha1sum

uuencode

uudecode

mimencode,
mmencode

crypt

openssl

shred

Miscellaneous

mktemp

make

install

dos2unix

ptx

more, **less**

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

rsync

ssh

scp

Local Network

write

netconfig

Mail

mail

mailto

mailstats

vacation

Terminal Control Commands

Command affecting the console or terminal

tput

infocmp

reset

clear

resize

script

Math Commands

"Doing the numbers"

factor

bc

dc

awk

Miscellaneous Commands

Command that fit in no special category

jot, **seq**

xmessage

zenity

doexec

dialog

sox

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

groups

chown,
chgrp

useradd,
userdel

usermod

groupmod

id

lid

who

w

logname

su

sudo

passwd

ac

last

newgrp

Terminals

tty

stty

setterm

tset

setserial

getty, **agetty**

mesg

hostid

sar

readelf

size

System Logs

logger

logrotate

Job Control

ps

pgrep,
pkill

pstree

top

nice

nohup

pidof

fuser

cron

Process Control and Booting

init

telinit

runlevel

halt, **shut-
down**, **reboot**

service

Network

nmap

ifconfig

netstat

iwconfig

ip

mkbootdisk

mkisofs

chroot

lockfile

flock

mknod

MAKEDEV

tmpwatch

Backup

dump, **restore**

fdformat

System Resources

ulimit

quota

setquota

umask

rdev

Modules

lsmod

insmod

rmmod

modprobe

depmod

modinfo

Miscellaneous

env

ldd

watch

strip

nm

[10] The *killall* system script should not be confused with the killall command in `/usr/bin`.

```
# --> 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 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: =~.

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 [1] 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.

-
-
-
-
-
-
-

[1]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.

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.
-
-
-
-
-

Note

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.

-
-
-
-
-
-

-
-

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.

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* [3] -- 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 [4] -- `*` 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`. [5] 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
```

[3] *Filename expansion* means expanding filename patterns or templates containing special characters. For example, `example.???` might expand to `example.001` and/or `example.txt`.
[4] A *wild card* character, analogous to a wild card in poker, can represent (almost) any other character.
[5] 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.
```

```
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
```

Note

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, "Operating on files with a *for* loop".

Caution

Filenames with embedded whitespace can cause *globbing* to choke. David Wheeler [http://www.dwheeler.com/essays/filenames-in-shell.html] 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 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 se
TESTVARIABLES

exit $?
```

Tip

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 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.
```

Tip

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, “Spammer Identification”, Example A.40, “Petals Around the Rose”, Example A.41, “Quacky: a Perquackey-type word game”, and Example A.42, “Nim” for more examples of self-documenting scripts.

Note

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)
```

Caution

Some utilities will not work inside a *here document*.

Warning

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. [1]

```
#!/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.
```

Caution

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 !!!
```

[1]Except, as Dennis Benzinger points out, if using <<- to suppress tabs.

```
   !


   # 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.

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* [1] 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, "Code blocks and I/O redirection" and Example 3.2, "Saving the output of a code block to a file") and sending it as input to another file, command, program, or script.

Each open file gets assigned a file descriptor. [2] 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. [3] This simplifies restoration to normal after complex redirection and reshuffling (see Example 20.1, "Redirecting `stdin` using *exec*").

```
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
```

[1]By convention in UNIX and Linux, data streams and peripherals (device files) are treated as files, in a fashion analogous to ordinary files.
[2]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**.
[3]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 [http://groups.google.com/group/gnu.bash.bug/browse_thread/thread/13955daafded3b5c/18c17050087f9f37]). Best leave this particular fd alone.

```
   # 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, “Unpacking an *rpm* archive” and Example A.14, “*fifo*: Making daily backups, using named pipes”.

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 scr

# Thanks, S.C.
```

For a more detailed introduction to I/O redirection see Appendix F, *A Detailed Introduction to I/O and I/O Redirection*.

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.

Important

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.
```

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, “Real name from username”). 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, “Saving the output of a code block to a file”.

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"
```

Applications

Clever use of I/O redirection permits parsing and stitching together snippets of command output (see Example 15.7, “Using *read* with file redirection”). 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, [1] 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"
```

[1]An external command invoked with an exec does *not* (usually) fork off a subprocess / subshell.

```
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, “Subshell Pitfalls”.

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.

Note

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 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)`**.

Note

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. [1]

Caution

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
```

Note

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)
```

[1]This has the same effect as a named pipe (temp file), and, in fact, named pipes were at one time used in process substitution.

```
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`...
}

Note

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.
```

Note

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, “The Gronsfeld Cipher”

Note

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.
```

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
```

Important

The shift command works on arguments passed to functions (see Example 36.18, “Return value trickery”).

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, “Indirect variable references - the new way”) 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

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, “Code blocks and I/O redirection”).

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 [
  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
# 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.
```

Note

Emmanuel Rouat's sample `bashrc` file contains some instructive examples of functions.

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. [2]

[2]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
```

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 funct

echo
echo "\"loc_var\" outside function = $loc_var"
                                      # $loc_var outside function
                                      # No, $loc_var not visible g
echo "\"global_var\" outside function = $global_var"
                                      # $global_var outside functi
                                      # $global_var is visible glo
echo

exit 0
```

```
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.*

```
#  In contrast to C, a Bash variable declared inside a function
#+ is local ONLY if declared as such.
```

Caution

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
                                 #+ so $global_var is not visib

func
echo "global_var = $global_var"  # global_var = 37
                                 # Has been set by function cal
```

Note

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
```

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,[3] an expression implicit in its own expression,[4] *a snake swallowing its own tail*,[5] or . . . a function that calls itself.[6]

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 $?
```

[3]Otherwise known as *redundancy*.
[4]Otherwise known as *tautology*.
[5]Otherwise known as a *metaphor*.
[6]Otherwise known as a *recursive function*.

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. [7]

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)."
```

[7] 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.
```

```
  #  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, "Generating prime numbers using the modulo operator" 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.

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`** [1] 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. [2] 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.
```

[1] ... as the first word of a command string. Obviously, an alias is only meaningful at the *beginning* of a command.
[2] However, aliases do seem to expand positional parameters.

```
# 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 [http://bash.deta.in/Tetris_Game.sh]:

```
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 $?
```

Caution

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

  # ==> . . .
```

Important

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, “*days-between*: Days between two dates” and Example 7.4, “Testing for broken links” 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.
```

Tip

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
```

Note

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, “Generating prime numbers using the modulo operator”, and Example 16.46, “Generating prime numbers”.

--

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, “*Game of Life*”.

--

For more interesting scripts using arrays, see:

- Example 12.3, “Finding anagrams”
- Example 16.46, “Generating prime numbers”
- Example A.22, “More on hash functions”
- Example A.44, “An all-purpose shell scripting homework assignment solution”
- Example A.41, “Quacky: a Perquackey-type word game”
- Example A.42, “Nim”

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 $?
```

Caution

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, “Indirect variable references - the new way” and Example A.22, “More on hash functions”) 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.

/dev

The `/dev` directory contains entries for the *physical devices* that may or may not be present in the hardware. [1] 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. [2] This permits mounting an entire filesystem within a single large file. See Example 17.8, “Creating a filesystem in a file” and Example 17.7, “Checking a CD image”.

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`. [3]

```
/dev/sda1    /mnt/flashdrive    auto    noauto,user,noatime    0 0
```

(See also Example A.23, “Mounting USB keychain storage devices”.)

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.)
```

[1]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.
[2]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.
[3]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.

```
#  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!
```

/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

    3     0    3007872 hda 4472 22260 114520 94240 3551 18703 50384 549710 0 11155
    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`. [4]

```
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!)
```

[4]Certain system commands, such as procinfo, free, vmstat, lsdev, and uptime do this as well.

```
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
```

Note

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 st
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.
```

Warning

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, “Analyzing a spam domain”
6. Example A.28, “Spammer Identification”
7. Example A.29, “Spammer Hunt”
8. Example 29.1, “Using `/dev/tcp` for troubleshooting”

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, "*Badname*, eliminate file names in current directory containing bad characters and white-space.").

```
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 mes
# If "$filename" does exist, the contents of the file will
# Therefore, no output at all will result from the above li
#
#  This can be useful in situations where the return code f
#+ 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, "*cleanup*: A script to clean up log files in /var/log " and Example 2.3, "*cleanup*: An enhanced and generalized version of above scripts."):

```
cat /dev/null > /var/log/messages
#  : > /var/log/messages   has same effect, but does not sp

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 sa
```

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 b
                                 #+ if nothing specified o
# 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 bloc
fi


####################################################################
echo "Creating swap file of size $blocks blocks (KB)."
dd if=/dev/zero of=$FILE bs=$BLOCKSIZE count=$blocks  # Zer
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 e
#+ 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, “Creating a filesystem in a file”) or “securely” deleting a file (see Example 16.61, “Securely deleting a file”).

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 pow
#+                less RAM available to system.
#
#  Of what use is a ramdisk?
#  Keeping a large dataset, such as a table or dictionary o
#+ speeds up data lookup, since memory access is much faste
```

```
E_NON_ROOT_USER=70             # Must run as root.
ROOTUSER_NAME=root

MOUNTPT=/mnt/ramdisk           # Create with mkdir /mnt/ram
SIZE=2000                      # 2K blocks (change as appro
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
then                           #+ so no error if this scrip
  mkdir $MOUNTPT               #+ multiple times.
fi

##############################################################
dd if=/dev/zero of=$DEVICE count=$SIZE bs=$BLOCKSIZE  # Zer
                                                      # Why
mke2fs $DEVICE                 # Create an ext2 filesystem
mount $DEVICE $MOUNTPT         # Mount it.
chmod 777 $MOUNTPT             # Enables ordinary user to a
                               # However, must be root to u
##############################################################
# Need to test whether above commands succeed. Could cause
# Exercise: modify this script to make it safer.

echo "\"$MOUNTPT\" now available for use."
# The ramdisk is now accessible for storing files, even by

#  Caution, the ramdisk is volatile, and its contents will
#+ 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

#  Suitably modified, this script can by invoked in /etc/rc
#+ to set up ramdisk automatically at bootup.
#  That may be appropriate on, for example, a database serv

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, "*test24*: another buggy script" 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.

   ### Tip

   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. [1] 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.
```

[1] By convention, `signal 0` is assigned to exit.

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 $?
```

Note

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, “A command-line stopwatch”).

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
```

Note

`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.
```

Note

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

Abbreviation	**Name**	**Effect**
`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, *Restricted Shells*).
`-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, "Testing whether a string is *null*", Example 20.5, "Redirected *while* loop", and Example 9.6, "*arglist*: Listing arguments with $* and $@".

- 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, “Testing large return values in a function”.

- 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, “Reassigning the positional parameters”).

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. [1]

- 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.

[1]Setting the suid permission on the script itself has no effect in Linux and most other UNIX flavors.

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*.

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,” [1] 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
}
```

[1] In this context, “magic numbers” have an entirely different meaning than the magic numbers used to designate file types.

```
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, *Exit Codes With Special Meanings*.

  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 the section called “Standard Command-Line Options”.

- Break complex scripts into simpler modules. Use functions where appropriate. See Example 37.4, “Using arrays and other miscellaneous trickery to deal four random hands from a deck of cards”.

- 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

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, “Variable assignment, using *read*”). “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
```

Note

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.

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. [1] 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
```

[1]Quite a number of Linux utilities are, in fact, shell wrappers. Some examples are `/usr/bin/pdf2ps`, `/usr/bin/batch`, and `/usr/bin/xmkmf`.

```
#+ 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 [http://sourceforge.net/projects/undvd/], which provides an easy-to-use command-line interface to the complex mencoder [http://www.mplayerhq.hu/DOCS/HTML/en/mencoder.html] utility. Another example is Itzchak Rehberg's Ext3Undel [http://projects.izzysoft.de/trac/ext3undel], a set of scripts to recover deleted file on an *ext3* filesystem.

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.
```

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.
```

Caution

Too many levels of recursion can exhaust the script's stack space, causing a segfault.

"Colorizing" Scripts

The ANSI [2] 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.
```

[2]ANSI is, of course, the acronym for the American National Standards Institute. This august body establishes and maintains various technical and industrial standards.

```
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"
```

Note

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
```

Note

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.

Note

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.
        tpute 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, “Colorizing text using hash functions”, Example A.44, “An all-purpose shell scripting homework assignment solution”, Example A.52, “Cycling through all the possible color backgrounds”, and Example A.40, “Petals Around the Rose”.

Caution

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 [http://code.google.com/p/ansi-color/] utility (based on Moshe Jacobson's color utility [http://bash.deta.in/color-1.1.tar.gz] 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.

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?
```

Note

Certain operators, notably expr, are very inefficient and might be replaced by double parentheses arithmetic expansion. See Example A.59, “Testing execution times of various commands”.

```
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, “Testing execution times of various commands”.

```
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
```

Note

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, “Associative arrays vs. conventional arrays (execution times)”.

```
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, [3] 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, “Monthly Payment on a Mortgage”.

Assorted Tips

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.

-

[3]This usually means liberal use of functions.

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 ~/.basl
#+ (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, “Prepending a line to a file” 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, “*mailformat*: Formatting an e-mail message” and Example 15.20, “Using *export* to pass a variable to an embedded *awk* script”.

•

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
```

Caution

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, "*Game of Life*".

- 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, "A C-style *for* loop" and Example 11.18, "C-style syntax in a *while* loop".

- 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, “On-line connect status”, Example 16.25, “Generating “Crypto-Quote” Puzzles”, and Example A.9, “Soundex conversion”.

•

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, “Commenting out a block of code”.

•

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:

```
# $Id$
```

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 [http://www.batse.msfc.nasa.gov/~mallozzi/home/software/xforms/src/widtools-2.0.tgz].

Caution

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).

Security Issues

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. [4]

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.

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 [http://www.datsi.fi.upm.es/~frosal/sources/] does exactly that.

Unfortunately, according to an article [http://www.linuxjournal.com/article/8256] 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.

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`.

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*. [5] 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.

[4]See Marius van Oers' article, Unix Shell Scripting Malware [http://www.virusbtn.com/magazine/archives/200204/malshell.xml], and also the *Denning* reference in the *bibliography*.
[5]Or, better yet, #!/bin/env sh.

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 [http://linux.oreillynet.com/pub/a/linux/2002/02/28/caldera.html] 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. [ftp://ftp.cwru.edu/pub/bash/FAQ] for a complete listing.

A Test Suite

Let us illustrate some of the incompatibilities between Bash and the classic Bourne shell. Download and install the “Heirloom Bourne Shell” [http://freshmeat.net/projects/bournesh] 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 $?
```

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 [http://sourceware.cygnus.com/cygwin/] package from Cygnus and the

MKS utilities [http://www.mkssoftware.com/] from Mortice Kern Associates add shell scripting capabilities to Windows.

Another alternative is UWIN [http://www2.research.att.com/~gsf/download/uwin/uwin.html], 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

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.
```

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, “Spammer Hunt”, Example 19.14, “Parsing a mailbox”, Example A.35, “Locating split paragraphs in a text file”, and Example A.24, “Converting to HTML”.

•

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, “Updating FC4”.

Caution

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, “Timed *read*”, for instance).

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
```

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 [[...]].

Caution

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 [https://bugs.launchpad.net/ubuntu-website/+bug/109931] and Wikinerds on Bash syntax [http://en.wikinerds.org/index.php/Bash_syntax_and_semantics].

Setting *shopt -s compat31* in a script causes reversion to the original behavior.

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. [1]

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.
```

[1]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.

```
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, “Morse Code Practice” for an interesting usage of an *associative array*.

Caution

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 [2] , ver. 4.01:

> There is a new 'coproc' reserved word that specifies a coprocess:
> an asynchronous command run with two pipes connected to the creating
> shell. Coprocs can be named. The input and output file descriptors
> and the PID of the coprocess are available to the calling shell in
> 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
```

[2]Copyright 1995-2009 by Chester Ramey.

```
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.
```

Caution

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 [3] and the `-i` option permits preloading the edit buffer. [4] 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
```

[3]This only works with pipes and certain other *special* files.
[4]But only in conjunction with readline, i.e., from the command-line.

```
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. [5] 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.

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.

[5] And while you're at it, consider fixing the notorious piped read problem.

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')
```

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 `\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
```

Note

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, “Piping the output of *echo* to a *read*” and Example 15.8, “Problems reading from a pipe”.

- 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

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. [1] 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.

About the Author

Who is this guy anyhow?

The author claims no credentials or special qualifications, [2] other than a compulsion to write. [3]

This book is somewhat of a departure from his other major work, HOW-2 Meet Women: The Shy Man's Guide to Relationships [http://bash.deta.in/hmw60.zip]. He has also written the Software-Building HOWTO [http://tldp.org/HOWTO/Software-Building-HOWTO.html]. Of late, he has been trying his (heavy) hand at fiction: Dave Dawson Over Berlin (First Installment) [http://bash.deta.in/dave-dawson-over-berlin.epub] Dave Dawson Over Berlin (Second Installment) [http://bash.deta.in/dave-dawson-over-berlin.II.epub] and Dave Dawson Over Berlin (Third Installment) [http://bash.deta.in/dave-dawson-over-berlin.III.epub] . He also has a few *Instructables* (here [http://www.instructables.com/id/Arduino-Morse-Code-Shield/], here [http://www.instructables.com/id/Haywired-Hackduino/], here [http://www.instructables.com/id/Arduino-DIY-SD-Card-Logging-Shield/], here [http://www.instructables.com/id/Binguino-An-Arduino-based-Bingo-Number-Generato/], here [http://www.instructables.com/id/The-Raspberry-Pi-Lapdock-Connection/], here [http://www.instructables.com/id/The-Raspberry-Pi-Arduino-Connection/], and here [http://www.instructables.com/id/Switchable-Dual-Voltage-33v5v-Hacduino/] 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 [http://ibiblio.org/pub/Linux/utils/file/cruft-0.2.tar.gz] one-time pad encryption utility, the mcalc [http://ibiblio.org/pub/Linux/apps/financial/mcalc-1.6.tar.gz] mortgage calculator, the judge [http://ibiblio.org/pub/Linux/games/amusements/judge-1.0.tar.gz] Scrabble® adjudicator, the yawl

[1] This is the notorious *flog it to death* technique that works so well with slow learners, eccentrics, odd ducks, fools and geniuses.

[2] 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.

[3] Those who can, do. Those who can't . . . get an MCSE.

[http://ibiblio.org/pub/Linux/libs/yawl-0.3.2.tar.gz] word gaming list package, and the Quacky [http://bash.deta.in/qky.README.html] 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. [4]

Where to Go For Help

The author [mailto:thegrendel.abs@gmail.com] 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. [5]

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*

Tools Used to Produce This Book

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.

Software and Printware

i. Bram Moolenaar's powerful SGML-aware vim [http://www.vim.org] text editor.

ii. OpenJade [http://www.netfolder.com/DSSSL/], a DSSSL rendering engine for converting SGML documents into other formats.

[4]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!*"
[5]Well, if you *absolutely* insist, you can try modifying Example A.44, "An all-purpose shell scripting homework assignment solution" to suit your purposes.

iii. Norman Walsh's DSSSL stylesheets [http://nwalsh.com/docbook/dsssl/].

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.

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 [mailto:feloy@free.fr] 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 [mailto:feloy@free.fr].

Philippe Martin also pointed out that positional parameters past $9 are possible using {bracket} notation. (See Example 4.5, “Positional Parameters”).

Stéphane Chazelas [mailto:stephane_chazelas@yahoo.fr] 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 [mailto:mikaku@fiwix.org] for giving permission to use his fine tree script (Example A.16, “*tree*: Displaying a directory tree”), and to Rick Boivie for revising it.

Likewise, thanks to Michel Charpentier [mailto:charpov@cs.unh.edu] for permission to use his dc factoring script (Example 16.52, “Factoring”).

Kudos to Noah Friedman [mailto:friedman@prep.ai.mit.edu] for permission to use his string function script (Example A.18, “*string functions*: C-style string functions”).

Emmanuel Rouat [mailto:emmanuel.rouat@wanadoo.fr] suggested corrections and additions on command substitution, aliases, and path management. He also contributed a very nice sample `.bashrc` file (Appendix M, *Sample `.bashrc` and `.bash_profile` Files*).

Heiner Steven [mailto:heiner.steven@odn.de] kindly gave permission to use his base conversion script, Example 16.48, “Base Conversion”. 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, “A (useful) script that recursively calls itself”), revised the *tree.sh* script (Example A.16, “*tree*: Displaying a directory tree”), and suggested performance improvements for the *monthlypmt.sh* script (Example 16.47, “Monthly Payment on a Mortgage”).

Florian Wisser enlightened me on some of the fine points of testing strings (see Example 7.6, “Testing whether a string is *null*”), 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, “Analyzing a spam domain” and Example A.28, “Spammer Identification”).

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 [mailto:chet@po.cwru.edu] 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 [http://www.tldp.org]. 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 [http://www.fsf.org], 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.

Disclaimer

(This is a variant of the standard LDP [http://www.tldp.org] 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

Peter Denning. *Computers Under Attack: Intruders, Worms, and Viruses*. ACM Press. Copyright © 1990. 0-201-53067-8.

Ken Burtch. *Linux Shell Scripting with Bash [http://www.samspublishing.com/title/0672326426]*. 1st edition. Sams Publishing (Pearson). Copyright © 2004. 0672326426.

Daniel Goldman. *Definitive Guide to Sed [http://www.sed-book.com/]*. 1st edition. Copyright © 2013.

Dale Dougherty and Arnold Robbins. *Sed and Awk*. 2nd edition. O'Reilly and Associates. Copyright © 1997. 1-156592-225-5.

Jeffrey Friedl. *Mastering Regular Expressions*. O'Reilly and Associates. Copyright © 2002. 0-596-00289-0.

Aeleen Frisch. *Essential System Administration*. 3rd edition. O'Reilly and Associates. Copyright © 2002. 0-596-00343-9.

Stephen Kochan and Patrick Wood. *Unix Shell Programming*. Hayden. Copyright © 1990. 067248448X.

Neil Matthew and Richard Stones. *Beginning Linux Programming*. Wrox Press. Copyright © 1996. 1874416680.

[mayerref] Herbert Mayer. *Advanced C Programming on the IBM PC*. Windcrest Books. Copyright © 1989. 0830693637.

David Medinets. *Unix Shell Programming Tools*. McGraw-Hill. Copyright © 1999. 0070397333.

Cameron Newham and Bill Rosenblatt. *Learning the Bash Shell*. 2nd edition. O'Reilly and Associates. Copyright © 1998. 1-56592-347-2.

Anatole Olczak. *Bourne Shell Quick Reference Guide*. ASP, Inc.. Copyright © 1991. 093573922X.

Jerry Peek, Tim O'Reilly, and Mike Loukides. *Unix Power Tools*. 3rd edition. O'Reilly and Associates. Random House. Copyright © 2002. 0-596-00330-7.

Clifford Pickover. *Computers, Pattern, Chaos, and Beauty*. St. Martin's Press. Copyright © 1990. 0-312-04123-3.

George Polya. *How To Solve It*. Princeton University Press. Copyright © 1973. 0-691-02356-5.

Chet Ramey and Brian Fox. *The GNU Bash Reference Manual [http://www.network-theory.co.uk/bash/manual/]*. Network Theory Ltd. Copyright © 2003. 0-9541617-7-7.

Arnold Robbins. *Bash Reference Card*. SSC. Copyright © 1998. 1-58731-010-5.

Arnold Robbins. *Effective Awk Programming*. Free Software Foundation / O'Reilly and Associates. Copyright © 2000. 1-882114-26-4.

Bill Rosenblatt. *Learning the Korn Shell*. O'Reilly and Associates. Copyright © 1993. 1-56592-054-6.

Paul Sheer. *LINUX: Rute User's Tutorial and Exposition*. 1st edition. . Copyright © 2002. 0-13-033351-4.

Ellen Siever and the staff of O'Reilly and Associates. *Linux in a Nutshell*. 2nd edition. O'Reilly and Associates. Copyright © 1999. 1-56592-585-8.

Dave Taylor. *Wicked Cool Shell Scripts: 101 Scripts for Linux, Mac OS X, and Unix Systems*. 1st edition. No Starch Press. Copyright © 2004. 1-59327-012-7.

The UNIX CD Bookshelf. 3rd edition. O'Reilly and Associates. Copyright © 2003. 0-596-00392-7.

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, “*lowercase*: Changes all filenames in working directory to lowercase.”.

```
#! /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 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 the section called “Parameter Substitution”.

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
# $Id$
# 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

# $Id$
# 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=${FLCHAR:=75}
         # cd_flchar is used for the @s & @h history
                cd_lchar=${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 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			

Operator	**Meaning**	**-----**	**Operator**	**Meaning**
			`-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 *

Operator	Tests Whether	-----	Operator	Tests Whether
!	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`*

Expression	**Meaning**
`${string:position:length}`	Extract *$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.

Note

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, “ A *shell wrapper* around another awk script” and Example 28.2, “Passing an indirect reference to *awk*”).

Sed

Sed is a non-interactive [1] **s**tream **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. [2] 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

[1]*Sed* executes without user intervention.
[2]If no address range is specified, the default is *all* lines.

Operator	**Name**	**Effect**
`[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

Note

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".
```

Note

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 instructio
```

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.

Note

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, "Checking all the binaries in a directory for authorship" and Example 16.32, "Stripping comments from C program files".

Tip

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, "*shell wrapper*"
2. Example 36.2, " A slightly more complex *shell wrapper*"
3. Example 16.3, "*Badname*, eliminate file names in current directory containing bad characters and white-space."
4. Example A.2, "*rn*: A simple-minded file renaming utility"
5. Example 16.17, "Emulating *grep* in a script"
6. Example 16.27, "Using *column* to format a directory listing"
7. Example A.12, "*behead*: Removing mail and news message headers"
8. Example A.16, "*tree*: Displaying a directory tree"
9. Example A.17, "*tree2*: Alternate directory tree script"
10. Example 16.32, "Stripping comments from C program files"
11. Example 11.10, "Checking all the binaries in a directory for authorship"
12. Example 16.48, "Base Conversion"
13. Example A.1, "*mailformat*: Formatting an e-mail message"
14. Example 16.14, "Generating 10-digit random numbers"
15. Example 16.12, "Word Frequency Analysis"
16. Example A.10, "*Game of Life*"
17. Example 19.12, "A self-documenting script"
18. Example 16.19, "Looking up definitions in *Webster's 1913 Dictionary*"
19. Example A.29, "Spammer Hunt"
20. Example A.31, "A *podcasting* script"
21. Example A.24, "Converting to HTML"

22.Example A.43, “A command-line stopwatch”

23.Example A.55, “Inserting text in a file using *sed*”

For a more extensive treatment of *sed*, refer to the pertinent references in the Bibliography.

Awk

Awk [3] 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 }
```

[3]Its name derives from the initials of its authors, **A**ho, **W**einberg, and **K**ernighan.

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, “Forcing a log-off”
2. Example 20.8, “Redirected *for* loop”
3. Example 16.32, “Stripping comments from C program files”
4. Example 36.5, “ A *shell wrapper* around another awk script”
5. Example 28.2, “Passing an indirect reference to *awk*”
6. Example 15.20, “Using *export* to pass a variable to an embedded *awk* script”
7. Example 29.3, “Finding the process associated with a PID”
8. Example 29.4, “On-line connect status”
9. Example 11.3, “*Fileinfo:* operating on a file list contained in a variable”
10. Example 16.61, “Securely deleting a file”
11. Example 9.16, “Reseeding RANDOM”
12. Example 16.4, “Deleting a file by its *inode* number”
13. Example 10.6, “Alternate ways of extracting and locating substrings”
14. Example 36.19, “Even more return value trickery”
15. Example 11.9, “Listing all users on the system”
16. Example 36.4, “ A *shell wrapper* around an awk script”

17.Example 16.53, “Calculating the hypotenuse of a triangle”

18.Example T.3, “A third ASCII table script, using *awk*”

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 [1] 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 trou-

[1]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).

bleshooting scripts more straightforward. [2] 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, “Timed Input”.

Note

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.

[2]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.

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`.

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).

Caution

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 [http://www.gnu.org/prep/standards/].

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 [1])
`/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, *Sample `.bashrc` and `.bash_profile` Files* 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, [2] 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 Controlle
 vendorId: 8086
```

Note

This file is present on Red Hat and Fedora Core installations, but may be missing from other distros.

[1]This does not apply to **csh**, **tcsh**, and other shells not related to or descended from the classic Bourne shell (**sh**).
[2]In older versions of UNIX, passwords *were* stored in `/etc/passwd`, and that explains the name of the file.

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` [1]

  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, *`/dev` and `/proc`*.

[1]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.

- `/proc`

  Process directory. Contains information and statistics about running processes and kernel parameters. See Chapter 29, *`/dev` and `/proc`*.

- `/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.

  Warning

  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. [1] 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
```

[1]This works only from the *command line*, of course, and not within a script.

```
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. [2]

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.* [3]

--

Resources:

Bash programmable completion [http://freshmeat.net/projects/bashcompletion] project

Mitch Frazier's *Linux Journal* [http://www.linuxjournal.com] article, *More on Using the Bash Complete Command* [http://www.linuxjournal.com/content/more-using-bash-complete-command]

Steve's excellent two-part article, “An Introduction to Bash Completion”: Part 1 [http://www.debian-administration.org/article/An_introduction_to_bash_completion_part_1] and Part 2 [http://www.debian-administration.org/article/An_introduction_to_bash_completion_part_2]

[2]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.
[3]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*.

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.

Note

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.

Note

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 [http://samrowe.com/wordpress/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 [mailto:emmanuel.rouat@wanadoo.fr] 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 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

Batch File Operator	**Shell Script Equivalent**	**Meaning**
`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

DOS Command	**UNIX Equivalent**	**Effect**
`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

Note

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. [1]

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 [http://www.maem.umr.edu/batch/] site had a set of comprehensive tutorials on the old-fashioned art of batch file programming. Unfortunately the page has vanished without a trace.

[1]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 [http://www.ntfs-3g.org]) to do batch or scripted operations.

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 inscription
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.)
```

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, “*Game of Life*”, 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.

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, “Simple *for* loops” 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, “Converting numbers to Roman numerals”.

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, “A script that generates an ASCII table” 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, “Petals Around the Rose”, 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-
Smith,Tom,404 Polk Ave.,Los Angeles,CA,90003,(213) 879-5
...
```

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, "Word Frequency Analysis" 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 tri
#+ of the square root.
#  Our first "guess" at a square root is the argument it

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 


while [ ABS( $guess $oldguess ) -gt $tolerance ]
#       ^^^^^^^^^^^^^^^^^^^^^^^ Fix up syntax, of course

#      "ABS" is a (floating point) function to find the a
#+      of the difference between the two terms.
#             So, as long as difference between current a
#+            trial solution (guess) exceeds the toleranc
```

```
do
   oldguess = $guess  # Update $oldguess to previous $gue

#  ====================================================
   guess = ( $oldguess + ( $argument / $oldguess ) ) / 2
#        = 1/2 ( ($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, “Analyzing a spam domain” and Example A.28, “Spammer Identification”.

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, “A *man page* editor” 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, “Emulating a push-down stack” 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 [1] 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.)

[1]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.
```

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, "Calculating PI", 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
"J". The arbitrarily chosen key word, "CODES" comes first
the rest of the alphabet, in order from left to right, sk
already used.

To encrypt, separate the plaintext message into digrams (
groups). If a group has two identical letters, delete the
form a new group. If there is a single letter left over a
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
   row. If necessary, wrap around left to the beginning o

or

2) Both letters will be in the same column of the key squ
   For each letter, substitute the one immediately below 
   row. If necessary, wrap around to the top of the colum
```

```
or

3) Both letters will form the corners of a rectangle with
   For each letter, substitute the one on the other corne
   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 un
and #2 (move in opposite direction for substitution). Und
just take the remaining two corners of the rectangle.


Helen Fouche Gaines' classic work, ELEMENTARY CRYPTANALYS
fairly detailed description of the Playfair Cipher and it
```

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, “The Gronsfeld Cipher” 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 [http://www.tldp.org].
```

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.

Release	**Date**	**Comments**
`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.

Release	Date	Comments
`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 [http://bash.deta.in/abs-guide-latest.tar.bz2]). A pdf version [http://bash.deta.in/abs-guide.pdf] is also available (mirror site [http://www.mediafire.com/file/xi34ape1bifcnlb/abs-guide.pdf]). There is likewise an epub version [http://bash.deta.in/abs-guide.epub], courtesy of Craig Barnes and Michael Satke. The change log [http://bash.deta.in/Change.log] gives a detailed revision history. The *ABS Guide* even has its own `freshmeat.net/freecode` page [http://freecode.com/projects/advancedbashscriptingguide/] 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 [http://www.tldp.org/LDP/abs/], which maintains many other Guides and HOWTOs as well.

Many thanks to Ronny Bangsund for donating server space [http://bash.deta.in/] 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.

Warning

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 [http://www.tldp.org/manifesto.html].

Hyun Jin Cha has done a Korean translation [http://kldp.org/HOWTO/html/Adv-Bash-Scr-HOWTO/index.html] of version 1.0.11 of this book. Spanish, Portuguese, French [http://abs.traduc.org/], German, Italian [http://it.tldp.org/guide/abs/index.html], Russian [http://gazette.linux.ru.net/rus/articles/index-abs-guide.html], Czech [http://premekvihan.net/bash], Chinese [http://www.linuxsir.org/bbs/showthread.php?t=256887], 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 [http://bash.deta.in/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.

* * *

^ (caret)

- Beginning-of-line, in a Regular Expression
- ^

  ^^

  Uppercase conversion in *parameter substitution*

~ *Tilde*

- ~ home directory, corresponds to `$HOME`
- ~/ *Current user's* home directory
- ~+ *Current* working directory
- ~- *Previous* working directory

= *Equals* sign

- = Variable assignment operator
- = String comparison operator

  == String comparison operator
- =~ *Regular Expression* match operator

  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