Advanced Bash-Scripting Guide

An in-depth exploration of the gentle art of shell scripting

Mendel Cooper

Brindle-Phlogiston Associates

thegrendel@theriver.com

16 June 2002

Revision History

Revision 0.1	14 June 2000	Revised by: mc
Initial release.		
Revision 0.2	30 October 2000	Revised by: mc
Bugs fixed, plus much additional material and more example scripts.		
Revision 0.3	12 February 2001	Revised by: mc
Another major update.		
Revision 0.4	08 July 2001	Revised by: mc
More bugfixes, much more material, more scripts - a complete revision and expansion of the book.		
Revision 0.5	03 September 2001	Revised by: mc
Major update. Bugfixes, material added, chapters and sections reorganized.		
Revision 1.0	14 October 2001	Revised by: mc
Bugfixes, reorganization, material added. Stable release.		
Revision 1.1	06 January 2002	Revised by: mc
Bugfixes, material and scripts added.		
Revision 1.2	31 March 2002	Revised by: mc
Bugfixes, material and scripts added.		
Revision 1.3	02 June 2002	Revised by: mc

'TANGERINE' release: A few bugfixes, much more material and scripts added.

Revision 1.4 16 June 2002 Revised by: mc

'MANGO' release: Quite a number of typos fixed, more material and scripts added.

This tutorial assumes no previous knowledge of scripting or programming, but progresses rapidly toward an intermediate/advanced level of instruction ...*all the while sneaking in little snippets of UNIX wisdom and lore*. It serves as a textbook, a manual for self-study, and 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`**.

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. See the change log for a revision history.

Dedication

For Anita, the source of all the magic

Table of Contents

List of Tables

List of Examples

Next
Introduction

12.5. File and Archiving Commands

Archiving

tar

The standard UNIX archiving utility. Originally a *Tape ARchiving* program, it has developed into a general purpose package that can handle all manner of archiving with all types of destination devices, ranging from tape drives to regular files to even `stdout` (see Example 4-4). GNU tar has been patched to accept various compression filters, such as **tar czvf archive_name.tar.gz ***, which recursively archives and gzips all files in a directory tree except dotfiles in the current working directory (**$PWD**). [1]

Some useful **tar** options:

1. `-c` create (a new archive)
2. `-x` extract (files from existing archive)
3. `--delete` delete (files from existing archive)

This option will not work on magnetic tape devices.

4. `-r` append (files to existing archive)
5. `-A` append (*tar* files to existing archive)
6. `-t` list (contents of existing archive)
7. `-u` update archive
8. `-d` compare archive with specified filesystem
9. `-z` gzip the archive

(compress or uncompress, depending on whether combined with the `-c` or `-x`) option

10. `-j` bzip2 the archive

It may be difficult to recover data from a corrupted *gzipped* tar archive. When archiving important files, make multiple backups.

shar

Shell archiving utility. The files in a shell archive are concatenated without compression, and the resultant archive is essentially a shell script, complete with #!/bin/sh header, and containing all the necessary unarchiving commands. Shar archives still show up in Internet newsgroups, but otherwise **shar** has been pretty well replaced by **tar**/**gzip**. The **unshar** command unpacks **shar** archives.

ar

Creation and manipulation utility for archives, mainly used for binary object file libraries.

cpio

This specialized archiving copy command (**co**p**y** **i**nput and **o**utput) is rarely seen any more, having been supplanted by **tar**/**gzip**. It still has its uses, such as moving a directory tree.

Example 12-22. Using cpio to move a directory tree

```
#!/bin/bash

# Copying a directory tree using cpio.

ARGS=2
E_BADARGS=65

if [ $# -ne "$ARGS" ]
then
  echo "Usage: `basename $0` source destination"
  exit $E_BADARGS
fi

source=$1
destination=$2

find "$source" -depth | cpio -admvp "$destination"
# Read the man page to decipher these cpio options.

exit 0
```

Example 12-23. Unpacking an *rpm* archive

```
#!/bin/bash
# de-rpm.sh: Unpack an 'rpm' archive

E_NO_ARGS=65
TEMPFILE=$$.cpio                          # Tempfile with "unique" name.
                                          # $$ is process ID of script.

if [ -z "$1" ]
then
  echo "Usage: `basename $0` filename"
exit $E_NO_ARGS
fi


rpm2cpio < $1 > $TEMPFILE                 # Converts rpm archive into cpio archive.
cpio --make-directories -F $TEMPFILE -i   # Unpacks cpio archive.
rm -f $TEMPFILE                           # Deletes cpio archive.

exit 0
```

Compression

gzip

The standard GNU/UNIX compression utility, replacing the inferior and proprietary **compress**. The corresponding decompression command is **gunzip**, which is the equivalent of **gzip -d**.

The **zcat** filter decompresses a *gzipped* file to `stdout`, as possible input to a pipe or redirection. This is, in effect, a **cat** command that works on compressed files (including files processed with the older **compress** utility). The **zcat** command is equivalent to **gzip -dc**.

On some commercial UNIX systems, **zcat** is a synonym for **uncompress -c**, and will not work on *gzipped* files.

See also Example 7-6.

bzip2

An alternate compression utility, usually more efficient (but slower) than **gzip**, especially on large files. The corresponding decompression command is **bunzip2**.

Newer versions of tar have been patched with **bzip2** support.

compress, **uncompress**

This is an older, proprietary compression utility found in commercial UNIX distributions. The more efficient **gzip** has largely replaced it. Linux distributions generally include a **compress** workalike for compatibility, although **gunzip** can unarchive files treated with **compress**.

The **znew** command transforms *compressed* files into *gzipped* ones.

sq

Yet another compression utility, a filter that works only on sorted ASCII word lists. It uses the standard invocation syntax for a filter, **sq < input-file > output-file**. Fast, but not nearly as efficient as gzip. The corresponding uncompression filter is **unsq**, invoked like **sq**.

The output of **sq** may be piped to **gzip** for further compression.

zip, **unzip**

Cross-platform file archiving and compression utility compatible with DOS *pkzip.exe*. "Zipped" archives seem to be a more acceptable medium of exchange on the Internet than "tarballs".

unarc, **unarj**, **unrar**

These Linux utilities permit unpacking archives compressed with the DOS *arc.exe*, *arj.exe*, and *rar.exe* programs.

File Information

file

A utility for identifying file types. The command **`file file-name`** will return a file specification for `file-name`, such as `ascii text` or `data`. It references the magic numbers found in `/usr/share/magic`, `/etc/magic`, or `/usr/lib/magic`, depending on the Linux/UNIX distribution.

The `-f` option causes **file** to run in batch mode, to read from a designated file a list of filenames to analyze. The `-z` option, when used on a compressed target file, forces an attempt to analyze the uncompressed file type.

```
bash$ file test.tar.gz
test.tar.gz: gzip compressed data, deflated, last modified: Sun Sep 16 13:34:51 2001,
os: Unix

bash file -z test.tar.gz
test.tar.gz: GNU tar archive (gzip compressed data, deflated, last modified: Sun Sep
16 13:34:51 2001, os: Unix)
```

Example 12-24. stripping comments from C program files

```
#!/bin/bash
# strip-comment.sh: Strips out the comments (/* COMMENT */) in a C program.

E_NOARGS=65
E_ARGERROR=66
E_WRONG_FILE_TYPE=67

if [ $# -eq "$E_NOARGS" ]
then
  echo "Usage: `basename $0` C-program-file" >&2 # Error message to stderr.
  exit $E_ARGERROR
fi

# Test for correct file type.
type=`eval file $1 | awk '{ print $2, $3, $4, $5 }'`
# "file $1" echoes file type...
# then awk removes the first field of this, the filename...
# then the result is fed into the variable "type".
correct_type="ASCII C program text"

if [ "$type" != "$correct_type" ]
then
  echo
  echo "This script works on C program files only."
  echo
  exit $E_WRONG_FILE_TYPE
fi


# Rather cryptic sed script:
#--------
sed '
/^\/\*/d
/.*\/\*/d
' $1
#--------
# Easy to understand if you take several hours to learn sed fundamentals.


#  Need to add one more line to the sed script to deal with
#+ case where line of code has a comment following it on same line.
#  This is left as a non-trivial exercise.

# Also, the above code deletes lines with a "*/" or "/*",
# not a desirable result.

exit 0


# ------------------------------------------------------------------
# Code below this line will not execute because of 'exit 0' above.

# Stephane Chazelas suggests the following alternative:

usage() {
  echo "Usage: `basename $0` C-program-file" >&2
  exit 1
}

WEIRD=`echo -n -e '\377'`   # or WEIRD=$'\377'
[[ $# -eq 1 ]] || usage
```

```
case `file "$1"` in
  *"C program text"*) sed -e "s%/\*%${WEIRD}%g;s%\*/%${WEIRD}%g" "$1" \
     | tr '\377\n' '\n\377' \
     | sed -ne 'p;n' \
     | tr -d '\n' | tr '\377' '\n';;
  *) usage;;
esac

# This is still fooled by things like:
# printf("/*");
# or
# /*  /* buggy embedded comment */
#
# To handle all special cases (comments in strings, comments in string
# where there is a \", \\" ...) the only way is to write a C parser
# (lex or yacc perhaps?).

exit 0
```

which

which command-xxx gives the full path to "command-xxx". This is useful for finding out whether a particular command or utility is installed on the system.

$bash which rm

```
/usr/bin/rm
```

whereis

Similar to **which**, above, **whereis command-xxx** gives the full path to "command-xxx", but also to its *manpage*.

$bash whereis rm

```
rm: /bin/rm /usr/share/man/man1/rm.1.bz2
```

whatis

whatis filexxx looks up "filexxx" in the *whatis* database. This is useful for identifying system commands and important configuration files. Consider it a simplified **man** command.

$bash whatis whatis

```
whatis               (1)  - search the whatis database for complete words
```

Example 12-25. Exploring /usr/X11R6/bin

```
#!/bin/bash

# What are all those mysterious binaries in /usr/X11R6/bin?

DIRECTORY="/usr/X11R6/bin"
# Try also "/bin", "/usr/bin", "/usr/local/bin", etc.

for file in $DIRECTORY/*
do
  whatis `basename $file`   # Echoes info about the binary.
done

exit 0
# You may wish to redirect output of this script, like so:
# ./what.sh >>whatis.db
# or view it a page at a time on stdout,
# ./what.sh | less
```

See also Example 10-3.

vdir

Show a detailed directory listing. The effect is similar to ls -l.

This is one of the GNU *fileutils*.

```
bash$ vdir
total 10
 -rw-r--r--    1 bozo  bozo      4034 Jul 18 22:04 data1.xrolo
 -rw-r--r--    1 bozo  bozo      4602 May 25 13:58 data1.xrolo.bak
 -rw-r--r--    1 bozo  bozo       877 Dec 17  2000 employment.xrolo

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

shred

Securely erase a file by overwriting it multiple times with random bit patterns before deleting it. This command has the same effect as Example 12-41, but does it in a more thorough and elegant manner.

This is one of the GNU *fileutils*.

Using **shred** on a file may not prevent recovery of some or all of its contents using advanced forensic technology.

locate, **slocate**

The **locate** command searches for files using a database stored for just that purpose. The **slocate** command is the secure version of **locate** (which may be aliased to **slocate**).

```
$bash locate hickson
```

```
/usr/lib/xephem/catalogs/hickson.edb
```

strings

Use the **strings** command to find printable strings in a binary or data file. It will list sequences of printable characters found in the target file. This might be handy for a quick 'n dirty examination of a core dump or for looking at an unknown graphic image file (**`strings image-file | more`** might show something like `JFIF`, which would identify the file as a *jpeg* graphic). In a script, you would probably parse the output of **strings** with grep or sed. See Example 10-7 and Example 10-9.

Example 12-26. An "improved" *strings* command

```
#!/bin/bash
# wstrings.sh: "word-strings" (enhanced "strings" command)
#
#  This script filters the output of "strings" by checking it
#+ against a standard word list file.
#  This effectively eliminates all the gibberish and noise,
#+ and outputs only recognized words.

# ============================================================
#                 Standard Check for Script Argument(s)
ARGS=1
E_BADARGS=65
E_NOFILE=66

if [ $# -ne $ARGS ]
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
# ============================================================


MINSTRLEN=3                          #  Minimum string length.
WORDFILE=/usr/share/dict/linux.words  #  Dictionary file.
                                     #  May specify a different
                                     #+ word list file
                                     #+ of format 1 word per line.


wlist=`strings "$1" | tr A-Z a-z | tr '[:space:]' Z | \
tr -cs '[:alpha:]' Z | tr -s '\173-\377' Z | tr Z ' '`

# Translate output of 'strings' command with multiple passes of 'tr'.
#  "tr A-Z a-z"  converts to lowercase.
#  "tr '[:space:]'"  converts whitespace characters to Z's.
#  "tr -cs '[:alpha:]' Z"  converts non-alphabetic characters to Z's,
#+ and squeezes multiple consecutive Z's.
#  "tr -s '\173-\377' Z"  converts all characters past 'z' to Z's
```

```
#+ and squeezes multiple consecutive Z's,
#+ which gets rid of all the weird characters that the previous
#+ translation failed to deal with.
#  Finally, "tr Z ' '" converts all those Z's to whitespace,
#+ which will be seen as word separators in the loop below.

#  Note the technique of feeding the output of 'tr' back to itself,
#+ but with different arguments and/or options on each pass.


for word in $wlist                    # Important:
                                      # $wlist must not be quoted here.
                                      # "$wlist" does not work.
                                      # Why?
do

  strlen=${#word}                     # String length.
  if [ "$strlen" -lt "$MINSTRLEN" ]   # Skip over short strings.
  then
    continue
  fi

  grep -Fw $word "$WORDFILE"          # Match whole words only.

done


exit 0
```

Comparison

diff, **patch**

> **diff**: flexible file comparison utility. It compares the target files line-by-line sequentially. In some applications, such as comparing word dictionaries, it may be helpful to filter the files through sort and **uniq** before piping them to **diff**. **`diff file-1 file-2`** outputs the lines in the files that differ, with carets showing which file each particular line belongs to.
>
> The `--side-by-side` option to **diff** outputs each compared file, line by line, in separate columns, with non-matching lines marked.
>
> There are available various fancy frontends for **diff**, such as **spiff**, **wdiff**, **xdiff**, and **mgdiff**.

The **diff** command returns an exit status of 0 if the compared files are identical, and 1 if they differ. This permits use of **diff** in a test construct within a shell script (see below).

A common use for **diff** is generating difference files to be used with **patch** The `-e` option outputs files suitable for **ed** or **ex** scripts.

patch: flexible versioning utility. Given a difference file generated by **diff**, **patch** can upgrade a previous version of a package to a newer version. It is much more convenient to distribute a relatively small "diff" file than the entire body of a newly revised package. Kernel "patches" have become the preferred method of distributing the frequent releases of the Linux kernel.

```
patch -p1 <patch-file
# Takes all the changes listed in 'patch-file'
# and applies them to the files referenced therein.
# This upgrades to a newer version of the package.
```

Patching the kernel:

```
cd /usr/src
gzip -cd patchXX.gz | patch -p0
# Upgrading kernel source using 'patch'.
# From the Linux kernel docs "README",
# by anonymous author (Alan Cox?).
```

The **diff** command can also recursively compare directories (for the filenames present).

```
bash$ diff -r ~/notes1 ~/notes2
Only in /home/bozo/notes1: file02
 Only in /home/bozo/notes1: file03
 Only in /home/bozo/notes2: file04
```

Use **zdiff** to compare *gzipped* files.

diff3

An extended version of **diff** that compares three files at a time. This command returns an exit value of 0 upon successful execution, but unfortunately this gives no information about the results of the comparison.

```
bash$ diff3 file-1 file-2 file-3
====
 1:1c
   This is line 1 of "file-1".
 2:1c
   This is line 1 of "file-2".
 3:1c
   This is line 1 of "file-3"
```

sdiff

Compare and/or edit two files in order to merge them into an output file. Because of its interactive nature, this command would find little use in a script.

cmp

The **cmp** command is a simpler version of **diff**, above. Whereas **diff** reports the differences between two files, **cmp** merely shows at what point they differ.

Like **diff**, **cmp** returns an exit status of 0 if the compared files are identical, and 1 if they differ. This permits use in a test construct within a shell script.

Example 12-27. Using cmp to compare two files within a script.

```
#!/bin/bash

ARGS=2  # Two args to script expected.
E_BADARGS=65
E_UNREADABLE=66

if [ $# -ne "$ARGS" ]
then
  echo "Usage: `basename $0` file1 file2"
  exit $E_BADARGS
fi

if [[ ! -r "$1" || ! -r "$2" ]]
then
  echo "Both files to be compared must exist and be readable."
  exit $E_UNREADABLE
fi

cmp $1 $2 &> /dev/null  # /dev/null buries the output of the "cmp" command.
# Also works with 'diff', i.e.,   diff $1 $2 &> /dev/null

if [ $? -eq 0 ]         # Test exit status of "cmp" command.
then
  echo "File \"$1\" is identical to file \"$2\"."
else
  echo "File \"$1\" differs from file \"$2\"."
fi

exit 0
```

Use **zcmp** on *gzipped* files.

comm

Versatile file comparison utility. The files must be sorted for this to be useful.

comm *-options first-file second-file*

`comm file-1 file-2` outputs three columns:

- column 1 = lines unique to `file-1`
- column 2 = lines unique to `file-2`
- column 3 = lines common to both.

The options allow suppressing output of one or more columns.

- `-1` suppresses column `1`
- `-2` suppresses column `2`
- `-3` suppresses column `3`
- `-12` suppresses both columns `1` and `2`, etc.

Utilities

basename

Strips the path information from a file name, printing only the file name. The construction **`basename $0`** lets the script know its name, that is, the name it was invoked by. This can be used for "usage" messages if, for example a script is called with missing arguments:

```
echo "Usage: `basename $0` arg1 arg2 ... argn"
```

dirname

Strips the **basename** from a filename, printing only the path information.

basename and **dirname** can operate on any arbitrary string. The argument does not need to refer to an existing file, or even be a filename for that matter (see Example A-8).

Example 12-28. basename and dirname

```
#!/bin/bash

a=/home/bozo/daily-journal.txt

echo "Basename of /home/bozo/daily-journal.txt = `basename $a`"
echo "Dirname of /home/bozo/daily-journal.txt = `dirname $a`"
echo
echo "My own home is `basename ~/`."         # Also works with just ~.
echo "The home of my home is `dirname ~/`."  # Also works with just ~.

exit 0
```

split

Utility for splitting a file into smaller chunks. Usually used for splitting up large files in order to back them up on floppies or preparatory to e-mailing or uploading them.

sum, cksum, md5sum

These are utilities for generating checksums. A *checksum* is a number mathematically calculated from the contents of a file, for the purpose of checking its integrity. A script might refer to a list of checksums for security purposes, such as ensuring that the contents of key system files have not been altered or corrupted. For security applications, use the 128-bit **md5sum** (**m**essage **d**igest check**sum**) command.

```
bash$ cksum /boot/vmlinuz
1670054224 804083 /boot/vmlinuz


bash$ md5sum /boot/vmlinuz
0f43eccea8f09e0a0b2b5cf1dcf333ba  /boot/vmlinuz
```

Note that **cksum** also shows the size, in bytes, of the target file.

Example 12-29. Checking file integrity

```
#!/bin/bash
# file-integrity.sh: Checking whether files in a given directory
#                    have been tampered with.

E_DIR_NOMATCH=70
E_BAD_DBFILE=71

dbfile=File_record.md5
# Filename for storing records.


set_up_database ()
{
  echo ""$directory"" > "$dbfile"
  # Write directory name to first line of file.
  md5sum "$directory"/* >> "$dbfile"
  # Append md5 checksums and filenames.
}

check_database ()
{
  local n=0
  local filename
  local checksum

  # ------------------------------------------- #
  #  This file check should be unnecessary,
  #+ but better safe than sorry.

  if [ ! -r "$dbfile" ]
  then
    echo "Unable to read checksum database file!"
    exit $E_BAD_DBFILE
  fi
  # ------------------------------------------- #

  while read record[n]
  do

    directory_checked="${record[0]}"
    if [ "$directory_checked" != "$directory" ]
    then
      echo "Directories do not match up!"
      # Tried to use file for a different directory.
      exit $E_DIR_NOMATCH
    fi

    if [ "$n" -gt 0 ]   # Not directory name.
    then
      filename[n]=$( echo ${record[$n]} | awk '{ print $2 }' )
      #  md5sum writes records backwards,
      #+ checksum first, then filename.
      checksum[n]=$( md5sum "${filename[n]}" )

      if [ "${record[n]}" = "${checksum[n]}" ]
      then
        echo "${filename[n]} unchanged."
      else
```

```
        echo "${filename[n]} : CHECKSUM ERROR!"
        # File has been changed since last checked.
      fi

    fi


    let "n+=1"
  done <"$dbfile"       # Read from checksum database file.

}

# ================================================= #
# main ()

if [ -z  "$1" ]
then
  directory="$PWD"      #  If not specified,
else                    #+ use current working directory.
  directory="$1"
fi

clear                   # Clear screen.

# ------------------------------------------------------------------- #
  if [ ! -r "$dbfile" ] # Need to create database file?
  then
    echo "Setting up database file, \""$directory"/"$dbfile"\"."; echo
    set_up_database
  fi
# ------------------------------------------------------------------- #

check_database          # Do the actual work.

echo

#  You may wish to redirect the stdout of this script to a file,
#+ especially if the directory checked has many files in it.

#  For a much more thorough file integrity check,
#+ consider the "Tripwire" package,
#+ http://sourceforge.net/projects/tripwire/.

exit 0
```

Encoding and Encryption

uuencode

This utility encodes binary files into ASCII characters, making them suitable for transmission in the body of an e-mail message or in a newsgroup posting.

uudecode

This reverses the encoding, decoding uuencoded files back into the original binaries.

Example 12-30. uudecoding encoded files

```
#!/bin/bash

lines=35        # Allow 35 lines for the header (very generous).

for File in *   # Test all the files in the current working directory...
do
  search1=`head -$lines $File | grep begin | wc -w`
  search2=`tail -$lines $File | grep end | wc -w`
  #  Uuencoded files have a "begin" near the beginning,
  #+ and an "end" near the end.
  if [ "$search1" -gt 0 ]
  then
    if [ "$search2" -gt 0 ]
    then
      echo "uudecoding - $File -"
      uudecode $File
    fi
  fi
done

#  Note that running this script upon itself fools it
#+ into thinking it is a uuencoded file,
#+ because it contains both "begin" and "end".

# Exercise:
# Modify this script to check for a newsgroup header.

exit 0
```

The fold -s command may be useful (possibly in a pipe) to process long uudecoded text messages downloaded from Usenet newsgroups.

mimencode, **mmencode**

The **mimencode** and **mmencode** commands process multimedia-encoded e-mail attachments. Although *mail user agents* (such as **pine** or **kmail**) normally handle this automatically, these particular utilities permit manipulating such attachments manually from the command line or in a batch by means of a shell script.

crypt

At one time, this was the standard UNIX file encryption utility. [2] Politically motivated government regulations prohibiting the export of encryption software resulted in the disappearance of **crypt** from much of the UNIX world, and it is still missing from most Linux distributions. Fortunately, programmers have come up with a number of decent alternatives to it, among them the author's very own cruft (see Example A-5).

Miscellaneous

make

Utility for building and compiling binary packages. This can also be used for any set of operations that is triggered by incremental changes in source files.

The **make** command checks a `Makefile`, a list of file dependencies and operations to be carried out.

install

Special purpose file copying command, similar to **cp**, but capable of setting permissions and attributes of the copied files. This command seems tailormade for installing software packages, and as such it shows up frequently in `Makefiles` (in the *`make`*

`install :` section). It could likewise find use in installation scripts.

ptx

The **ptx [targetfile]** command outputs a permuted index (cross-reference list) of the targetfile. This may be further filtered and formatted in a pipe, if necessary.

more, **less**

Pagers that display a text file or stream to `stdout`, one screenful at a time. These may be used to filter the output of a script.

Notes

[1] A **tar czvf archive_name.tar.gz *** *will* include dotfiles in directories *below* the current working directory. This is an undocumented GNU **tar** "feature".

[2] This is a symmetric block cipher, used to encrypt files on a single system or local network, as opposed to the "public key" cipher class, of which **pgp** is a well-known example.

Prev Text Processing Commands | Home Up | Next Communications Commands

12.4. Text Processing Commands

Commands affecting text and text files

sort

File sorter, often used as a filter in a pipe. This command sorts a text stream or file forwards or backwards, or according to various keys or character positions. Using the `-m` option, it merges presorted input files. The *info page* lists its many capabilities and options. See Example 10-9, Example 10-10, and Example A-9.

tsort

Topological sort, reading in pairs of whitespace-separated strings and sorting according to input patterns.

uniq

This filter removes duplicate lines from a sorted file. It is often seen in a pipe coupled with sort.

```
cat list-1 list-2 list-3 | sort | uniq > final.list
# Concatenates the list files,
# sorts them,
# removes duplicate lines,
# and finally writes the result to an output file.
```

The useful `-c` option prefixes each line of the input file with its number of occurrences.

```
bash$ cat testfile
This line occurs only once.
 This line occurs twice.
 This line occurs twice.
 This line occurs three times.
 This line occurs three times.
 This line occurs three times.


bash$ uniq -c testfile
      1 This line occurs only once.
       2 This line occurs twice.
       3 This line occurs three times.


bash$ sort testfile | uniq -c | sort -nr
      3 This line occurs three times.
       2 This line occurs twice.
       1 This line occurs only once.
```

The **`sort INPUTFILE | uniq -c | sort -nr`** command string produces a *frequency of occurrence* listing on the `INPUTFILE` file (the `-nr` options to **sort** cause a reverse numerical sort). This template finds use in analysis of log files

and dictionary lists, and wherever the lexical structure of a document needs to be examined.

Example 12-8. Word Frequency Analysis

```
#!/bin/bash
# wf.sh: Crude word frequency analysis on a text file.


# Check for input file on command line.
ARGS=1
E_BADARGS=65
E_NOFILE=66

if [ $# -ne "$ARGS" ]  # Correct number of arguments passed to script?
then
  echo "Usage: `basename $0` filename"
  exit $E_BADARGS
fi

if [ ! -f "$1" ]       # Check if file exists.
then
  echo "File \"$1\" does not exist."
  exit $E_NOFILE
fi



############################################################
# main ()
sed -e 's/\.//g'  -e 's/ /\
/g' "$1" | tr 'A-Z' 'a-z' | sort | uniq -c | sort -nr
#                           ========================
#                            Frequency of occurrence

#  Filter out periods and
#+ change space between words to linefeed,
#+ then shift characters to lowercase, and
#+ finally prefix occurrence count and sort numerically.
############################################################

# Exercises:
# ---------
# 1) Add 'sed' commands to filter out other punctuation, such as commas.
# 2) Modify to also filter out multiple spaces and other whitespace.
# 3) Add a secondary sort key, so that instances of equal occurrence
#+   are sorted alphabetically.

exit 0
```

```
bash$ cat testfile
This line occurs only once.
 This line occurs twice.
 This line occurs twice.
 This line occurs three times.
 This line occurs three times.
 This line occurs three times.


bash$ ./wf.sh testfile
      6 this
       6 occurs
       6 line
       3 times
       3 three
       2 twice
       1 only
       1 once
```

expand, **unexpand**

The **expand** filter converts tabs to spaces. It is often used in a pipe.

The **unexpand** filter converts spaces to tabs. This reverses the effect of **expand**.

cut

A tool for extracting fields from files. It is similar to the **`print $N`** command set in awk, but more limited. It may be simpler to use **cut** in a script than **awk**. Particularly important are the `-d` (delimiter) and `-f` (field specifier) options.

Using **cut** to obtain a listing of the mounted filesystems:

```
cat /etc/mtab | cut -d ' ' -f1,2
```

Using **cut** to list the OS and kernel version:

```
uname -a | cut -d" " -f1,3,11,12
```

Using **cut** to extract message headers from an e-mail folder:

```
bash$ grep '^Subject:' read-messages | cut -c10-80
Re: Linux suitable for mission-critical apps?
 MAKE MILLIONS WORKING AT HOME!!!
 Spam complaint
 Re: Spam complaint
```

Using **cut** to parse a file:

```
# List all the users in /etc/passwd.

FILENAME=/etc/passwd

for user in $(cut -d: -f1 $FILENAME)
do
  echo $user
done

# Thanks, Oleg Philon for suggesting this.
```

`cut -d ' ' -f2,3 filename` is equivalent to **`awk -F'[ ]' '{ print $2, $3 }' filename`**

See also Example 12-33.

paste

Tool for merging together different files into a single, multi-column file. In combination with **cut**, useful for creating system log files.

join

Consider this a special-purpose cousin of **paste**. This powerful utility allows merging two files in a meaningful fashion, which essentially creates a simple version of a relational database.

The **join** command operates on exactly two files, but pastes together only those lines with a common tagged field (usually a numerical label), and writes the result to `stdout`. The files to be joined should be sorted according to the tagged field for the matchups to work properly.

```
File: 1.data

100 Shoes
200 Laces
300 Socks
```

```
File: 2.data

100 $40.00
200 $1.00
300 $2.00
```

```
bash$ join 1.data 2.data
File: 1.data 2.data

 100 Shoes $40.00
 200 Laces $1.00
 300 Socks $2.00
```

The tagged field appears only once in the output.

head

lists the beginning of a file to `stdout` (the default is `10` lines, but this can be changed). It has a number of interesting options.

Example 12-9. Which files are scripts?

```
#!/bin/bash
# script-detector.sh: Detects scripts within a directory.

TESTCHARS=2    # Test first 2 characters.
SHABANG='#!'   # Scripts begin with a "sha-bang."

for file in *  # Traverse all the files in current directory.
do
  if [[ `head -c$TESTCHARS "$file"` = "$SHABANG" ]]
  #      head -c2                    #!
  #  The '-c' option to "head" outputs a specified
  #+ number of characters, rather than lines (the default).
  then
    echo "File \"$file\" is a script."
  else
    echo "File \"$file\" is *not* a script."
  fi
done

exit 0
```

Example 12-10. Generating 10-digit random numbers

```
#!/bin/bash
# rnd.sh: Outputs a 10-digit random number

# Script by Stephane Chazelas.

head -c4 /dev/urandom | od -N4 -tu4 | sed -ne '1s/.* //p'


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

# Analysis
# --------

# head:
# -c4 option takes first 4 bytes.

# od:
# -N4 option limits output to 4 bytes.
# -tu4 option selects unsigned decimal format for output.

# sed:
# -n option, in combination with "p" flag to the "s" command,
# outputs only matched lines.



# The author of this script explains the action of 'sed', as follows.
```

```
# head -c4 /dev/urandom | od -N4 -tu4 | sed -ne '1s/.* //p'
# ---------------------------------> |

# Assume output up to "sed" --------> |
# is 0000000 1198195154\n

# sed begins reading characters: 0000000 1198195154\n.
# Here it finds a newline character,
# so it is ready to process the first line (0000000 1198195154).
# It looks at its <range><action>s. The first and only one is

#   range     action
#   1         s/.* //p

# The line number is in the range, so it executes the action:
# tries to substitute the longest string ending with a space in the line
# ("0000000 ") with nothing (//), and if it succeeds, prints the result
# ("p" is a flag to the "s" command here, this is different from the "p" command).

# sed is now ready to continue reading its input. (Note that before
# continuing, if -n option had not been passed, sed would have printed
# the line once again).

# Now, sed reads the remainder of the characters, and finds the end of the file.
# It is now ready to process its 2nd line (which is also numbered '$' as
# it's the last one).
# It sees it is not matched by any <range>, so its job is done.

# In few word this sed commmand means:
# "On the first line only, remove any character up to the right-most space,
# then print it."

# A better way to do this would have been:
#           sed -e 's/.* //;q'

# Here, two <range><action>s (could have been written
#           sed -e 's/.* //' -e q):

#   range                    action
#   nothing (matches line)   s/.* //
#   nothing (matches line)   q (quit)

# Here, sed only reads its first line of input.
# It performs both actions, and prints the line (substituted) before quitting
# (because of the "q" action) since the "-n" option is not passed.

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

# A simpler altenative to the above 1-line script would be:
#           head -c4 /dev/urandom| od -An -tu4

exit 0
```

See also Example 12-30.

tail

lists the end of a file to `stdout` (the default is `10` lines). Commonly used to keep track of changes to a system logfile,

using the `-f` option, which outputs lines appended to the file.

Example 12-11. Using tail to monitor the system log

```
#!/bin/bash

filename=sys.log

cat /dev/null > $filename; echo "Creating / cleaning out file."
#  Creates file if it does not already exist,
#+ and truncates it to zero length if it does.
#  : > filename   and   > filename also work.

tail /var/log/messages > $filename
# /var/log/messages must have world read permission for this to work.

echo "$filename contains tail end of system log."

exit 0
```

See also Example 12-4, Example 12-30 and Example 30-6.

grep

A multi-purpose file search tool that uses regular expressions. It was originally a command/filter in the venerable **ed** line editor, **g/re/p**, that is, *global - regular expression - print*.

grep *pattern* [*file*...]

Search the target file(s) for occurrences of *pattern*, where *pattern* may be literal text or a regular expression.

```
bash$ grep '[rst]ystem.$' osinfo.txt
The GPL governs the distribution of the Linux operating system.
```

If no target file(s) specified, **grep** works as a filter on `stdout`, as in a pipe.

```
bash$ ps ax | grep clock
765 tty1     S      0:00 xclock
 901 pts/1    S      0:00 grep clock
```

The `-i` option causes a case-insensitive search.

The `-w` option matches only whole words.

The `-l` option lists only the files in which matches were found, but not the matching lines.

The `-r` (recursive) option searches files in the current working directory and all subdirectories below it.

The `-n` option lists the matching lines, together with line numbers.

```
bash$ grep -n Linux osinfo.txt
2:This is a file containing information about Linux.
 6:The GPL governs the distribution of the Linux operating system.
```

The `-v` (or `--invert-match`) option *filters out* matches.

```
grep pattern1 *.txt | grep -v pattern2

# Matches all lines in "*.txt" files containing "pattern1",
# but ***not*** "pattern2".
```

The `-c` (`--count`) option gives a numerical count of matches, rather than actually listing the matches.

```
grep -c txt *.sgml   # (number of occurrences of "txt" in "*.sgml" files)


#   grep -cz .
#            ^ dot
# means count (-c) zero-separated (-z) items matching "."
# that is, non-empty ones (containing at least 1 character).
#
printf 'a b\nc  d\n\n\n\n\n\000\n\000e\000\000\nf' | grep -cz .     # 4
printf 'a b\nc  d\n\n\n\n\n\000\n\000e\000\000\nf' | grep -cz '$'   # 5
printf 'a b\nc  d\n\n\n\n\n\000\n\000e\000\000\nf' | grep -cz '^'   # 5
#
printf 'a b\nc  d\n\n\n\n\n\000\n\000e\000\000\nf' | grep -c '$'    # 9
# By default, newline chars (\n) separate items to match.

# Note that the -z option is GNU "grep" specific.


# Thanks, S.C.
```

When invoked with more than one target file given, **grep** specifies which file contains matches.

```
bash$ grep Linux osinfo.txt misc.txt
osinfo.txt:This is a file containing information about Linux.
 osinfo.txt:The GPL governs the distribution of the Linux operating system.
 misc.txt:The Linux operating system is steadily gaining in popularity.
```

To force **grep** to show the filename when searching only one target file, simply give `/dev/null` as the second file.

```
bash$ grep Linux osinfo.txt /dev/null
osinfo.txt:This is a file containing information about Linux.
 osinfo.txt:The GPL governs the distribution of the Linux operating system.
```

If there is a successful match, **grep** returns an exit status of 0, which makes it useful in a condition test in a script, especially in combination with the `-q` option to suppress output.

```
SUCCESS=0                      # if grep lookup succeeds
word=Linux
filename=data.file

grep -q "$word" "$filename"    # The "-q" option causes nothing to echo to stdout.

if [ $? -eq $SUCCESS ]
then
  echo "$word found in $filename"
else
  echo "$word not found in $filename"
fi
```

Example 30-6 demonstrates how to use **grep** to search for a word pattern in a system logfile.

Example 12-12. Emulating "grep" in a script

```
#!/bin/bash
# grp.sh: Very crude reimplementation of 'grep'.

E_BADARGS=65

if [ -z "$1" ]    # Check for argument to script.
then
  echo "Usage: `basename $0` pattern"
  exit $E_BADARGS
fi

echo

for file in *     # Traverse all files in $PWD.
do
  output=$(sed -n /"$1"/p $file)  # Command substitution.

  if [ ! -z "$output" ]           # What happens if "$output" is not quoted?
  then
    echo -n "$file: "
    echo $output
  fi              #  sed -ne "/$1/s|^|${file}: |p"  is equivalent to above.

  echo
```

```
done

echo

exit 0

# Exercises:
# ---------
# 1) Add newlines to output, if more than one match in any given file.
# 2) Add features.
```

egrep is the same as **grep -E**. This uses a somewhat different, extended set of regular expressions, which can make the search somewhat more flexible.

fgrep is the same as **grep -F**. It does a literal string search (no regular expressions), which allegedly speeds things up a bit.

agrep extends the capabilities of **grep** to approximate matching. The search string may differ by a specified number of characters from the resulting matches. This utility is not part of the core Linux distribution.

To search compressed files, use **zgrep**, **zegrep**, or **zfgrep**. These also work on non-compressed files, though slower than plain **grep**, **egrep**, **fgrep**. They are handy for searching through a mixed set of files, some compressed, some not.

To search bzipped files, use **bzgrep**.

look

The command **look** works like **grep**, but does a lookup on a "dictionary", a sorted word list. By default, **look** searches for a match in `/usr/dict/words`, but a different dictionary file may be specified.

Example 12-13. Checking words in a list for validity

```
#!/bin/bash
# lookup: Does a dictionary lookup on each word in a data file.

file=words.data  # Data file from which to read words to test.

echo

while [ "$word" != end ]  # Last word in data file.
do
  read word      # From data file, because of redirection at end of loop.
  look $word > /dev/null  # Don't want to display lines in dictionary file.
  lookup=$?      # Exit status of 'look' command.

  if [ "$lookup" -eq 0 ]
  then
    echo "\"$word\" is valid."
  else
    echo "\"$word\" is invalid."
  fi

done <"$file"    # Redirects stdin to $file, so "reads" come from there.
```

```
echo

exit 0

# ----------------------------------------------------------------
# Code below line will not execute because of "exit" command above.


# Stephane Chazelas proposes the following, more concise alternative:

while read word && [[ $word != end ]]
do if look "$word" > /dev/null
   then echo "\"$word\" is valid."
   else echo "\"$word\" is invalid."
   fi
done <"$file"

exit 0
```

sed, **awk**

Scripting languages especially suited for parsing text files and command output. May be embedded singly or in combination in pipes and shell scripts.

sed

Non-interactive "stream editor", permits using many **ex** commands in batch mode. It finds many uses in shell scripts.

awk

Programmable file extractor and formatter, good for manipulating and/or extracting fields (columns) in structured text files. Its syntax is similar to C.

wc

wc gives a "word count" on a file or I/O stream:

```
bash $ wc /usr/doc/sed-3.02/README
20     127     838 /usr/doc/sed-3.02/README
[20 lines  127 words  838 characters]
```

wc -w gives only the word count.

wc -l gives only the line count.

wc -c gives only the character count.

wc -L gives only the length of the longest line.

Using **wc** to count how many *.txt* files are in current working directory:

```
$ ls *.txt | wc -l
# Will work as long as none of the "*.txt" files have a linefeed in their name.

# Alternative ways of doing this are:
#      find . -maxdepth 1 -name \*.txt -print0 | grep -cz .
#      (shopt -s nullglob; set -- *.txt; echo $#)

# Thanks, S.C.
```

Using **wc** to total up the size of all the files whose names begin with letters in the range d - h

```
bash$ wc [d-h]* | grep total | awk '{print $3}'
71832
```

Using **wc** to count the instances of the word "Linux" in the main source file for this book.

```
bash$ grep Linux abs-book.sgml | wc -l
50
```

See also Example 12-30 and Example 16-7.

Certain commands include some of the functionality of **wc** as options.

```
... | grep foo | wc -l
# This frequently used construct can be more concisely rendered.

... | grep -c foo
# Just use the "-c" (or "--count") option of grep.

# Thanks, S.C.
```

tr

character translation filter.

Must use quoting and/or brackets, as appropriate. Quotes prevent the shell from reinterpreting the special characters in **tr** command sequences. Brackets should be quoted to prevent expansion by the shell.

Either **`tr "A-Z" "*" <filename`** or **`tr A-Z \* <filename`** changes all the uppercase letters in `filename` to asterisks (writes to `stdout`). On some systems this may not work, but **`tr A-Z '[**]'`** will.

The `-d` option deletes a range of characters.

```
echo "abcdef"                 # abcdef
echo "abcdef" | tr -d b-d     # aef


tr -d 0-9 <filename
# Deletes all digits from the file "filename".
```

The `--squeeze-repeats` (or `-s`) option deletes all but the first instance of a string of consecutive characters. This option is useful for removing excess whitespace.

```
bash$ echo "XXXXX" | tr --squeeze-repeats 'X'
X
```

The `-c` "complement" option *inverts* the character set to match. With this option, **tr** acts only upon those characters *not* matching the specified set.

```
bash$ echo "acfdeb123" | tr -c b-d +
+c+d+b++++
```

Note that **tr** recognizes POSIX character classes. [1]

```
bash$ echo "abcd2ef1" | tr '[:alpha:]' -
----2--1
```

Example 12-14. toupper: Transforms a file to all uppercase.

```
#!/bin/bash
# Changes a file to all uppercase.

E_BADARGS=65

if [ -z "$1" ]  # Standard check for command line arg.
then
  echo "Usage: `basename $0` filename"
  exit $E_BADARGS
fi

tr a-z A-Z <"$1"

# Same effect as above, but using POSIX character set notation:
#       tr '[:lower:]' '[:upper:]' <"$1"
# Thanks, S.C.

exit 0
```

Example 12-15. lowercase: Changes all filenames in working directory to lowercase.

```
#! /bin/bash
#
# Changes every filename in working directory to all lowercase.
#
# Inspired by a script of John Dubois,
# which was translated into into Bash by Chet Ramey,
# and considerably simplified by Mendel Cooper, author of this document.


for filename in *               # Traverse all files in directory.
do
   fname=`basename $filename`
   n=`echo $fname | tr A-Z a-z`  # Change name to lowercase.
   if [ "$fname" != "$n" ]       # Rename only files not already lowercase.
   then
     mv $fname $n
   fi
done

exit 0


# Code below this line will not execute because of "exit".
#--------------------------------------------------------#
# To run it, delete script above line.

# The above script will not work on filenames containing blanks or newlines.

# Stephane Chazelas therefore suggests the following alternative:


for filename in *    # Not necessary to use basename,
                     # since "*" won't return any file containing "/".
do n=`echo "$filename/" | tr '[:upper:]' '[:lower:]'`
#                            POSIX char set notation.
#                    Slash added so that trailing newlines are not
#                    removed by command substitution.
   # Variable substitution:
   n=${n%/}          # Removes trailing slash, added above, from filename.
   [[ $filename == $n ]] || mv "$filename" "$n"
                     # Checks if filename already lowercase.
done

exit 0
```

Example 12-16. du: DOS to UNIX text file conversion.

```
#!/bin/bash
# du.sh: DOS to UNIX text file converter.

E_WRONGARGS=65

if [ -z "$1" ]
then
  echo "Usage: `basename $0` filename-to-convert"
  exit $E_WRONGARGS
fi

NEWFILENAME=$1.unx

CR='\015'  # Carriage return.
# Lines in a DOS text file end in a CR-LF.

tr -d $CR < $1 > $NEWFILENAME
# Delete CR and write to new file.

echo "Original DOS text file is \"$1\"."
echo "Converted UNIX text file is \"$NEWFILENAME\"."

exit 0
```

Example 12-17. rot13: rot13, ultra-weak encryption.

```
#!/bin/bash
# rot13.sh: Classic rot13 algorithm, encryption that might fool a 3-year old.

# Usage: ./rot13.sh filename
# or     ./rot13.sh <filename
# or     ./rot13.sh and supply keyboard input (stdin)

cat "$@" | tr 'a-zA-Z' 'n-za-mN-ZA-M'   # "a" goes to "n", "b" to "o", etc.
# The 'cat "$@"' construction
# permits getting input either from stdin or from files.

exit 0
```

Example 12-18. Generating "Crypto-Quote" Puzzles

```
#!/bin/bash
# crypto-quote.sh: Encrypt quotes

# Will encrypt famous quotes in a simple monoalphabetic substitution.
#  The result is similar to the "Crypto Quote" puzzles
#+ seen in the Op Ed pages of the Sunday paper.


key=ETAOINSHRDLUBCFGJMQPVWZYXK
# The "key" is nothing more than a scrambled alphabet.
# Changing the "key" changes the encryption.

# The 'cat "$@"' construction gets input either from stdin or from files.
# If using stdin, terminate input with a Control-D.
```

```
# Otherwise, specify filename as command-line parameter.

cat "$@" | tr "a-z" "A-Z" | tr "A-Z" "$key"
#        |  to uppercase  |     encrypt
# Will work on lowercase, uppercase, or mixed-case quotes.
# Passes non-alphabetic characters through unchanged.


# Try this script with something like
# "Nothing so needs reforming as other people's habits."
# --Mark Twain
#
# Output is:
# "CFPHRCS QF CIIOQ MINFMBRCS EQ FPHIM GIFGUI'Q HETRPQ."
# --BEML PZERC

# To reverse the encryption:
# cat "$@" | tr "$key" "A-Z"


#  This simple-minded cipher can be broken by an average 12-year old
#+ using only pencil and paper.

exit 0
```

tr variants

The **tr** utility has two historic variants. The BSD version does not use brackets (**`tr a-z A-Z`**), but the SysV one does (**`tr '[a-z]' '[A-Z]'`**). The GNU version of **tr** resembles the BSD one, so quoting letter ranges within brackets is mandatory.

fold

A filter that wraps lines of input to a specified width. This is especially useful with the `-s` option, which breaks lines at word spaces (see Example 12-19 and Example A-2).

fmt

Simple-minded file formatter, used as a filter in a pipe to "wrap" long lines of text output.

Example 12-19. Formatted file listing.

```
#!/bin/bash

WIDTH=40                    # 40 columns wide.

b=`ls /usr/local/bin`       # Get a file listing...

echo $b | fmt -w $WIDTH

# Could also have been done by
#  echo $b | fold - -s -w $WIDTH

exit 0
```

See also Example 12-4.

A powerful alternative to **fmt** is Kamil Toman's **par** utility, available from http://www.cs.berkeley.edu/~amc/Par/.

col

This deceptively named filter removes reverse line feeds from an input stream. It also attempts to replace whitespace with equivalent tabs. The chief use of **col** is in filtering the output from certain text processing utilities, such as **groff** and **tbl**.

column

Column formatter. This filter transforms list-type text output into a "pretty-printed" table by inserting tabs at appropriate places.

Example 12-20. Using column to format a directory listing

```
#!/bin/bash
# This is a slight modification of the example file in the "column" man page.


(printf "PERMISSIONS LINKS OWNER GROUP SIZE MONTH DAY HH:MM PROG-NAME\n" \
; ls -l | sed 1d) | column -t

#  The "sed 1d" in the pipe deletes the first line of output,
#+ which would be "total       N",
#+ where "N" is the total number of files found by "ls -l".

# The -t option to "column" pretty-prints a table.

exit 0
```

colrm

Column removal filter. This removes columns (characters) from a file and writes the file, lacking the range of specified columns, back to `stdout`. **`colrm 2 4 <filename`** removes the second through fourth characters from each line of the text file `filename`.

If the file contains tabs or nonprintable characters, this may cause unpredictable behavior. In such cases, consider using expand and **unexpand** in a pipe preceding **colrm**.

nl

Line numbering filter. **`nl filename`** lists `filename` to `stdout`, but inserts consecutive numbers at the beginning of each non-blank line. If `filename` omitted, operates on `stdin`.

The output of **nl** is very similar to **`cat -n`**, however, by default **nl** does not list blank lines.

Example 12-21. nl: A self-numbering script.

```
#!/bin/bash

# This script echoes itself twice to stdout with its lines numbered.

# 'nl' sees this as line 3 since it does not number blank lines.
# 'cat -n' sees the above line as number 5.

nl `basename $0`

echo; echo  # Now, let's try it with 'cat -n'

cat -n `basename $0`
# The difference is that 'cat -n' numbers the blank lines.
# Note that 'nl -ba' will also do so.

exit 0
```

pr

Print formatting filter. This will paginate files (or `stdout`) into sections suitable for hard copy printing or viewing on screen. Various options permit row and column manipulation, joining lines, setting margins, numbering lines, adding page headers, and merging files, among other things. The **pr** command combines much of the functionality of **nl**, **paste**, **fold**, **column**, and **expand**.

`pr -o 5 --width=65 fileZZZ | more` gives a nice paginated listing to screen of `fileZZZ` with margins set at 5 and 65.

A particularly useful option is `-d`, forcing double-spacing (same effect as **sed -G**).

gettext

A GNU utility for localization and translating the text output of programs into foreign languages. While primarily intended for C programs, **gettext** also finds use in shell scripts. See the *`info page`*.

iconv

A utility for converting file(s) to a different encoding (character set). Its chief use is for localization.

recode

Consider this a fancier version of **iconv**, above. This very versatile utility for converting a file to a different encoding is not part of the standard Linux installation.

TeX, **gs**

TeX and **Postscript** are text markup languages used for preparing copy for printing or formatted video display.

TeX is Donald Knuth's elaborate typsetting system. It is often convenient to write a shell script encapsulating all the options and arguments passed to one of these markup languages.

Ghostscript (**gs**) is a GPL-ed Postscript interpreter.

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

Yet another text markup and display formatting language is **groff**. This is the enhanced GNU version of the venerable UNIX **roff/troff** display and typesetting package. *Manpages* use **groff** (see Example A-1).

The **tbl** table processing utility is considered part of **groff**, as its function is to convert table markup into **groff** commands.

The **eqn** equation processing utility is likewise part of **groff**, and its function is to convert equation markup into **groff** commands.

lex, **yacc**

The **lex** lexical analyzer produces programs for pattern matching. This has been replaced by the nonproprietary **flex** on Linux systems.

The **yacc** utility creates a parser based on a set of specifications. This has been replaced by the nonproprietary **bison** on Linux systems.

Notes

[1] This is only true of the GNU version of **tr**, not the generic version often found on commercial UNIX systems.

Prev Home Next
Time / Date Commands Up File and Archiving Commands

12.3. Time / Date Commands

Manipulating the time and date

date

Simply invoked, **date** prints the date and time to `stdout`. Where this command gets interesting is in its formatting and parsing options.

Example 12-7. Using date

```
#!/bin/bash
# Exercising the 'date' command

echo "The number of days since the year's beginning is `date +%j`."
# Needs a leading '+' to invoke formatting.
# %j gives day of year.

echo "The number of seconds elapsed since 01/01/1970 is `date +%s`."
#  %s yields number of seconds since "UNIX epoch" began,
#+ but how is this useful?

prefix=temp
suffix=`eval date +%s`  # The "+%s" option to 'date' is GNU-specific.
filename=$prefix.$suffix
echo $filename
#  It's great for creating "unique" temp filenames,
#+ even better than using $$.

# Read the 'date' man page for more formatting options.

exit 0
```

The `-u` option gives the UTC (Universal Coordinated Time).

```
bash$ date
Fri Mar 29 21:07:39 MST 2002


bash$ date -u
Sat Mar 30 04:07:42 UTC 2002
```

zdump

Echoes the time in a specified time zone.

```
bash$ zdump EST
EST  Tue Sep 18 22:09:22 2001 EST
```

time

Outputs very verbose timing statistics for executing a command.

`time ls -l /` gives something like this:

```
0.00user 0.01system 0:00.05elapsed 16%CPU (0avgtext+0avgdata 0maxresident)k
 0inputs+0outputs (149major+27minor)pagefaults 0swaps
```

See also the very similar times command in the previous section.

As of version 2.0 of Bash, **time** became a shell reserved word, with slightly altered behavior in a pipeline.

touch

Utility for updating access/modification times of a file to current system time or other specified time, but also useful for creating a new file. The command **`touch zzz`** will create a new file of zero length, named `zzz`, assuming that `zzz` did not previously exist. Time-stamping empty files in this way is useful for storing date information, for example in keeping track of modification times on a project.

The **touch** command is equivalent to **`: >> newfile`** or **`>> newfile`** (for ordinary files).

at

The **at** job control command executes a given set of commands at a specified time. Superficially, it resembles crond, however, **at** is chiefly useful for one-time execution of a command set.

`at 2pm January 15` prompts for a set of commands to execute at that time. These commands should be shell-script compatible, since, for all practical purposes, the user is typing in an executable shell script a line at a time. Input terminates with a Ctl-D.

Using either the `-f` option or input redirection (<), **at** reads a command list from a file. This file is an executable shell script, though it should, of course, be noninteractive. Particularly clever is including the run-parts command in the file to execute a different set of scripts.

```
bash$ at 2:30 am Friday < at-jobs.list
job 2 at 2000-10-27 02:30
```

batch

The **batch** job control command is similar to **at**, but it runs a command list when the system load drops below `.8`. Like **at**, it can read commands from a file with the `-f` option.

cal

Prints a neatly formatted monthly calendar to `stdout`. Will do current year or a large range of past and future years.

sleep

This is the shell equivalent of a wait loop. It pauses for a specified number of seconds, doing nothing. This can be useful for timing or in processes running in the background, checking for a specific event every so often (see Example 30-6).

```
sleep 3
# Pauses 3 seconds.
```

The **sleep** command defaults to seconds, but minute, hours, or days may also be specified.

```
sleep 3 h
# Pauses 3 hours!
```

usleep

Microsleep (the "u" may be read as the Greek "mu", or micro prefix). This is the same as **sleep**, above, but "sleeps" in microsecond intervals. This can be used for fine-grain timing, or for polling an ongoing process at very frequent intervals.

```
usleep 30
# Pauses 30 microseconds.
```

The **usleep** command does not provide particularly accurate timing, and is therefore unsuitable for critical timing loops.

hwclock, **clock**

The **hwclock** command accesses or adjusts the machine's hardware clock. Some options require root privileges. The `/etc/rc.d/rc.sysinit` startup file uses **hwclock** to set the system time from the hardware clock at bootup.

The **clock** command is a synonym for **hwclock**.

Prev
Complex Commands

Home
Up

Next
Text Processing Commands

12.2. Complex Commands

Commands for more advanced users

find

-exec *COMMAND* \;

Carries out *COMMAND* on each file that **find** scores a hit on. *COMMAND* terminates with \; (the ; is escaped to make certain the shell passes it to **find** literally, which concludes the command sequence). If *COMMAND* contains {}, then **find** substitutes the full path name of the selected file.

```
bash$ find ~/ -name '*.txt'
/home/bozo/.kde/share/apps/karm/karmdata.txt
 /home/bozo/misc/irmeyc.txt
 /home/bozo/test-scripts/1.txt
```

```
find /home/bozo/projects -mtime 1
# Lists all files in /home/bozo/projects directory tree
# that were modified within the last day.
```

```
find /etc -exec grep '[0-9][0-9]*[.][0-9][0-9]*[.][0-9][0-9]*[.][0-9][0-9]*' {} \;

# Finds all IP addresses (xxx.xxx.xxx.xxx) in /etc directory files.
# There a few extraneous hits - how can they be filtered out?

# Perhaps by:

find /etc -type f -exec cat '{}' \; | tr -c '.[:digit:]' '\n' \
 | grep '^[^.][^.]*\.[^.][^.]*\.[^.][^.]*\.[^.][^.]*$'
# [:digit:] is one of the character classes
# introduced with the POSIX 1003.2 standard.

# Thanks, S.C.
```

The `-exec` option to **find** should not be confused with the exec shell builtin.

Example 12-2. Badname, eliminate file names in current directory containing bad characters and whitespace.

```
#!/bin/bash

# Delete filenames in current directory containing bad characters.

for filename in *
do
badname=`echo "$filename" | sed -n /[\+\{\;\"\\\=\?~\(\)\<\>\&\*\|\$]/p`
# Files containing those nasties:     + { ; " \ = ? ~ ( ) < > & * | $
rm $badname 2>/dev/null    # So error messages deep-sixed.
done

# Now, take care of files containing all manner of whitespace.
find . -name "* *" -exec rm -f {} \;
# The path name of the file that "find" finds replaces the "{}".
# The '\' ensures that the ';' is interpreted literally, as end of command.

exit 0

#-----------------------------------------------------------------------
# Commands below this line will not execute because of "exit" command.

# An alternative to the above script:
find . -name '*[+{;"\\=?~()<>&*|$ ]*' -exec rm -f '{}' \;
exit 0
# (Thanks, S.C.)
```

Example 12-3. Deleting a file by its *inode* number

```
#!/bin/bash
# idelete.sh: Deleting a file by its inode number.

#  This is useful when a filename starts with an illegal character,
#+ such as ? or -.

ARGCOUNT=1                      # Filename arg must be passed to script.
E_WRONGARGS=70
E_FILE_NOT_EXIST=71
E_CHANGED_MIND=72

if [ $# -ne "$ARGCOUNT" ]
then
  echo "Usage: `basename $0` filename"
  exit $E_WRONGARGS
fi

if [ ! -e "$1" ]
then
  echo "File \""$1"\" does not exist."
  exit $E_FILE_NOT_EXIST
fi

inum=`ls -i | grep "$1" | awk '{print $1}'`
# inum = inode (index node) number of file
# Every file has an inode, a record that hold its physical address info.

echo; echo -n "Are you absolutely sure you want to delete \"$1\" (y/n)? "
read answer
```

```
case "$answer" in
[nN]) echo "Changed your mind, huh?"
      exit $E_CHANGED_MIND
      ;;
*)    echo "Deleting file \"$1\".";;
esac

find . -inum $inum -exec rm {} \;
echo "File "\"$1"\" deleted!"

exit 0
```

See Example 12-22, Example 4-4, and Example 10-9 for scripts using **find**. Its manpage provides more detail on this complex and powerful command.

xargs

A filter for feeding arguments to a command, and also a tool for assembling the commands themselves. It breaks a data stream into small enough chunks for filters and commands to process. Consider it as a powerful replacement for backquotes. In situations where backquotes fail with a too many arguments error, substituting **xargs** often works. Normally, **xargs** reads from `stdin` or from a pipe, but it can also be given the output of a file.

The default command for **xargs** is echo. This means that input piped to **xargs** may have linefeeds and other whitespace characters stripped out.

```
bash$ ls -l
total 0
 -rw-rw-r--    1 bozo  bozo         0 Jan 29 23:58 file1
 -rw-rw-r--    1 bozo  bozo         0 Jan 29 23:58 file2


bash$ ls -l | xargs
total 0 -rw-rw-r-- 1 bozo bozo 0 Jan 29 23:58 file1 -rw-rw-r-- 1 bozo bozo 0 Jan 29
23:58 file2
```

ls | xargs -p -l gzip gzips every file in current directory, one at a time, prompting before each operation.

An interesting **xargs** option is `-n` *NN*, which limits to *NN* the number of arguments passed.

ls | xargs -n 8 echo lists the files in the current directory in 8 columns.

Another useful option is `-0`, in combination with **find -print0** or **grep -lZ**. This allows handling arguments containing whitespace or quotes.

```
find / -type f -print0 | xargs -0 grep -liwZ GUI | xargs -0 rm -f

grep -rliwZ GUI / | xargs -0 rm -f
```

Either of the above will remove any file containing "GUI". *(Thanks, S.C.)*

Example 12-4. Logfile using xargs to monitor system log

```
#!/bin/bash

# Generates a log file in current directory
# from the tail end of /var/log/messages.

# Note: /var/log/messages must be world readable
# if this script invoked by an ordinary user.
#         #root chmod 644 /var/log/messages

LINES=5

( date; uname -a ) >>logfile
# Time and machine name
echo ------------------------------------------------------------------- >>logfile
tail -$LINES /var/log/messages | xargs |  fmt -s >>logfile
echo >>logfile
echo >>logfile

exit 0
```

Example 12-5. copydir, copying files in current directory to another, using xargs

```
#!/bin/bash

# Copy (verbose) all files in current directory
# to directory specified on command line.

if [ -z "$1" ]   # Exit if no argument given.
then
  echo "Usage: `basename $0` directory-to-copy-to"
  exit 65
fi

ls . | xargs -i -t cp ./{} $1
# This is the exact equivalent of
#    cp * $1
# unless any of the filenames has "whitespace" characters.

exit 0
```

expr

All-purpose expression evaluator: Concatenates and evaluates the arguments according to the operation given (arguments must be separated by spaces). Operations may be arithmetic, comparison, string, or logical.

expr 3 + 5

returns 8

expr 5 % 3

returns 2

expr 5 * 3

returns 15

The multiplication operator must be escaped when used in an arithmetic expression with **expr**.

y=`expr $y + 1`

Increment a variable, with the same effect as **let y=y+1** and **y=$(($y+1))**. This is an example of arithmetic expansion.

z=`expr substr $string $position $length`

Extract substring of $length characters, starting at $position.

Example 12-6. Using expr

```
#!/bin/bash

# Demonstrating some of the uses of 'expr'
# =======================================

echo

# Arithmetic Operators
# ---------- ---------

echo "Arithmetic Operators"
echo
a=`expr 5 + 3`
echo "5 + 3 = $a"

a=`expr $a + 1`
echo
echo "a + 1 = $a"
echo "(incrementing a variable)"

a=`expr 5 % 3`
# modulo
echo
echo "5 mod 3 = $a"

echo
echo

# Logical Operators
# ------- ---------

#  Returns 1 if true, 0 if false,
#+ opposite of normal Bash convention.

echo "Logical Operators"
echo

x=24
y=25
b=`expr $x = $y`         # Test equality.
echo "b = $b"            # 0  ( $x -ne $y )
echo

a=3
b=`expr $a \> 10`
echo 'b=`expr $a \> 10`, therefore...'
echo "If a > 10, b = 0 (false)"
echo "b = $b"            # 0  ( 3 ! -gt 10 )
```

```
echo

b=`expr $a \< 10`
echo "If a < 10, b = 1 (true)"
echo "b = $b"             # 1  ( 3 -lt 10 )
echo
# Note escaping of operators.

b=`expr $a \<= 3`
echo "If a <= 3, b = 1 (true)"
echo "b = $b"             # 1  ( 3 -le 3 )
# There is also a "\>=" operator (greater than or equal to).


echo
echo

# Comparison Operators
# ---------- ---------

echo "Comparison Operators"
echo
a=zipper
echo "a is $a"
if [ `expr $a = snap` ]
# Force re-evaluation of variable 'a'
then
   echo "a is not zipper"
fi

echo
echo



# String Operators
# ------ ---------

echo "String Operators"
echo

a=1234zipper43231
echo "The string being operated upon is \"$a\"."

# length: length of string
b=`expr length $a`
echo "Length of \"$a\" is $b."

# index: position of first character in substring
#        that matches a character in string
b=`expr index $a 23`
echo "Numerical position of first \"2\" in \"$a\" is \"$b\"."

# substr: extract substring, starting position & length specified
b=`expr substr $a 2 6`
echo "Substring of \"$a\", starting at position 2,\
and 6 chars long is \"$b\"."


#  The default behavior of the 'match' operations is to
```

```
#+ search for the specified match at the ***beginning*** of the string.
#
#       uses Regular Expressions
b=`expr match "$a" '[0-9]*'`              #  Numerical count.
echo Number of digits at the beginning of \"$a\" is $b.
b=`expr match "$a" '\([0-9]*\)'`          #  Note that escaped parentheses
#                  ==      ==              + trigger substring match.
echo "The digits at the beginning of \"$a\" are \"$b\"."

echo

exit 0
```

The : operator can substitute for **match**. For example, **b=`expr $a : [0-9]*`** is the exact equivalent of **b=`expr match $a [0-9]*`** in the above listing.

```
#!/bin/bash

echo
echo "String operations using \"expr \$string : \" construct"
echo "==================================================="
echo

a=1234zipper5FLIPPER43231

echo "The string being operated upon is \"`expr "$a" : '\(.*\)'`\"."
#     Escaped parentheses grouping operator.           ==  ==

#       ***************************
#+          Escaped parentheses
#+           match a substring
#       ***************************


#  If no escaped parentheses...
#+ then 'expr' converts the string operand to an integer.

echo "Length of \"$a\" is `expr "$a" : '.*'`."   # Length of string

echo "Number of digits at the beginning of \"$a\" is `expr "$a" : '[0-9]*'`."

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

echo

echo "The digits at the beginning of \"$a\" are `expr "$a" : '\([0-9]*\)'`."
#                                                       ==      ==
echo "The first 7 characters of \"$a\" are `expr "$a" : '\(.......\)'`."
#         =====                                          ==       ==
# Again, escaped parentheses force a substring match.
#
echo "The last 7 characters of \"$a\" are `expr "$a" : '.*\(.......\)'`."
#         ====                  end of string operator  ^^
#  (actually means skip over one or more of any characters until specified
#+  substring)

echo
```

```
exit 0
```

The above example illustrates how **expr** uses the *escaped parentheses* -- \(... \) -- grouping operator in tandem with regular expression parsing to match a substring.

Perl and sed have far superior string parsing facilities. A short **Perl** or **sed** "subroutine" within a script (see Section 34.2) is an attractive alternative to using **expr**.

See Section 9.2 for more on string operations.

12.1. Basic Commands

The first commands a novice learns

ls

The basic file "list" command. It is all too easy to underestimate the power of this humble command. For example, using the `-R`, recursive option, **ls** provides a tree-like listing of a directory structure. Other interesting options are `-S`, sort listing by file size, `-t`, sort by file modification time, and `-i`, show file inodes (see Example 12-3).

Example 12-1. Using ls to create a table of contents for burning a CDR disk

```
#!/bin/bash

SPEED=2          # May use higher speed if your hardware supports it.
IMAGEFILE=cdimage.iso
CONTENTSFILE=contents
DEFAULTDIR=/opt  # Make sure this directory exists.

# Script to automate burning a CDR.

# Uses Joerg Schilling's "cdrecord" package.
# (http://www.fokus.gmd.de/nthp/employees/schilling/cdrecord.html)

#  If this script invoked as an ordinary user, need to suid cdrecord
#+ (chmod u+s /usr/bin/cdrecord, as root).

if [ -z "$1" ]
then
  IMAGE_DIRECTORY=$DEFAULTDIR
  # Default directory, if not specified on command line.
else
    IMAGE_DIRECTORY=$1
fi

ls -lRF $IMAGE_DIRECTORY > $IMAGE_DIRECTORY/$CONTENTSFILE
# The "l" option gives a "long" file listing.
# The "R" option makes the listing recursive.
# The "F" option marks the file types (directories get a trailing /).
echo "Creating table of contents."

mkisofs -r -o $IMAGFILE $IMAGE_DIRECTORY
echo "Creating ISO9660 file system image ($IMAGEFILE)."

cdrecord -v -isosize speed=$SPEED dev=0,0 $IMAGEFILE
echo "Burning the disk."
echo "Please be patient, this will take a while."

exit 0
```

cat, **tac**

cat, an acronym for *concatenate*, lists a file to `stdout`. When combined with redirection (> or >>), it is commonly used to concatenate files.

```
cat filename cat file.1 file.2 file.3 > file.123
```

The `-n` option to **cat** inserts consecutive numbers before all lines of the target file(s). The `-b` option numbers only the non-blank lines. The `-v` option echoes nonprintable characters, using ^ notation. The `-s` option squeezes multiple consecutive blank lines into a single blank line.

See also Example 12-21 and Example 12-17.

tac, is the inverse of *cat*, listing a file backwards from its end.

rev

reverses each line of a file, and outputs to `stdout`. This is not the same effect as **tac**, as it preserves the order of the lines, but flips each one around.

```
bash$ cat file1.txt
This is line 1.
 This is line 2.

bash$ tac file1.txt
This is line 2.
 This is line 1.

bash$ rev file1.txt
.1 enil si sihT
 .2 enil si sihT
```

cp

This is the file copy command. **`cp file1 file2`** copies `file1` to `file2`, overwriting `file2` if it already exists (see Example 12-5).

Particularly useful are the `-a` archive flag (for copying an entire directory tree) and the `-r` and `-R` recursive flags.

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 9-15 and Example A-3.

When used in a non-interactive script, **mv** takes the `-f` (*force*) option to bypass user input.

When a directory is moved to a preexisting directory, it becomes a subdirectory of the destination directory.

```
bash$ mv source_directory target_directory

bash$ ls -lF target_directory
total 1
 drwxrwxr-x    2 bozo  bozo      1024 May 28 19:20 source_directory/
```

rm

Delete (remove) a file or files. The `-f` option forces removal of even readonly files, and is useful for bypassing user input in a script.

When used with the recursive flag `-r`, this command removes files all the way down the directory tree.

rmdir

Remove directory. The directory must be empty of all files, including invisible "dotfiles", [1] for this command to succeed.

mkdir

Make directory, creates a new directory. **`mkdir -p project/programs/December`** creates the named directory. The *`-p`* option automatically creates any necessary parent directories.

chmod

Changes the attributes of an existing file (see Example 11-9).

```
chmod +x filename
# Makes "filename" executable for all users.

chmod u+s filename
# Sets "suid" bit on "filename" permissions.
# An ordinary user may execute "filename" with same privileges as the file's owner.
# (This does not apply to shell scripts.)
```

```
chmod 644 filename
# Makes "filename" readable/writable to owner, readable to
# others
# (octal mode).
```

```
chmod 1777 directory-name
# Gives everyone read, write, and execute permission in directory,
# however also sets the "sticky bit".
# This means that only the owner of the directory,
# owner of the file, and, of course, root
# can delete any particular file in that directory.
```

chattr

Change file attributes. This has the same effect as **chmod** above, but with a different invocation syntax, and it works only on an *ext2* filesystem.

ln

Creates links to pre-existings files. Most often used with the `-s`, symbolic or "soft" link flag. This permits referencing the linked file by more than one name and is a superior alternative to aliasing (see Example 5-6).

`ln -s oldfile newfile` links the previously existing `oldfile` to the newly created link, `newfile`.

Notes

[1] These are files whose names begin with a dot, such as `~/.Xdefaults`. Such filenames do not show up in a normal **ls** listing, and they cannot be deleted by an accidental **rm -rf ***. Dotfiles are generally used as setup and configuration files in a user's home directory.

Chapter 12. External Filters, Programs and Commands

Table of Contents

Standard UNIX commands make shell scripts more versatile. The power of scripts comes from coupling system commands and shell directives with simple programming constructs.

11.1. Job Control Commands

Certain of the following job control commands take a "job identifier" as an argument. See the table at end of the chapter.

jobs

Lists the jobs running in the background, giving the job number. Not as useful as **ps**.

It is all too easy to confuse *jobs* and *processes*. Certain builtins, such as **kill**, **disown**, and **wait** accept either a job number or a process number as an argument. The **fg**, **bg** and **jobs** commands accept only a job number.

```
bash$ sleep 100 &
[1] 1384

bash $ jobs
[1]+  Running                 sleep 100 &
```

"1" is the job number (jobs are maintained by the current shell), and "1384" is the process number (processes are maintained by the system). To kill this job/process, either a **kill %1** or a **kill 1384** works.

Thanks, S.C.

disown

Remove job(s) from the shell's table of active jobs.

fg, **bg**

The **fg** command switches a job running in the background into the foreground. The **bg** command restarts a suspended job, and runs it in the background. If no job number is specified, then the **fg** or **bg** command acts upon the currently running job.

wait

Stop script execution until all jobs running in background have terminated, or until the job number or process id specified as an option terminates. Returns the exit status of waited-for command.

You may use the **wait** command to prevent a script from exiting before a background job finishes executing (this would create a dreaded orphan process).

Example 11-19. Waiting for a process to finish before proceeding

```
#!/bin/bash

ROOT_UID=0   # Only users with $UID 0 have root privileges.
E_NOTROOT=65
E_NOPARAMS=66

if [ "$UID" -ne "$ROOT_UID" ]
then
  echo "Must be root to run this script."
  # "Run along kid, it's past your bedtime."
  exit $E_NOTROOT
fi

if [ -z "$1" ]
then
  echo "Usage: `basename $0` find-string"
  exit $E_NOPARAMS
fi


echo "Updating 'locate' database..."
echo "This may take a while."
updatedb /usr &     # Must be run as root.

wait
# Don't run the rest of the script until 'updatedb' finished.
# You want the the database updated before looking up the file name.

locate $1

# Without the wait command, in the worse case scenario,
# the script would exit while 'updatedb' was still running,
# leaving it as an orphan process.

exit 0
```

Optionally, **wait** can take a job identifier as an argument, for example, **wait%1** or **wait $PPID**. See the job id table.

Within a script, running a command in the background with an ampersand (&) may cause the script to hang until **ENTER** is hit. This seems to occur with commands that write to `stdout`. It can be a major annoyance.

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

ls -l &
echo "Done."
```

```
bash$ ./test.sh
Done.
 [bozo@localhost test-scripts]$ total 1
 -rwxr-xr-x    1 bozo     bozo           34 Oct 11 15:09 test.sh
 _
```

Placing a **wait** after the background command seems to remedy this.

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

ls -l &
echo "Done."
wait
```

```
bash$ ./test.sh
Done.
 [bozo@localhost test-scripts]$ total 1
 -rwxr-xr-x    1 bozo     bozo           34 Oct 11 15:09 test.sh
```

Redirecting the output of the command to a file or even to `/dev/null` also takes care of this problem.

suspend

This has a similar effect to **Control-Z**, but it suspends the shell (the shell's parent process should resume it at an appropriate time).

logout

Exit a login shell, optionally specifying an exit status.

times

Gives statistics on the system time used in executing commands, in the following form:

```
0m0.020s 0m0.020s
```

This capability is of very limited value, since it is uncommon to profile and benchmark shell scripts.

kill

Forcibly terminate a process by sending it an appropriate *terminate* signal (see Example 13-4).

Example 11-20. A script that kills itself

```
#!/bin/bash
# self-destruct.sh

kill $$  # Script kills its own process here.
         # Recall that "$$" is the script's PID.

echo "This line will not echo."
# Instead, the shell sends a "Terminated" message to stdout.

exit 0
```

`kill -l` lists all the signals. A **`kill -9`** is a "sure kill", which will usually terminate a process that stubbornly refuses to die with a plain **kill**. Sometimes, a **`kill -15`** works. A "zombie process", that is, a process whose parent has terminated, cannot be killed (you can't kill something that is already dead), but **init** will usually clean it up sooner or later.

command

The **command COMMAND** directive disables aliases and functions for the command "COMMAND".

This is one of three shell directives that effect script command processing. The others are builtin and enable.

builtin

Invoking **builtin BUILTIN_COMMAND** runs the command "BUILTIN_COMMAND" as a shell builtin, temporarily disabling both functions and external system commands with the same

name.

enable

This either enables or disables a shell builtin command. As an example, **enable -n kill** disables the shell builtin kill, so that when Bash subsequently encounters **kill**, it invokes `/bin/kill`.

The `-a` option to **enable** lists all the shell builtins, indicating whether or not they are enabled. The `-f filename` option lets **enable** load a builtin as a shared library (DLL) module from a properly compiled object file. [1].

autoload

This is a port to Bash of the *ksh* autoloader. With **autoload** in place, a function with an "autoload" declaration will load from an external file at its first invocation. [2] This saves system resources.

Note that **autoload** is not a part of the core Bash installation. It needs to be loaded in with **enable -f** (see above).

Table 11-1. 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

Notes

[1] The C source for a number of loadable builtins is typically found in the `/usr/share/doc/bash-?.??/functions` directory.

Note that the `-f` option to **enable** is not portable to all systems.

[2] The same effect as **autoload** can be achieved with typeset -fu.

Chapter 11. 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 a separate process -- or because a particular builtin needs direct access to the shell internals.

> When a command or the shell itself initiates (or *spawns*) a new subprocess to carry out a task, this is called *forking*. This new process is the "child", and the process that *forked* it off is the "parent". While the *child process* is doing its work, the *parent process* is still executing.
>
> 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 by itself a command, but part of a larger command structure. [1]

I/O

echo

prints (to `stdout`) an expression or variable (see Example 5-1).

```
echo Hello
echo $a
```

An **echo** requires the `-e` option to print escaped characters. See Example 6-2.

Normally, each **echo** command prints a terminal newline, but the `-n` option suppresses this.

An **echo** can be used to feed a sequence of commands down a pipe.

```
if echo "$VAR" | grep -q txt   # if [[ $VAR = *txt* ]]
then
  echo "$VAR contains the substring sequence \"txt\""
fi
```

An **echo**, in combination with command substitution can set a variable.

```
a=`echo "HELLO" | tr A-Z a-z`
```

See also Example 12-15, Example 12-2, Example 12-32, and Example 12-33.

Be aware that **echo `command`** deletes any linefeeds that the output of *command* generates.

The $IFS (internal field separator) variable normally contains \n (linefeed) as one of its set of whitespace characters. Bash therefore splits the output of *command* at linefeeds into arguments to **echo**. Then **echo** outputs these arguments, separated by spaces.

```
bash$ ls -l /usr/share/apps/kjezz/sounds
-rw-r--r--    1 root     root         1407 Nov  7  2000 reflect.au
 -rw-r--r--    1 root     root          362 Nov  7  2000 seconds.au



bash$ echo `ls -l /usr/share/apps/kjezz/sounds`
total 40 -rw-r--r-- 1 root root 716 Nov 7 2000 reflect.au -rw-r--r-- 1 root root 362
Nov 7 2000 seconds.au
```

This command is a shell builtin, and not the same as /bin/echo, although its behavior is similar.

```
bash$ type -a echo
echo is a shell builtin
 echo is /bin/echo
```

printf

The **printf**, formatted print, command is an enhanced **echo**. It is a limited variant of the C language printf() library function, and its syntax is somewhat different.

printf *format-string... parameter...*

This is the Bash builtin version of the /bin/printf or /usr/bin/printf command. See the **printf** manpage (of the system command) for in-depth coverage.

Older versions of Bash may not support **printf**.

Example 11-1. printf in action

```
#!/bin/bash
# printf demo

PI=3.14159265358979
DecimalConstant=31373
Message1="Greetings,"
Message2="Earthling."

echo

printf "Pi to 2 decimal places = %1.2f" $PI
echo
printf "Pi to 9 decimal places = %1.9f" $PI  # It even rounds off correctly.

printf "\n"                                  # Prints a line feed,
                                             # equivalent to 'echo'.

printf "Constant = \t%d\n" $DecimalConstant  # Inserts tab (\t)

printf "%s %s \n" $Message1 $Message2

echo

# ===========================================#
# Simulation of C function, 'sprintf'.
# Loading a variable with a formatted string.

echo

Pi12=$(printf "%1.12f" $PI)
echo "Pi to 12 decimal places = $Pi12"

Msg=`printf "%s %s \n" $Message1 $Message2`
echo $Msg; echo $Msg

# As it happens, the 'sprintf' function can now be accessed
# as a loadable module to Bash, but this is not portable.

exit 0
```

Formatting error messages is a useful application of **printf**

```
E_BADDIR=65

var=nonexistent_directory

error()
{
  printf "$@" >&2
  # Formats positional params passed, and sents them to stderr.
  echo
  exit $E_BADDIR
}

cd $var || error $"Can't cd to %s." "$var"

# Thanks, S.C.
```

read

"Reads" the value of a variable from `stdin`, that is, interactively fetches input from the keyboard. The `-a` option lets **read** get array variables (see Example 26-2).

Example 11-2. Variable assignment, using read

```
#!/bin/bash

echo -n "Enter the value of variable 'var1': "
# The -n option to echo suppresses newline.

read var1
# Note no '$' in front of var1, since it is being set.

echo "var1 = $var1"


echo

# A single 'read' statement can set multiple variables.
echo -n "Enter the values of variables 'var2' and 'var3' (separated by a space or
tab): "
read var2 var3
echo "var2 = $var2      var3 = $var3"
# If you input only one value, the other variable(s) will remain unset (null).

exit 0
```

A **read** without an associated variable assigns its input to the dedicated variable $REPLY.

Example 11-3. What happens when read has no variable

```
#!/bin/bash

echo

# ------------------------- #
# First code block.
echo -n "Enter a value: "
read var
echo "\"var\" = "$var""
# Everything as expected here.
# ------------------------- #

echo

echo -n "Enter another value: "
read           #  No variable supplied for 'read', therefore...
               #+ Input to 'read' assigned to default variable, $REPLY.
var="$REPLY"
echo "\"var\" = "$var""
# This is equivalent to the first code block.

echo

exit 0
```

Normally, inputting a **** suppresses a newline during input to a **read**. The `-r` option causes an inputted **** to be interpreted literally.

Example 11-4. Multi-line input to read

```
#!/bin/bash

echo

echo "Enter a string terminated by a \\, then press <ENTER>."
echo "Then, enter a second string, and again press <ENTER>."
read var1     # The "\" suppresses the newline, when reading "var1".
              #     first line \
              #     second line

echo "var1 = $var1"
#     var1 = first line second line

# For each line terminated by a "\",
# you get a prompt on the next line to continue feeding characters into var1.

echo; echo

echo "Enter another string terminated by a \\ , then press <ENTER>."
read -r var2  # The -r option causes the "\" to be read literally.
              #     first line \

echo "var2 = $var2"
#     var2 = first line \
```

```
# Data entry terminates with the first <ENTER>.

echo

exit 0
```

The **read** command has some interesting options that permit echoing a prompt and even reading keystrokes without hitting **ENTER**.

```
# Read a keypress without hitting ENTER.

read -s -n1 -p "Hit a key " keypress
echo; echo "Keypress was "\"$keypress\""."

# -s option means do not echo input.
# -n N option means accept only N characters of input.
# -p option means echo the following prompt before reading input.

# Using these options is tricky, since they need to be in the correct order.
```

The `-t` option to **read** permits timed input (see Example 9-4).

The **read** command may also "read" its variable value from a file redirected to `stdin`. If the file contains more than one line, only the first line is assigned to the variable. If **read** has more than one parameter, then each of these variables gets assigned a successive whitespace-delineated string. Caution!

Example 11-5. Using read with file redirection

```
#!/bin/bash

read var1 <data-file
echo "var1 = $var1"
# var1 set to the entire first line of the input file "data-file"

read var2 var3 <data-file
echo "var2 = $var2   var3 = $var3"
# Note non-intuitive behavior of "read" here.
# 1) Rewinds back to the beginning of input file.
# 2) Each variable is now set to a corresponding string,
#    separated by whitespace, rather than to an entire line of text.
# 3) The final variable gets the remainder of the line.
# 4) If there are more variables to be set than whitespace-terminated strings
#    on the first line of the file, then the excess variables remain empty.

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

# How to resolve the above problem with a loop:
while read line
do
  echo "$line"
done <data-file
# Thanks, Heiner Steven for pointing this out.
```

```
echo "----------------------------------------------"

# Use $IFS (Internal File Separator variable) to split a line of input to
# "read", if you do not want the default to be whitespace.

echo "List of all users:"
OIFS=$IFS; IFS=:       # /etc/passwd uses ":" for field separator.
while read name passwd uid gid fullname ignore
do
  echo "$name ($fullname)"
done </etc/passwd   # I/O redirection.
IFS=$OIFS              # Restore originial $IFS.
# This code snippet also by Heiner Steven.

exit 0
```

Piping output to a **read**, using echo to set variables will fail.

However, piping the output of cat does seem to work.

```
cat file1 file2 |
while read line
do
echo $line
done
```

Filesystem

cd

The familiar **cd** change directory command finds use in scripts where execution of a command requires being in a specified directory.

```
(cd /source/directory && tar cf - . ) | (cd /dest/directory && tar xpvf -)
```

[from the previously cited example by Alan Cox]

The `-P` (physical) option to **cd** causes it to ignore symbolic links.

cd - changes to $OLDPWD, the previous working directory.

pwd

Print Working Directory. This gives the user's (or script's) current directory (see Example 11-6). The effect is identical to reading the value of the builtin variable $PWD.

pushd, **popd**, **dirs**

This command set is a mechanism for bookmarking working directories, a means of moving back and forth through directories in an orderly manner. A pushdown stack is used to keep track of directory names. Options allow various

manipulations of the directory stack.

pushd dir-name pushes the path *dir-name* onto the directory stack and simultaneously changes the current working directory to *dir-name*

popd removes (pops) the top directory path name off the directory stack and simultaneously changes the current working directory to that directory popped from the stack.

dirs lists the contents of the directory stack (compare this with the $DIRSTACK variable). A successful **pushd** or **popd** will automatically invoke **dirs**.

Scripts that require various changes to the current working directory without hard-coding the directory name changes can make good use of these commands. Note that the implicit `$DIRSTACK` array variable, accessible from within a script, holds the contents of the directory stack.

Example 11-6. Changing the current working directory

```
#!/bin/bash

dir1=/usr/local
dir2=/var/spool

pushd $dir1
# Will do an automatic 'dirs' (list directory stack to stdout).
echo "Now in directory `pwd`." # Uses back-quoted 'pwd'.

# Now, do some stuff in directory 'dir1'.
pushd $dir2
echo "Now in directory `pwd`."

# Now, do some stuff in directory 'dir2'.
echo "The top entry in the DIRSTACK array is $DIRSTACK."
popd
echo "Now back in directory `pwd`."

# Now, do some more stuff in directory 'dir1'.
popd
echo "Now back in original working directory `pwd`."

exit 0
```

Variables

let

The **let** command carries out arithmetic operations on variables. In many cases, it functions as a less complex version of expr.

Example 11-7. Letting let do some arithmetic.

```
#!/bin/bash

echo

let a=11          # Same as 'a=11'
let a=a+5         # Equivalent to  let "a = a + 5"
                  # (double quotes and spaces make it more readable)
echo "11 + 5 = $a"

let "a <<= 3"     # Equivalent to  let "a = a << 3"
echo "\"\$a\" (=16) left-shifted 3 places = $a"

let "a /= 4"      # Equivalent to  let "a = a / 4"
echo "128 / 4 = $a"

let "a -= 5"      # Equivalent to  let "a = a - 5"
echo "32 - 5 = $a"

let "a = a * 10"  # Equivalent to  let "a = a * 10"
echo "27 * 10 = $a"

let "a %= 8"      # Equivalent to  let "a = a % 8"
echo "270 modulo 8 = $a  (270 / 8 = 33, remainder $a)"

echo

exit 0
```

eval

`eval arg1 [arg2] ... [argN]`

Translates into commands the arguments in a list (useful for code generation within a script).

Example 11-8. Showing the effect of eval

```
#!/bin/bash

y=`eval ls -l`  # Similar to y=`ls -l`
echo $y         # but linefeeds removed because "echoed" variable is unquoted.
echo
echo "$y"       # Linefeeds preserved when variable is quoted.

echo; echo

y=`eval df`     # Similar to y=`df`
echo $y         # but linefeeds removed.

#  When LF's not preserved, it may make it easier to parse output,
#+ using utilities such as "awk".

exit 0
```

Example 11-9. Forcing a log-off

```
#!/bin/bash

y=`eval ps ax | sed -n '/ppp/p' | awk '{ print $1 }'`
# Finding the process number of 'ppp'.

kill -9 $y   # Killing it

# Above lines may be replaced by
#  kill -9 `ps ax | awk '/ppp/ { print $1 }'


chmod 666 /dev/ttyS3
# Doing a SIGKILL on ppp changes the permissions
# on the serial port. Restore them to previous state.

rm /var/lock/LCK..ttyS3   # Remove the serial port lock file.

exit 0
```

Example 11-10. A version of "rot13"

```
#!/bin/bash
# A version of "rot13" using 'eval'.
# Compare to "rot13.sh" example.

setvar_rot_13()             # "rot13" scrambling
{
  local varname=$1 varvalue=$2
  eval $varname='$(echo "$varvalue" | tr a-z n-za-m)'
}


setvar_rot_13 var "foobar"   # Run "foobar" through rot13.
echo $var                    # sbbone

echo $var | tr a-z n-za-m    # foobar
                             # Back to original variable.

# This example by Stephane Chazelas.

exit 0
```

The **eval** command can be risky, and normally should be avoided when there exists a reasonable alternative. An **eval $COMMANDS** executes the contents of *COMMANDS*, which may contain such unpleasant surprises as **rm -rf ***. Running an **eval** on unfamiliar code written by persons unknown is living dangerously.

set

The **set** command changes the value of internal script variables. One use for this is to toggle option flags which help determine the behavior of the script. Another application for it is to reset the positional parameters that a script sees as the result of a command (**set `command`**). The script can then parse the fields of the command output.

Example 11-11. Using set with positional parameters

```
#!/bin/bash

# script "set-test"

# Invoke this script with three command line parameters,
# for example, "./set-test one two three".

echo
echo "Positional parameters before  set \`uname -a\` :"
echo "Command-line argument #1 = $1"
echo "Command-line argument #2 = $2"
echo "Command-line argument #3 = $3"

echo

set `uname -a` # Sets the positional parameters to the output
               # of the command `uname -a`

echo "Positional parameters after  set \`uname -a\` :"
# $1, $2, $3, etc. reinitialized to result of `uname -a`
echo "Field #1 of 'uname -a' = $1"
echo "Field #2 of 'uname -a' = $2"
echo "Field #3 of 'uname -a' = $3"
echo

exit 0
```

Invoking **set** without any options or arguments simply lists all the environmental and other variables that have been initialized.

```
bash$ set
AUTHORCOPY=/home/bozo/posts
 BASH=/bin/bash
 BASH_VERSION=$'2.05.8(1)-release'
 ...
 XAUTHORITY=/home/bozo/.Xauthority
 _=/etc/bashrc
 variable22=abc
 variable23=xzy
```

Using **set** with the `--` option explicitly assigns the contents of a variable to the positional parameters. When no variable follows the `--`, it *unsets* the positional parameters.

Example 11-12. Reassigning the positional parameters

```
#!/bin/bash

variable="one two three four five"

set -- $variable
# Sets positional parameters to the contents of "$variable".

first_param=$1
second_param=$2
shift; shift        # Shift past first two positional params.
remaining_params="$*"

echo
echo "first parameter = $first_param"             # one
echo "second parameter = $second_param"           # two
echo "remaining parameters = $remaining_params"   # three four five

echo; echo

# Again.
set -- $variable
first_param=$1
second_param=$2
echo "first parameter = $first_param"             # one
echo "second parameter = $second_param"           # two

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

set --
# Unsets positional parameters if no variable specified.

first_param=$1
second_param=$2
echo "first parameter = $first_param"             # (null value)
echo "second parameter = $second_param"           # (null value)

exit 0
```

See also Example 10-2 and Example 12-39.

unset

The **unset** command deletes a shell variable, effectively setting it to *null*. Note that this command does not affect positional parameters.

```
bash$ unset PATH

bash$ echo $PATH

bash$
```

Example 11-13. "unsetting" a variable

```
#!/bin/bash
# unset.sh: Unsetting a variable.

variable=hello                        # Initialized.
echo "variable = $variable"

unset variable                        # Unset.
                                      # Same effect as   variable=
echo "(unset) variable = $variable"   # $variable is null.

exit 0
```

export

The **export** command makes available variables to all child processes of the running script or shell. Unfortunately, there is no way to **export** variables back to the parent process, to the process that called or invoked the script or shell. One important use of **export** command is in startup files, to initialize and make accessible environmental variables to subsequent user processes.

Example 11-14. Using export to pass a variable to an embedded awk script

```
#!/bin/bash

# Yet another version of the "column totaler" script (col-totaler.sh)
# that adds up a specified column (of numbers) in the target file.
# This uses the environment to pass a script variable to 'awk'.

ARGS=2
E_WRONGARGS=65

if [ $# -ne "$ARGS" ] # Check for proper no. of command line args.
then
   echo "Usage: `basename $0` filename column-number"
   exit $E_WRONGARGS
fi

filename=$1
column_number=$2

#===== Same as original script, up to this point =====#

export column_number
# Export column number to environment, so it's available for retrieval.


# Begin awk script.
# ------------------------------------------------
awk '{ total += $ENVIRON["column_number"]
}
END { print total }' $filename
# ------------------------------------------------
# End awk script.


# Thanks, Stephane Chazelas.
```

```
exit 0
```

It is possible to initialize and export variables in the same operation, as in **export var1=xxx**.

declare, **typeset**

The declare and typeset commands specify and/or restrict properties of variables.

readonly

Same as declare -r, sets a variable as read-only, or, in effect, as a constant. Attempts to change the variable fail with an error message. This is the shell analog of the C language **const** type qualifier.

getopts

This powerful tool parses command-line arguments passed to the script. This is the Bash analog of the getopt external command and the **getopt** library function familiar to C programmers. It permits passing and concatenating multiple options [2] and associated arguments to a script (for example **`scriptname -abc -e /usr/local`**).

The **getopts** construct uses two implicit variables. `$OPTIND` is the argument pointer (*OPTion INDex*) and `$OPTARG` (*OPTion ARGument*) the (optional) argument attached to an option. A colon following the option name in the declaration tags that option as having an associated argument.

A **getopts** construct usually comes packaged in a while loop, which processes the options and arguments one at a time, then decrements the implicit `$OPTIND` variable to step to the next.

1. The arguments passed from the command line to the script must be preceded by a minus (`-`) or a plus (`+`). It is the prefixed `-` or `+` that lets **getopts** recognize command-line arguments as *options*. In fact, **getopts** will not process arguments without the prefixed `-` or `+`, and will terminate option processing at the first argument encountered lacking them.
2. The **getopts** template differs slightly from the standard **while** loop, in that it lacks condition brackets.
3. The **getopts** construct replaces the obsolete and less powerful getopt external command.

```
while getopts ":abcde:fg" Option
# Initial declaration.
# a, b, c, d, e, f, and g are the options (flags) expected.
# The : after option 'e' shows it will have an argument passed with it.
do
  case $Option in
    a ) # Do something with variable 'a'.
    b ) # Do something with variable 'b'.
    ...
    e)  # Do something with 'e', and also with $OPTARG,
        # which is the associated argument passed with option 'e'.
    ...
    g ) # Do something with variable 'g'.
  esac
done
shift $(($OPTIND - 1))
# Move argument pointer to next.

# All this is not nearly as complicated as it looks <grin>.
```

Example 11-15. Using getopts to read the options/arguments passed to a script

```
#!/bin/bash

# 'getopts' processes command line arguments to script.
# The arguments are parsed as "options" (flags) and associated arguments.

# Try invoking this script with
# 'scriptname -mn'
# 'scriptname -oq qOption' (qOption can be some arbitrary string.)
# 'scriptname -qXXX -r'
#
# 'scriptname -qr'    - Unexpected result, takes "r" as the argument to option "q"
# 'scriptname -q -r'  - Unexpected result, same as above
#  If an option expects an argument ("flag:"), then it will grab
#  whatever is next on the command line.

NO_ARGS=0
E_OPTERROR=65

if [ $# -eq "$NO_ARGS" ]  # Script invoked with no command-line args?
then
  echo "Usage: `basename $0` options (-mnopqrs)"
  exit $E_OPTERROR        # Exit and explain usage, if no argument(s) given.
fi
# Usage: scriptname -options
# Note: dash (-) necessary


while getopts ":mnopq:rs" Option
do
  case $Option in
    m     ) echo "Scenario #1: option -m-";;
    n | o ) echo "Scenario #2: option -$Option-";;
    p     ) echo "Scenario #3: option -p-";;
    q     ) echo "Scenario #4: option -q-, with argument \"$OPTARG\"";;
    # Note that option 'q' must have an associated argument,
    # otherwise it falls through to the default.
    r | s ) echo "Scenario #5: option -$Option-"';;
    *     ) echo "Unimplemented option chosen.";;   # DEFAULT
  esac
done

shift $(($OPTIND - 1))
# Decrements the argument pointer so it points to next argument.

exit 0
```

Script Behavior

source, . (dot command)

This command, when invoked from the command line, executes a script. Within a script, a **source file-name** loads the file `file-name`. This is the shell scripting equivalent of a C/C++ **#include** directive. It is useful in situations when multiple scripts use a common data file or function library.

Example 11-16. "Including" a data file

```
#!/bin/bash

. data-file    # Load a data file.
# Same effect as "source data-file", but more portable.

#  The file "data-file" must be present in current working directory,
#+ since it is referred to by its 'basename'.

# Now, reference some data from that file.

echo "variable1 (from data-file) = $variable1"
echo "variable3 (from data-file) = $variable3"

let "sum = $variable2 + $variable4"
echo "Sum of variable2 + variable4 (from data-file) = $sum"
echo "message1 (from data-file) is \"$message1\""
# Note:                           escaped quotes

print_message This is the message-print function in the data-file.


exit 0
```

File `data-file` for Example 11-16, above. Must be present in same directory.

```
# This is a data file loaded by a script.
# Files of this type may contain variables, functions, etc.
# It may be loaded with a 'source' or '.' command by a shell script.

# Let's initialize some variables.

variable1=22
variable2=474
variable3=5
variable4=97

message1="Hello, how are you?"
message2="Enough for now. Goodbye."

print_message ()
{
# Echoes any message passed to it.

  if [ -z "$1" ]
  then
    return 1
    # Error, if argument missing.
  fi
```

```
  echo

  until [ -z "$1" ]
  do
    # Step through arguments passed to function.
    echo -n "$1"
    # Echo args one at a time, suppressing line feeds.
    echo -n " "
    # Insert spaces between words.
    shift
    # Next one.
  done

  echo

  return 0
}
```

exit

Unconditionally terminates a script. The **exit** command may optionally take an integer argument, which is returned to the shell as the exit status of the script. It is a good practice to end all but the simplest scripts with an **`exit 0`**, indicating a successful run.

If a script terminates with an **exit** lacking an argument, the exit status of the script is the exit status of the last command executed in the script, not counting the **exit**.

exec

This shell builtin replaces the current process with a specified command. Normally, when the shell encounters a command, it forks off a child process to actually execute the command. Using the **exec** builtin, the shell does not fork, and the command exec'ed replaces the shell. When used in a script, therefore, it forces an exit from the script when the **exec**'ed command terminates. For this reason, if an **exec** appears in a script, it would probably be the final command.

Example 11-17. Effects of exec

```
#!/bin/bash

exec echo "Exiting \"$0\"."   # Exit from script here.

# ---------------------------------
# The following lines never execute.

echo "This echo will never echo."

exit 99                      #  This script will not exit here.
                             #  Check exit value after script terminates
                             #+ with an 'echo $?'.
                             #  It will *not* be 99.
```

Example 11-18. A script that exec's itself

```
#!/bin/bash
# self-exec.sh

echo

echo "This line appears ONCE in the script, yet it keeps echoing."
echo "The PID of this instance of the script is still $$."
#     Demonstrates that a subshell is not forked off.

echo "==================== Hit Ctl-C to exit ===================="

sleep 1

exec $0   #  Spawns another instance of this same script
          #+ that replaces the previous one.

echo "This line will never echo!"  # Why not?

exit 0
```

An **exec** also serves to reassign file descriptors. **exec <zzz-file** replaces `stdin` with the file `zzz-file` (see Example 16-1).

The `-exec` option to find is *not* the same as the **exec** shell builtin.

shopt

This command permits changing shell options on the fly (see Example 24-1 and Example 24-2). It often appears in the Bash startup files, but also has its uses in scripts. Needs version 2 or later of Bash.

```
shopt -s cdspell
# Allows minor misspelling directory names with 'cd'
command.
```

Commands

true

A command that returns a successful (zero) exit status, but does nothing else.

```
# Endless loop
while true   # alias for ":"
do
   operation-1
   operation-2
   ...
   operation-n
   # Need a way to break out of loop.
done
```

false

A command that returns an unsuccessful exit status, but does nothing else.

```
# Null loop
while false
do
   # The following code will not execute.
   operation-1
   operation-2
   ...
   operation-n
   # Nothing happens!
done
```

type [cmd]

Similar to the which external command, **type cmd** gives the full pathname to "cmd". Unlike **which**, **type** is a Bash builtin. The useful `-a` option to **type** identifies *`keywords`* and *`builtins`*, and also locates system commands with identical names.

```
bash$ type '['
[ is a shell builtin
bash$ type -a '['
[ is a shell builtin
 [ is /usr/bin/[
```

hash [cmds]

Record the path name of specified commands (in the shell hash table), so the shell or script will not need to search the `$PATH` on subsequent calls to those commands. When **hash** is called with no arguments, it simply lists the commands that have been hashed. The `-r` option resets the hash table.

help

help COMMAND looks up a short usage summary of the shell builtin COMMAND. This is the counterpart to whatis, but for builtins.

```
bash$ help exit
exit: exit [n]
    Exit the shell with a status of N.  If N is omitted, the exit status
    is that of the last command executed.
```

Notes

[1] An exception to this is the time command, listed in the official Bash documentation as a keyword.

[2] A option is an argument that acts as a flag, switching script behaviors on or off. The argument associated with a particular option indicates the behavior that the option (flag) switches on or off.

10.4. Testing and Branching

The **case** and **select** constructs are technically not loops, since they do not iterate the execution of a code block. Like loops, however, they direct program flow according to conditions at the top or bottom of the block.

Controlling program flow in a code block

case (in) / esac

The **case** construct is the shell equivalent of **switch** in C/C++. It permits branching to one of a number of code blocks, depending on condition tests. It serves as a kind of shorthand for multiple if/then/else statements and is an appropriate tool for creating menus.

case "$*variable*" in

"$*condition1*")
command...
;;

"$*condition2*")
command...
;;

esac

- Quoting the variables is not mandatory, since word splitting does not take place.
- Each test line ends with a right paren).
- Each condition block ends with a *double* semicolon ;;.
- The entire **case** block terminates with an **esac** (*case* spelled backwards).

Example 10-23. Using case

```
#!/bin/bash

echo; echo "Hit a key, then hit return."
read Keypress

case "$Keypress" in
  [a-z]   ) echo "Lowercase letter";;
  [A-Z]   ) echo "Uppercase letter";;
  [0-9]   ) echo "Digit";;
  *       ) echo "Punctuation, whitespace, or other";;
esac  # Allows ranges of characters in [square brackets].
```

```
# Exercise:
# --------
# As the script stands, # it accepts a single keystroke, then terminates.
# Change the script so it accepts continuous input,
# reports on each keystroke, and terminates only when "X" is hit.
# Hint: enclose everything in a "while" loop.

exit 0
```

Example 10-24. Creating menus using case

```
#!/bin/bash

# Crude address database

clear # Clear the screen.

echo "          Contact List"
echo "          ------- ----"
echo "Choose one of the following persons:"
echo
echo "[E]vans, Roland"
echo "[J]ones, Mildred"
echo "[S]mith, Julie"
echo "[Z]ane, Morris"
echo

read person

case "$person" in
# Note variable is quoted.

  "E" | "e" )
  # Accept upper or lowercase input.
  echo
  echo "Roland Evans"
  echo "4321 Floppy Dr."
  echo "Hardscrabble, CO 80753"
  echo "(303) 734-9874"
  echo "(303) 734-9892 fax"
  echo "revans@zzy.net"
  echo "Business partner & old friend"
  ;;
# Note double semicolon to terminate
# each option.

  "J" | "j" )
  echo
  echo "Mildred Jones"
  echo "249 E. 7th St., Apt. 19"
  echo "New York, NY 10009"
```

```
  echo "(212) 533-2814"
  echo "(212) 533-9972 fax"
  echo "milliej@loisaida.com"
  echo "Girlfriend"
  echo "Birthday: Feb. 11"
  ;;

# Add info for Smith & Zane later.

         * )
   # Default option.
   # Empty input (hitting RETURN) fits here, too.
   echo
   echo "Not yet in database."
  ;;

esac

echo

#  Exercise:
#  --------
#  Change the script so it accepts continuous input,
#+ instead of terminating after displaying just one address.

exit 0
```

An exceptionally clever use of **case** involves testing for command-line parameters.

```
#! /bin/bash

case "$1" in
"") echo "Usage: ${0##*/} <filename>"; exit 65;;  # No command-line parameters,
                                                  # or first parameter empty.
# Note that ${0##*/} is ${var##pattern} param substitution. Net result is $0.

-*) FILENAME=./$1;;   # If filename passed as argument ($1) starts with a dash,
                      # replace it with ./$1
                      # so further commands don't interpret it as an option.

* ) FILENAME=$1;;     # Otherwise, $1.
esac
```

Example 10-25. Using command substitution to generate the case variable

```
#!/bin/bash
# Using command substitution to generate a "case" variable.

case $( arch ) in   # "arch" returns machine architecture.
i386 ) echo "80386-based machine";;
i486 ) echo "80486-based machine";;
i586 ) echo "Pentium-based machine";;
i686 ) echo "Pentium2+-based machine";;
*    ) echo "Other type of machine";;
esac

exit 0
```

A **case** construct can filter strings for globbing patterns.

Example 10-26. Simple string matching

```
#!/bin/bash
# match-string.sh: simple string matching

match_string ()
{
  MATCH=0
  NOMATCH=90
  PARAMS=2     # Function requires 2 arguments.
  BAD_PARAMS=91

  [ $# -eq $PARAMS ] || return $BAD_PARAMS

  case "$1" in
  "$2") return $MATCH;;
  *   ) return $NOMATCH;;
  esac

}


a=one
b=two
c=three
d=two


match_string $a     # wrong number of parameters
echo $?             # 91

match_string $a $b  # no match
echo $?             # 90

match_string $b $d  # match
echo $?             # 0
```

```
exit 0
```

Example 10-27. Checking for alphabetic input

```
#!/bin/bash
# Using "case" structure to filter a string.

SUCCESS=0
FAILURE=-1

isalpha ()  # Tests whether *first character* of input string is alphabetic.
{
if [ -z "$1" ]               # No argument passed?
then
  return $FAILURE
fi

case "$1" in
[a-zA-Z]*) return $SUCCESS;;  # Begins with a letter?
*        ) return $FAILURE;;
esac
}             # Compare this with "isalpha ()" function in C.


isalpha2 ()   # Tests whether *entire string* is alphabetic.
{
  [ $# -eq 1 ] || return $FAILURE

  case $1 in
  *[!a-zA-Z]*|"") return $FAILURE;;
               *) return $SUCCESS;;
  esac
}



check_var ()  # Front-end to isalpha().
{
if isalpha "$@"
then
  echo "$* = alpha"
else
  echo "$* = non-alpha"  # Also "non-alpha" if no argument passed.
fi
}

a=23skidoo
b=H3llo
c=-What?
d=`echo $b`   # Command substitution.
```

```
check_var $a
check_var $b
check_var $c
check_var $d
check_var     # No argument passed, so what happens?


# Script improved by S.C.

exit 0
```

select

The **select** construct, adopted from the Korn Shell, is yet another tool for building menus.

select *variable* [in *list*]
do
 command...
 break
done

This prompts the user to enter one of the choices presented in the variable list. Note that **select** uses the PS3 prompt (#?) by default, but that this may be changed.

Example 10-28. Creating menus using select

```
#!/bin/bash

PS3='Choose your favorite vegetable: ' # Sets the prompt string.

echo

select vegetable in "beans" "carrots" "potatoes" "onions" "rutabagas"
do
  echo
  echo "Your favorite veggie is $vegetable."
  echo "Yuck!"
  echo
  break  # if no 'break' here, keeps looping forever.
done

exit 0
```

If **in** *list* is omitted, then **select** uses the list of command line arguments ($@) passed to the script or to the function in which the **select** construct is embedded.

Compare this to the behavior of a

for *variable* [in *list*]

construct with the **in** *list* omitted.

Example 10-29. Creating menus using select in a function

```
#!/bin/bash

PS3='Choose your favorite vegetable: '

echo

choice_of()
{
select vegetable
# [in list] omitted, so 'select' uses arguments passed to function.
do
  echo
  echo "Your favorite veggie is $vegetable."
  echo "Yuck!"
  echo
  break
done
}

choice_of beans rice carrots radishes tomatoes spinach
#         $1    $2   $3      $4       $5       $6
#         passed to choice_of() function

exit 0
```

See also Example 35-3.

10.3. Loop Control

Commands Affecting Loop Behavior

break, continue

The **break** and **continue** loop control commands [1] correspond exactly to their counterparts in other programming languages. The **break** command terminates the loop (breaks out of it), while **continue** causes a jump to the next iteration of the loop, skipping all the remaining commands in that particular loop cycle.

Example 10-20. Effects of break and continue in a loop

```
#!/bin/bash

LIMIT=19  # Upper limit

echo
echo "Printing Numbers 1 through 20 (but not 3 and 11)."

a=0

while [ $a -le "$LIMIT" ]
do
 a=$(($a+1))

 if [ "$a" -eq 3 ] || [ "$a" -eq 11 ]  # Excludes 3 and 11
 then
   continue  # Skip rest of this particular loop iteration.
 fi

 echo -n "$a "
done

# Exercise:
# Why does loop print up to 20?

echo; echo

echo Printing Numbers 1 through 20, but something happens after 2.

##############################################################
```

```
# Same loop, but substituting 'break' for 'continue'.

a=0

while [ "$a" -le "$LIMIT" ]
do
 a=$(($a+1))

 if [ "$a" -gt 2 ]
 then
   break  # Skip entire rest of loop.
 fi

 echo -n "$a "
done

echo; echo; echo

exit 0
```

The **break** command may optionally take a parameter. A plain **break** terminates only the innermost loop in which it is embedded, but a **break N** breaks out of *N* levels of loop.

Example 10-21. Breaking out of multiple loop levels

```
#!/bin/bash
# break-levels.sh: Breaking out of loops.

# "break N" breaks out of N level loops.

for outerloop in 1 2 3 4 5
do
  echo -n "Group $outerloop:   "

  for innerloop in 1 2 3 4 5
  do
    echo -n "$innerloop "

    if [ "$innerloop" -eq 3 ]
    then
      break  # Try   break 2   to see what happens.
             # ("Breaks" out of both inner and outer loops.)
    fi
  done

  echo
done
```

```
echo

exit 0
```

The **continue** command, similar to **break**, optionally takes a parameter. A plain **continue** cuts short the current iteration within its loop and begins the next. A **continue N** terminates all remaining iterations at its loop level and continues with the next iteration at the loop N levels above.

Example 10-22. Continuing at a higher loop level

```
#!/bin/bash
# The "continue N" command, continuing at the Nth level loop.

for outer in I II III IV V           # outer loop
do
  echo; echo -n "Group $outer: "

  for inner in 1 2 3 4 5 6 7 8 9 10  # inner loop
  do

    if [ "$inner" -eq 7 ]
    then
      continue 2  # Continue at loop on 2nd level, that is "outer loop".
                  # Replace above line with a simple "continue"
                  # to see normal loop behavior.
    fi

    echo -n "$inner "  # 8 9 10 will never echo.
  done

done

echo; echo

# Exercise:
# Come up with a meaningful use for "continue N" in a script.

exit 0
```

The **continue N** construct is difficult to understand and tricky to use in any meaningful context. It is probably best avoided.

Notes

[1] These are shell builtins, whereas other loop commands, such as while and case, are keywords.

Prev Nested Loops | Home | Up | Next Testing and Branching

10.2. Nested Loops

A nested loop is a loop within a loop, an inner loop within the body of an outer one. What happens 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 may interrupt this process.

Example 10-19. Nested Loop

```
#!/bin/bash
# Nested "for" loops.

outer=1             # Set outer loop counter.

# Beginning of outer loop.
for a in 1 2 3 4 5
do
  echo "Pass $outer in outer loop."
  echo "---------------------"
  inner=1           # Reset inner loop counter.

  # Beginning of inner loop.
  for b in 1 2 3 4 5
  do
    echo "Pass $inner in inner loop."
    let "inner+=1"  # Increment inner loop counter.
  done
  # End of inner loop.

  let "outer+=1"    # Increment outer loop counter.
  echo              # Space between output in pass of outer loop.
done
# End of outer loop.

exit 0
```

See Example 26-4 for an illustration of nested "while" loops, and Example 26-5 to see a "while" loop nested inside an "until" loop.

10.1. Loops

A *loop* is a block of code that iterates (repeats) a list of commands as long as the loop control condition is true.

for loops

for (in)

This is the basic looping construct. It differs significantly from its C counterpart.

for *arg* in [*list*]
do
 command(s)...
done


During each pass through the loop, *arg* takes on the value of each variable in the *list*.

```
for arg in "$var1" "$var2" "$var3" ... "$varN"
# In pass 1 of the loop, $arg = $var1
# In pass 2 of the loop, $arg = $var2
# In pass 3 of the loop, $arg = $var3
# ...
# In pass N of the loop, $arg = $varN

# Arguments in [list] quoted to prevent possible word splitting.
```

The argument *list* may contain wild cards.

If **do** is on same line as **for**, there needs to be a semicolon after list.

for *arg* in [*list*] ; do

Example 10-1. Simple for loops

```
#!/bin/bash
# List the planets.

for planet in Mercury Venus Earth Mars Jupiter Saturn Uranus Neptune Pluto
do
  echo $planet
done

echo

# Entire 'list' enclosed in quotes creates a single variable.
for planet in "Mercury Venus Earth Mars Jupiter Saturn Uranus Neptune Pluto"
do
  echo $planet
done

exit 0
```

Each **[list]** element may contain multiple parameters. This is useful when processing parameters in groups. In such cases, use the **set** command (see Example 11-11) to force parsing of each **[list]** element and assignment of each component to the positional parameters.

Example 10-2. for loop with two parameters in each [list] element

```
#!/bin/bash
# Planets revisited.

# Associate the name of each planet with its distance from the sun.

for planet in "Mercury 36" "Venus 67" "Earth 93"  "Mars 142" "Jupiter 483"
do
  set -- $planet  # Parses variable "planet" and sets positional parameters.
  # the "--" prevents nasty surprises if $planet is null or begins with a dash.

  # May need to save original positional parameters, since they get overwritten.
  # One way of doing this is to use an array,
  #        original_params=("$@")

  echo "$1              $2,000,000 miles from the sun"
  #-------two  tabs---concatenate zeroes onto parameter $2
done

# (Thanks, S.C., for additional clarification.)

exit 0
```

A variable may supply the **[list]** in a **for** loop.

Example 10-3. *Fileinfo:* operating on a file list contained in a variable

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

FILES="/usr/sbin/privatepw
/usr/sbin/pwck
/usr/sbin/go500gw
/usr/bin/fakefile
/sbin/mkreiserfs
/sbin/ypbind"     # List of files you are curious about.
                  # Threw in a dummy file, /usr/bin/fakefile.

echo

for file in $FILES
do

  if [ ! -e "$file" ]       # Check if file exists.
  then
    echo "$file does not exist."; echo
    continue                # On to next.
   fi

  ls -l $file | awk '{ print $9 "         file size: " $5 }'  # Print 2 fields.
  whatis `basename $file`   # File info.
  echo
done

exit 0
```

The **[list]** in a **for** loop may contain filename globbing, that is, using wildcards for filename expansion.

Example 10-4. Operating on files with a for loop

```
#!/bin/bash
# list-glob.sh: Generating [list] in a for-loop using "globbing".

echo

for file in *
do
  ls -l "$file"  # Lists all files in $PWD (current directory).
  # Recall that the wild card character "*" matches everything,
  # however, in "globbing", it doesn't match dot-files.

  # If the pattern matches no file, it is expanded to itself.
  # To prevent this, set the nullglob option
  # (shopt -s nullglob).
  # Thanks, S.C.
done

echo; echo
```

```
for file in [jx]*
do
  rm -f $file    # Removes only files beginning with "j" or "x" in $PWD.
  echo "Removed file \"$file\"".
done

echo

exit 0
```

Omitting the **in [list]** part of a **for** loop causes the loop to operate on $@, the list of arguments given on the command line to the script. A particularly clever illustration of this is Example A-16.

Example 10-5. Missing `in [list]` in a for loop

```
#!/bin/bash

# Invoke both with and without arguments, and see what happens.

for a
do
 echo -n "$a "
done

#  The 'in list' missing, therefore the loop operates on '$@'
#+ (command-line argument list, including whitespace).

echo

exit 0
```

It is possible to use command substitution to generate the **[list]** in a **for** loop. See also Example 12-38, Example 10-10 and Example 12-33.

Example 10-6. Generating the [list] in a for loop with command substitution

```
#!/bin/bash
# A for-loop with [list] generated by command substitution.

NUMBERS="9 7 3 8 37.53"

for number in `echo $NUMBERS`  # for number in 9 7 3 8 37.53
do
  echo -n "$number "
done

echo
exit 0
```

This is a somewhat more complex example of using command substitution to create the [list].

Example 10-7. A grep replacement for binary files

```
#!/bin/bash
# bin-grep.sh: Locates matching strings in a binary file.

# A "grep" replacement for binary files.
# Similar effect to "grep -a"

E_BADARGS=65
E_NOFILE=66

if [ $# -ne 2 ]
then
  echo "Usage: `basename $0` string filename"
  exit $E_BADARGS
fi

if [ ! -f "$2" ]
then
  echo "File \"$2\" does not exist."
  exit $E_NOFILE
fi


for word in $( strings "$2" | grep "$1" )
# The "strings" command lists strings in binary files.
# Output then piped to "grep", which tests for desired string.
do
  echo $word
done

# As S.C. points out, the above for-loop could be replaced with the simpler
#    strings "$2" | grep "$1" | tr -s "$IFS" '[\n*]'


# Try something like  "./bin-grep.sh mem /bin/ls"  to exercise this script.

exit 0
```

More of the same.

Example 10-8. Listing all users on the system

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

PASSWORD_FILE=/etc/passwd
n=1           # User number

for name in $(awk 'BEGIN{FS=":"}{print $1}' < "$PASSWORD_FILE" )
# Field separator = :    ^^^^^^
# Print first field              ^^^^^^^^
# Get input from password file               ^^^^^^^^^^^^^^^^^
do
  echo "USER #$n = $name"
  let "n += 1"
done


# USER #1 = root
# USER #2 = bin
# USER #3 = daemon
# ...
# USER #30 = bozo

exit 0
```

A final example of the [list] resulting from command substitution.

Example 10-9. Checking all the binaries in a directory for authorship

```
#!/bin/bash
# findstring.sh:
# Find a particular string in binaries in a specified directory.

directory=/usr/bin/
fstring="Free Software Foundation"  # See which files come from the FSF.

for file in $( find $directory -type f -name '*' | sort )
do
  strings -f $file | grep "$fstring" | sed -e "s%$directory%%"
  #  In the "sed" expression,
  #+ it is necessary to substitute for the normal "/" delimiter
  #+ because "/" happens to be one of the characters filtered out.
  #  Failure to do so gives an error message (try it).
done

exit 0

#  Exercise (easy):
#  ----------------
#  Convert this script to taking command-line parameters
#+ for $directory and $fstring.
```

The output of a **for** loop may be piped to a command or commands.

Example 10-10. Listing the symbolic links in a directory

```
#!/bin/bash
# symlinks.sh: Lists symbolic links in a directory.

ARGS=1                # Expect one command-line argument.

if [ $# -ne "$ARGS" ]  # If not 1 arg...
then
  directory=`pwd`      # current working directory
else
  directory=$1
fi

echo "symbolic links in directory \"$directory\""

for file in "$( find $directory -type l )"   # -type l = symbolic links
do
  echo "$file"
done | sort                                  # Otherwise file list is unsorted.

#  As Dominik 'Aeneas' Schnitzer points out,
#+ failing to quote  $( find $directory -type l )
#+ will choke on filenames with embedded whitespace.

exit 0
```

The `stdout` of a loop may be redirected to a file, as this slight modification to the previous example shows.

Example 10-11. Symbolic links in a directory, saved to a file

```
#!/bin/bash
# symlinks.sh: Lists symbolic links in a directory.

ARGS=1                # Expect one command-line argument.
OUTFILE=symlinks.list  # save file

if [ $# -ne "$ARGS" ]  # If not 1 arg...
then
  directory=`pwd`      # current working directory
else
  directory=$1
fi

echo "symbolic links in directory \"$directory\""

for file in "$( find $directory -type l )"   # -type l = symbolic links
do
  echo "$file"
done | sort > "$OUTFILE"                    # stdout of loop
```

```
#           ^^^^^^^^^^^^                          redirected to save file.

exit 0
```

There is an alternative syntax to a **for** loop that will look very familiar to C programmers. This requires double parentheses.

Example 10-12. A C-like for loop

```
#!/bin/bash
# Two ways to count up to 10.

echo

# Standard syntax.
for a in 1 2 3 4 5 6 7 8 9 10
do
  echo -n "$a "
done

echo; echo

# +=========================================+

# Now, let's do the same, using C-like syntax.

LIMIT=10

for ((a=1; a <= LIMIT ; a++))  # Double parentheses, and "LIMIT" with no "$".
do
  echo -n "$a "
done                           # A construct borrowed from 'ksh93'.

echo; echo

# +=================================================================+

# Let's use the C "comma operator" to increment two variables simultaneously.

for ((a=1, b=1; a <= LIMIT ; a++, b++))  # The comma chains together operations.
do
  echo -n "$a-$b "
done

echo; echo

exit 0
```

See also Example 26-7, Example 26-8, and Example A-7.

Now, a *for-loop* used in a "real-life" context.

Example 10-13. Using efax in batch mode

```
#!/bin/bash

EXPECTED_ARGS=2
E_BADARGS=65

if [ $# -ne $EXPECTED_ARGS ]
# Check for proper no. of command line args.
then
   echo "Usage: `basename $0` phone# text-file"
   exit $E_BADARGS
fi


if [ ! -f "$2" ]
then
  echo "File $2 is not a text file"
  exit $E_BADARGS
fi


fax make $2              # Create fax formatted files from text files.

for file in $(ls $2.0*)  # Concatenate the converted files.
                         # Uses wild card in variable list.
do
  fil="$fil $file"
done

efax -d /dev/ttyS3 -o1 -t "T$1" $fil   # Do the work.


# As S.C. points out, the for-loop can be eliminated with
#    efax -d /dev/ttyS3 -o1 -t "T$1" $2.0*
# but it's not quite as instructive [grin].

exit 0
```

while

This construct tests for a condition at the top of a loop, and keeps looping as long as that condition is true (returns a 0 exit status). In contrast to a for loop, a *while loop* finds use in situations where the number of loop repetitions is not known beforehand.

while [*condition*]
do
  command...
done

As is the case with for/in loops, placing the **do** on the same line as the condition test requires a semicolon.

while [*condition*] ; do

Note that certain specialized **while** loops, as, for example, a getopts construct, deviate somewhat from the standard template given here.

Example 10-14. Simple while loop

```
#!/bin/bash

var0=0
LIMIT=10

while [ "$var0" -lt "$LIMIT" ]
do
  echo -n "$var0 "        # -n suppresses newline.
  var0=`expr $var0 + 1`   # var0=$(($var0+1)) also works.
done

echo

exit 0
```

Example 10-15. Another while loop

```
#!/bin/bash

echo

while [ "$var1" != "end" ]     # while test "$var1" != "end"
do                             # also works.
  echo "Input variable #1 (end to exit) "
  read var1                    # Not 'read $var1' (why?).
  echo "variable #1 = $var1"   # Need quotes because of "#".
  # If input is 'end', echoes it here.
  # Does not test for termination condition until top of loop.
  echo
done

exit 0
```

A **while** loop may have multiple conditions. Only the final condition determines when the loop terminates. This necessitates a slightly different loop syntax, however.

Example 10-16. while loop with multiple conditions

```
#!/bin/bash

var1=unset
previous=$var1

while echo "previous-variable = $previous"
      echo
      previous=$var1
      [ "$var1" != end ] # Keeps track of what $var1 was previously.
      # Four conditions on "while", but only last one controls loop.
      # The *last* exit status is the one that counts.
do
echo "Input variable #1 (end to exit) "
  read var1
  echo "variable #1 = $var1"
done

# Try to figure out how this all works.
# It's a wee bit tricky.

exit 0
```

As with a **for** loop, a **while** loop may employ C-like syntax by using the double parentheses construct (see also Example 9-25).

Example 10-17. C-like syntax in a while loop

```
#!/bin/bash
# wh-loopc.sh: Count to 10 in a "while" loop.

LIMIT=10
a=1

while [ "$a" -le $LIMIT ]
do
  echo -n "$a "
  let "a+=1"
done           # No surprises, so far.

echo; echo

# +=========================================================================+

# Now, repeat with C-like syntax.

((a = 1))      # a=1
# Double parentheses permit space when setting a variable, as in C.

while (( a <= LIMIT ))   # Double parentheses, and no "$" preceding variables.
do
  echo -n "$a "
  ((a += 1))   # let "a+=1"
```

```
  # Yes, indeed.
  # Double parentheses permit incrementing a variable with C-like syntax.
done

echo

# Now, C programmers can feel right at home in Bash.

exit 0
```

A **while** loop may have its `stdin` redirected to a file by a < at its end.

until

This construct tests for a condition at the top of a loop, and keeps looping as long as that condition is false (opposite of **while** loop).

until [*condition-is-true*]
do
  command...
done

Note that an **until** loop tests for the terminating condition at the top of the loop, differing from a similar construct in some programming languages.

As is the case with for/in loops, placing the **do** on the same line as the condition test requires a semicolon.

until [*condition-is-true*] ; do

Example 10-18. until loop

```
#!/bin/bash

until [ "$var1" = end ] # Tests condition here, at top of loop.
do
  echo "Input variable #1 "
  echo "(end to exit)"
  read var1
  echo "variable #1 = $var1"
done

exit 0
```

Chapter 10. Loops and Branches

Table of Contents

Operations on code blocks are the key to structured, organized shell scripts. Looping and branching constructs provide the tools for accomplishing this.

5.1. Variable Substitution

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

$

Let us carefully distinguish between the *name* of a variable and its *value*. If **variable1** is the name of a variable, then **$variable1** is a reference to its *value*, the data item it contains. The only time a variable appears "naked", without the $ prefix, is when declared or assigned, when *unset*, when exported, or in the special case of a variable representing a signal (see Example 30-5). Assignment may be with an = (as in *var1=27*), in a read statement, and at the head of a loop (*for var2 in 1 2 3*).

Enclosing a referenced value in double quotes (" ") does not interfere with variable substitution. This is called partial quoting, sometimes referred to as "weak quoting". Using single quotes (' ') causes the variable name to be used literally, and no substitution will take place. This is full quoting, sometimes referred to as "strong quoting". See Chapter 6 for a detailed discussion.

Note that **$variable** is actually a simplified alternate form of **${variable}**. In contexts where the **$variable** syntax causes an error, the longer form may work (see Section 9.3, below).

Example 5-1. Variable assignment and substitution

```
#!/bin/bash

# Variables: assignment and substitution

a=375
hello=$a

#-------------------------------------------------------------------------
# No space permitted on either side of = sign when initializing variables.

#  If "VARIABLE =value",
#+ script tries to run "VARIABLE" command with one argument, "=value".

#  If "VARIABLE= value",
#+ script tries to run "value" command with
#+ the environmental variable "VARIABLE" set to "".
#-------------------------------------------------------------------------


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

```
echo $hello
echo ${hello} # Identical to above.

echo "$hello"
echo "${hello}"

echo

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

echo

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

# Notice the effect of different types of quoting.


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

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

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

var1=variable1  var2=variable2  var3=variable3
echo
echo "var1=$var1   var2=$var2  var3=$var3"

# May cause problems with older versions of "sh".

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

echo; echo

numbers="one two three"
other_numbers="1 2 3"
# If whitespace within a variable, then quotes necessary.
echo "numbers = $numbers"
echo "other_numbers = $other_numbers"   # other_numbers = 1 2 3
echo

echo "uninitialized_variable = $uninitialized_variable"
```

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

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

echo

exit 0
```

An uninitialized variable has a "null" value - no assigned value at all (not zero!). Using a variable before assigning a value to it will inevitably cause problems.

Chapter 6. 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, such as the wild card character, *.)

```
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
```

Certain programs and utilities can still reinterpret or expand special characters in a quoted string. This is an important use of quoting, 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.
```

Of course, **grep [Ff]irst *.txt** would not work.

When referencing a variable, it is generally advisable in enclose it in double quotes (" "). This preserves all special characters within the variable name, except $, ` (backquote), and \ (escape). Keeping $ as a special character permits referencing a quoted variable (*"$variable"*), that is, replacing the variable with its value (see Example 5-1, above).

Use double quotes to prevent word splitting. [1] An argument enclosed in double quotes presents itself as a single word, even if it contains whitespace separators.

```
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, S.C.
```

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

Example 6-1. Echoing Weird Variables

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

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

echo

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

# Examples above supplied by S.C.

exit 0
```

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

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

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

echo

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

# This example courtesy of Stephane Chazelas.
```

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

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

Special meanings of certain escaped characters

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

\n

means newline

\r

means return

\t

means tab

\v

means vertical tab

\b

means backspace

\a

means "alert" (beep or flash)

\0xx

translates to the octal ASCII equivalent of *0xx*

Example 6-2. Escaped Characters

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

echo; echo

echo "\v\v\v\v"      # Prints \v\v\v\v
# Use the -e option with 'echo' to print escaped characters.
echo -e "\v\v\v\v"   # Prints 4 vertical tabs.
echo -e "\042"       # Prints " (quote, octal ASCII character 42).

# The $'\X' construct makes the -e option unnecessary.
echo $'\n'           # Newline.
echo $'\a'           # Alert (beep).

# Version 2 and later of Bash permits using the $'\xxx' construct.
echo $'\t \042 \t'   # Quote (") framed by tabs.


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

echo

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

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

echo; echo

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

echo; echo

exit 0
```

See Example 35-1 for another example of the **$' '** string expansion construct.

\"

gives the quote its literal meaning

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

\$

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

```
echo "\$variable01"  # results in $variable01
```

\\

gives the backslash its literal meaning

```
echo "\\"  # results in \
```

The behavior of \ depends on whether it is itself escaped, 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 Stephane Chazelas.
```

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

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

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

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

variable=\\
echo "$variable"        # \
```

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

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

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

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

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

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

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

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

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

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

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

```
echo "foo
bar"
#foo
#bar

echo

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

echo

echo foo\
bar     # Newline escaped.
#foobar

echo

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

echo

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

# Examples suggested by Stephane Chazelas.
```

Notes

[1] "Word splitting", in this context, means dividing a character string into a number of separate and discrete arguments.

Chapter 14. Command Substitution

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

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_listing
# Same result.

# A possible problem with putting a list of files into a single string
# is that a newline may creep in.
#
# A safer way to assign a list of files to a parameter is with an array.
#      shopt -s nullglob    # If no match, filename expands to nothing.
#      textfile_listing=( *.txt )
#
# Thanks, S.C.
```

Command substitution may result in word splitting.

```
COMMAND `echo a b`     # 2 args: a and b

COMMAND "`echo a b`"   # 1 arg: "a b"

COMMAND `echo`         # no arg

COMMAND "`echo`"       # one empty arg


# Thanks, S.C.
```

Even when there is no word splitting, command substitution can remove trailing newlines.

```
# cd "`pwd`"  # This should always work.
# However...

mkdir 'dir with trailing newline
'

cd 'dir with trailing newline
'

cd "`pwd`"  # Error message:
# bash: cd: /tmp/file with trailing newline: No such file or directory

cd "$PWD"   # Works fine.




old_tty_setting=$(stty -g)   # Save old terminal setting.
echo "Hit a key "
stty -icanon -echo           # Disable "canonical" mode for terminal.
                             # Also, disable *local* echo.
key=$(dd bs=1 count=1 2> /dev/null)   # Using 'dd' to get a keypress.
stty "$old_tty_setting"      # Restore old setting.
echo "You hit ${#key} key."  # ${#variable} = number of characters in $variable
#
# Hit any key except RETURN, and the output is "You hit 1 key."
# Hit RETURN, and it's "You hit 0 key."
# The newline gets eaten in the command substitution.

Thanks, S.C.
```

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".

#  Be aware that the variables may contain embedded whitespace,
#+ or even (horrors), control characters.
```

```
#  Excerpts from system file, /etc/rc.d/rc.sysinit
#+ (on a Red Hat Linux installation)


if [ -f /fsckoptions ]; then
        fsckoptions=`cat /fsckoptions`
...
fi
#
#
if [ -e "/proc/ide/${disk[$device]}/media" ] ; then
             hdmedia=`cat /proc/ide/${disk[$device]}/media`
...
fi
#
#
if [ ! -n "`uname -r | grep -- "-"`" ]; then
       ktag="`cat /proc/version`"
...
fi
#
#
if [ $usb = "1" ]; then
```

```
    sleep 5
    mouseoutput=`cat /proc/bus/usb/devices 2>/dev/null|grep -E "^I.*Cls=03.*Prot=02"`
    kbdoutput=`cat /proc/bus/usb/devices 2>/dev/null|grep -E "^I.*Cls=03.*Prot=01"`
...
fi
```

Do not set a variable to the contents of a *long* text file unless you have a very good reason for doing so. Do not set a variable to the contents of a *binary* file, even as a joke.

Example 14-1. Stupid script tricks

```
#!/bin/bash
# stupid-script-tricks.sh: Don't try this at home, folks.


dangerous_variable=`cat /boot/vmlinuz`   # The compressed Linux kernel itself.

echo "string-length of \$dangerous_variable = ${#dangerous_variable}"
# string-length of $dangerous_variable = 794151
# (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 14-2. 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.

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

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." );
  return (0);
}
```

```
bash$ gcc -o hello hello.c
```

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

greeting=`./hello`
echo $greeting
```

```
bash$ sh hello.sh
Hello, world.
```

The **$(COMMAND)** form has superseded backticks for command substitution.

```
output=$(sed -n /"$1"/p $file)
# From "grp.sh" example.
```

Examples of command substitution in shell scripts:

1. Example 10-7
2. Example 10-25
3. Example 9-23
4. Example 12-2

5. Example 12-15
6. Example 12-12
7. Example 12-38
8. Example 10-13
9. Example 10-10
10. Example 12-24
11. Example 16-7
12. Example A-17
13. Example 28-1
14. Example 12-32
15. Example 12-33
16. Example 12-34

Notes

[1] For purposes of *command substitution*, a **command** may be an external system command, an internal scripting *builtin*, or even a script function.

12.8. Math Commands

"Doing the numbers"

factor

Decompose an integer into prime factors.

```
bash$ factor 27417
27417: 3 13 19 37
```

bc, **dc**

These are flexible, arbitrary precision calculation utilities.

bc has a syntax vaguely resembling C.

dc is stack-oriented and uses RPN ("Reverse Polish Notation").

Of the two, **bc** seems more useful in scripting. It is a fairly well-behaved UNIX utility, and may therefore be used in a pipe.

Bash can't handle floating point calculations, and it lacks operators for certain important mathematical functions. Fortunately, **bc** comes to the rescue.

Here is a simple template for using **bc** to calculate a script variable. This uses command substitution.

```
variable=$(echo "OPTIONS; OPERATIONS" | bc)
```

Example 12-32. Monthly Payment on a Mortgage

```
#!/bin/bash
# monthlypmt.sh: Calculates monthly payment on a mortgage.


# This is a modification of code in the "mcalc" (mortgage calculator) package,
# by Jeff Schmidt and Mendel Cooper (yours truly, the author of this document).
#   http://www.ibiblio.org/pub/Linux/apps/financial/mcalc-1.6.tar.gz  [15k]

echo
echo "Given the principal, interest rate, and term of a mortgage,"
echo "calculate the monthly payment."

bottom=1.0

echo
echo -n "Enter principal (no commas) "
read principal
echo -n "Enter interest rate (percent) "  # If 12%, enter "12", not ".12".
read interest_r
echo -n "Enter term (months) "
read term


 interest_r=$(echo "scale=9; $interest_r/100.0" | bc) # Convert to decimal.
                # "scale" determines how many decimal places.


 interest_rate=$(echo "scale=9; $interest_r/12 + 1.0" | bc)


 top=$(echo "scale=9; $principal*$interest_rate^$term" | bc)

 echo; echo "Please be patient. This may take a while."

 let "months = $term - 1"
# ============================================================================
 for ((x=$months; x > 0; x--))
 do
   bot=$(echo "scale=9; $interest_rate^$x" | bc)
   bottom=$(echo "scale=9; $bottom+$bot" | bc)
#  bottom = $(($bottom + $bot"))
 done
# ----------------------------------------------------------------------------
#  Rick Boivie pointed out a more efficient implementation
#+ of the above loop, which decreases computation time by 2/3.

# for ((x=1; x <= $months; x++))
# do
#   bottom=$(echo "scale=9; $bottom * $interest_rate + 1" | bc)
# done


#  And then he came up with an even more efficient alternative,
#+ one that cuts down the run time by about 95%!

# bottom=`{
#     echo "scale=9; bottom=$bottom; interest_rate=$interest_rate"
```

```
#     for ((x=1; x <= $months; x++))
#     do
#          echo 'bottom = bottom * interest_rate + 1'
#     done
#     echo 'bottom'
#     } | bc`       # Embeds a 'for loop' within command substitution.

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

 # let "payment = $top/$bottom"
 payment=$(echo "scale=2; $top/$bottom" | bc)
 # Use two decimal places for dollars and cents.

 echo
 echo "monthly payment = \$$payment"  # Echo a dollar sign in front of amount.
 echo


 exit 0

 # Exercises:
 #   1) Filter input to permit commas in principal amount.
 #   2) Filter input to permit interest to be entered as percent or decimal.
 #   3) If you are really ambitious,
 #      expand this script to print complete amortization tables.
```

Example 12-33. Base Conversion

```
:
##########################################################################
# Shellscript:  base.sh - print number to different bases (Bourne Shell)
# Author     :  Heiner Steven (heiner.steven@odn.de)
# Date       :  07-03-95
# Category   :  Desktop
# $Id: base.sh,v 1.2 2000/02/06 19:55:35 heiner Exp $
##########################################################################
# Description
#
# Changes
# 21-03-95 stv  fixed error occuring with 0xb as input (0.2)
##########################################################################

# ==> Used in this document with the script author's permission.
# ==> Comments added by document author.

NOARGS=65
PN=`basename "$0"`                              # Program name
VER=`echo '$Revision: 1.2 $' | cut -d' ' -f2`  # ==> VER=1.2

Usage () {
    echo "$PN - print number to different bases, $VER (stv '95)
usage: $PN [number ...]

If no number is given, the numbers are read from standard input.
```

```
A number may be
    binary (base 2)             starting with 0b (i.e. 0b1100)
    octal (base 8)              starting with 0  (i.e. 014)
    hexadecimal (base 16)       starting with 0x (i.e. 0xc)
    decimal                     otherwise (i.e. 12)" >&2
    exit $NOARGS
}   # ==> Function to print usage message.

Msg () {
    for i   # ==> in [list] missing.
    do echo "$PN: $i" >&2
    done
}

Fatal () { Msg "$@"; exit 66; }

PrintBases () {
    # Determine base of the number
    for i      # ==> in [list] missing...
    do         # ==> so operates on command line arg(s).
        case "$i" in
            0b*)                ibase=2;;       # binary
            0x*|[a-f]*|[A-F]*)  ibase=16;;      # hexadecimal
            0*)                 ibase=8;;       # octal
            [1-9]*)             ibase=10;;      # decimal
            *)
                Msg "illegal number $i - ignored"
                continue;;
        esac

        # Remove prefix, convert hex digits to uppercase (bc needs this)
        number=`echo "$i" | sed -e 's:^0[bBxX]::' | tr '[a-f]' '[A-F]'`
        # ==> Uses ":" as sed separator, rather than "/".

        # Convert number to decimal
        dec=`echo "ibase=$ibase; $number" | bc`  # ==> 'bc' is calculator utility.
        case "$dec" in
            [0-9]*)     ;;                      # number ok
            *)          continue;;              # error: ignore
        esac

        # Print all conversions in one line.
        # ==> 'here document' feeds command list to 'bc'.
        echo `bc <<!
            obase=16; "hex="; $dec
            obase=10; "dec="; $dec
            obase=8;  "oct="; $dec
            obase=2;  "bin="; $dec
!
    ` | sed -e 's: :    :g'

    done
}

while [ $# -gt 0 ]
do
```

```
    case "$1" in
        --)     shift; break;;
        -h)     Usage;;                 # ==> Help message.
        -*)     Usage;;
        *)      break;;                 # first number
    esac   # ==> More error checking for illegal input would be useful.
    shift
done

if [ $# -gt 0 ]
then
    PrintBases "$@"
else                                    # read from stdin
    while read line
    do
        PrintBases $line
    done
fi
```

An alternate method of invoking **bc** involves using a here document embedded within a command substitution block. This is especially appropriate when a script needs to pass a list of options and commands to **bc**.

```
variable=`bc << LIMIT_STRING
options
statements
operations
LIMIT_STRING
`

...or...


variable=$(bc << LIMIT_STRING
options
statements
operations
LIMIT_STRING
)
```

Example 12-34. Another way to invoke bc

```
#!/bin/bash
# Invoking 'bc' using command substitution
# in combination with a 'here document'.


var1=`bc << EOF
18.33 * 19.78
EOF
`
echo $var1       # 362.56


#  $( ... ) notation also works.
v1=23.53
v2=17.881
v3=83.501
v4=171.63

var2=$(bc << EOF
scale = 4
a = ( $v1 + $v2 )
b = ( $v3 * $v4 )
a * b + 15.35
EOF
)
echo $var2       # 593487.8452


var3=$(bc -l << EOF
scale = 9
s ( 1.7 )
EOF
)
# Returns the sine of 1.7 radians.
# The "-l" option calls the 'bc' math library.
echo $var3       # .991664810


# Now, try it in a function...
hyp=             # Declare global variable.
hypotenuse ()    # Calculate hypotenuse of a right triangle.
{
hyp=$(bc -l << EOF
scale = 9
sqrt ( $1 * $1 + $2 * $2 )
EOF
)
# Unfortunately, can't return floating point values from a Bash function.
}

hypotenuse 3.68 7.31
echo "hypotenuse = $hyp"    # 8.184039344


exit 0
```

Most persons avoid **dc**, since it requires non-intuitive RPN input. Yet it has its uses.

Example 12-35. Converting a decimal number to hexadecimal

```
#!/bin/bash
# hexconvert.sh: Convert a decimal number to hexadecimal.

BASE=16     # Hexadecimal.

if [ -z "$1" ]
then
  echo "Usage: $0 number"
  exit $E_NOARGS
  # Need a command line argument.
fi
# Exercise: add argument validity checking.


hexcvt ()
{
if [ -z "$1" ]
then
  echo 0
  return    # "Return" 0 if no arg passed to function.
fi

echo ""$1" "$BASE" o p" | dc
#                 "o" sets radix (numerical base) of output.
#                   "p" prints the top of stack.
# See 'man dc' for other options.
return
}

hexcvt "$1"

exit 0
```

Studying the *info* page for **dc** gives some insight into its intricacies. However, there seems to be a small, select group of *dc wizards* who delight in showing off their mastery of this powerful, but arcane utility.

Example 12-36. Factoring

```
#!/bin/bash
# factr.sh: Factor a number

MIN=2       # Will not work for number smaller than this.
E_NOARGS=65
E_TOOSMALL=66

if [ -z $1 ]
then
  echo "Usage: $0 number"
  exit $E_NOARGS
fi

if [ "$1" -lt "$MIN" ]
then
  echo "Number to factor must be $MIN or greater."
  exit $E_TOOSMALL
fi

# Exercise: Add type checking (to reject non-integer arg).

echo "Factors of $1:"
# -------------------------------------------------------------------------------
echo "$1[p]s2[lip/dli%0=1dvsr]s12sid2%0=13sidvsr[dli%0=1lrli2+dsi!>.]ds.xd1<2" | dc
# -------------------------------------------------------------------------------
# Above line of code written by Michel Charpentier <charpov@cs.unh.edu>.
# Used with permission (thanks).

 exit 0
```

awk

Yet another way of doing floating point math in a script is using awk's built-in math functions in a shell wrapper.

Example 12-37. Calculating the hypotenuse of a triangle

```
#!/bin/bash
# hypotenuse.sh: Returns the "hypotenuse" of a right triangle.
#               ( square root of sum of squares of the "legs")

ARGS=2                # Script needs sides of triangle passed.
E_BADARGS=65          # Wrong number of arguments.

if [ $# -ne "$ARGS" ] # Test number of arguments to script.
then
  echo "Usage: `basename $0` side_1 side_2"
  exit $E_BADARGS
fi


AWKSCRIPT=' { printf( "%3.7f\n", sqrt($1*$1 + $2*$2) ) } '
#            command(s) / parameters passed to awk


echo -n "Hypotenuse of $1 and $2 = "
```

```
echo $1 $2 | awk "$AWKSCRIPT"

exit 0
```

9.1. Internal Variables

Builtin variables

variables affecting bash script behavior

$BASH

the path to the *Bash* binary itself

```
bash$ echo $BASH
/bin/bash
```

$BASH_ENV

an environmental variable pointing to a Bash startup file to be read when a script is invoked

$BASH_VERSINFO[n]

a 6-element array containing version information about the installed release of Bash. This is similar to $BASH_VERSION, below, but a bit more detailed.

```
# Bash version info:

for n in 0 1 2 3 4 5
do
  echo "BASH_VERSINFO[$n] = ${BASH_VERSINFO[$n]}"
done

# BASH_VERSINFO[0] = 2                      # Major version no.
# BASH_VERSINFO[1] = 05                     # Minor version no.
# BASH_VERSINFO[2] = 8                      # Patch level.
# BASH_VERSINFO[3] = 1                      # Build version.
# BASH_VERSINFO[4] = release                # Release status.
# BASH_VERSINFO[5] = i386-redhat-linux-gnu  # Architecture
                                            # (same as $MACHTYPE).
```

$BASH_VERSION

the version of Bash installed on the system

```
bash$ echo $BASH_VERSION
2.04.12(1)-release
```

```
tcsh% echo $BASH_VERSION
BASH_VERSION: Undefined variable.
```

Checking $BASH_VERSION is a good method of determining which shell is running. $SHELL does not necessarily give the correct answer.

$DIRSTACK

the top value in the directory stack (affected by pushd and popd)

This builtin variable corresponds to the dirs command, however **dirs** shows the entire contents of the directory stack.

$EDITOR

the default editor invoked by a script, usually **vi** or **emacs**.

$EUID

"effective" user id number

Identification number of whatever identity the current user has assumed, perhaps by means of su.

The $EUID is not necessarily the same as the $UID.

$FUNCNAME

name of the current function

```
xyz23 ()
{
  echo "$FUNCNAME now executing."  # xyz23 now executing.
}

xyz23

echo "FUNCNAME = $FUNCNAME"        # FUNCNAME =
                                   # Null value outside a function.
```

$GLOBIGNORE

A list of filename patterns to be excluded from matching in globbing.

$GROUPS

groups current user belongs to

This is a listing (array) of the group id numbers for current user, as recorded in /etc/passwd.

```
root# echo $GROUPS
0


root# echo ${GROUPS[1]}
1


root# echo ${GROUPS[5]}
6
```

$HOME

home directory of the user, usually /home/username (see Example 9-12)

$HOSTNAME

The hostname command assigns the system name at bootup in an init script. However, the gethostname() function sets the Bash internal variable $HOSTNAME. See also Example 9-12.

$HOSTTYPE

host type

Like $MACHTYPE, identifies the system hardware.

```
bash$ echo $HOSTTYPE
i686
```

$IFS

input field separator

This defaults to whitespace (space, tab, and newline), but may be changed, for example, to parse a comma-separated data file. Note that $* uses the first character held in $IFS. See Example 6-1.

```
bash$ echo $IFS | cat -vte
$


bash$ bash -c 'set w x y z; IFS=":-;"; echo "$*"'
w:x:y:z
```

$IFS does not handle whitespace the same as it does other characters.

Example 9-1. $IFS and whitespace

```
#!/bin/bash
# $IFS treats whitespace differently than other characters.

output_args_one_per_line()
{
  for arg
  do echo "[$arg]"
  done
}

echo; echo "IFS=\" \""
echo "-------"

IFS=" "
var=" a  b c   "
output_args_one_per_line $var  # output_args_one_per_line `echo " a  b c   "`
#
# [a]
# [b]
# [c]


echo; echo "IFS=:"
echo "-----"

IFS=:
var=":a::b:c:::"               # Same as above, but substitute ":" for " ".
output_args_one_per_line $var
#
# []
# [a]
# []
# [b]
# [c]
# []
# []
# []

# The same thing happens with the "FS" field separator in awk.

# Thank you, Stephane Chazelas.

echo

exit 0
```

(Thanks, S. C., for clarification and examples.)

$IGNOREEOF

ignore EOF: how many end-of-files (control-D) the shell will ignore before logging out.

$LC_COLLATE

Often set in the `.bashrc` or `/etc/profile` files, this variable controls collation order in filename expansion and pattern matching. If mishandled, `LC_COLLATE` can cause unexpected results in filename globbing.

As of version 2.05 of Bash, filename globbing no longer distinguishes between lowercase and uppercase letters in a character range between brackets. For example, **ls [A-M]*** would match both `File1.txt` and `file1.txt`. To revert to the customary behavior of bracket matching, set `LC_COLLATE` to `C` by an **`export LC_COLLATE=C`** in `/etc/profile` and/or `~/.bashrc`.

`$LC_CTYPE`

This internal variable controls character interpretation in globbing and pattern matching.

`$LINENO`

This variable is the line number of the shell script in which this variable appears. It has significance only within the script in which it appears, and is chiefly useful for debugging purposes.

```
# *** BEGIN DEBUG BLOCK ***
last_cmd_arg=$_  # Save it.

echo "At line number $LINENO, variable \"v1\" = $v1"
echo "Last command argument processed = $last_cmd_arg"
# *** END DEBUG BLOCK ***
```

`$MACHTYPE`

machine type

Identifies the system hardware.

```
bash$ echo $MACHTYPE
i686-debian-linux-gnu
```

`$OLDPWD`

old working directory ("OLD-print-working-directory", previous directory you were in)

`$OSTYPE`

operating system type

```
bash$ echo $OSTYPE
linux-gnu
```

`$PATH`

path to binaries, usually `/usr/bin/`, `/usr/X11R6/bin/`, `/usr/local/bin`, etc.

When given a command, the shell automatically does a hash table search on the directories listed in the *path* for the executable. The path is stored in the environmental variable, `$PATH`, a list of directories, separated by colons. Normally, the system stores the `$PATH` definition in `/etc/profile` and/or `~/.bashrc` (see Chapter 27).

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

PATH=${PATH}:/opt/bin appends the /opt/bin directory to the current path. In a script, it may be expedient to temporarily add a directory to the path in this way. When the script exits, this restores the original $PATH (a child process, such as a script, may not change the environment of the parent process, the shell).

The current "working directory", ./, is usually omitted from the $PATH as a security measure.

$PIPESTATUS

Exit status of last executed pipe. Interestingly enough, this does not give the same result as the exit status of the last executed command.

```
bash$ echo $PIPESTATUS
0

bash$ ls -al | bogus_command
bash: bogus_command: command not found
bash$ echo $PIPESTATUS
141

bash$ ls -al | bogus_command
bash: bogus_command: command not found
bash$ echo $?
127
```

$PPID

The $PPID of a process is the process id (pid) of its parent process. [1]

Compare this with the pidof command.

$PS1

This is the main prompt, seen at the command line.

$PS2

The secondary prompt, seen when additional input is expected. It displays as ">".

$PS3

The tertiary prompt, displayed in a select loop (see Example 10-28).

$PS4

The quartenary prompt, shown at the beginning of each line of output when invoking a script with the -x option. It displays as "+".

$PWD

working directory (directory you are in at the time)

This is the analog to the pwd builtin command.

```
#!/bin/bash

E_WRONG_DIRECTORY=73

clear # Clear screen.

TargetDirectory=/home/bozo/projects/GreatAmericanNovel

cd $TargetDirectory
echo "Deleting stale files in $TargetDirectory."

if [ "$PWD" != "$TargetDirectory" ]
then    # Keep from wiping out wrong directory by accident.
  echo "Wrong directory!"
  echo "In $PWD, rather than $TargetDirectory!"
  echo "Bailing out!"
  exit $E_WRONG_DIRECTORY
fi

rm -rf *
rm .[A-Za-z0-9]*    # Delete dotfiles.
# rm -f .[^.]* ..?*   to remove filenames beginning with multiple dots.
# (shopt -s dotglob; rm -f *)   will also work.
# Thanks, S.C. for pointing this out.

# Filenames may contain all characters in the 0 - 255 range, except "/".
# Deleting files beginning with weird characters is left as an exercise.

# Various other operations here, as necessary.

echo
echo "Done."
echo "Old files deleted in $TargetDirectory."
echo


exit 0
```

$REPLY

The default value when a variable is not supplied to read. Also applicable to select menus, but only supplies the item number of the variable chosen, not the value of the variable itself.

```
#!/bin/bash

echo
echo -n "What is your favorite vegetable? "
read

echo "Your favorite vegetable is $REPLY."
# REPLY holds the value of last "read" if and only if
# no variable supplied.

echo
echo -n "What is your favorite fruit? "
read fruit
```

```
echo "Your favorite fruit is $fruit."
echo "but..."
echo "Value of \$REPLY is still $REPLY."
# $REPLY is still set to its previous value because
# the variable $fruit absorbed the new "read" value.

echo

exit 0
```

$SECONDS

The number of seconds the script has been running.

```
#!/bin/bash

ENDLESS_LOOP=1
INTERVAL=1

echo
echo "Hit Control-C to exit this script."
echo

while [ $ENDLESS_LOOP ]
do
  if [ "$SECONDS" -eq 1 ]
  then
    units=second
  else
    units=seconds
  fi

  echo "This script has been running $SECONDS $units."
  sleep $INTERVAL
done


exit 0
```

$SHELLOPTS

the list of enabled shell options, a readonly variable

```
bash$ echo $SHELLOPTS
braceexpand:hashall:histexpand:monitor:history:interactive-comments:emacs
```

$SHLVL

Shell level, how deeply Bash is nested. If, at the command line, $SHLVL is 1, then in a script it will increment to 2.

$TMOUT

If the *$TMOUT* environmental variable is set to a non-zero value *time*, then the shell prompt will time out after *time* seconds. This will cause a logout.

Unfortunately, this works only while waiting for input at the shell prompt console or in an xterm. While it would be nice to speculate on the uses of this internal variable for timed input, for example in combination with read, *$TMOUT* does not work in that context and is virtually useless for shell scripting. (Reportedly the *ksh* version of a timed **read** does work.)

Implementing timed input in a script is certainly possible, but may require complex machinations. One method is to set up a timing loop to signal the script when it times out. This also requires a signal handling routine to trap (see Example 30-5) the interrupt generated by the timing loop (whew!).

Example 9-2. Timed Input

```
#!/bin/bash
# timed-input.sh

# TMOUT=3            useless in a script

TIMELIMIT=3  # Three seconds in this instance, may be set to different value.

PrintAnswer()
{
  if [ "$answer" = TIMEOUT ]
  then
    echo $answer
  else       # Don't want to mix up the two instances.
    echo "Your favorite veggie is $answer"
    kill $!  # Kills no longer needed TimerOn function running in background.
             # $! is PID of last job running in background.
  fi

}



TimerOn()
{
  sleep $TIMELIMIT && kill -s 14 $$ &
  # Waits 3 seconds, then sends sigalarm to script.
}

Int14Vector()
{
  answer="TIMEOUT"
  PrintAnswer
  exit 14
}

trap Int14Vector 14   # Timer interrupt (14) subverted for our purposes.

echo "What is your favorite vegetable "
TimerOn
read answer
PrintAnswer
```

```
#  Admittedly, this is a kludgy implementation of timed input,
#+ however the "-t" option to "read" simplifies this task.
#  See "t-out.sh", below.

#  If you need something really elegant...
#+ consider writing the application in C or C++,
#+ using appropriate library functions, such as 'alarm' and 'setitimer'.

exit 0
```

An alternative is using stty.

Example 9-3. Once more, timed input

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

# Written by Stephane Chazelas,
# and modified by the document author.

INTERVAL=5                # timeout interval

timedout_read() {
  timeout=$1
  varname=$2
  old_tty_settings=`stty -g`
  stty -icanon min 0 time ${timeout}0
  eval read $varname      # or just    read $varname
  stty "$old_tty_settings"
  # See man page for "stty".
}

echo; echo -n "What's your name? Quick! "
timedout_read $INTERVAL your_name

# This may not work on every terminal type.
# The maximum timeout depends on the terminal.
# (it is often 25.5 seconds).

echo

if [ ! -z "$your_name" ]  # If name input before timeout...
then
  echo "Your name is $your_name."
else
  echo "Timed out."
fi

echo

# The behavior of this script differs somewhat from "timed-input.sh".
# At each keystroke, the counter resets.

exit 0
```

Perhaps the simplest method is using the `-t` option to read.

Example 9-4. Timed read

```
#!/bin/bash
# t-out.sh (per a suggestion by "syngin seven)

TIMELIMIT=4        # 4 seconds

read -t $TIMELIMIT variable <&1

echo

if [ -z "$variable" ]
then
  echo "Timed out, variable still unset."
else
  echo "variable = $variable"
fi

exit 0
```

`$UID`

user id number

current user's user identification number, as recorded in `/etc/passwd`

This is the current user's real id, even if she has temporarily assumed another identity through su. `$UID` is a readonly variable, not subject to change from the command line or within a script, and is the counterpart to the id builtin.

Example 9-5. Am I root?

```
#!/bin/bash
# am-i-root.sh:   Am I root or not?

ROOT_UID=0   # Root has $UID 0.

if [ "$UID" -eq "$ROOT_UID" ]  # Will the real "root" please stand up?
then
  echo "You are root."
else
  echo "You are just an ordinary user (but mom loves you just the same)."
fi

exit 0


# ============================================================= #
# Code below will not execute, because the script already exited.

# An alternate method of getting to the root of matters:

ROOTUSER_NAME=root

username=`id -nu`              # Or...   username=`whoami`
if [ "$username" = "$ROOTUSER_NAME" ]
```

```
then
  echo "Rooty, toot, toot. You are root."
else
  echo "You are just a regular fella."
fi
```

See also Example 2-2.

The variables $ENV, $LOGNAME, $MAIL, $TERM, $USER, and $USERNAME are *not* Bash builtins. These are, however, often set as environmental variables in one of the Bash startup files. $SHELL, the name of the user's login shell, may be set from /etc/passwd or in an "init" script, and it is likewise not a Bash builtin.

```
tcsh% echo $LOGNAME
bozo
tcsh% echo $SHELL
/bin/tcsh
tcsh% echo $TERM
rxvt

bash$ echo $LOGNAME
bozo
bash$ echo $SHELL
/bin/tcsh
bash$ echo $TERM
rxvt
```

Positional Parameters

$0, $1, $2, etc.

positional parameters, passed from command line to script, passed to a function, or set to a variable (see Example 5-5 and Example 11-11)

$#

number of command line arguments [2] or positional parameters (see Example 34-2)

$*

All of the positional parameters, seen as a single word

$@

Same as $*, but each parameter is a quoted string, that is, the parameters are passed on intact, without interpretation or expansion. This means, among other things, that each parameter in the argument list is seen as a separate word.

Example 9-6. arglist: Listing arguments with $* and $@

```
#!/bin/bash
# Invoke this script with several arguments, such as "one two three".

E_BADARGS=65

if [ ! -n "$1" ]
then
  echo "Usage: `basename $0` argument1 argument2 etc."
  exit $E_BADARGS
fi

echo

index=1

echo "Listing args with \"\$*\":"
for arg in "$*"  # Doesn't work properly if "$*" isn't quoted.
do
  echo "Arg #$index = $arg"
  let "index+=1"
done             # $* sees all arguments as single word.
echo "Entire arg list seen as single word."

echo

index=1

echo "Listing args with \"\$@\":"
for arg in "$@"
do
  echo "Arg #$index = $arg"
  let "index+=1"
done             # $@ sees arguments as separate words.
echo "Arg list seen as separate words."

echo

exit 0
```

Following a **shift**, the $@ holds the remaining command-line parameters, lacking the previous $1, which was lost.

```
#!/bin/bash
# Invoke with ./scriptname 1 2 3 4 5

echo "$@"    # 1 2 3 4 5
shift
echo "$@"    # 2 3 4 5
shift
echo "$@"    # 3 4 5

# Each "shift" loses parameter $1.
# "$@" then contains the remaining parameters.
```

The $@ special parameter finds use as a tool for filtering input into shell scripts. The **cat "$@"** construction accepts input to a script either from stdin or from files given as parameters to the script. See Example 12-17 and Example 12-18.

The $* and $@ parameters sometimes display inconsistent and puzzling behavior, depending on the setting of $IFS.

Example 9-7. Inconsistent $* and $@ behavior

```
#!/bin/bash

# Erratic behavior of the "$*" and "$@" internal Bash variables,
# depending on whether these are quoted or not.
# Word splitting and linefeeds handled inconsistently.

# This example script by Stephane Chazelas,
# and slightly modified by the document author.


set -- "First one" "second" "third:one" "" "Fifth: :one"
# Setting the script arguments, $1, $2, etc.

echo

echo 'IFS unchanged, using "$*"'
c=0
for i in "$*"               # quoted
do echo "$((c+=1)): [$i]"   # This line remains the same in every instance.
                            # Echo args.
done
echo ---

echo 'IFS unchanged, using $*'
c=0
for i in $*                 # unquoted
do echo "$((c+=1)): [$i]"
done
echo ---

echo 'IFS unchanged, using "$@"'
c=0
for i in "$@"
do echo "$((c+=1)): [$i]"
done
echo ---

echo 'IFS unchanged, using $@'
c=0
for i in $@
do echo "$((c+=1)): [$i]"
done
echo ---

IFS=:
echo 'IFS=":", using "$*"'
c=0
for i in "$*"
do echo "$((c+=1)): [$i]"
done
echo ---
```

```
echo 'IFS=":", using $*'
c=0
for i in $*
do echo "$((c+=1)): [$i]"
done
echo ---

var=$*
echo 'IFS=":", using "$var" (var=$*)'
c=0
for i in "$var"
do echo "$((c+=1)): [$i]"
done
echo ---

echo 'IFS=":", using $var (var=$*)'
c=0
for i in $var
do echo "$((c+=1)): [$i]"
done
echo ---

var="$*"
echo 'IFS=":", using $var (var="$*")'
c=0
for i in $var
do echo "$((c+=1)): [$i]"
done
echo ---

echo 'IFS=":", using "$var" (var="$*")'
c=0
for i in "$var"
do echo "$((c+=1)): [$i]"
done
echo ---

echo 'IFS=":", using "$@"'
c=0
for i in "$@"
do echo "$((c+=1)): [$i]"
done
echo ---

echo 'IFS=":", using $@'
c=0
for i in $@
do echo "$((c+=1)): [$i]"
done
echo ---

var=$@
echo 'IFS=":", using $var (var=$@)'
c=0
for i in $var
do echo "$((c+=1)): [$i]"
done
```

```
echo ---

echo 'IFS=":", using "$var" (var=$@)'
c=0
for i in "$var"
do echo "$((c+=1)): [$i]"
done
echo ---

var="$@"
echo 'IFS=":", using "$var" (var="$@")'
c=0
for i in "$var"
do echo "$((c+=1)): [$i]"
done
echo ---

echo 'IFS=":", using $var (var="$@")'
c=0
for i in $var
do echo "$((c+=1)): [$i]"
done

echo

# Try this script with ksh or zsh -y.

exit 0
```

The **$@** and **$*** parameters differ only when between double quotes.

Example 9-8. $* and $@ when $IFS is empty

```
#!/bin/bash

# If $IFS set, but empty,
# then "$*" and "$@" do not echo positional params as expected.

mecho ()       # Echo positional parameters.
{
echo "$1,$2,$3";
}


IFS=""         # Set, but empty.
set a b c      # Positional parameters.

mecho "$*"     # abc,,
mecho $*       # a,b,c

mecho $@       # a,b,c
mecho "$@"     # a,b,c

# The behavior of $* and $@ when $IFS is empty depends
# on whatever Bash or sh version being run.
# It is therefore inadvisable to depend on this "feature" in a script.
```

```
# Thanks, S.C.

exit 0
```

Other Special Parameters

$-

Flags passed to script

This was originally a *ksh* construct adopted into Bash, and unfortunately it does not seem to work reliably in Bash scripts. One possible use for it is to have a script self-test whether it is interactive.

$!

PID (process id) of last job run in background

$_

Special variable set to last argument of previous command executed.

Example 9-9. underscore variable

```
#!/bin/bash

echo $_              # /bin/bash
                     # Just called /bin/bash to run the script.

du >/dev/null        # So no output from command.
echo $_              # du

ls -al >/dev/null    # So no output from command.
echo $_              # -al  (last argument)

:
echo $_              # :
```

$?

exit status of a command, function, or the script itself (see Example 23-3)

$$

process id of the script itself, often used in scripts to construct "unique" temp file names (see Example A-13, Example 30-6, Example 12-23, and Example 11-20)

Notes

[1] The pid of the currently running script is $$, of course.

[2] The words "argument" and "parameter" are often used interchangeably. In the context of this document, they have the same precise meaning, that of a variable passed to a script or function.

Variables Revisited Up Manipulating Strings

Chapter 4. Special Characters

Special Characters Found In Scripts and Elsewhere

#

Comments. Lines beginning with a # (with the exception of #!) are comments.

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

Comments may also occur at the end of a command.

```
echo "A comment will follow." # Comment here.
```

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

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

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

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

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

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

# Thanks, S.C.
```

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

Certain pattern matching operations also use the #.

;

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

```
echo hello; echo there
```

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

;;

Terminator in a case option. [Double semicolon]

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

.

"dot" command. [period] Equivalent to source (see Example 11-16). This is a bash builtin.

.

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

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


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

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

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

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

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

The *dot* often appears as the destination (directory) of a file movement command.

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

.

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

"

partial quoting. [double quote] *"STRING"* preserves (from interpretation) most of the special characters within *STRING*. See also Chapter 6.

'

full quoting. [single quote] *'STRING'* preserves all special characters within *STRING*. This is a stronger form of quoting than using ". See also Chapter 6.

,

comma operator. The **comma operator** 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" and calculate "t2".
```

\

escape. [backslash] **\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 6 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. [backticks] `*command*` makes available the output of *command* for setting a variable. This is also known as backticks or backquotes.

:

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

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

Endless loop:

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

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

Placeholder in if/then test:

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

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

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

Provide a placeholder where a command is expected in a here document. See Example 17-9.

Evaluate string of variables using parameter substitution (as in Example 9-12).

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

Variable expansion / substring replacement.

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

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

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

See also Example 12-11.

In combination with the >> redirection operator, updates a file access/modification time (**`: >> new_file`**). If the file did not previously exist, creates it. This is equivalent to touch.

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

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

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

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

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

!

reverse (or negate) the sense of a test or exit status. The ! operator inverts the exit status of the command to which it is applied (see Example 3-2). It also inverts the meaning of a test operator. This can, for example, change the sense of "equal" (=) to "not-equal" (!=). The ! operator is a Bash keyword.

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

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

*

wild card. [asterisk] The * character serves as a "wild card" for filename expansion in globbing, as well as representing any number (or zero) characters in a regular expression.

*

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

A double asterisk, **, is the exponentiation operator.

?

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

In a double parentheses construct, the ? serves as a C-style trinary operator. See Example 9-25.

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

?

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

$

Variable substitution.

```
var1=5
var2=23skidoo

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

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

$

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

${}

Parameter substitution.

$*, $@

positional parameters.

$?

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

$$

process id variable. The $$ variable holds the *process id* of the script in which it appears.

()

command group.

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

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

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

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

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

array initialization.

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

{xxx,yyy,zzz,...}

Brace expansion.

```
grep Linux file*.{txt,htm*}
# Finds all instances of the word "Linux"
# in the files "fileA.txt", "file2.txt", "fileR.html", "file-87.htm", etc.
```

A command may act upon a comma-separated list of file specs within *braces*. [1] Filename expansion (globbing) applies to the file specs between the braces.

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

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

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

{}

Block of code. [curly brackets] Also referred to as an "inline group", this construct, in effect, creates an anonymous function.

However, unlike a function, the variables in a code block remain visible to the remainder of the script.

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

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

# Thanks, S.C.
```

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

Example 4-1. Code blocks and I/O redirection

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

File=/etc/fstab

{
read line1
read line2
} < $File

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

exit 0
```

Example 4-2. Saving the results of a code block to a file

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

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

SUCCESS=0
E_NOARGS=65

if [ -z "$1" ]
then
  echo "Usage: `basename $0` rpm-file"
  exit $E_NOARGS
```

```
fi

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

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

# See rpm man page for explanation of options.

exit 0
```

Unlike a command group within (parentheses), as above, a code block enclosed by {braces} will *not* normally launch a subshell. [2]

{} \;

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

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

[ ]

test.

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

[[ ]]

test.

Test expression between [[ ]] (shell keyword).

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

[ ]

array element.

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

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

[ ]

range of characters.

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

(( ))

integer expansion.

Expand and evaluate integer expression between (( )).

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

> &> >& >> <

redirection.

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

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

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

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

process substitution.

`(command)>`

`<(command)`

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

In yet another context, the "<" and ">" characters act as integer comparison operators. See also Example 12-6.

<<

redirection used in a here document.

<, >

ASCII comparison.

```
veg1=carrots
veg2=tomatoes

if [[ "$veg1" < "$veg2" ]]
then
  echo "Although $veg1 precede $veg2 in the dictionary,"
  echo "this implies nothing about my culinary preferences."
else
  echo "What kind of dictionary are you using, anyhow?"
fi
```

\<, \>

word boundary in a regular expression.

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

|

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

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


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

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

```
cat $filename | grep $search_word
```

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

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

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

exit 0
```

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

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

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

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

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

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

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

>|

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

||

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

&

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

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

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

Example 4-3. Running a loop in the background

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

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

echo   # This 'echo' sometimes will not display.

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

echo   # This 'echo' sometimes will not display.

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

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

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

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

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

exit 0
```

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

&&

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

-

option, prefix. Option flag for a command or filter. Prefix for an operator.

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

```
ls -al
```

sort -dfu $filename

set -- $variable

```
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
```

-

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

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

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



# More elegant than, but equivalent to:
#   cd source-directory
#   tar cf - . | (cd ../target-directory; tar xzf -)
#
```

```
# cp -a /source/directory /dest     also has same effect.
```

```
bunzip2 linux-2.4.3.tar.bz2 | tar xvf -
# --uncompress tar file--    | --then pass it to "tar"--
# If "tar" has not been patched to handle "bunzip2",
# this needs to be done in two discrete steps, using a pipe.
# The purpose of the exercise is to unarchive "bzipped" kernel source.
```

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

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

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

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

By itself on the command line, filefails with an error message.

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

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


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

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

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

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

```
grep Linux file1 | diff file2 -
```

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

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

```
#!/bin/bash

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

NOARGS=0
E_BADARGS=65

if [ $# = $NOARGS ]
then
  echo "Usage: `basename $0` filename"
  exit $E_BADARGS
fi

tar cvf - `find . -mtime -1 -type f -print` > $1.tar
gzip $1.tar


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

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


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


exit 0
```

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

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

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

-

previous working directory. [dash] **cd -** changes to the previous working directory. This uses the $OLDPWD environmental variable.

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

-

Minus. Minus sign in an arithmetic operation.

=

Equals. Assignment operator

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

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

+

Plus. Addition arithmetic operator.

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

+

Option. Option flag for a command or filter.

Certain commands and builtins use the + to enable certain options and the - to disable them.

%

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

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.

^

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

Control Characters

change the behavior of the terminal or text display. A control character is a **CONTROL** + **key** combination.

- **Ctl-C**

  Terminate a foreground job.

- **Ctl-D**

  Log out from a shell (similar to exit).

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

- **Ctl-G**

  "BEL" (beep).

- **Ctl-H**

  Backspace.

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

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

- **Ctl-J**

  Carriage return.

- **Ctl-L**

Formfeed (clear the terminal screen). This has the same effect as the clear command.

- **Ctl-M**

  Newline.

- **Ctl-U**

  Erase a line of input.

- **Ctl-Z**

  Pause a foreground job.

Whitespace

functions as a separator, separating commands or variables. Whitespace consists of either spaces, tabs, blank lines, or any combination thereof. 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, defaults to whitespace.

Notes

[1] The shell does the *brace expansion*. The command itself acts upon the *result* of the expansion.

[2] Exception: a code block in braces as part of a pipe *may* be run as a subshell.

```
ls | { read firstline; read secondline; }
# Error. The code block in braces runs as a subshell,
# so the output of "ls" cannot be passed to variables within the block.
echo "First line is $firstline; second line is $secondline"  # Will not work.

# Thanks, S.C.
```

Chapter 26. 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 (find the contents of) an array element, use *curly bracket* notation, that is, **${variable[xx]}**.

Example 26-1. 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 o.k.


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.
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
```

Array variables have a syntax all their own, and even standard Bash commands and operators have special options adapted for array use.

```
array=( zero one two three four five )

echo ${array[0]}       #  zero
echo ${array:0}        #  zero
                       #  Parameter expansion of first element.
echo ${array:1}        #  ero
                       #  Parameter expansion of first element,
                       #+ starting at position #1 (2nd character).

echo ${#array}         #  4
                       #  Length of first element of array.
```

In an array context, some Bash builtins have a slightly altered meaning. For example, unset deletes array elements, or even an entire array.

Example 26-2. Some special properties of arrays

```
#!/bin/bash

declare -a colors
# Permits declaring an array without specifying its size.

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).

index=0

while [ "$index" -lt "$element_count" ]
do    # List all the elements in the array.
  echo ${colors[$index]}
  let "index = $index + 1"
done
# Each array element listed on a separate line.
# If this is not desired, use  echo -n "${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 26-3. Of empty arrays and empty elements

```
#!/bin/bash
# empty-array.sh

# An empty array is not the same as an array with empty elements.

array0=( first second third )
array1=( '' )   # "array1" has one empty element.
array2=( )      # No elements... "array2" is empty.

echo

echo "Elements in array0:  ${array0[@]}"
echo "Elements in array1:  ${array1[@]}"
echo "Elements in array2:  ${array2[@]}"
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
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

exit 0  # Thanks, S.C.
```

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[@]}"

# Adding an element to an array.
array=( "${array[@]}" "new element" )
# or
array[${#array[*]}]="new element"

# Thanks, S.C.
```

The **array=(element1 element2 ... elementN)** initialization operation, with the help of command substitution, makes it possible to load the contents of a text file into an array.

```
#!/bin/bash

filename=sample_file

#            cat sample_file
#
#            1 a b c
#            2 d e fg


declare -a array1

array1=( `cat "$filename" | tr '\n' ' '`)  # Loads contents
                                          # of $filename into array1.
#         list file to stdout.
#                          change linefeeds in file 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
```

Arrays permit deploying old familiar algorithms as shell scripts. Whether this is necessarily a good idea is left to the reader to decide.

Example 26-4. An old friend: *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"
  done # End of inner loop


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
```

--

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 26-5. Complex array application: *Sieve of Eratosthenes*

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

# Sieve of Eratosthenes
# Ancient algorithm for finding prime numbers.

# This runs a couple of orders of magnitude
# slower than the equivalent C program.

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.


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
# We know 1 is prime, so 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


}


# Invoke the functions sequentially.
```

```
initialize
sift
print_primes
# This is what they call structured programming.

echo

exit 0



# ----------------------------------------------- #
# Code below line will not execute.

# 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 0
```

Compare this array-based prime number generator with an alternative that does not use arrays, Example A-16.

--

Arrays lend themselves, to some extent, to emulating data structures for which Bash has no native support.

Example 26-6. 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 local variable,
                  #+ because of limitation on function return range.

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=FINAL; push $value3

pop               # FINAL
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
# ==========================================================


# 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)  Using this script as a jumping-off point,
#   + write a stack-based 4-function calculator.

exit 0
```

--

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 26-7. 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 conditions.
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 + $T1`      # Q[n - Q[n-1]] + Q[n - ![n-2]]
echo -n "${Q[n]} "

if [ `expr $n % $LINEWIDTH` -eq 0 ]    # Format output.
then   #     mod
  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.
```

--

Bash supports only one-dimensional arrays, however a little trickery permits simulating multi-dimensional ones.

Example 26-8. Simulating a two-dimensional array, then tilting it

```
#!/bin/bash
# Simulating a two-dimensional array.

# A two-dimensional array stores rows sequentially.

Rows=5
Columns=5

declare -a alpha     # char alpha [Rows] [Columns];
                     # Unnecessary declaration.

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
  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
```

```
  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.

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-- ))  # Step through the array backwards.
do

  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.
```

```
  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).

}


#--------------------------------------------------#
load_alpha     # Load the array.
print_alpha    # Print it out.
rotate         # Rotate it 45 degrees counterclockwise.
#--------------------------------------------------#


# This is a rather contrived, not to mention kludgy simulation.
#
# Exercises:
# ---------
# 1)  Rewrite the array loading and printing functions
#   + in a more intuitive and elegant fashion.
#
# 2)  Figure out how the array rotation functions work.
#     Hint: think about the implications of backwards-indexing an array.

exit 0
```

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.

Chapter 16. I/O Redirection

Table of Contents

There are always three default "files" 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 4-1 and Example 4-2) and sending it as input to another file, command, program, or script.

Each open file gets assigned a file descriptor. [1] 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. [2] This simplifies restoration to normal after complex redirection and reshuffling (see Example 16-1).

```
COMMAND_OUTPUT >
   # Redirect stdout to a file.
   # Creates the file if not present, otherwise overwrites it.

   ls -lR > dir-tree.list
   # Creates a file containing a listing of the directory tree.

: > filename
   # The > truncates file "filename" to zero length.
   # If file not present, creates zero-length file (same effect as 'touch').
   # The : serves as a dummy placeholder, producing no output.

> filename
   # The > truncates file "filename" to zero length.
   # If file not present, creates zero-length file (same effect as 'touch').
   # (Same result as ": >", above, but this does not work with some shells.)

COMMAND_OUTPUT >>
   # Redirect stdout to a file.
   # Creates the file if not present, otherwise appends to it.


   # Single-line redirection commands (affect only the line they are on):
   # --------------------------------------------------------------------

1>filename
   # Redirect stdout to file "filename".
1>>filename
   # Redirect and append stdout to file "filename".
2>filename
   # Redirect stderr to file "filename".
2>>filename
   # Redirect and append stderr to file "filename".
```

```
   &>filename
      # Redirect both stdout and stderr to file "filename".

      #==============================================================================
      # 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.

   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

command1 | command2 | command3 > output-file
```

See Example 12-23 and Example A-15.

Multiple output streams may be redirected to one file.

```
ls -yz >> command.log 2>&1
# Capture result of illegal options "yz" to "ls" in file "command.log".
# Because stderr redirected to the file, any error messages will also be there.
```

Closing File Descriptors

n<&-

Close input file descriptor *n*.

0<&-, <&-

Close `stdin`.

n>&-

Close output file descriptor *n*.

1>&-, >&-

Close `stdout`.

Child processes inherit open file descriptors. This is why pipes work. To prevent an fd from being inherited, close it.

```
# Redirecting only stderr to a pipe.

exec 3>&1                                # Save current "value" of stdout.
ls -l 2>&1 >&3 3>&- | grep bad 3>&-    # Close fd 3 for 'grep' (but not 'ls').
#              ^^^^   ^^^^
exec 3>&-                                # Now close it for the remainder of the
script.

# Thanks, S.C.
```

For a more detailed introduction to I/O redirection see Appendix D.

Notes

[1] 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 version of a file pointer. It is analogous to a *file handle* in C.

[2] Using *`file descriptor 5`* might cause problems. When Bash creates a child process, as with exec, the child inherits fd 5 (see Chet Ramey's archived e-mail, SUBJECT: RE: File descriptor 5 is held open). Best leave this particular fd alone.

Chapter 32. Gotchas

Turandot: Gli enigmi sono tre, la morte una!

Caleph: No, no! Gli enigmi sono tre, una la vita!

Puccini

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 o.k.

# However...      using just the underscore will not work.
_=25
echo $_           # $_ is a special variable set to last arg of last command.

xyz((!*=value2    # Causes severe problems.
```

Using a hyphen or other reserved characters in a variable name.

```
var-1=23
# Use 'var_1' 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 - $b   # 'let c=$a-$b' or 'let "c = $a - $b"' are correct.

if [ $a -le 5]    # if [ $a -le 5 ]   is correct.
# if [ "$a" -le 5 ]   is even better.
# [[ $a -le 5 ]] also works.
```

Assuming uninitialized variables (variables before a value is assigned to them) are "zeroed out". An uninitialized variable has a value of "null", *not* zero.

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 mix up -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
```

Mixing up integer and string comparison operators.

```
#!/bin/bash
# bad-op.sh

number=1

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: 5: No such file or directory
```

Sometimes variables within "test" brackets ([ ]) need to be quoted (double quotes). Failure to do so may cause unexpected behavior. See Example 7-5, Example 16-4, and Example 9-6.

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, 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.

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.

A shell script headed by **#!/bin/sh** may 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**.

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 32-1. 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"
```

```
# End subshell
)

echo
echo "inner_variable outside subshell = $inner_variable"  # Unset.
echo "outer_variable outside subshell = $outer_variable"  # Unchanged.
echo

exit 0
```

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 11-12).

Example 32-2. Piping the output of echo to a read

```
#!/bin/bash
#  badread.sh:
#  Attempting to use 'echo and 'read'
#+ to assign variables non-interactively.

a=aaa
b=bbb
c=ccc

echo "one two three" | read a b c
# Try to reassign a, b, and c.

echo "a = $a"  # a = aaa
echo "b = $b"  # b = bbb
echo "c = $c"  # c = ccc
# Reassignment failed.

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

# 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.

exit 0
```

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 scripts written for Linux or BSD systems may need fixups to run on a commercial UNIX machine. Such scripts often employ GNU commands and filters which have greater functionality than their generic UNIX counterparts. This is particularly true of such text processing utilites as tr.

Danger is near thee --

Beware, beware, beware, beware.

Many brave hearts are asleep in the deep.

So beware --

Beware.

A.J. Lamb and H.W. Petrie

Notes

[1] Setting the *suid* permission on the script itself has no effect.

Chapter 31. Options

Options are settings that change shell and/or script behavior.

The set command enables options within a script. At the point in the script where you want the options to take effect, use **set -o option-name** or, in short form, **set -option-abbrev**. These two forms are equivalent.

```
#!/bin/bash

set -o verbose
# Echoes all commands before executing.
```

```
#!/bin/bash

set -v
# Exact same effect as above.
```

To *disable* an option within a script, use **set +o option-name** or **set +option-abbrev**.

```
#!/bin/bash

set -o verbose
# Command echoing on.
command
...
command

set +o verbose
# Command echoing off.
command
# Not echoed.


set -v
# Command echoing on.
command
...
command

set +v
# Command echoing off.
command

exit 0
```

An alternate method of enabling options in a script is to specify them immediately following the *#!* script header.

```
#!/bin/bash -x
#
# Body of script follows.
```

It is also possible to enable script options from the command line. Some options that will not work with **set** are available this way. Among these are *-i*, force script to run interactive.

```
bash -v script-name
```

```
bash -o verbose script-name
```

The following is a listing of some useful options. They may be specified in either abbreviated form or by complete name.

Table 31-1. bash options

Abbreviation	Name	Effect
`-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 **...**
`-f`	noglob	Filename expansion (globbing) disabled
`-i`	interactive	Script runs in *interactive* mode
`-p`	privileged	Script runs as "suid" (caution!)
`-r`	restricted	Script runs in *restricted* mode (see Chapter 21).
`-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
`-e`	errexit	Abort script at first error (when a command exits with non-zero status)
`-n`	noexec	Read commands in script, but do not execute them (syntax check)
`-s`	stdin	Read commands from `stdin`
`-t`	(none)	Exit after first command

-	(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.

5.4. Special Variable Types

local variables

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

environmental variables

variables that affect the behavior of the shell and user interface

In a more general context, each process has an "environment", that is, a group of variables that hold information 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 shell variables causes the shell to update its environment, and all the shell's child processes (the commands it executes) inherit this environment.

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

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

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

(Thank you, S. C. for the clarification, and for providing the above example.)

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

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

---

positional parameters

arguments passed to the script from the command line - $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. [1] After $9, the arguments must be enclosed in brackets, for example, ${10}, ${11}, ${12}.

Example 5-5. Positional Parameters

```
#!/bin/bash

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

echo

echo "The name of this script is \"$0\"."
# Adds ./ for current directory
echo "The name of this script is \"`basename $0`\"."
# Strips out path name info (see 'basename')

echo

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

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

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

# ...

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

echo

exit 0
```

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. There must also exist symbolic links to all the alternate names of the script.

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_
# This will prevent an error, even if positional parameter is absent.

critical_argument01=$variable1_

# The extra character can be stripped off later, if desired, like so.
variable1=${variable1_/_/}   # Side effects only if $variable1_ begins with "_".
# This uses one of the parameter substitution templates discussed in Chapter 9.
# Leaving out the replacement pattern results in a deletion.

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

Example 5-6. wh, whois domain name lookup

```
#!/bin/bash

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

# Place this script, named '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-cw
# ln -s /usr/local/bin/wh /usr/local/bin/wh-radb


if [ -z "$1" ]
then
  echo "Usage: `basename $0` [domain-name]"
  exit 65
fi

case `basename $0` in
# Checks script name and calls proper server
    "wh"     ) whois $1@whois.ripe.net;;
    "wh-ripe") whois $1@whois.ripe.net;;
    "wh-radb") whois $1@whois.radb.net;;
    "wh-cw"  ) whois $1@whois.cw.net;;
    *        ) echo "Usage: `basename $0` [domain-name]";;
esac

exit 0
```

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 5-7. Using shift

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

#  Name this script something like shft,
#+ and invoke it with some parameters, for example
#          ./shft a b c def 23 skidoo

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

echo               # Extra line feed.

exit 0
```

The **shift** command also works on parameters passed to a function. See Example 34-6.

Notes

[1] The process calling the script sets the *$0* parameter. By convention, this parameter is the name of the script. See the manpage for **execv**.

12.9. Miscellaneous Commands

Command that fit in no special category

jot, **seq**

These utilities emit a sequence of integers, with a user-selected increment.

The normal separator character between each integer is a newline, but this can be changed with the `-s` option.

```
bash$ seq 5
1
 2
 3
 4
 5


bash$ seq -s : 5
1:2:3:4:5
```

Both **jot** and **seq** come in handy in a for loop.

Example 12-38. Using seq to generate loop arguments

```
#!/bin/bash

for a in `seq 80`  # or   for a in $( seq 80 )
# Same as   for a in 1 2 3 4 5 ... 80   (saves much typing!).
# May also use 'jot' (if present on system).
do
  echo -n "$a "
done
# Example of using the output of a command to generate
# the [list] in a "for" loop.

echo; echo


COUNT=80  # Yes, 'seq' may also take a replaceable parameter.

for a in `seq $COUNT`  # or   for a in $( seq $COUNT )
do
  echo -n "$a "
done

echo
```

```
exit 0
```

getopt

The **getopt** command parses command-line options preceded by a dash. This external command corresponds to the getopts Bash builtin, but it is not nearly as flexible.

Example 12-39. Using getopt to parse command-line options

```
#!/bin/bash

# Try the following when invoking this script.
#   sh ex33a -a
#   sh ex33a -abc
#   sh ex33a -a -b -c
#   sh ex33a -d
#   sh ex33a -dXYZ
#   sh ex33a -d XYZ
#   sh ex33a -abcd
#   sh ex33a -abcdZ
#   sh ex33a -z
#   sh ex33a a
# Explain the results of each of the above.

E_OPTERR=65

if [ "$#" -eq 0 ]
then   # Script needs at least one command-line argument.
  echo "Usage $0 -[options a,b,c]"
  exit $E_OPTERR
fi

set -- `getopt "abcd:" "$@"`
# Sets positional parameters to command-line arguments.
# What happens if you use "$*" instead of "$@"?

while [ ! -z "$1" ]
do
  case "$1" in
    -a) echo "Option \"a\"";;
    -b) echo "Option \"b\"";;
    -c) echo "Option \"c\"";;
    -d) echo "Option \"d\" $2";;
     *) break;;
  esac

  shift
done

#  It is better to use the 'getopts' builtin in a script,
#+ rather than 'getopt'.
#  See "ex33.sh".

exit 0
```

run-parts

The **run-parts** command [1] executes all the scripts in a target directory, sequentially in ASCII-sorted filename order. Of course, the scripts need to have execute permission.

The crond daemon invokes **run-parts** to run the scripts in the `/etc/cron.*` directories.

yes

In its default behavior the **yes** command feeds a continuous string of the character `y` followed by a line feed to `stdout`. A **control-c** terminates the run. A different output string may be specified, as in **`yes different string`**, which would continually output `different string` to `stdout`. One might well ask the purpose of this. From the command line or in a script, the output of **yes** can be redirected or piped into a program expecting user input. In effect, this becomes a sort of poor man's version of **expect**.

`yes | fsck /dev/hda1` runs **fsck** non-interactively (careful!).

`yes | rm -r dirname` has same effect as **`rm -rf dirname`** (careful!).

Be very cautious when piping **yes** to a potentially dangerous system command, such as fsck or fdisk.

banner

Prints arguments as a large vertical banner to `stdout`, using an ASCII character (default '#'). This may be redirected to a printer for hardcopy.

printenv

Show all the environmental variables set for a particular user.

```
bash$ printenv | grep HOME
HOME=/home/bozo
```

lp

The **lp** and **lpr** commands send file(s) to the print queue, to be printed as hard copy. [2] These commands trace the origin of their names to the line printers of another era.

`bash$ lp file1.txt` or `bash lp <file1.txt`

It is often useful to pipe the formatted output from **pr** to **lp**.

```
bash$ pr -options file1.txt | lp
```

Formatting packages, such as **groff** and *Ghostscript* may send their output directly to **lp**.

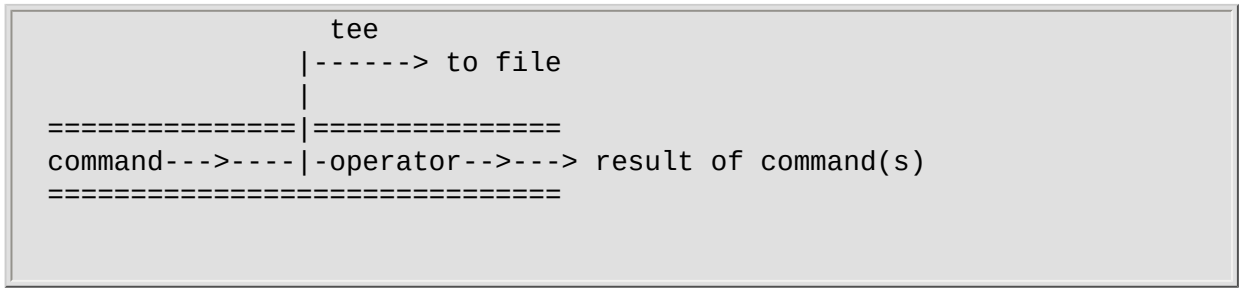
```
bash$ groff -Tascii file.tr | lp

bash$ gs -options | lp file.ps
```

Related commands are **lpq**, for viewing the print queue, and **lprm**, for removing jobs from the print queue.

tee

[UNIX borrows an idea here from the plumbing trade.]

This is a redirection operator, but with a difference. Like the plumber's *tee*, it permits "siponing off" the output of a command or commands within a pipe, but without affecting the result. This is useful for printing an ongoing process to a file or paper, perhaps to keep track of it for debugging purposes.




```
cat listfile* | sort | tee check.file | uniq > result.file
```

(The file `check.file` contains the concatenated sorted "listfiles", before the duplicate lines are removed by uniq.)

mkfifo

This obscure command creates a *named pipe*, a temporary *first-in-first-out buffer* for transferring data between processes. [3] Typically, one process writes to the FIFO, and the other reads from it. See Example A-15.

pathchk

This command checks the validity of a filename. If the filename exceeds the maximum allowable length (255 characters) or one or more of the directories in its path is not searchable, then an error message results. Unfortunately, **pathchk** does not return a recognizable error code, and it is therefore pretty much useless in a script.

dd

This is the somewhat obscure and much feared "data duplicator" command. Originally a utility for exchanging data on magnetic tapes between UNIX minicomputers and IBM mainframes, this command still has its uses. The **dd** command simply copies a file (or `stdin/stdout`), but with conversions. Possible conversions are ASCII/EBCDIC, [4] upper/lower case, swapping of byte pairs between input and output, and skipping and/or truncating the head or tail of the input file. A **`dd --help`** lists the conversion and other options that this powerful utility takes.

```
# Exercising 'dd'.

n=3
p=5
input_file=project.txt
output_file=log.txt

dd if=$input_file of=$output_file bs=1 skip=$((n-1)) count=$((p-n+1)) 2> /dev/null
# Extracts characters n to p from file $input_file.



echo -n "hello world" | dd cbs=1 conv=unblock 2> /dev/null
# Echoes "hello world" vertically.


# Thanks, S.C.
```

To demonstrate just how versatile **dd** is, let's use it to capture keystrokes.

Example 12-40. Capturing Keystrokes

```
#!/bin/bash
# Capture keystrokes without needing to press ENTER.


keypresses=4                      # Number of keypresses to capture.


old_tty_setting=$(stty -g)        # Save old terminal settings.

echo "Press $keypresses keys."
stty -icanon -echo                # Disable canonical mode.
                                  # Disable local echo.
keys=$(dd bs=1 count=$keypresses 2> /dev/null)
# 'dd' uses stdin, if "if" not specified.

stty "$old_tty_setting"           # Restore old terminal settings.

echo "You pressed the \"$keys\" keys."

# Thanks, S.C. for showing the way.
exit 0
```

The **dd** command can do random access on a data stream.

```
echo -n . | dd bs=1 seek=4 of=file conv=notrunc
# The "conv=notrunc" option means that the output file will not be truncated.

# Thanks, S.C.
```

The **dd** command can copy raw data and disk images to and from devices, such as floppies and tape drives (Example A-6). A common use is creating boot floppies.

```
dd if=kernel-image of=/dev/fd0H1440
```

Similarly, **dd** can copy the entire contents of a floppy, even one formatted with a "foreign" OS, to the hard drive as an image file.

```
dd if=/dev/fd0 of=/home/bozo/projects/floppy.img
```

Other applications of **dd** include initializing temporary swap files (Example 29-2) and ramdisks (Example 29-3). It can even do a low-level copy of an entire hard drive partition, although this is not necessarily recommended.

People (with presumably nothing better to do with their time) are constantly thinking of interesting applications of **dd**.

Example 12-41. Securely deleting a file

```
#!/bin/bash
# blotout.sh: Erase all traces of a file.

#  This script overwrites a target file alternately
#+ with random bytes, then zeros before finally deleting it.
#  After that, even examining the raw disk sectors
#+ will not reveal the original file data.

PASSES=7         # Number of file-shredding passes.
BLOCKSIZE=1      #  I/O with /dev/urandom requires unit block size,
                 #+ otherwise you get weird results.
E_BADARGS=70
E_NOT_FOUND=71
E_CHANGED_MIND=72

if [ -z "$1" ]   # No filename specified.
then
  echo "Usage: `basename $0` filename"
  exit $E_BADARGS
fi

file=$1

if [ ! -e "$file" ]
then
  echo "File \"$file\" not found."
  exit $E_NOT_FOUND
fi

echo; echo -n "Are you absolutely sure you want to blot out \"$file\" (y/n)? "
read answer
case "$answer" in
[nN]) echo "Changed your mind, huh?"
      exit $E_CHANGED_MIND
      ;;
*)    echo "Blotting out file \"$file\".";;
esac


flength=$(ls -l "$file" | awk '{print $5}')  # Field 5 is file length.

pass_count=1

echo

while [ "$pass_count" -le "$PASSES" ]
do
  echo "Pass #$pass_count"
  sync         # Flush buffers.
  dd if=/dev/urandom of=$file bs=$BLOCKSIZE count=$flength
               # Fill with random bytes.
  sync         # Flush buffers again.
  dd if=/dev/zero of=$file bs=$BLOCKSIZE count=$flength
               # Fill with zeros.
  sync         # Flush buffers yet again.
  let "pass_count += 1"
  echo
done
```

```
rm -f $file    # Finally, delete scrambled and shredded file.
sync           # Flush buffers a final time.

echo "File \"$file\" blotted out and deleted."; echo


#  This is a fairly secure, if inefficient and slow method
#+ of thoroughly "shredding" a file. The "shred" command,
#+ part of the GNU "fileutils" package, does the same thing,
#+ but more efficiently.

#  The file cannot not be "undeleted" or retrieved by normal methods.
#  However...
#+ this simple method will likely *not* withstand forensic analysis.


#  Tom Vier's "wipe" file-deletion package does a much more thorough job
#+ of file shredding than this simple script.
#     http://www.ibiblio.org/pub/Linux/utils/file/wipe-2.0.0.tar.bz2

#  For an in-depth analysis on the topic of file deletion and security,
#+ see Peter Gutmann's paper,
#+     "Secure Deletion of Data From Magnetic and Solid-State Memory".
#           http://www.cs.auckland.ac.nz/~pgut001/secure_del.html


exit 0
```

od

The **od**, or *octal dump* filter converts input (or files) to octal (base-8) or other bases. This is useful for viewing or processing binary data files or otherwise unreadable system device files, such as `/dev/urandom`, and as a filter for binary data. See Example 9-23 and Example 12-10.

hexdump

Performs a hexadecimal, octal, decimal, or ASCII dump of a binary file. This command is the rough equivalent of **od**, above, but not nearly as useful.

mcookie

This command generates a "magic cookie", a 128-bit (32-character) pseudorandom hexadecimal number, normally used as an authorization "signature" by the X server. This also available for use in a script as a "quick 'n dirty" random number.

```
random000=`mcookie | sed -e '2p'`
# Uses 'sed' to strip off extraneous characters.
```

Of course, a script could use md5 for the same purpose.

```
# Generate md5 checksum on the script itself.
random001=`md5sum $0 | awk '{print $1}'`
# Uses 'awk' to strip off the filename.
```

m4

A hidden treasure, **m4** is a powerful macro processing filter, [5] virtually a complete language. Although originally written as a pre-processor for *RatFor*, **m4** turned out to be useful as a stand-alone utility. In fact, **m4** combines some of the functionality of eval, tr, and awk, in addition to its extensive macro expansion facilities.

The April, 2002 issue of *Linux Journal* has a very nice article on **m4** and its uses.

Example 12-42. Using m4

```
#!/bin/bash
# m4.sh: Using the m4 macro processor

# Strings
string=abcdA01
echo "len($string)" | m4                           # 7
echo "substr($string,4)" | m4                      # A01
echo "regexp($string,[0-1][0-1],\&Z)" | m4     # 01Z

# Arithmetic
echo "incr(22)" | m4                               # 23
echo "eval(99 / 3)" | m4                           # 33

exit 0
```

Notes

[1] This is actually a script adapted from the Debian Linux distribution.

[2] The *print queue* is the group of jobs "waiting in line" to be printed.

[3] For an excellent overview of this topic, see Andy Vaught's article, Introduction to Named Pipes, in the September, 1997 issue of Linux Journal.

[4] EBCDIC (pronounced "ebb-sid-ic") is an acronym for Extended Binary Coded Decimal Interchange Code. This is an IBM data format no longer in much use. A bizarre application of the `conv=ebcdic` option of **dd** is as a quick 'n easy, but not very secure text file encoder.

```
cat $file | dd conv=swab,ebcdic > $file_encrypted
# Encode (looks like gibberish).
# Might as well switch bytes (swab), too, for a little extra obscurity.

cat $file_encrypted | dd conv=swab,ascii > $file_plaintext
# Decode.
```

[5] A *macro* is a symbolic constant that expands into a command string or a set of operations on parameters.

Chapter 27. 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 G 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.

Notes

[1] This does not apply to **csh**, **tcsh**, and other shells not related to or descended from the classic Bourne shell (**sh**).

Prev Arrays | Home Up | Next /dev and /proc

B.2. Awk

Awk 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 couple of these here - the ones most useful for shell scripting.

Awk breaks each line of input passed to it into *fields*. By default, a field is a string of consecutive characters separated by whitespace, though there are options for changing the delimiter. Awk parses and operates on each separate field. This makes awk ideal for handling structured text files, especially tables, data organized into consistent chunks, such as rows and columns.

Strong quoting (single quotes) and curly brackets enclose segments of awk code within a shell script.

```
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.
```

We have just seen the awk **print** command in action. The only other feature of awk we need to deal with here is variables. Awk handles variables similarly to shell scripts, though a bit more flexibly.

```
{ total += ${column_number} }
```

This adds the value of *column_number* to the running total of "total". Finally, to print "total", there is an **END** command block, executed after the script has processed all its input.

```
END { print total }
```

Corresponding to the **END**, there is a **BEGIN**, for a code block to be performed before awk starts processing its input.

For examples of awk within shell scripts, see:

1. Example 11-9
2. Example 16-7
3. Example 12-24
4. Example 34-3
5. Example 9-20
6. Example 11-14
7. Example 28-1
8. Example 28-2
9. Example 10-3
10. Example 12-41
11. Example 9-23
12. Example 12-3
13. Example 9-11
14. Example 34-7
15. Example 10-8

That's all the awk we'll cover here, folks, but there's lots more to learn. See the appropriate references in the *Bibliography*.

9.4. Typing variables: declare or typeset

The **declare** or **typeset** builtins (they are exact synonyms) permit restricting the properties of variables. This is a very weak form of the typing 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
```

(**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.

-i *integer*

```
declare -i number
# The script will treat subsequent occurrences of "number" as an integer.

number=3
echo "number = $number"     # number = 3

number=three
echo "number = $number"     # number = 0
# Tries to evaluate "three" as an integer.
```

Note that certain arithmetic operations are permitted for declared integer variables without the need for expr or let.

-a *array*

```
declare -a indices
```

The variable `indices` will be treated as an array.

-f *functions*

```
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.

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 9-18. 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.
```

Chapter 3. Exit and Exit Status

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

Chet Ramey

The **exit** command may be used to terminate 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*). A successful command returns a 0, while an unsuccessful one returns a non-zero value that usually may 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 a decimal number in the 0 - 255 range).

When a script ends with an **exit** that has no parameter, the exit status of the script is the exit status of the last command executed in the script (*not* counting the **exit**).

$? 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". 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 3-1. 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 because 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.
```

$? is especially useful for testing the result of a command in a script (see Example 12-27 and Example 12-13).

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

Example 3-2. 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.
#    !true   leads to a "command not found" error

# Thanks, S.C.
```

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

16.1. Using exec

An **exec <filename** command redirects `stdin` to a file. From that point on, all `stdin` comes from that file, rather than its normal source (usually keyboard input). This provides a method of reading a file line by line and possibly parsing each line of input using sed and/or awk.

Example 16-1. 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.

Example 16-2. 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 16-3. 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.

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.
# -----------------------------------------------

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
```

Prev
I/O Redirection

Home
Up

Next
Redirecting Code Blocks

Chapter 24. 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** 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 losing important files.

In a script, aliases have very limited usefulness. It would be quite 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. [1] 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 24-1. Aliases within a script

```
#!/bin/bash
# Invoke with command line parameter to exercise last section of this script.

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.
# Alias handles 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.
  let count+=1
done

echo; echo

alias xyz="cat $1"   #  Try a positional parameter in an alias.
xyz                  #  Assumes you invoke the script
                     #+ with a filename as a parameter.
#  This seems to work,
#+ although the Bash documentation suggests that it shouldn't.
#
#  However, as Steve Jacobson points out,
#+ the "$1" parameter expands immediately upon declaration of the alias,
#+ so, in the strictest sense, this is not an example
#+ of parameterizing an alias.

exit 0
```

The **unalias** command removes a previously set *alias*.

Example 24-2. unalias: Setting and unsetting an alias

```
#!/bin/bash

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
```

Notes

[1] However, aliases do seem to expand positional parameters.

Chapter 35. Bash, version 2

The current version of *Bash*, the one you have running on your machine, is actually version 2.XX.Y.

```
bash$ echo $BASH_VERSION
2.05.8(1)-release
```

This update of the classic Bash scripting language added array variables, [1] string and parameter expansion, and a better method of indirect variable references, among other features.

Example 35-1. 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'
echo $'Three form feeds \f \f \f'
echo $'10 newlines \n\n\n\n\n\n\n\n\n\n'

exit 0
```

Example 35-2. 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 ${!variable} notation is greatly superior to the old "eval var1=\$$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

# This is useful for referencing members of an array or table,
# or for simulating a multi-dimensional array.
# An indexing option would have been nice (sigh).

exit 0
```

Example 35-3. Simple database application, using indirect variable referencing

```
#!/bin/bash
# resistor-inventory.sh
# Simple database application using indirect variable referencing.

# ============================================================== #
# 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=.25
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:"
  echo "There are ${!Inv} of [${!Val} ohm / ${!Pdissip} watt] resistors in stock."
  echo "These are located in bin # ${!Loc}."
  echo "Their color code is \"${!Ccode}\"."

  break
done

echo; echo

# Exercise:
# --------
# Rewrite this script using arrays, rather than indirect variable referencing.
# Which method is more straightforward and intuitive?


# 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 35-4. Using arrays and other miscellaneous trickery to deal four random hands from a deck of cards

```
#!/bin/bash
# May need to be invoked with  #!/bin/bash2  on older machines.

# Cards:
# 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 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 %= $CARDS"
if [ "${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.
{
seed=`eval date +%s`
let "seed %= 32766"
RANDOM=$seed
}
```

```
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).
    let "u %= $CARDS_IN_SUIT"
    if [ "$u" -eq 0 ]   # Nested if/then condition test.
    then
     echo
     echo
    fi
    # Separate hands.

    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 0
#================



# Exercise 1:
# Add comments to thoroughly document this script.

# Exercise 2:
# Revise the script 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.
```

Notes

[1] Chet Ramey promises associative arrays (a Perl feature) in a future Bash release.

Part 3. Beyond the Basics

Table of Contents

19.2. Globbing

Bash itself cannot recognize Regular Expressions. In scripts, commands and utilities, such as sed and awk, interpret RE's.

Bash does carry out filename expansion, a process known as "globbing", but this does *not* use the standard RE set. Instead, globbing recognizes and expands wildcards. Globbing interprets the standard wildcard characters, * and ?, character lists in square brackets, and certain other special characters (such as ^ for negating the sense of a match). There are some important limitations on wildcard characters in globbing, however. Strings containing * will not match filenames that start with a dot, as, for example, `.bashrc`. [1] Likewise, the *?* has a different meaning in globbing than as part of an RE.

```
bash$ ls -l
total 2
 -rw-rw-r--    1 bozo  bozo         0 Aug  6 18:42 a.1
 -rw-rw-r--    1 bozo  bozo         0 Aug  6 18:42 b.1
 -rw-rw-r--    1 bozo  bozo         0 Aug  6 18:42 c.1
 -rw-rw-r--    1 bozo  bozo       466 Aug  6 17:48 t2.sh
 -rw-rw-r--    1 bozo  bozo       758 Jul 30 09:02 test1.txt

bash$ ls -l t?.sh
-rw-rw-r--    1 bozo  bozo       466 Aug  6 17:48 t2.sh

bash$ ls -l [ab]*
-rw-rw-r--    1 bozo  bozo         0 Aug  6 18:42 a.1
 -rw-rw-r--    1 bozo  bozo         0 Aug  6 18:42 b.1

bash$ ls -l [a-c]*
-rw-rw-r--    1 bozo  bozo         0 Aug  6 18:42 a.1
 -rw-rw-r--    1 bozo  bozo         0 Aug  6 18:42 b.1
 -rw-rw-r--    1 bozo  bozo         0 Aug  6 18:42 c.1

bash$ ls -l [^ab]*
-rw-rw-r--    1 bozo  bozo         0 Aug  6 18:42 c.1
 -rw-rw-r--    1 bozo  bozo       466 Aug  6 17:48 t2.sh
 -rw-rw-r--    1 bozo  bozo       758 Jul 30 09:02 test1.txt

bash$ ls -l {b*,c*,*est*}
-rw-rw-r--    1 bozo  bozo         0 Aug  6 18:42 b.1
 -rw-rw-r--    1 bozo  bozo         0 Aug  6 18:42 c.1
 -rw-rw-r--    1 bozo  bozo       758 Jul 30 09:02 test1.txt


bash$ echo *
a.1 b.1 c.1 t2.sh test1.txt

bash$ echo t*
t2.sh test1.txt
```

Even an echo command performs wildcard expansion on filenames.

See also Example 10-4.

Notes

[1] Filename expansion *can* match dotfiles, but only if the pattern explicitly includes the dot.

```
~/[.]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.
```

19.1. A Brief Introduction to Regular Expressions

An expression is a string of characters. Those characters that have 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 for the symbols that follow. Regular Expressions are sets of characters and/or metacharacters that UNIX endows with special features. [1]

The main uses for Regular Expressions (REs) are text searches and string manipulation. An RE *matches* a single character or a set of characters (a substring or an entire string).

- The asterisk -- * -- matches any number of repeats of the character string or RE preceding it, *including zero*.

  "1133*" matches *11 + one or more 3's + possibly other characters*: *113*, *1133*, *111312*, and so forth.
- The dot -- . -- matches any one character, except a newline. [2]

  "13." matches *13 + at least one of any character (including a space)*: *1133*, *11333*, but not *13* (additional character missing).
- The caret -- ^ -- matches the beginning of a line, but sometimes, depending on context, negates the meaning of a set of characters in an RE.
- The dollar sign -- $ -- at the end of an RE matches the end of a line.

  "^$" matches blank lines.
- Brackets -- [...] -- enclose a set of characters to match in a single RE.

  "[xyz]" matches the characters *x*, *y*, or *z*.

  "[c-n]" matches any of the characters in the range *c* to *n*.

  "[B-Pk-y]" matches any of the characters in the ranges *B* to *P* and *k* to *y*.

  "[a-z0-9]" matches any lowercase letter or any digit.

"[^b-d]" matches all characters *except* those in the range *b* to *d*. This is an instance of ^ negating or inverting the meaning of the following RE (taking on a role similar to ! in a different context).

Combined sequences of bracketed characters match common word patterns. "[Yy][Ee][Ss]" matches *yes*, *Yes*, *YES*, *yEs*, and so forth. "[0-9][0-9][0-9]-[0-9][0-9]-[0-9][0-9][0-9][0-9]" matches any Social Security number.

- The backslash -- \ -- escapes a special character, which means that character gets interpreted literally.

  A "\$" reverts back to its literal meaning of "$", rather than its RE meaning of end-of-line. Likewise a "\\" has the literal meaning of "\".

- Escaped "angle brackets" -- \<...\> -- mark word boundaries.

  The angle brackets must be escaped, since otherwise they have only their literal character meaning.

  "\<the\>" matches the word "the", but not the words "them", "there", "other", etc.

```
bash$ cat textfile
This is line 1, of which there is only one instance.
 This is the only instance of line 2.
 This is line 3, another line.
 This is line 4.


bash$ grep 'the' textfile
This is line 1, of which there is only one instance.
 This is the only instance of line 2.
 This is line 3, another line.


bash$ grep '\<the\>' textfile
This is the only instance of line 2.
```

- **Extended REs.** Used in egrep, awk, and Perl

- The question mark -- ? -- matches zero or one of the previous RE. It is generally used for matching

single characters.

- The plus -- + -- matches one or more of the previous RE. It serves a role similar to the *, but does *not* match zero occurrences.

```
# GNU versions of sed and awk can use "+",
# but it needs to be escaped.

echo a111b | sed -ne '/a1\+b/p'
echo a111b | grep 'a1\+b'
echo a111b | gawk '/a1+b/'
# All of above are equivalent.

# Thanks, S.C.
```

- Escaped "curly brackets" -- \{ \} -- indicate the number of occurrences of a preceding RE to match.

  It is necessary to escape the curly brackets since they have only their literal character meaning otherwise. This usage is technically not part of the basic RE set.

  "[0-9]\{5\}" matches exactly five digits (characters in the range of 0 to 9).

  
  Curly brackets are not available as an RE in the "classic" version of awk. However, **gawk** has the `--re-interval` option that permits them (without being escaped).

```
bash$ echo 2222 | gawk --re-interval '/2{3}/'
2222
```

- Parentheses -- **( )** -- enclose groups of REs. They are useful with the following "|" operator and in substring extraction using expr.
- The -- | -- "or" RE operator matches any of a set of alternate characters.

```
bash$ egrep 're(a|e)d' misc.txt
People who read seem to be better informed than those who do not.
 The clarinet produces sound by the vibration of its reed.
```

POSIX Character Classes. `[:class:]`

This is an alternate method of specifying a range of characters to match.

- **`[:alnum:]`** matches alphabetic or numeric characters. This is equivalent to **`[A-Za-z0-9]`**.
- **`[:alpha:]`** matches alphabetic characters. This is equivalent to **`[A-Za-z]`**.
- **`[:blank:]`** matches a space or a tab.
- **`[:cntrl:]`** matches control characters.
- **`[:digit:]`** matches (decimal) digits. This is equivalent to **`[0-9]`**.
- **`[:graph:]`** (graphic printable characters). Matches characters in the range of ASCII 33 - 126. This is the same as **`[:print:]`**, below, but excluding the space character.
- **`[:lower:]`** matches lowercase alphabetic characters. This is equivalent to **`[a-z]`**.
- **`[:print:]`** (printable characters). Matches characters in the range of ASCII 32 - 126. This is the same as **`[:graph:]`**, above, but adding the space character.
- **`[:space:]`** matches whitespace characters (space and horizontal tab).
- **`[:upper:]`** matches uppercase alphabetic characters. This is equivalent to **`[A-Z]`**.
- **`[:xdigit:]`** matches hexadecimal digits. This is equivalent to **`[0-9A-Fa-f]`**.

POSIX character classes generally require quoting or double brackets ([[ ]]).

```
bash$ grep [[:digit:]] test.file
abc=723
```

These character classes may even be used with globbing, to a limited extent.

```
bash$ ls -l ?[[:digit:]][[:digit:]]?
-rw-rw-r--    1 bozo  bozo         0 Aug 21 14:47 a33b
```

To see POSIX character classes used in scripts, refer to Example 12-14 and Example 12-15.

Sed, awk, and Perl, used as filters in scripts, take REs as arguments when "sifting" or transforming files or I/O streams. See Example A-12 and Example A-17 for illustrations of this.

"Sed & Awk", by Dougherty and Robbins gives a very complete and lucid treatment of REs (see the *Bibliography*).

Notes

[1] The simplest type of Regular Expression is a character string that retains its literal meaning, not containing any metacharacters.

[2] Since sed, awk, and grep process single lines, there will usually not be a newline to match. In those cases where there is a newline in a multiple line expression, the dot will match the newline.

```
#!/bin/bash

sed -e 'N;s/.*/[&]/' << EOF   # Here Document
line1
line2
EOF
# OUTPUT:
# [line1
# line2]



echo

awk '{ $0=$1 "\n" $2; if (/line.1/) {print}}' << EOF
line 1
line 2
EOF
# OUTPUT:
# line
# 1


# Thanks, S.C.

exit 0
```

Chapter 20. Subshells

Running a shell script launches another instance of the command processor. Just as your commands are interpreted at the command line prompt, similarly does a script batch process a list of commands in a file. Each shell script running is, in effect, a subprocess of the parent shell, the one that gives you the prompt at the console or in an xterm window.

A shell script can also launch subprocesses. These *subshells* let the script do parallel processing, in effect executing multiple subtasks simultaneously.

Command List in 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, local variables.

Example 20-1. Variable scope in a subshell

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

echo

outer_variable=Outer

(
inner_variable=Inner
echo "From subshell, \"inner_variable\" = $inner_variable"
echo "From subshell, \"outer\" = $outer_variable"
)

echo

if [ -z "$inner_variable" ]
then
  echo "inner_variable undefined in main body of shell"
else
  echo "inner_variable defined in main body of shell"
fi

echo "From main body of shell, \"inner_variable\" = $inner_variable"
# $inner_variable will show as uninitialized because
# variables defined in a subshell are "local variables".
```

```
echo

exit 0
```

See also Example 32-1.

+

Directory changes made in a subshell do not carry over to the parent shell.

Example 20-2. 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
```

One application of this is testing whether a variable is defined.

```
if (set -u; : $variable) 2> /dev/null
then
  echo "Variable is set."
fi

# Could also be written [[ ${variable-x} != x || ${variable-y} != y ]]
# or                    [[ ${variable-x} != x$variable ]]
# or                    [[ ${variable+x} = x ]])
```

Another application is checking for a lock file:

```
if (set -C; : > lock_file) 2> /dev/null
then
  echo "Another user is already running that script."
  exit 65
fi

# Thanks, S.C.
```

Processes may execute in parallel within different subshells. This permits breaking a complex task into subcomponents processed concurrently.

Example 20-3. Running parallel processes in subshells

```
        (cat list1 list2 list3 | sort | uniq > list123) &
        (cat list4 list5 list6 | sort | uniq > list456) &
        # Merges and sorts both sets of lists simultaneously.
        # Running in background ensures parallel execution.
        #
        # Same effect as
        #   cat list1 list2 list3 | sort | uniq > list123 &
        #   cat list4 list5 list6 | sort | uniq > list456 &

        wait   # Don't execute the next command until subshells finish.

        diff list123 list456
```

Redirecting I/O to a subshell uses the "|" pipe operator, as in **`ls -al | (command)`**.

A command block between *`curly braces`* does *not* launch a subshell.

{ command1; command2; command3; ... }

Globbing Up Restricted Shells

Appendix B. A Sed and Awk Micro-Primer

Table of Contents

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-like syntax

For all their differences, the two utilities share a similar invocation syntax, both use regular expressions , both read input by default from `stdin`, and both output to `stdout`. These are well-behaved UNIX tools, and they work together well. The output from one can be piped into the other, and their combined capabilities give shell scripts some of the power of Perl.

One important difference between the utilities is that while shell scripts can easily pass arguments to sed, it is more complicated for awk (see Example 34-3 and Example 9-20).

Chapter 19. Regular Expressions

Table of Contents

To fully utilize the power of shell scripting, you need to master Regular Expressions. Certain commands and utilities commonly used in scripts, such as expr, sed and awk interpret and use REs.

34.9. Shell Scripting Under Windows

Even users running *that other* OS can run UNIX-like shell scripts, and therefore benefit from many of the lessons of this book. The Cygwin package from Cygnus and the MKS utilities from Mortice Kern Associates add shell scripting capabilities to Windows.

Chapter 36. Endnotes

Table of Contents

Part 4. Advanced Topics

Table of Contents

Prev | Home | Next
Recess Time | Regular Expressions

Part 1. Introduction

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, give additional power and flexibility to scripts. Shell scripts lend themselves exceptionally well to administrative system tasks and other routine repetitive jobs not requiring the bells and whistles of a full-blown tightly structured programming language.

Table of Contents

Chapter 1. Why Shell Programming?

A working knowledge of shell scripting is essential to everyone wishing to become reasonably adept 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.

Writing shell scripts is not hard to learn, since the 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 and straightforward, similar to that of invoking and chaining together utilities at the command line, and there are only a few "rules" to learn. Most short scripts work right the first time, and debugging even the longer ones is straightforward.

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 shell script, even if slowly, is often a useful first stage in project development. This way, the structure of the application can be tested and played with, and the major pitfalls found before proceeding to the final coding in C, C++, Java, or Perl.

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

When not to use shell scripts

- resource-intensive tasks, especially where speed is a factor (sorting, hashing, etc.)
- 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 instead)
- complex applications, where structured programming is a necessity (need typechecking of variables, function prototypes, etc.)
- mission-critical applications upon which you are betting the ranch, or 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 multi-dimensional arrays
- need data structures, such as linked lists or trees
- need to generate or manipulate graphics or GUIs
- need direct access to system hardware
- need port or socket I/O
- need to use libraries or interface with legacy code
- proprietary, closed-source applications (shell scripts are necessarily Open Source)

If any of the above applies, consider a more powerful scripting language, perhaps Perl, Tcl, Python, or possibly a high-level 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 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 all flavors of UNIX. Most of the principles dealt with in this book apply equally well to scripting with other shells, such as the Korn Shell, from which Bash derives some of its features, [2] and the C Shell and its variants. (Note that C Shell programming is not recommended due to certain inherent problems, as pointed out in a news group posting by Tom Christiansen in October of 1993).

The following is a tutorial in shell scripting. It relies heavily on examples to illustrate features of the shell. As far as possible, the example scripts have been tested, and some of them may actually be useful in real life. The reader should use the actual examples in the source archive (`something-or-other.sh`), [3] give them execute permission (**`chmod u+rx scriptname`**), then run them to see what happens. Should the source archive not be available, then cut-and-paste from the HTML, pdf, or text rendered versions. Be aware that some of the scripts below introduce features before they are explained, and this may require the reader to temporarily skip ahead for enlightenment.

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

Notes

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

[2] Many of the features of *ksh88*, and even a few from the updated *ksh93* have been merged into Bash.

[3] 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 follow this guideline.

Prev Introduction | Home | Up | Next Starting Off With a Sha-Bang

Chapter 2. Starting Off With a Sha-Bang

Table of Contents

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 the log files in /var/log

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

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

There is nothing unusual here, just a set of commands that could just as easily be invoked one by one from the command line on the console or in an xterm. The advantages of placing the commands in a script go beyond not having to retype them time and again. The script can easily be modified, customized, or generalized for a particular application.

Example 2-2. cleanup: An enhanced and generalized version of above script.

```
#!/bin/bash
# cleanup, version 2
# Run as root, of course.

LOG_DIR=/var/log
ROOT_UID=0     # Only users with $UID 0 have root privileges.
LINES=50       # Default number of lines saved.
E_XCD=66       # Can't change directory?
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

if [ -n "$1" ]
# Test if command line argument 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=65  # Non-numerical argument (bad arg format)
#
#    case "$1" in
#    ""      ) lines=50;;
#    *[!0-9]*) echo "Usage: `basename $0` file-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 -$lines messages > mesg.temp # Saves last section of message log file.
mv mesg.temp messages             # Becomes new log directory.


# 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 "Logs cleaned up."

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 variant of the first script keeps the last section of the message log intact. You will constantly discover ways of refining previously written scripts for increased effectiveness.

The *sha-bang* (#!) 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 [1] "magic number", a special marker that designates a file type, or in this case an executable shell script (see **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 (line 1 of the script), ignoring comments. [2]

```
#!/bin/sh
#!/bin/bash
#!/usr/bin/perl
#!/usr/bin/tcl
#!/bin/sed -f
#!/usr/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. [3] Using **`#!/bin/sh`**, the default Bourne Shell in most commercial variants of UNIX, makes the script portable to non-Linux machines, though you may have to sacrifice a few Bash-specific features (the script will conform to the POSIX [4] **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.

#! can be omitted if the script consists only of a set of generic system commands, using no internal shell directives. Example 2, above, requires the initial #!, since the variable assignment line, **`lines=50`**, uses a shell-specific construct. Note that **`#!/bin/sh`** invokes the default shell interpreter, which defaults to `/bin/bash` on a Linux machine.

This tutorial encourages a modular approach to constructing a script. Make note of and collect "boilerplate" code snippets that might be useful in future scripts. Eventually you can build a quite 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.

```
if [ $# -ne Number_of_expected_args ]
then
  echo "Usage: `basename $0` whatever"
  exit $WRONG_ARGS
fi
```

Notes

[1] Some flavors of UNIX (those based on 4.2BSD) take a four-byte magic number, requiring a blank after the !, **`#! /bin/sh`**.

[2] 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.

[3] 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=65

echo "This line will never print (betcha!)."

exit $WHATEVER  # Doesn't matter. The script will not exit here.
```

Also, try starting a README file with a **#!/bin/more**, and making it executable. The result is a self-listing documentation file.

[4] **P**ortable **O**perating **S**ystem *I*nterface, an attempt to standardize UNI**X**-like OSes.

Part 2. Basics

Table of Contents

Chapter 5. Introduction to Variables and Parameters

Table of Contents

Variables are at the heart of every programming and scripting language. They appear in arithmetic operations and manipulation of quantities, string parsing, and are indispensable for working in the abstract with symbols - tokens that represent something else. A variable is nothing more than a location or set of locations in computer memory holding an item of data.

Chapter 7. Tests

Table of Contents

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.

Chapter 8. Operations and Related Topics

Table of Contents

Chapter 9. Variables Revisited

Table of Contents

Used properly, variables can add power and flexibility to scripts. This requires learning their subtleties and nuances.

Chapter 13. 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

chown, **chgrp**

The **chown** command changes the ownership of a file or files. This command is a useful method that *root* can use to shift file ownership from one user to another. An ordinary user may not change the ownership of files, not even her own files. [1]

```
root# chown bozo *.txt
```

The **chgrp** command changes the *group* ownership of a file or files. You must be owner of the file(s) as well as a member of the destination group (or *root*) to use this operation.

```
chgrp --recursive dunderheads *.data
#  The "dunderheads" group will now own all the "*.data" files
#+ all the way down the $PWD directory tree (that's what "recursive" means).
```

useradd, **userdel**

The **useradd** administrative command adds a user account to the system and creates a home directory for that particular user, if so specified. The corresponding **userdel** command removes a user account from the system [2] and deletes associated files.

The **adduser** command is a synonym for **useradd** and is usually a symbolic link to it.

id

The **id** command lists the real and effective user IDs and the group IDs of the current user. This is the counterpart to the $UID, $EUID, and $GROUPS internal Bash variables.

```
bash$ id
uid=501(bozo) gid=501(bozo) groups=501(bozo),22(cdrom),80(cdwriter),81(audio)

bash$ echo $UID
501
```

Also see Example 9-5.

who

Show all users logged on to the system.

```
bash$ who
bozo  tty1     Apr 27 17:45
 bozo  pts/0    Apr 27 17:46
 bozo  pts/1    Apr 27 17:47
 bozo  pts/2    Apr 27 17:49
```

The `-m` gives detailed information about only the current user. Passing any two arguments to **who** is the equivalent of **who -m**, as in **who am i** or **who The Man**.

```
bash$ who -m
localhost.localdomain!bozo  pts/2    Apr 27 17:49
```

whoami is similar to **who -m**, but only lists the user name.

```
bash$ whoami
bozo
```

w

Show all logged on users and the processes belonging to them. This is an extended version of **who**. The output of **w** may be piped to **grep** to find a specific user and/or process.

```
bash$ w | grep startx
bozo  tty1     -                4:22pm  6:41   4.47s  0.45s  startx
```

logname

Show current user's login name (as found in `/var/run/utmp`). This is a near-equivalent to whoami, above.

```
bash$ logname
bozo

bash$ whoami
bozo
```

However...

```
bash$ su
Password: ......

bash# whoami
root
bash# logname
bozo
```

su

Runs a program or script as a *s*ubstitute *u*ser. **su rjones** starts a shell as user *rjones*. A naked **su** defaults to *root*. See Example A-15.

sudo

Runs a command as root (or another user). This may be used in a script, thus permitting a regular user to run the script.

```
#!/bin/bash

# Some commands.
sudo cp /root/secretfile /home/bozo/secret
# Some more commands.
```

The file `/etc/sudoers` holds the names of users permitted to invoke **sudo**.

users

Show all logged on users. This is the approximate equivalent of **who -q**.

ac

Show users' logged in time, as read from `/var/log/wtmp`. This is one of the GNU accounting utilities.

```
bash$ ac
        total       68.08
```

last

List *last* logged in users, as read from `/var/log/wtmp`. This command can also show remote logins.

groups

Lists the current user and the groups she belongs to. This corresponds to the $GROUPS internal variable, but gives the group names, rather than the numbers.

```
bash$ groups
bozita cdrom cdwriter audio xgrp

bash$ echo $GROUPS
501
```

newgrp

Change user's group ID without logging out. This permits access to the new group's files. Since users may be members of multiple groups simultaneously, this command finds little use.

Terminals

tty

Echoes the name of the current user's terminal. Note that each separate xterm window counts as a different terminal.

```
bash$ tty
/dev/pts/1
```

stty

Shows and/or changes terminal settings. This complex command, used in a script, can control terminal behavior and the way output displays. See the info page, and study it carefully.

Example 13-1. setting an erase character

```
#!/bin/bash
# erase.sh: Using "stty" to set an erase character when reading input.

echo -n "What is your name? "
read name                       # Try to erase characters of input.
                                # Won't work.
echo "Your name is $name."

stty erase '#'                  # Set "hashmark" (#) as erase character.
echo -n "What is your name? "
read name                       # Use # to erase last character typed.
echo "Your name is $name."

exit 0
```

Example 13-2. secret password: Turning off terminal echoing

```
#!/bin/bash

echo
echo -n "Enter password "
read passwd
echo "password is $passwd"
echo -n "If someone had been looking over your shoulder, "
echo "your password would have been compromised."

echo && echo  # Two line-feeds in an "and list".

stty -echo    # Turns off screen echo.

echo -n "Enter password again "
read passwd
echo
echo "password is $passwd"
echo

stty echo     # Restores screen echo.

exit 0
```

A creative use of **stty** is detecting a user keypress (without hitting **ENTER**).

Example 13-3. Keypress detection

```
#!/bin/bash
# keypress.sh: Detect a user keypress ("hot keyboard").

echo

old_tty_settings=$(stty -g)   # Save old settings.
stty -icanon
Keypress=$(head -c1)          # or $(dd bs=1 count=1 2> /dev/null)
                              # on non-GNU systems

echo
echo "Key pressed was \""$Keypress"\"."
echo

stty "$old_tty_settings"      # Restore old settings.

# Thanks, Stephane Chazelas.

exit 0
```

Also see Example 9-3.

terminals and modes

Normally, a terminal works in the *canonical* mode. When a user hits a key, the resulting character does not immediately go to the program actually running in this terminal. A buffer local to the terminal stores keystrokes. When the user hits the **ENTER** key, this sends all the stored keystrokes to the program running. There is even a basic line editor inside the terminal.

```
bash$ stty -a
speed 9600 baud; rows 36; columns 96; line = 0;
 intr = ^C; quit = ^\; erase = ^H; kill = ^U; eof = ^D; eol = <undef>; eol2 =
<undef>;
 start = ^Q; stop = ^S; susp = ^Z; rprnt = ^R; werase = ^W; lnext = ^V; flush = ^O;
 ...
 isig icanon iexten echo echoe echok -echonl -noflsh -xcase -tostop -echoprt
```

Using canonical mode, it is possible to redefine the special keys for the local terminal line editor.

```
bash$ cat > filexxx
wha<ctl-W>I<ctl-H>foo bar<ctl-U>hello world<ENTER>
<ctl-D>
bash$ cat filexxx
hello world
bash$ bash$ wc -c < file
13
```

The process controlling the terminal receives only 13 characters (12 alphabetic ones, plus a newline), although the user hit 26 keys.

In non-canonical ("raw") mode, every key hit (including special editing keys such as **ctl-H**) sends a character immediately to the controlling process.

The Bash prompt disables both `icanon` and `echo`, since it replaces the basic terminal line editor with its own more elaborate one.

For example, when you hit **ctl-A** at the Bash prompt, there's no **^A** echoed by the terminal, but Bash gets a **\1** character, interprets it, and moves the cursor to the begining of the line.

Stephane Chazelas

tset

Show or initialize terminal settings. This is a less capable version of **stty**.

```
bash$ tset -r
Terminal type is xterm-xfree86.
 Kill is control-U (^U).
 Interrupt is control-C (^C).
```

setserial

Set or display serial port parameters. This command must be run by root user and is usually found in a system setup script.

```
# From /etc/pcmcia/serial script:

IRQ=`setserial /dev/$DEVICE | sed -e 's/.*IRQ: //'`
setserial /dev/$DEVICE irq 0 ; setserial /dev/$DEVICE irq $IRQ
```

getty, **agetty**

The initialization process for a terminal uses **getty** or **agetty** to set it up for login by a user. These commands are not used within user shell scripts. Their scripting counterpart is **stty**.

mesg

Enables or disables write access to the current user's terminal. Disabling access would prevent another user on the network to write to the terminal.

It can be very annoying to have a message about ordering pizza suddenly appear in the middle of the text file you are editing. On a multi-user network, you might therefore wish to disable write access to your terminal when you need to avoid interruptions.

wall

This is an acronym for "write all", i.e., sending a message to all users at every terminal logged into the network. It is primarily a system administrator's tool, useful, for example, when warning everyone that the system will shortly go down due to a problem (see Example 17-2).

```
bash$ wall System going down for maintenance in 5 minutes!
Broadcast message from bozo (pts/1) Sun Jul  8 13:53:27 2001...

 System going down for maintenance in 5 minutes!
```

If write access to a particular terminal has been disabled with **mesg**, then **wall** cannot send a message to it.

dmesg

Lists all system bootup messages to `stdout`. Handy for debugging and ascertaining which device drivers were installed and which system interrupts in use. The output of **dmesg** may, of course, be parsed with grep, sed, or awk from within a script.

Information and Statistics

uname

Output system specifications (OS, kernel version, etc.) to `stdout`. Invoked with the `-a` option, gives verbose system info (see Example 12-4). The `-s` option shows only the OS type.

```
bash$ uname -a
Linux localhost.localdomain 2.2.15-2.5.0 #1 Sat Feb 5 00:13:43 EST 2000 i686 unknown

bash$ uname -s
Linux
```

arch

Show system architecture. Equivalent to **uname -m**. See Example 10-25.

```
bash$ arch
i686

bash$ uname -m
i686
```

lastcomm

Gives information about previous commands, as stored in the `/var/account/pacct` file. Command name and user name can be specified by options. This is one of the GNU accounting utilities.

lastlog

List the last login time of all system users. This references the `/var/log/lastlog` file.

```
bash$ lastlog
root         tty1                      Fri Dec  7 18:43:21 -0700 2001
 bin                                    **Never logged in**
 daemon                                 **Never logged in**
 ...
 bozo          tty1                     Sat Dec  8 21:14:29 -0700 2001


bash$ lastlog | grep root
root         tty1                      Fri Dec  7 18:43:21 -0700 2001
```

This command will fail if the user invoking it does not have read permission for the `/var/log/lastlog` file.

lsof

List open files. This command outputs a detailed table of all currently open files and gives information about their owner, size,

the processes associated with them, and more. Of course, **lsof** may be piped to grep and/or awk to parse and analyze its results.

```
bash$ lsof
COMMAND    PID    USER   FD   TYPE     DEVICE    SIZE     NODE NAME
 init         1    root  mem    REG        3,5   30748    30303 /sbin/init
 init         1    root  mem    REG        3,5   73120     8069 /lib/ld-2.1.3.so
 init         1    root  mem    REG        3,5  931668     8075 /lib/libc-2.1.3.so
 cardmgr    213    root  mem    REG        3,5   36956    30357 /sbin/cardmgr
 ...
```

strace

Diagnostic and debugging tool for tracing system calls and signals. The simplest way of invoking it is **strace COMMAND**.

```
bash$ strace df
execve("/bin/df", ["df"], [/* 45 vars */]) = 0
 uname({sys="Linux", node="bozo.localdomain", ...}) = 0
 brk(0)                                  = 0x804f5e4
 ...
```

This is the Linux equivalent of **truss**.

free

Shows memory and cache usage in tabular form. The output of this command lends itself to parsing, using grep, awk or **Perl**. The **procinfo** command shows all the information that **free** does, and much more.

```
bash$ free
                total       used       free     shared    buffers     cached
   Mem:         30504      28624       1880      15820       1608      16376
   -/+ buffers/cache:      10640      19864
   Swap:        68540       3128      65412
```

To show unused RAM memory:

```
bash$ free | grep Mem | awk '{ print $4 }'
1880
```

procinfo

Extract and list information and statistics from the /proc pseudo-filesystem. This gives a very extensive and detailed listing.

```
bash$ procinfo | grep Bootup
Bootup: Wed Mar 21 15:15:50 2001    Load average: 0.04 0.21 0.34 3/47 6829
```

lsdev

List devices, that is, show installed hardware.

```
bash$ lsdev
Device            DMA   IRQ  I/O Ports
 ------------------------------------------------
 cascade             4     2
 dma                            0080-008f
 dma1                           0000-001f
 dma2                           00c0-00df
 fpu                            00f0-00ff
 ide0                       14  01f0-01f7 03f6-03f6
 ...
```

du

Show (disk) file usage, recursively. Defaults to current working directory, unless otherwise specified.

```
bash$ du -ach
1.0k    ./wi.sh
 1.0k    ./tst.sh
 1.0k    ./random.file
 6.0k    .
 6.0k    total
```

df

Shows filesystem usage in tabular form.

```
bash$ df
Filesystem           1k-blocks      Used Available Use% Mounted on
 /dev/hda5               273262     92607    166547  36% /
 /dev/hda8               222525    123951     87085  59% /home
 /dev/hda7              1408796   1075744    261488  80% /usr
```

stat

Gives detailed and verbose *stat*istics on a given file (even a directory or device file) or set of files.

```
bash$ stat test.cru
  File: "test.cru"
   Size: 49970        Allocated Blocks: 100          Filetype: Regular File
   Mode: (0664/-rw-rw-r--)         Uid: (  501/ bozo)  Gid: (  501/ bozo)
 Device:  3,8   Inode: 18185     Links: 1
 Access: Sat Jun  2 16:40:24 2001
 Modify: Sat Jun  2 16:40:24 2001
 Change: Sat Jun  2 16:40:24 2001
```

If the target file does not exist, **stat** returns an error message.

```
bash$ stat nonexistent-file
nonexistent-file: No such file or directory
```

vmstat

Display virtual memory statistics.

```
bash$ vmstat
   procs                  memory    swap          io system         cpu
 r  b  w   swpd   free   buff  cache  si  so    bi    bo   in    cs  us  sy id
 0  0  0      0  11040   2636  38952   0   0    33     7  271    88   8   3 89
```

netstat

Show current network statistics and information, such as routing tables and active connections. This utility accesses information in `/proc/net` (Chapter 28). See Example 28-2.

netstat -r is equivalent to route.

uptime

Shows how long the system has been running, along with associated statistics.

```
bash$ uptime
10:28pm  up  1:57,  3 users,  load average: 0.17, 0.34, 0.27
```

hostname

Lists the system's host name. This command sets the host name in an `/etc/rc.d` setup script (`/etc/rc.d/rc.sysinit` or similar). It is equivalent to **uname -n**, and a counterpart to the $HOSTNAME internal variable.

```
bash$ hostname
localhost.localdomain

bash$ echo $HOSTNAME
localhost.localdomain
```

hostid

Echo a 32-bit hexadecimal numerical identifier for the host machine.

```
bash$ hostid
7f0100
```

This command allegedly fetches a "unique" serial number for a particular system. Certain product registration procedures use this number to brand a particular user license. Unfortunately, **hostid** only returns the machine network address in hexadecimal, with pairs of bytes transposed.

The network address of a typical non-networked Linux machine, is found in `/etc/hosts`.

```
bash$ cat /etc/hosts
127.0.0.1               localhost.localdomain localhost

```

As it happens, transposing the bytes of **127.0.0.1**, we get **0.127.1.0**, which translates in hex to **007f0100**, the exact equivalent of what **hostid** returns, above. There exist only a few million other Linux machines with this identical *hostid*.

sar

Invoking **sar** (system activity report) gives a very detailed rundown on system statistics. This command is found on some commercial UNIX systems, but is not part of the base Linux distribution. It is contained in the sysstat utilities package, written by Sebastien Godard.

```
bash$ sar
Linux 2.4.7-10 (localhost.localdomain)  12/31/2001

 10:30:01 AM       CPU     %user     %nice   %system     %idle
 10:40:00 AM       all      1.39      0.00      0.77     97.84
 10:50:00 AM       all     76.83      0.00      1.45     21.72
 11:00:00 AM       all      1.32      0.00      0.69     97.99
 11:10:00 AM       all      1.17      0.00      0.30     98.53
 11:20:00 AM       all      0.51      0.00      0.30     99.19
 06:30:00 PM       all    100.00      0.00    100.01      0.00
 Average:          all      1.39      0.00      0.66     97.95


```

System Logs

logger

Appends a user-generated message to the system log (`/var/log/messages`). You do not have to be root to invoke **logger**.

```
logger Experiencing instability in network connection at 23:10, 05/21.
# Now, do a 'tail /var/log/messages'.

```

By embedding a **logger** command in a script, it is possible to write debugging information to `/var/log/messages`.

```
logger -t $0 -i Logging at line "$LINENO".
# The "-t" option specifies the tag for the logger entry.
# The "-i" option records the process ID.

# tail /var/log/message
# ...
# Jul  7 20:48:58 localhost ./test.sh[1712]: Logging at line 3.

```

logrotate

This utility manages the system log files, rotating, compressing, deleting, and/or mailing them, as appropriate. Usually crond runs **logrotate** on a daily basis.

Adding an appropriate entry to `/etc/logrotate.conf` makes it possible to manage personal log files, as well as system-wide ones.

Job Control

ps

*P*rocess *S*tatistics: lists currently executing processes by owner and PID (process id). This is usually invoked with `ax` options, and may be piped to grep or sed to search for a specific process (see Example 11-9 and Example 28-1).

```
bash$  ps ax | grep sendmail
295 ?      S      0:00 sendmail: accepting connections on port 25
```

pstree

Lists currently executing processes in "tree" format. The `-p` option shows the PIDs, as well as the process names.

top

Continuously updated display of most cpu-intensive processes. The `-b` option displays in text mode, so that the output may be parsed or accessed from a script.

```
bash$ top -b
  8:30pm  up 3 min,  3 users,  load average: 0.49, 0.32, 0.13
 45 processes: 44 sleeping, 1 running, 0 zombie, 0 stopped
 CPU states: 13.6% user,  7.3% system,  0.0% nice, 78.9% idle
 Mem:    78396K av,   65468K used,   12928K free,       0K shrd,    2352K buff
 Swap:  157208K av,       0K used,  157208K free                   37244K cached

   PID USER     PRI  NI  SIZE  RSS SHARE STAT %CPU %MEM   TIME COMMAND
   848 bozo      17   0   996  996   800 R     5.6  1.2   0:00 top
     1 root       8   0   512  512   444 S     0.0  0.6   0:04 init
     2 root       9   0     0    0     0 SW    0.0  0.0   0:00 keventd
   ...
```

nice

Run a background job with an altered priority. Priorities run from 19 (lowest) to -20 (highest). Only *root* may set the negative (higher) priorities. Related commands are **renice**, **snice**, and **skill**.

nohup

Keeps a command running even after user logs off. The command will run as a foreground process unless followed by &. If you use **nohup** within a script, consider coupling it with a wait to avoid creating an orphan or zombie process.

pidof

Identifies *process id (pid)* of a running job. Since job control commands, such as kill and **renice** act on the *pid* of a process (not its name), it is sometimes necessary to identify that *pid*. The **pidof** command is the approximate counterpart to the $PPID internal variable.

```
bash$ pidof xclock
880
```

Example 13-4. pidof helps kill a process

```
#!/bin/bash
# kill-process.sh

NOPROCESS=2

process=xxxyyyzzz  # Use nonexistent process.
# For demo purposes only...
# ... don't want to actually kill any actual process with this script.
#
# If, for example, you wanted to use this script to logoff the Internet,
#     process=pppd

t=`pidof $process`       # Find pid (process id) of $process.
# The pid is needed by 'kill' (can't 'kill' by program name).

if [ -z "$t" ]           # If process not present, 'pidof' returns null.
then
  echo "Process $process was not running."
  echo "Nothing killed."
  exit $NOPROCESS
fi

kill $t                  # May need 'kill -9' for stubborn process.

# Need a check here to see if process allowed itself to be killed.
# Perhaps another " t=`pidof $process` ".


# This entire script could be replaced by
#    kill $(pidof -x process_name)
# but it would not be as instructive.

exit 0
```

fuser

Identifies the processes (by pid) that are accessing a given file, set of files, or directory. May also be invoked with the `-k` option, which kills those processes. This has interesting implications for system security, especially in scripts preventing unauthorized users from accessing system services.

crond

Administrative program scheduler, performing such duties as cleaning up and deleting system log files and updating the slocate database. This is the superuser version of at (although each user may have their own `crontab` file which can be changed with the **crontab** command). It runs as a daemon and executes scheduled entries from `/etc/crontab`.

Process Control and Booting

init

The **init** command is the parent of all processes. Called in the final step of a bootup, **init** determines the runlevel of the system from `/etc/inittab`. Invoked by its alias **telinit**, and by root only.

telinit

Symlinked to **init**, this is a means of changing the system runlevel, usually done for system maintenance or emergency filesystem repairs. Invoked only by root. This command can be dangerous - be certain you understand it well before using!

runlevel

Shows the current and last runlevel, that is, whether the system is halted (runlevel `0`), in single-user mode (`1`), in multi-user mode (`2` or `3`), in X Windows (`5`), or rebooting (`6`). This command accesses the `/var/run/utmp` file.

halt, **shutdown**, **reboot**

Command set to shut the system down, usually just prior to a power down.

Network

ifconfig

Network interface configuration and tuning utility. It is most often used at bootup to set up the interfaces, or to shut them down when rebooting.

```
# Code snippets from /etc/rc.d/init.d/network

# ...

# Check that networking is up.
[ ${NETWORKING} = "no" ] && exit 0

[ -x /sbin/ifconfig ] || exit 0

# ...

for i in $interfaces ; do
  if ifconfig $i 2>/dev/null | grep -q "UP" >/dev/null 2>&1 ; then
    action "Shutting down interface $i: " ./ifdown $i boot
  fi
# The GNU-specific "-q" option to "grep" means "quiet", i.e., producing no output.
# Redirecting output to /dev/null is therefore not strictly necessary.

# ...

echo "Currently active devices:"
echo `/sbin/ifconfig | grep ^[a-z] | awk '{print $1}'`
#                            ^^^^^  should be quoted to prevent globbing.
#  The following also work.
#    echo $(/sbin/ifconfig | awk '/^[a-z]/ { print $1 })'
#    echo $(/sbin/ifconfig | sed -e 's/ .*//')
#  Thanks, S.C., for additional comments.
```

See also Example 30-6.

route

Show info about or make changes to the kernel routing table.

```
bash$ route
Destination     Gateway         Genmask         Flags   MSS Window  irtt Iface
 pm3-67.bozosisp *               255.255.255.255 UH        40 0          0 ppp0
 127.0.0.0       *               255.0.0.0       U         40 0          0 lo
 default         pm3-67.bozosisp 0.0.0.0         UG        40 0          0 ppp0
```

chkconfig

Check network configuration. This command lists and manages the network services started at bootup in the `/etc/rc?.d` directory.

Originally a port from IRIX to Red Hat Linux, **chkconfig** may not be part of the core installation of some Linux flavors.

```
bash$ chkconfig --list
atd             0:off   1:off   2:off   3:on    4:on    5:on    6:off
 rwhod           0:off   1:off   2:off   3:off   4:off   5:off   6:off
 ...
```

tcpdump

Network packet "sniffer". This is a tool for analyzing and troubleshooting traffic on a network by dumping packet headers that match specified criteria.

Dump ip packet traffic between hosts *bozoville* and *caduceus*:

```
bash$ tcpdump ip host bozoville and caduceus
```

Of course, the output of **tcpdump** can be parsed, using certain of the previously discussed text processing utilities.

Filesystem

mount

Mount a filesystem, usually on an external device, such as a floppy or CDROM. The file `/etc/fstab` provides a handy listing of available filesystems, partitions, and devices, including options, that may be automatically or manually mounted. The file `/etc/mtab` shows the currently mounted filesystems and partitions (including the virtual ones, such as `/proc`).

mount -a mounts all filesystems and partitions listed in `/etc/fstab`, except those with a `noauto` option. At bootup, a startup script in `/etc/rc.d` (`rc.sysinit` or something similar) invokes this to get everything mounted.

```
mount -t iso9660 /dev/cdrom /mnt/cdrom
# Mounts CDROM
mount /mnt/cdrom
# Shortcut, if /mnt/cdrom listed in /etc/fstab
```

This versatile command can even mount an ordinary file on a block device, and the file will act as if it were a filesystem.

Mount accomplishes that by associating the file with a loopback device. One application of this is to mount and examine an ISO9660 image before burning it onto a CDR. [3]

Example 13-5. Checking a CD image

```
# As root...

mkdir /mnt/cdtest  # Prepare a mount point, if not already there.

mount -r -t iso9660 -o loop cd-image.iso /mnt/cdtest   # Mount the image.
#                  "-o loop" option equivalent to "losetup /dev/loop0"
cd /mnt/cdtest     # Now, check the image.
ls -alR            # List the files in the directory tree there.
                   # And so forth.
```

umount

Unmount a currently mounted filesystem. Before physically removing a previously mounted floppy or CDROM disk, the device must be **umount**ed, else filesystem corruption may result.

```
umount /mnt/cdrom
# You may now press the eject button and safely remove the disk.
```

The **automount** utility, if properly installed, can mount and unmount floppies or CDROM disks as they are accessed or removed. On laptops with swappable floppy and CDROM drives, this can cause problems, though.

sync

Forces an immediate write of all updated data from buffers to hard drive (synchronize drive with buffers). While not strictly necessary, a **sync** assures the sys admin or user that the data just changed will survive a sudden power failure. In the olden days, a **`sync; sync`** (twice, just to make absolutely sure) was a useful precautionary measure before a system reboot.

At times, you may wish to force an immediate buffer flush, as when securely deleting a file (see Example 12-41) or when the lights begin to flicker.

losetup

Sets up and configures loopback devices.

Example 13-6. Creating a filesystem in a file

```
SIZE=1000000  # 1 meg

head -c $SIZE < /dev/zero > file  # Set up file of designated size.
losetup /dev/loop0 file           # Set it up as loopback device.
mke2fs /dev/loop0                 # Create filesystem.
mount -o loop /dev/loop0 /mnt     # Mount it.

# Thanks, S.C.
```

mkswap

Creates a swap partition or file. The swap area must subsequently be enabled with **swapon**.

swapon, **swapoff**

Enable / disable swap partitition or file. These commands usually take effect at bootup and shutdown.

mke2fs

Create a Linux ext2 filesystem. This command must be invoked as root.

Example 13-7. Adding a new hard drive

```
#!/bin/bash

# Adding a second hard drive to system.
# Software configuration. Assumes hardware already mounted.
# From an article by the author of this document.
# in issue #38 of "Linux Gazette", http://www.linuxgazette.com.

ROOT_UID=0     # This script must be run as root.
E_NOTROOT=67   # Non-root exit error.

if [ "$UID" -ne "$ROOT_UID" ]
then
  echo "Must be root to run this script."
  exit $E_NOTROOT
fi

# Use with extreme caution!
# If something goes wrong, you may wipe out your current filesystem.


NEWDISK=/dev/hdb         # Assumes /dev/hdb vacant. Check!
MOUNTPOINT=/mnt/newdisk  # Or choose another mount point.


fdisk $NEWDISK
mke2fs -cv $NEWDISK1   # Check for bad blocks & verbose output.
#  Note:    /dev/hdb1, *not* /dev/hdb!
mkdir $MOUNTPOINT
chmod 777 $MOUNTPOINT  # Makes new drive accessible to all users.


# Now, test...
# mount -t ext2 /dev/hdb1 /mnt/newdisk
# Try creating a directory.
# If it works, umount it, and proceed.

# Final step:
# Add the following line to /etc/fstab.
# /dev/hdb1  /mnt/newdisk  ext2  defaults  1 1

exit 0
```

See also Example 13-6 and Example 29-3.

tune2fs

Tune ext2 filesystem. May be used to change filesystem parameters, such as maximum mount count. This must be invoked as root.

This is an extremely dangerous command. Use it at your own risk, as you may inadvertently destroy your filesystem.

dumpe2fs

Dump (list to `stdout`) very verbose filesystem info. This must be invoked as root.

```
root# dumpe2fs /dev/hda7 | grep 'ount count'
dumpe2fs 1.19, 13-Jul-2000 for EXT2 FS 0.5b, 95/08/09
 Mount count:              6
 Maximum mount count:      20
```

hdparm

List or change hard disk parameters. This command must be invoked as root, and it may be dangerous if misused.

fdisk

Create or change a partition table on a storage device, usually a hard drive. This command must be invoked as root.

Use this command with extreme caution. If something goes wrong, you may destroy an existing filesystem.

fsck, **e2fsck**, **debugfs**

Filesystem check, repair, and debug command set.

fsck: a front end for checking a UNIX filesystem (may invoke other utilities). The actual filesystem type generally defaults to ext2.

e2fsck: ext2 filesystem checker.

debugfs: ext2 filesystem debugger. One of the uses of this versatile, but dangerous command is to (attempt to) recover deleted files. For advanced users only!

All of these should be invoked as root, and they can damage or destroy a filesystem if misused.

badblocks

Checks for bad blocks (physical media flaws) on a storage device. This command finds use when formatting a newly installed hard drive or testing the integrity of backup media. [4] As an example, **badblocks /dev/fd0** tests a floppy disk.

The **badblocks** command may be invoked destructively (overwrite all data) or in non-destructive read-only mode. If root user owns the device to be tested, as is generally the case, then root must invoke this command.

mkbootdisk

Creates a boot floppy which can be used to bring up the system if, for example, the MBR (master boot record) becomes corrupted. The **mkbootdisk** command is actually a Bash script, written by Erik Troan, in the `/sbin` directory.

chroot

CHange ROOT directory. Normally commands are fetched from $PATH, relative to `/`, the default root directory. This changes the root directory to a different one (and also changes the working directory to there). This is useful for security purposes, for instance when the system administrator wishes to restrict certain users, such as those telnetting in, to a secured portion of the filesystem (this is sometimes referred to as confining a guest user to a "chroot jail"). Note that after a **chroot**, the execution path for system binaries is no longer valid.

A **`chroot /opt`** would cause references to `/usr/bin` to be translated to `/opt/usr/bin`. Likewise, **`chroot`**

`/aaa/bbb /bin/ls` would redirect future instances of **ls** to `/aaa/bbb` as the base directory, rather than `/` as is normally the case. An **alias XX 'chroot /aaa/bbb ls'** in a user's `~/.bashrc` effectively restricts which portion of the filesystem she may run command "XX" on.

The **chroot** command is also handy when running from an emergency boot floppy (**chroot** to `/dev/fd0`), or as an option to **lilo** when recovering from a system crash. Other uses include installation from a different filesystem (an **rpm** option) or running a readonly filesystem from a CD ROM. Invoke only as root, and use with care.

It might be necessary to copy certain system files to a *chrooted* directory, since the normal `$PATH` can no longer be relied upon.

lockfile

This utility is part of the **procmail** package (www.procmail.org). It creates a *lock file*, a semaphore file that controls access to a file, device, or resource. The lock file serves as a flag that this particular file, device, or resource is in use by a particular process ("busy"), and this permits only restricted access (or no access) to other processes.

Lock files are used in such applications as protecting system mail folders from simultaneously being changed by multiple users, indicating that a modem port is being accessed, and showing that an instance of Netscape is using its cache. Scripts may check for the existence of a lock file created by a certain process to check if that process is running. Note that if a script attempts create a lock file that already exists, the script will likely hang.

Normally, applications create and check for lock files in the `/var/lock` directory. A script can test for the presence of a lock file by something like the following.

```
appname=xyzip
# Application "xyzip" created lock file "/var/lock/xyzip.lock".

if [ -e "/var/lock/$appname.lock ]
then
  ...
```

mknod

Creates block or character device files (may be necessary when installing new hardware on the system).

tmpwatch

Automatically deletes files which have not been accessed within a specified period of time. Usually invoked by crond to remove stale log files.

MAKEDEV

Utility for creating device files. It must be run as root, and in the `/dev` directory.

```
root# ./MAKEDEV
```

This is a sort of advanced version of **mknod**.

Backup

dump, **restore**

The **dump** command is an elaborate filesystem backup utility, generally used on larger installations and networks. [5] It reads raw disk partitions and writes a backup file in a binary format. Files to be backed up may be saved to a variety of storage media, including disks and tape drives. The **restore** command restores backups made with **dump**.

fdformat

Perform a low-level format on a floppy disk.

System Resources

ulimit

Sets an *upper limit* on system resources. Usually invoked with the `-f` option, which sets a limit on file size (**ulimit -f 1000** limits files to 1 meg maximum). The `-t` option limits the coredump size (**ulimit -c 0** eliminates coredumps). Normally, the value of **ulimit** would be set in `/etc/profile` and/or `~/.bash_profile` (see Chapter 27).

umask

User file creation MASK. Limit the default file attributes for a particular user. All files created by that user take on the attributes specified by **umask**. The (octal) value passed to **umask** defines the file permissions *disabled*. For example, **umask 022** ensures that new files will have at most 755 permissions (777 NAND 022). [6] Of course, the user may later change the attributes of particular files with chmod. The usual practice is to set the value of **umask** in `/etc/profile` and/or `~/.bash_profile` (see Chapter 27).

rdev

Get info about or make changes to root device, swap space, or video mode. The functionality of **rdev** has generally been taken over by **lilo**, but **rdev** remains useful for setting up a ram disk. This is another dangerous command, if misused.

Modules

lsmod

List installed kernel modules.

```
bash$ lsmod
Module                  Size  Used by
 autofs                  9456   2 (autoclean)
 opl3                   11376   0
 serial_cs               5456   0 (unused)
 sb                     34752   0
 uart401                 6384   0 [sb]
 sound                  58368   0 [opl3 sb uart401]
 soundlow                 464   0 [sound]
 soundcore               2800   6 [sb sound]
 ds                      6448   2 [serial_cs]
 i82365                 22928   2
 pcmcia_core            45984   0 [serial_cs ds i82365]
```

insmod

Force installation of a kernel module. Must be invoked as root.

rmmod

Force unloading of a kernel module. Must be invoked as root.

modprobe

Module loader that is normally invoked automatically in a startup script.

depmod

Creates module dependency file, usually invoked from startup script.

Miscellaneous

env

Runs a program or script with certain environmental variables set or changed (without changing the overall system environment). The `[varname=xxx]` permits changing the environmental variable `varname` for the duration of the script. With no options specified, this command lists all the environmental variable settings.

In Bash and other Bourne shell derivatives, it is possible to set variables in a single command's environment.

```
var1=value1 var2=value2 commandXXX
# $var1 and $var2 set in the environment of 'commandXXX' only.
```

The first line of a script (the "sha-bang" line) may use **env** when the path to the shell or interpreter is unknown.

```
#! /usr/bin/env perl

print "This Perl script will run,\n";
print "even when I don't know where to find Perl.\n";

# Good for portable cross-platform scripts,
# where the Perl binaries may not be in the expected place.
# Thanks, S.C.
```

ldd

Show shared lib dependencies for an executable file.

```
bash$ ldd /bin/ls
libc.so.6 => /lib/libc.so.6 (0x4000c000)
/lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x80000000)
```

strip

Remove the debugging symbolic references from an executable binary. This decreases its size, but makes debugging of it impossible.

This command often occurs in a Makefile, but rarely in a shell script.

nm

List symbols in an unstripped compiled binary.

rdist

Remote distribution client: synchronizes, clones, or backs up a file system on a remote server.

Using our knowledge of administrative commands, let us examine a system script. One of the shortest and simplest to understand scripts is **killall**, used to suspend running processes at system shutdown.

Example 13-8. killall, from `/etc/rc.d/init.d`

```
#!/bin/sh

# --> Comments added by the author of this document marked by "# -->".

# --> This is part of the 'rc' script package
# --> by Miquel van Smoorenburg, <miquels@drinkel.nl.mugnet.org>

# --> This particular script seems to be Red Hat 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, and since 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
        # --> using the 'stop' 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 "action" commands to "echos".

Exercise 2. Look at some of the more complex scripts in `/etc/rc.d/init.d`. See if you can understand parts 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.

Notes

[1] This is the case on a Linux machine or a UNIX system with disk quotas.

[2] The **userdel** command will fail if the particular user being deleted is still logged on.

[3] For more detail on burning CDRs, see Alex Withers' article, Creating CDs, in the October, 1999 issue of Linux Journal.

[4] The `-c` option to mke2fs also invokes a check for bad blocks.

[5] Operators of single-user Linux systems generally prefer something simpler for backups, such as **tar**.

[6] NAND is the logical "not-and" operator. Its effect is somewhat similar to subtraction.

Prev
Miscellaneous Commands

Home
Up

Next
Command Substitution

Chapter 15. Arithmetic Expansion

Arithmetic expansion provides a powerful tool for performing arithmetic operations in scripts. Translating a string into a numerical expression is relatively straightforward using backticks, double parentheses, or let.

Variations

Arithmetic expansion with backticks (often used in conjunction with expr)

```
z=`expr $z + 3`          # 'expr' does the expansion.
```

Arithmetic expansion with double parentheses, and using **let**

The use of backticks in arithmetic expansion has been superseded by double parentheses **$((...))** or the very convenient **let** construction.

```
z=$(($z+3))
# $((EXPRESSION)) is arithmetic expansion.  # Not to be confused with
                                            # command substitution.

let z=z+3
let "z += 3"  #If quotes, then spaces and special operators allowed.
# 'let' is actually arithmetic evaluation, rather than expansion.
```

All the above are equivalent. You may use whichever one "rings your chimes".

Examples of arithmetic expansion in scripts:

1. Example 12-6
2. Example 10-14
3. Example 26-1
4. Example 26-4
5. Example A-17

Chapter 17. Here Documents

A *here document* uses a special form of I/O redirection to feed a command list to an interactive program or command, such as ftp, telnet, or **ex**. A "limit string" delineates (frames) the command list. The special symbol << designates the limit string. This has the effect of redirecting the output of a file into the program, similar to **`interactive-program < command-file`**, where `command-file` contains

```
command #1
command #2
...
```

The "here document" alternative looks like this:

```
#!/bin/bash
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.

Example 17-1. dummyfile: Creates a 2-line dummy file

```
#!/bin/bash

# Non-interactive use of 'vi' to edit a file.
# Emulates 'sed'.

E_BADARGS=65

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 0
```

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*.

Example 17-2. 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: Comment lines printed by 'wall'.
zzz23EndOfMessagezzz23

# Could have been done more efficiently by
#         wall <message-file
# However, saving a message template in a script saves work.

exit 0
```

Example 17-3. Multi-line message using cat

```
#!/bin/bash

# 'echo' is fine for printing single line messages,
#  but somewhat problematic for for message blocks.
#  A 'cat' here document overcomes this limitation.

cat <<End-of-message
-------------------------------------
This is line 1 of the message.
This is line 2 of the message.
This is line 3 of the message.
This is line 4 of the message.
This is the last line of the message.
-------------------------------------
End-of-message

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 tabs (but not spaces) in the output. This may be useful in making a script more readable.

Example 17-4. Multi-line message, with tabs suppressed

```
#!/bin/bash
# Same as previous example, but...

#  The - option to a here document <<-
#  suppresses 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   <<-  .


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 17-5. Here document with parameter substitution

```
#!/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 0
```

Quoting or escaping the "limit string" at the head of a here document disables parameter substitution within its body. This has very limited usefulness.

Example 17-6. 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

exit 0
```

This is a useful script containing a here document with parameter substitution.

Example 17-7. upload: Uploads a file pair to "Sunsite" incoming directory

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

# Upload file pair (Filename.lsm, Filename.tar.gz)
# to incoming directory at Sunsite (metalab.unc.edu).

E_ARGERROR=65

if [ -z "$1" ]
then
  echo "Usage: `basename $0` filename"
  exit $E_ARGERROR
fi


Filename=`basename $1`           # Strips pathname out of file name.

Server="metalab.unc.edu"
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"
binary
bell                # Ring 'bell' after each file transfer
cd $Directory
put "$Filename.lsm"
put "$Filename.tar.gz"
bye
End-Of-Session

exit 0
```

A here document can supply input to a function in the same script.

Example 17-8. Here documents and functions

```
#!/bin/bash
# here-function.sh

GetPersonalData ()
{
  read firstname
  read lastname
  read address
  read city
  read state
  read zipcode
} # This certainly looks like an interactive function, but...


# Supply input to the above function.
GetPersonalData <<RECORD001
Bozo
Bozeman
2726 Nondescript Dr.
Baltimore
MD
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 17-9. "Anonymous" Here Document

```
#!/bin/bash

: <<TESTVARIABLES
${HOSTNAME?}${USER?}${MAIL?}  # Print error message if one of the variables not set.
TESTVARIABLES

exit 0
```

A variation of the above technique permits "commenting out" blocks of code.

Example 17-10. 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.


#  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.

: << DEBUGXXX
for file in *
do
 cat "$file"
done
DEBUGXXX

exit 0
```

Yet another twist of this nifty trick makes "self-documenting" scripts possible.

Example 17-11. 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
  cat "$0" | sed --silent -e '/DOCUMENTATIONXX$/,/^DOCUMENTATION/p' |
  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.
```