```
#include <unistd.h>
```

These also refer to header files, but they refer to header files that live outside the current source tree. They are supplied by the system to support the compilation of every program. If we look in `/usr/include`, we can see them:

```
[me@linuxbox diction-1.11]$ ls /usr/include
```

The header files in this directory were installed when we installed the compiler.

Building The Program

Most programs build with a simple two command sequence:

```
./configure
make
```

The `configure` program is a shell script which is supplied with the source tree. Its job is to analyze the *build environment*. Most source code is designed to be *portable*. That is, it is designed to build on more than one kind of Unix-like system. But in order to do that, the source code may need to undergo slight adjustments during the build to accommodate differences between systems. `configure` also checks to see that necessary external tools and components are installed. Let's run `configure`. Since `configure` is not located where the shell normally expects programs to be located, we must explicitly tell the shell its location by prefixing the command with `./` to indicate that the program is located in the current working directory:

```
[me@linuxbox diction-1.11]$ ./configure
```

`configure` will output a lot of messages as it tests and configures the build. When it finishes, it will look something like this:

```
checking libintl.h presence... yes
checking for libintl.h... yes
checking for library containing gettext... none required
configure: creating ./config.status
```

```
config.status: creating Makefile
config.status: creating diction.1
config.status: creating diction.texi
config.status: creating diction.spec
config.status: creating style.1
config.status: creating test/rundiction
config.status: creating config.h
[me@linuxbox diction-1.11]$
```

What's important here is that there are no error messages. If there were, the configuration failed, and the program will not build until the errors are corrected.

We see `configure` created several new files in our source directory. The most important one is `Makefile`. `Makefile` is a configuration file that instructs the `make` program exactly how to build the program. Without it, `make` will refuse to run. `Makefile` is an ordinary text file, so we can view it:

```
[me@linuxbox diction-1.11]$ less Makefile
```

The `make` program takes as input a *makefile* (which is normally named `Makefile`), which describes the relationships and dependencies among the components that comprise the finished program.

The first part of the makefile defines variables that are substituted in later sections of the makefile. For example we see the line:

```
CC=             gcc
```

which defines the C compiler to be `gcc`. Later in the makefile, we see one instance where it gets used:

```
diction:        diction.o sentence.o misc.o getopt.o getopt1.o
                $(CC) -o $@ $(LDFLAGS) diction.o sentence.o misc.o \
                getopt.o getopt1.o $(LIBS)
```

A substitution is performed here, and the value `$(CC)` is replaced by `gcc` at run time.

Most of the makefile consists of lines, which define a *target*, in this case the executable file `diction`, and the files on which it is dependent. The remaining lines describe the command(s) needed to create the target from its components. We see in this example that

the executable file `diction` (one of the final end products) depends on the existence of `diction.o`, `sentence.o`, `misc.o`, `getopt.o`, and `getopt1.o`. Later on, in the makefile, we see definitions of each of these as targets:

```
diction.o:      diction.c config.h getopt.h misc.h sentence.h
getopt.o:       getopt.c getopt.h getopt_int.h
getopt1.o:      getopt1.c getopt.h getopt_int.h
misc.o:         misc.c config.h misc.h
sentence.o:     sentence.c config.h misc.h sentence.h
style.o:        style.c config.h getopt.h misc.h sentence.h
```

However, we don't see any command specified for them. This is handled by a general target, earlier in the file, that describes the command used to compile any `.c` file into a `.o` file:

```
.c.o:
                $(CC) -c $(CPPFLAGS) $(CFLAGS) $<
```

This all seems very complicated. Why not simply list all the steps to compile the parts and be done with it? The answer to this will become clear in a moment. In the meantime, let's run `make` and build our program:

```
[me@linuxbox diction-1.11]$ make
```

The `make` program will run, using the contents of `Makefile` to guide its actions. It will produce a lot of messages.

When it finishes, we will see that all the targets are now present in our directory:

```
[me@linuxbox diction-1.11]$  ls
config.guess  de.po            en            install-sh   sentence.c
config.h      diction          en_GB         Makefile     sentence.h
config.h.in   diction.1        en_GB.mo      Makefile.in  sentence.o
config.log    diction.1.in     en_GB.po      misc.c       style
config.status diction.c        getopt1.c     misc.h       style.1
config.sub    diction.o        getopt1.o     misc.o       style.1.in
configure     diction.pot      getopt.c      NEWS         style.c
configure.in  diction.spec     getopt.h      nl           style.o
COPYING       diction.spec.in  getopt_int.h  nl.mo        test
de            diction.texi     getopt.o      nl.po
```

```
de.mo          diction.texi.in  INSTALL       README
```

Among the files, we see `diction` and `style`, the programs that we set out to build. Congratulations are in order! We just compiled our first programs from source code!

But just out of curiosity, let's run `make` again:

```
[me@linuxbox diction-1.11]$ make
make: Nothing to be done for `all'.
```

It only produces this strange message. What's going on? Why didn't it build the program again? Ah, this is the magic of `make`. Rather than simply building everything again, `make` only builds what needs building. With all of the targets present, `make` determined that there was nothing to do. We can demonstrate this by deleting one of the targets and running make again to see what it does. Let's get rid of one of the intermediate targets:

```
[me@linuxbox diction-1.11]$ rm getopt.o
[me@linuxbox diction-1.11]$ make
```

We see that `make` rebuilds it and re-links the `diction` and `style` programs, since they depend on the missing module. This behavior also points out another important feature of `make`: it keeps targets up to date. `make` insists that targets be newer than their dependencies. This makes perfect sense, as a programmer will often update a bit of source code and then use `make` to build a new version of the finished product. `make` ensures that everything that needs building based on the updated code is built. If we use the `touch` program to "update" one of the source code files, we can see this happen:

```
[me@linuxbox diction-1.11]$ ls -l diction getopt.c
-rwxr-xr-x 1 me      me      37164 2009-03-05 06:14 diction
-rw-r--r-- 1 me      me      33125 2007-03-30 17:45 getopt.c
[me@linuxbox diction-1.11]$ touch getopt.c
[me@linuxbox diction-1.11]$ ls -l diction getopt.c
-rwxr-xr-x 1 me      me      37164 2009-03-05 06:14 diction
-rw-r--r-- 1 me      me      33125 2009-03-05 06:23 getopt.c
[me@linuxbox diction-1.11]$ make
```

After `make` runs, we see that it has restored the target to being newer than the dependency:

```
[me@linuxbox diction-1.11]$  ls -l diction getopt.c
-rwxr-xr-x 1 me      me      37164 2009-03-05 06:24 diction
-rw-r--r-- 1 me      me      33125 2009-03-05 06:23 getopt.c
```

The ability of `make` to intelligently build only what needs building is a great benefit to programmers. While the time savings may not be very apparent with our small project, it is very significant with larger projects. Remember, the Linux kernel (a program that undergoes continuous modification and improvement) contains several *million* lines of code.

Installing The Program

Well-packaged source code will often include a special `make` target called `install`. This target will install the final product in a system directory for use. Usually, this directory is `/usr/local/bin`, the traditional location for locally built software. However, this directory is not normally writable by ordinary users, so we must become the superuser to perform the installation:

```
[me@linuxbox diction-1.11]$ sudo make install
```

After we perform the installation, we can check that the program is ready to go:

```
[me@linuxbox diction-1.11]$ which diction
/usr/local/bin/diction
[me@linuxbox diction-1.11]$ man diction
```

And there we have it!

Summing Up

In this chapter, we have seen how three simple commands:

```
./configure
make
make install
```

can be used to build many source code packages. We have also seen the important role that `make` plays in the maintenance of programs. The `make` program can be used for any task that needs to maintain a target/dependency relationship, not just for compiling source code.

Further Reading

- The Wikipedia has good articles on compilers and the `make` program:
  http://en.wikipedia.org/wiki/Compiler
  http://en.wikipedia.org/wiki/Make_(software)
- *The GNU Make Manual*:
  http://www.gnu.org/software/make/manual/html_node/index.html

Part 5 – Writing Shell Scripts

25 – Writing Your First Script

In the preceding chapters, we have assembled an arsenal of command line tools. While these tools can solve many kinds of computing problems, we are still limited to manually using them one by one on the command line. Wouldn't it be great if we could get the shell to do more of the work? We can. By joining our tools together into programs of our own design, the shell can carry out complex sequences of tasks all by itself. We can enable it to do this by writing *shell scripts*.

What Are Shell Scripts?

In the simplest terms, a shell script is a file containing a series of commands. The shell reads this file and carries out the commands as though they have been entered directly on the command line.

The shell is somewhat unique, in that it is both a powerful command line interface to the system and a scripting language interpreter. As we will see, most of the things that can be done on the command line can be done in scripts, and most of the things that can be done in scripts can be done on the command line.

We have covered many shell features, but we have focused on those features most often used directly on the command line. The shell also provides a set of features usually (but not always) used when writing programs.

How To Write A Shell Script

To successfully create and run a shell script, we need to do three things:

1. **Write a script.** Shell scripts are ordinary text files. So we need a text editor to write them. The best text editors will provide *syntax highlighting*, allowing us to see a color-coded view of the elements of the script. Syntax highlighting will help us spot certain kinds of common errors. `vim`, `gedit`, `kate`, and many other editors are good candidates for writing scripts.
2. **Make the script executable.** The system is rather fussy about not letting any old text file be treated as a program, and for good reason! We need to set the script file's permissions to allow execution.

3. **Put the script somewhere the shell can find it.** The shell automatically searches certain directories for executable files when no explicit pathname is specified. For maximum convenience, we will place our scripts in these directories.

Script File Format

In keeping with programming tradition, we'll create a "hello world" program to demonstrate an extremely simple script. So let's fire up our text editors and enter the following script:

```
#!/bin/bash

# This is our first script.

echo 'Hello World!'
```

The last line of our script is pretty familiar, just an `echo` command with a string argument. The second line is also familiar. It looks like a comment that we have seen used in many of the configuration files we have examined and edited. One thing about comments in shell scripts is that they may also appear at the end of lines, like so:

```
echo 'Hello World!' # This is a comment too
```

Everything from the # symbol onward on the line is ignored.

Like many things, this works on the command line, too:

```
[me@linuxbox ~]$ echo 'Hello World!' # This is a comment too
Hello World!
```

Though comments are of little use on the command line, they will work.

The first line of our script is a little mysterious. It looks like it should be a comment, since it starts with #, but it looks too purposeful to be just that. The `#!` character sequence is, in fact, a special construct called a *shebang*. The shebang is used to tell the system the name of the interpreter that should be used to execute the script that follows. Every shell script should include this as its first line.

Let's save our script file as `hello_world`.

Executable Permissions

The next thing we have to do is make our script executable. This is easily done using `chmod`:

```
[me@linuxbox ~]$ ls -l hello_world
-rw-r--r-- 1 me      me      63 2009-03-07 10:10 hello_world
[me@linuxbox ~]$ chmod 755 hello_world
[me@linuxbox ~]$ ls -l hello_world
-rwxr-xr-x 1 me      me      63 2009-03-07 10:10 hello_world
```

There are two common permission settings for scripts; 755 for scripts that everyone can execute, and 700 for scripts that only the owner can execute. Note that scripts must be readable in order to be executed.

Script File Location

With the permissions set, we can now execute our script:

```
[me@linuxbox ~]$ ./hello_world
Hello World!
```

In order for the script to run, we must precede the script name with an explicit path. If we don't, we get this:

```
[me@linuxbox ~]$ hello_world
bash: hello_world: command not found
```

Why is this? What makes our script different from other programs? As it turns out, nothing. Our script is fine. Its location is the problem. Back in Chapter 12, we discussed the PATH environment variable and its effect on how the system searches for executable programs. To recap, the system searches a list of directories each time it needs to find an executable program, if no explicit path is specified. This is how the system knows to execute `/bin/ls` when we type `ls` at the command line. The `/bin` directory is one of the directories that the system automatically searches. The list of directories is held within an environment variable named PATH. The PATH variable contains a colon-separated list of directories to be searched. We can view the contents of PATH:

```
[me@linuxbox ~]$ echo $PATH
/home/me/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:
/bin:/usr/games
```

Here we see our list of directories. If our script were located in any of the directories in the list, our problem would be solved. Notice the first directory in the list, `/home/me/bin`. Most Linux distributions configure the `PATH` variable to contain a `bin` directory in the user's home directory, to allow users to execute their own programs. So if we create the `bin` directory and place our script within it, it should start to work like other programs:

```
[me@linuxbox ~]$ mkdir bin
[me@linuxbox ~]$ mv hello_world bin
[me@linuxbox ~]$ hello_world
Hello World!
```

And so it does.

If the `PATH` variable does not contain the directory, we can easily add it by including this line in our `.bashrc` file:

```
export PATH=~/bin:"$PATH"
```

After this change is made, it will take effect in each new terminal session. To apply the change to the current terminal session, we must have the shell re-read the `.bashrc` file. This can be done by "sourcing" it:

```
[me@linuxbox ~]$ . .bashrc
```

The dot (`.`) command is a synonym for the `source` command, a shell builtin which reads a specified file of shell commands and treats it like input from the keyboard.

Note: Ubuntu automatically adds the `~/bin` directory to the `PATH` variable if the `~/bin` directory exists when the user's `.bashrc` file is executed. So, on Ubuntu systems, if we create the `~/bin` directory and then log out and log in again, everything works.

Good Locations For Scripts

The `~/bin` directory is a good place to put scripts intended for personal use. If we write a script that everyone on a system is allowed to use, the traditional location is `/usr/local/bin`. Scripts intended for use by the system administrator are often located in `/usr/local/sbin`. In most cases, locally supplied software, whether scripts or compiled programs, should be placed in the `/usr/local` hierarchy and not in `/bin` or `/usr/bin`. These directories are specified by the Linux Filesystem Hierarchy Standard to contain only files supplied and maintained by the Linux distributor.

More Formatting Tricks

One of the key goals of serious script writing is ease of *maintenance*, that is, the ease with which a script may be modified by its author or others to adapt it to changing needs. Making a script easy to read and understand is one way to facilitate easy maintenance.

Long Option Names

Many of the commands we have studied feature both short and long option names. For instance, the `ls` command has many options that can be expressed in either short or long form. For example:

```
[me@linuxbox ~]$ ls -ad
```

and:

```
[me@linuxbox ~]$ ls --all --directory
```

are equivalent commands. In the interests of reduced typing, short options are preferred when entering options on the command line, but when writing scripts, long options can provide improved readability.

Indentation And Line Continuation

When employing long commands, readability can be enhanced by spreading the command over several lines. In Chapter 18, we looked at a particularly long example of the find command:

```
[me@linuxbox ~]$ find playground \( -type f -not -perm 0600 -exec
chmod 0600 '{}' ';' \) -or \( -type d -not -perm 0711 -exec chmod
0711 '{}' ';' \)
```

Obviously, this command is a little hard to figure out at first glance. In a script, this command might be easier to understand if written this way:

```
find playground \
      \( \
            -type f \
            -not -perm 0600 \
            -exec chmod 0600 '{}' ';' \
      \) \
      -or \
      \( \
            -type d \
            -not -perm 0711 \
            -exec chmod 0711 '{}' ';' \
      \)
```

By using line continuations (backslash-linefeed sequences) and indentation, the logic of this complex command is more clearly described to the reader. This technique works on the command line, too, though it is seldom used, as it is very awkward to type and edit. One difference between a script and the command line is that the script may employ tab characters to achieve indentation, whereas the command line cannot, since tabs are used to activate completion.

Configuring vim For Script Writing

The `vim` text editor has many, many configuration settings. There are several common options that can facilitate script writing:

`:syntax on`

turns on syntax highlighting. With this setting, different elements of shell syntax will be displayed in different colors when viewing a script. This is helpful for identifying certain kinds of programming errors. It looks cool, too. Note that for this feature to work, you must have a complete version of `vim` installed, and the file you are editing must have a shebang indicating the file is a shell script. If you have difficulty with the command above, try **`:set syntax=sh`** instead.

`:set hlsearch`

turns on the option to highlight search results. Say we search for the word "echo." With this option on, each instance of the word will be highlighted.

`:set tabstop=4`

sets the number of columns occupied by a tab character. The default is eight columns. Setting the value to four (which is a common practice) allows long lines to fit more easily on the screen.

`:set autoindent`

turns on the "auto indent" feature. This causes `vim` to indent a new line the same amount as the line just typed. This speeds up typing on many kinds of programming constructs. To stop indentation, type `Ctrl-d`.

These changes can be made permanent by adding these commands (without the leading colon characters) to your `~/.vimrc` file.

Summing Up

In this first chapter of scripting, we have looked at how scripts are written and made to easily execute on our system. We also saw how we may use various formatting techniques to improve the readability (and thus, the maintainability) of our scripts. In future chapters, ease of maintenance will come up again and again as a central principle in good script writing.

Further Reading

- For "Hello World" programs and examples in various programming languages, see:
  http://en.wikipedia.org/wiki/Hello_world
- This Wikipedia article talks more about the shebang mechanism:
  http://en.wikipedia.org/wiki/Shebang_(Unix)

26 – Starting A Project

Starting with this chapter, we will begin to build a program. The purpose of this project is to see how various shell features are used to create programs and, more importantly, create *good* programs.

The program we will write is a *report generator*. It will present various statistics about our system and its status, and will produce this report in HTML format, so we can view it with a web browser such as Firefox or Konqueror.

Programs are usually built up in a series of stages, with each stage adding features and capabilities. The first stage of our program will produce a very minimal HTML page that contains no system information. That will come later.

First Stage: Minimal Document

The first thing we need to know is the format of a well-formed HTML document. It looks like this:

```
<HTML>
      <HEAD>
            <TITLE>Page Title</TITLE>
      </HEAD>
      <BODY>
            Page body.
      </BODY>
</HTML>
```

If we enter this into our text editor and save the file as `foo.html`, we can use the following URL in Firefox to view the file:

`file:///home/`*`username`*`/foo.html`

The first stage of our program will be able to output this HTML file to standard output. We can write a program to do this pretty easily. Let's start our text editor and create a new file named `~/bin/sys_info_page`:

```
[me@linuxbox ~]$ vim ~/bin/sys_info_page
```

and enter the following program:

```
#!/bin/bash

# Program to output a system information page

echo "<HTML>"
echo "      <HEAD>"
echo "            <TITLE>Page Title</TITLE>"
echo "      </HEAD>"
echo "      <BODY>"
echo "            Page body."
echo "      </BODY>"
echo "</HTML>"
```

Our first attempt at this problem contains a shebang, a comment (always a good idea) and a sequence of `echo` commands, one for each line of output. After saving the file, we'll make it executable and attempt to run it:

```
[me@linuxbox ~]$ chmod 755 ~/bin/sys_info_page
[me@linuxbox ~]$ sys_info_page
```

When the program runs, we should see the text of the HTML document displayed on the screen, since the `echo` commands in the script send their output to standard output. We'll run the program again and redirect the output of the program to the file `sys_info_page.html`, so that we can view the result with a web browser:

```
[me@linuxbox ~]$ sys_info_page > sys_info_page.html
[me@linuxbox ~]$ firefox sys_info_page.html
```

So far, so good.

When writing programs, it's always a good idea to strive for simplicity and clarity. Maintenance is easier when a program is easy to read and understand, not to mention, it can make the program easier to write by reducing the amount of typing. Our current version of the program works fine, but it could be simpler. We could actually combine all the echo commands into one, which will certainly make it easier to add more lines to the program's output. So, let's change our program to this:

```
#!/bin/bash

# Program to output a system information page

echo "<HTML>
      <HEAD>
            <TITLE>Page Title</TITLE>
      </HEAD>
      <BODY>
            Page body.
      </BODY>
</HTML>"
```

A quoted string may include newlines, and therefore contain multiple lines of text. The shell will keep reading the text until it encounters the closing quotation mark. It works this way on the command line, too:

```
[me@linuxbox ~]$ echo "<HTML>
>         <HEAD>
>                 <TITLE>Page Title</TITLE>
>         </HEAD>
>         <BODY>
>                 Page body.
>         </BODY>
> </HTML>"
```

The leading ">" character is the shell prompt contained in the PS2 shell variable. It appears whenever we type a multi-line statement into the shell. This feature is a little obscure right now, but later, when we cover multi-line programming statements, it will turn out to be quite handy.

Second Stage: Adding A Little Data

Now that our program can generate a minimal document, let's put some data in the report. To do this, we will make the following changes:

```
#!/bin/bash

# Program to output a system information page

echo "<HTML>
        <HEAD>
                <TITLE>System Information Report</TITLE>
```

```
        </HEAD>
        <BODY>
                <H1>System Information Report</H1>
        </BODY>
</HTML>"
```

We added a page title and a heading to the body of the report.

Variables And Constants

There is an issue with our script, however. Notice how the string "System Information Report" is repeated? With our tiny script it's not a problem, but let's imagine that our script was really long and we had multiple instances of this string. If we wanted to change the title to something else, we would have to change it in multiple places, which could be a lot of work. What if we could arrange the script so that the string only appeared once and not multiple times? That would make future maintenance of the script much easier. Here's how we could do that:

```
#!/bin/bash

# Program to output a system information page

title="System Information Report"

echo "<HTML>
        <HEAD>
                <TITLE>$title</TITLE>
        </HEAD>
        <BODY>
                <H1>$title</H1>
        </BODY>
</HTML>"
```

By creating a *variable* named `title` and assigning it the value "System Information Report," we can take advantage of parameter expansion and place the string in multiple locations.

So, how do we create a variable? Simple, we just use it. When the shell encounters a variable, it automatically creates it. This differs from many programming languages in which variables must be explicitly *declared* or defined before use. The shell is very lax about this, which can lead to some problems. For example, consider this scenario played out on the command line:

```
[me@linuxbox ~]$ foo="yes"
[me@linuxbox ~]$ echo $foo
yes
[me@linuxbox ~]$ echo $fool

[me@linuxbox ~]$
```

We first assign the value “yes” to the variable `foo`, then display its value with `echo`. Next we display the value of the variable name misspelled as “fool” and get a blank result. This is because the shell happily created the variable `fool` when it encountered it, and gave it the default value of nothing, or empty. From this, we learn that we must pay close attention to our spelling! It’s also important to understand what really happened in this example. From our previous look at how the shell performs expansions, we know that the command:

```
[me@linuxbox ~]$ echo $foo
```

undergoes parameter expansion and results in:

```
[me@linuxbox ~]$ echo yes
```

Whereas the command:

```
[me@linuxbox ~]$ echo $fool
```

expands into:

```
[me@linuxbox ~]$ echo
```

The empty variable expands into nothing! This can play havoc with commands that require arguments. Here’s an example:

```
[me@linuxbox ~]$ foo=foo.txt
[me@linuxbox ~]$ foo1=foo1.txt
[me@linuxbox ~]$ cp $foo $fool
cp: missing destination file operand after `foo.txt'
```

```
Try `cp --help' for more information.
```

We assign values to two variables, `foo` and `foo1`. We then perform a `cp`, but misspell the name of the second argument. After expansion, the `cp` command is only sent one argument, though it requires two.

There are some rules about variable names:

1. Variable names may consist of alphanumeric characters (letters and numbers) and underscore characters.
2. The first character of a variable name must be either a letter or an underscore.
3. Spaces and punctuation symbols are not allowed.

The word "variable" implies a value that changes, and in many applications, variables are used this way. However, the variable in our application, `title`, is used as a *constant*. A constant is just like a variable in that it has a name and contains a value. The difference is that the value of a constant does not change. In an application that performs geometric calculations, we might define `PI` as a constant, and assign it the value of `3.1415`, instead of using the number literally throughout our program. The shell makes no distinction between variables and constants; they are mostly for the programmer's convenience. A common convention is to use upper case letters to designate constants and lower case letters for true variables. We will modify our script to comply with this convention:

```
#!/bin/bash

# Program to output a system information page

TITLE="System Information Report For $HOSTNAME"

echo "<HTML>
        <HEAD>
                <TITLE>$TITLE</TITLE>
        </HEAD>
        <BODY>
                <H1>$TITLE</H1>
        </BODY>
</HTML>"
```

We also took the opportunity to jazz up our title by adding the value of the shell variable `HOSTNAME`. This is the network name of the machine.

Note: The shell actually does provide a way to enforce the immutability of constants, through the use of the `declare` builtin command with the `-r` (read-only) option. Had we assigned `TITLE` this way:

```
declare -r TITLE="Page Title"
```

the shell would prevent any subsequent assignment to `TITLE`. This feature is rarely used, but it exists for very formal scripts.

Assigning Values To Variables And Constants

Here is where our knowledge of expansion really starts to pay off. As we have seen, variables are assigned values this way:

`variable=value`

where *variable* is the name of the variable and *value* is a string. Unlike some other programming languages, the shell does not care about the type of data assigned to a variable; it treats them all as strings. You can force the shell to restrict the assignment to integers by using the `declare` command with the `-i` option, but, like setting variables as read-only, this is rarely done.

Note that in an assignment, there must be no spaces between the variable name, the equals sign, and the value. So what can the value consist of? Anything that we can expand into a string:

```
a=z                     # Assign the string "z" to variable a.
b="a string"            # Embedded spaces must be within quotes.
c="a string and $b"     # Other expansions such as variables can be
                        # expanded into the assignment.
d=$(ls -l foo.txt)      # Results of a command.
e=$((5 * 7))            # Arithmetic expansion.
f="\t\ta string\n"      # Escape sequences such as tabs and newlines.
```

Multiple variable assignments may be done on a single line:

```
a=5 b="a string"
```

During expansion, variable names may be surrounded by optional curly braces “{}”. This is useful in cases where a variable name becomes ambiguous due to its surrounding

context. Here, we try to change the name of a file from `myfile` to `myfile1`, using a variable:

```
[me@linuxbox ~]$ filename="myfile"
[me@linuxbox ~]$ touch $filename
[me@linuxbox ~]$ mv $filename $filename1
mv: missing destination file operand after `myfile'
Try `mv --help' for more information.
```

This attempt fails because the shell interprets the second argument of the `mv` command as a new (and empty) variable. The problem can be overcome this way:

```
[me@linuxbox ~]$ mv $filename ${filename}1
```

By adding the surrounding braces, the shell no longer interprets the trailing `1` as part of the variable name.

We'll take this opportunity to add some data to our report, namely the date and time the report was created and the user name of the creator:

```
#!/bin/bash

# Program to output a system information page

TITLE="System Information Report For $HOSTNAME"
CURRENT_TIME=$(date +"%x %r %Z")
TIME_STAMP="Generated $CURRENT_TIME, by $USER"

echo "<HTML>
        <HEAD>
                <TITLE>$TITLE</TITLE>
        </HEAD>
        <BODY>
                <H1>$TITLE</H1>
                <P>$TIME_STAMP</P>
        </BODY>
</HTML>"
```


Here Documents

We've looked at two different methods of outputting our text, both using the `echo`

command. There is a third way called a *here document* or *here script*. A here document is an additional form of I/O redirection in which we embed a body of text into our script and feed it into the standard input of a command. It works like this:

```
command << token
text
token
```

where *command* is the name of command that accepts standard input and *token* is a string used to indicate the end of the embedded text. We'll modify our script to use a here document:

```
#!/bin/bash

# Program to output a system information page

TITLE="System Information Report For $HOSTNAME"
CURRENT_TIME=$(date +"%x %r %Z")
TIME_STAMP="Generated $CURRENT_TIME, by $USER"

cat << _EOF_
<HTML>
        <HEAD>
                <TITLE>$TITLE</TITLE>
        </HEAD>
        <BODY>
                <H1>$TITLE</H1>
                <P>$TIME_STAMP</P>
        </BODY>
</HTML>
_EOF_
```

Instead of using `echo`, our script now uses `cat` and a here document. The string `_EOF_` (meaning "End Of File," a common convention) was selected as the token, and marks the end of the embedded text. Note that the token must appear alone and that there must not be trailing spaces on the line.

So what's the advantage of using a here document? It's mostly the same as `echo`, except that, by default, single and double quotes within here documents lose their special meaning to the shell. Here is a command line example:

```
[me@linuxbox ~]$ foo="some text"
[me@linuxbox ~]$ cat << _EOF_
> $foo
```

```
> "$foo"
> '$foo'
> \$foo
> _EOF_
some text
"some text"
'some text'
$foo
```

As we can see, the shell pays no attention to the quotation marks. It treats them as ordinary characters. This allows us to embed quotes freely within a here document. This could turn out to be handy for our report program.

Here documents can be used with any command that accepts standard input. In this example, we use a here document to pass a series of commands to the `ftp` program in order to retrieve a file from a remote FTP server:

```
#!/bin/bash

# Script to retrieve a file via FTP

FTP_SERVER=ftp.nl.debian.org
FTP_PATH=/debian/dists/lenny/main/installer-i386/current/images/cdrom
REMOTE_FILE=debian-cd_info.tar.gz

ftp -n << _EOF_
open $FTP_SERVER
user anonymous me@linuxbox
cd $FTP_PATH
hash
get $REMOTE_FILE
bye
_EOF_
ls -l $REMOTE_FILE
```

If we change the redirection operator from "<<" to "<<-", the shell will ignore leading tab characters in the here document. This allows a here document to be indented, which can improve readability:

```
#!/bin/bash

# Script to retrieve a file via FTP

FTP_SERVER=ftp.nl.debian.org
```

```
FTP_PATH=/debian/dists/lenny/main/installer-i386/current/images/cdrom
REMOTE_FILE=debian-cd_info.tar.gz

ftp -n <<- _EOF_
      open $FTP_SERVER
      user anonymous me@linuxbox
      cd $FTP_PATH
      hash
      get $REMOTE_FILE
      bye
      _EOF_

ls -l $REMOTE_FILE
```

Summing Up

In this chapter, we started a project that will carry us through the process of building a successful script. We introduced the concept of variables and constants and how they can be employed. They are the first of many applications we will find for parameter expansion. We also looked at how to produce output from our script, and various methods for embedding blocks of text.

Further Reading

- For more information about HTML, see the following articles and tutorials:
  http://en.wikipedia.org/wiki/Html
  http://en.wikibooks.org/wiki/HTML_Programming
  http://html.net/tutorials/html/
- The `bash` man page includes a section entitled “HERE DOCUMENTS,” which has a full description of this feature.

27 – Top-Down Design

As programs get larger and more complex, they become more difficult to design, code and maintain. As with any large project, it is often a good idea to break large, complex tasks into a series of small, simple tasks. Let's imagine that we are trying to describe a common, everyday task, going to the market to buy food, to a person from Mars. We might describe the overall process as the following series of steps:

1. Get in car.
2. Drive to market.
3. Park car.
4. Enter market.
5. Purchase food.
6. Return to car.
7. Drive home.
8. Park car.
9. Enter house.

However, a person from Mars is likely to need more detail. We could further break down the subtask "Park car" into this series of steps:

1. Find parking space.
2. Drive car into space.
3. Turn off motor.
4. Set parking brake.
5. Exit car.
6. Lock car.

The "Turn off motor" subtask could further be broken down into steps including "Turn off ignition," "Remove ignition key" and so on, until every step of the entire process of going to the market has been fully defined.

This process of identifying the top-level steps and developing increasingly detailed views of those steps is called *top-down design*. This technique allows us to break large complex tasks into many small, simple tasks. Top-down design is a common method of designing

programs and one that is well suited to shell programming in particular.

In this chapter, we will use top-down design to further develop our report generator script.

Shell Functions

Our script currently performs the following steps to generate the HTML document:

1. Open page.
2. Open page header.
3. Set page title.
4. Close page header.
5. Open page body.
6. Output page heading.
7. Output time stamp.
8. Close page body.
9. Close page.

For our next stage of development, we will add some additional tasks between steps 7 and 8. These will include:

- System uptime and load. This is the amount of time since the last shutdown or reboot and the average number of tasks currently running on the processor over several time intervals.
- Disk space. The overall use of space on the system's storage devices.
- Home space. The amount of storage space being used by each user.

If we had a command for each of these tasks, we could add them to our script simply through command substitution:

```
#!/bin/bash

# Program to output a system information page

TITLE="System Information Report For $HOSTNAME"
CURRENT_TIME=$(date +"%x %r %Z")
TIME_STAMP="Generated $CURRENT_TIME, by $USER"

cat << _EOF_
<HTML>
        <HEAD>
```

```
                <TITLE>$TITLE</TITLE>
        </HEAD>
        <BODY>
                <H1>$TITLE</H1>
                <P>$TIME_STAMP</P>
                $(report_uptime)
                $(report_disk_space)
                $(report_home_space)
        </BODY>
</HTML>
_EOF_
```

We could create these additional commands two ways. We could write three separate scripts and place them in a directory listed in our PATH, or we could embed the scripts within our program as *shell functions*. As we have mentioned before, shell functions are "mini-scripts" that are located inside other scripts and can act as autonomous programs. Shell functions have two syntactic forms:

```
function name {
     commands
     return
)
```

and

```
name () {
     commands
     return
}
```

where *name* is the name of the function and *commands* are a series of commands contained within the function. Both forms are equivalent and may be used interchangeably. Below we see a script that demonstrates the use of a shell function:

```
 1    #!/bin/bash
 2
 3    # Shell function demo
 4
 5    function funct {
 6         echo "Step 2"
 7         return
 8    }
 9
10    # Main program starts here
11
12    echo "Step 1"
```

```
13    funct
14    echo "Step 3"
```

As the shell reads the script, it passes over lines 1 through 11, as those lines consist of comments and the function definition. Execution begins at line 12, with an `echo` command. Line 13 *calls* the shell function `funct` and the shell executes the function just as it would any other command. Program control then moves to line 6, and the second `echo` command is executed. Line 7 is executed next. Its `return` command terminates the function and returns control to the program at the line following the function call (line 14), and the final `echo` command is executed. Note that in order for function calls to be recognized as shell functions and not interpreted as the names of external programs, shell function definitions must appear in the script before they are called.

We'll add minimal shell function definitions to our script:

```
#!/bin/bash

# Program to output a system information page

TITLE="System Information Report For $HOSTNAME"
CURRENT_TIME=$(date +"%x %r %Z")
TIME_STAMP="Generated $CURRENT_TIME, by $USER"

report_uptime () {
      return
}

report_disk_space () {
      return
}

report_home_space () {
      return
}

cat << _EOF_
<HTML>
      <HEAD>
            <TITLE>$TITLE</TITLE>
      </HEAD>
      <BODY>
            <H1>$TITLE</H1>
            <P>$TIME_STAMP</P>
            $(report_uptime)
            $(report_disk_space)
```

```
            $(report_home_space)
      </BODY>
</HTML>
_EOF_
```

Shell function names follow the same rules as variables. A function must contain at least one command. The `return` command (which is optional) satisfies the requirement.

Local Variables

In the scripts we have written so far, all the variables (including constants) have been *global variables*. Global variables maintain their existence throughout the program. This is fine for many things, but it can sometimes complicate the use of shell functions. Inside shell functions, it is often desirable to have *local variables*. Local variables are only accessible within the shell function in which they are defined and cease to exist once the shell function terminates.

Having local variables allows the programmer to use variables with names that may already exist, either in the script globally or in other shell functions, without having to worry about potential name conflicts.

Here is an example script that demonstrates how local variables are defined and used:

```
#!/bin/bash

# local-vars: script to demonstrate local variables

foo=0 # global variable foo

funct_1 () {

      local foo   # variable foo local to funct_1

      foo=1
      echo "funct_1: foo = $foo"
}

funct_2 () {

      local foo   # variable foo local to funct_2

      foo=2
      echo "funct_2: foo = $foo"
}

echo "global:  foo = $foo"
```

```
funct_1
echo "global:  foo = $foo"
funct_2
echo "global:  foo = $foo"
```

As we can see, local variables are defined by preceding the variable name with the word `local`. This creates a variable that is local to the shell function in which it is defined. Once outside the shell function, the variable no longer exists. When we run this script, we see the results:

```
[me@linuxbox ~]$ local-vars
global:  foo = 0
funct_1: foo = 1
global:  foo = 0
funct_2: foo = 2
global:  foo = 0
```

We see that the assignment of values to the local variable `foo` within both shell functions has no effect on the value of `foo` defined outside the functions.

This feature allows shell functions to be written so that they remain independent of each other and of the script in which they appear. This is very valuable, as it helps prevent one part of a program from interfering with another. It also allows shell functions to be written so that they can be portable. That is, they may be cut and pasted from script to script, as needed.

Keep Scripts Running

While developing our program, it is useful to keep the program in a runnable state. By doing this, and testing frequently, we can detect errors early in the development process. This will make debugging problems much easier. For example, if we run the program, make a small change, then run the program again and find a problem, it's very likely that the most recent change is the source of the problem. By adding the empty functions, called *stubs* in programmer-speak, we can verify the logical flow of our program at an early stage. When constructing a stub, it's a good idea to include something that provides feedback to the programmer, which shows the logical flow is being carried out. If we look at the output of our script now:


```
[me@linuxbox ~]$ sys_info_page
<HTML>
      <HEAD>
```

```
            <TITLE>System Information Report For twin2</TITLE>
      </HEAD>
      <BODY>
            <H1>System Information Report For linuxbox</H1>
            <P>Generated 03/19/2009 04:02:10 PM EDT, by me</P>



      </BODY>
</HTML>
```

we see that there are some blank lines in our output after the time stamp, but we can't be sure of the cause. If we change the functions to include some feedback:

```
report_uptime () {
        echo "Function report_uptime executed."
        return
}

report_disk_space () {
        echo "Function report_disk_space executed."
        return
}

report_home_space () {
        echo "Function report_home_space executed."
        return
}
```

and run the script again:

```
[me@linuxbox ~]$ sys_info_page
<HTML>
      <HEAD>
            <TITLE>System Information Report For linuxbox</TITLE>
      </HEAD>
      <BODY>
            <H1>System Information Report For linuxbox</H1>
            <P>Generated 03/20/2009 05:17:26 AM EDT, by me</P>
            Function report_uptime executed.
            Function report_disk_space executed.
            Function report_home_space executed.
      </BODY>
</HTML>
```

we now see that, in fact, our three functions are being executed.

With our function framework in place and working, it's time to flesh out some of the function code. First, the `report_uptime` function:

```
report_uptime () {
      cat <<- _EOF_
            <H2>System Uptime</H2>
            <PRE>$(uptime)</PRE>
            _EOF_
      return
}
```

It's pretty straightforward. We use a here document to output a section header and the output of the `uptime` command, surrounded by `<PRE>` tags to preserve the formatting of the command. The `report_disk_space` function is similar:

```
report_disk_space () {
      cat <<- _EOF_
            <H2>Disk Space Utilization</H2>
            <PRE>$(df -h)</PRE>
            _EOF_
      return
}
```

This function uses the `df -h` command to determine the amount of disk space. Lastly, we'll build the `report_home_space` function:

```
report_home_space () {
      cat <<- _EOF_
            <H2>Home Space Utilization</H2>
            <PRE>$(du -sh /home/*)</PRE>
            _EOF_
      return
}
```

We use the `du` command with the `-sh` options to perform this task. This, however, is not a complete solution to the problem. While it will work on some systems (Ubuntu, for example), it will not work on others. The reason is that many systems set the permissions of home directories to prevent them from being world-readable, which is a reasonable security measure. On these systems, the `report_home_space` function, as written,

will only work if our script is run with superuser privileges. A better solution would be to have the script could adjust its behavior according to the privileges of the user. We will take this up in the next chapter.

Shell Functions In Your `.bashrc` File

Shell functions make excellent replacements for aliases, and are actually the preferred method of creating small commands for personal use. Aliases are very limited in the kind of commands and shell features they support, whereas shell functions allow anything that can be scripted. For example, if we liked the `report_disk_space` shell function that we developed for our script, we could create a similar function named `ds` for our `.bashrc` file:

```
ds () {
      echo "Disk Space Utilization For $HOSTNAME"
      df -h
}
```

Summing Up

In this chapter, we have introduced a common method of program design called top-down design, and we have seen how shell functions are used to build the stepwise refinement that it requires. We have also seen how local variables can be used to make shell functions independent from one another and from the program in which they are placed. This makes it possible for shell functions to be written in a portable manner and to be *reusable* by allowing them to be placed in multiple programs; a great time saver.

Further Reading

- The Wikipedia has many articles on software design philosophy. Here are a couple of good ones:
  http://en.wikipedia.org/wiki/Top-down_design
  http://en.wikipedia.org/wiki/Subroutines

28 – Flow Control: Branching With if

In the last chapter, we were presented with a problem. How can we make our report generator script adapt to the privileges of the user running the script? The solution to this problem will require us to find a way to "change directions" within our script, based on a the results of a test. In programming terms, we need the program to *branch*.

Let's consider a simple example of logic expressed in *pseudocode*, a simulation of a computer language intended for human consumption:

X = 5

If X = 5, then:

    Say "X equals 5."

Otherwise:

    Say "X is not equal to 5."

This is an example of a branch. Based on the condition, "Does X = 5?" do one thing, "Say X equals 5," otherwise do another thing, "Say X is not equal to 5."

`if`

Using the shell, we can code the logic above as follows:

```
x=5

if [ $x = 5 ]; then
    echo "x equals 5."
else
    echo "x does not equal 5."
fi
```

or we can enter it directly at the command line (slightly shortened):

```
[me@linuxbox ~]$ x=5
[me@linuxbox ~]$ if [ $x = 5 ]; then echo "equals 5"; else echo "does
not equal 5"; fi
equals 5
[me@linuxbox ~]$ x=0
[me@linuxbox ~]$ if [ $x = 5 ]; then echo "equals 5"; else echo "does
not equal 5"; fi
does not equal 5
```

In this example, we execute the command twice. Once, with the value of `x` set to 5, which results in the string "equals 5" being output, and the second time with the value of `x` set to 0, which results in the string "does not equal 5" being output.

The `if` statement has the following syntax:

```
if commands; then
     commands
[elif commands; then
     commands...]
[else
     commands]
fi
```

where *commands* is a list of commands. This is a little confusing at first glance. But before we can clear this up, we have to look at how the shell evaluates the success or failure of a command.

Exit Status

Commands (including the scripts and shell functions we write) issue a value to the system when they terminate, called an *exit status*. This value, which is an integer in the range of 0 to 255, indicates the success or failure of the command's execution. By convention, a value of zero indicates success and any other value indicates failure. The shell provides a parameter that we can use to examine the exit status. Here we see it in action:

```
[me@linuxbox ~]$ ls -d /usr/bin
/usr/bin
[me@linuxbox ~]$ echo $?
0
[me@linuxbox ~]$ ls -d /bin/usr
ls: cannot access /bin/usr: No such file or directory
[me@linuxbox ~]$ echo $?
2
```

In this example, we execute the `ls` command twice. The first time, the command executes successfully. If we display the value of the parameter `$?`, we see that it is zero. We execute the `ls` command a second time, producing an error and examine the parameter `$?` again. This time it contains a 2, indicating that the command encountered an error. Some commands use different exit status values to provide diagnostics for errors, while many commands simply exit with a value of one when they fail. Man pages often include a section entitled "Exit Status," describing what codes are used. However, a zero always indicates success.

The shell provides two extremely simple builtin commands that do nothing except terminate with either a zero or one exit status. The `true` command always executes successfully and the `false` command always executes unsuccessfully:

```
[me@linuxbox ~]$ true
[me@linuxbox ~]$ echo $?
0
[me@linuxbox ~]$ false
[me@linuxbox ~]$ echo $?
1
```

We can use these commands to see how the `if` statement works. What the `if` statement really does is evaluate the success or failure of commands:

```
[me@linuxbox ~]$ if true; then echo "It's true."; fi
It's true.
[me@linuxbox ~]$ if false; then echo "It's true."; fi
[me@linuxbox ~]$
```

The command `echo "It's true."` is executed when the command following `if` executes successfully, and is not executed when the command following `if` does not execute successfully. If a list of commands follows `if`, the last command in the list is evaluated:

```
[me@linuxbox ~]$ if false; true; then echo "It's true."; fi
It's true.
[me@linuxbox ~]$ if true; false; then echo "It's true."; fi
[me@linuxbox ~]$
```

test

By far, the command used most frequently with `if` is `test`. The `test` command performs a variety of checks and comparisons. It has two equivalent forms:

`test` *`expression`*

and the more popular:

`[` *`expression`* `]`

where *expression* is an expression that is evaluated as either true or false. The `test` command returns an exit status of zero when the expression is true and a status of one when the expression is false.

File Expressions

The following expressions are used to evaluate the status of files:

Table 28-1: test File Expressions

Expression	Is True If:
file1 `-ef` *file2*	*file1* and *file2* have the same inode numbers (the two filenames refer to the same file by hard linking).
file1 `-nt` *file2*	*file1* is newer than *file2*.
file1 `-ot` *file2*	*file1* is older than *file2*.
`-b` *file*	*file* exists and is a block special (device) file.
`-c` *file*	*file* exists and is a character special (device) file.
`-d` *file*	*file* exists and is a directory.
`-e` *file*	*file* exists.
`-f` *file*	*file* exists and is a regular file.
`-g` *file*	*file* exists and is set-group-ID.
`-G` *file*	*file* exists and is owned by the effective group ID.
`-k` *file*	*file* exists and has its “sticky bit” set.
`-L` *file*	*file* exists and is a symbolic link.
`-O` *file*	*file* exists and is owned by the effective user ID.
`-p` *file*	*file* exists and is a named pipe.
`-r` *file*	*file* exists and is readable (has readable permission for

	the effective user).
`-s` *file*	*file* exists and has a length greater than zero.
`-S` *file*	*file* exists and is a network socket.
`-t` *fd*	*fd* is a file descriptor directed to/from the terminal. This can be used to determine whether standard input/output/error is being redirected.
`-u` *file*	*file* exists and is setuid.
`-w` *file*	*file* exists and is writable (has write permission for the effective user).
`-x` *file*	*file* exists and is executable (has execute/search permission for the effective user).

Here we have a script that demonstrates some of the file expressions:

```
#!/bin/bash

# test-file: Evaluate the status of a file

FILE=~/.bashrc

if [ -e "$FILE" ]; then
      if [ -f "$FILE" ]; then
            echo "$FILE is a regular file."
      fi
      if [ -d "$FILE" ]; then
            echo "$FILE is a directory."
      fi
      if [ -r "$FILE" ]; then
            echo "$FILE is readable."
      fi
      if [ -w "$FILE" ]; then
            echo "$FILE is writable."
      fi
      if [ -x "$FILE" ]; then
            echo "$FILE is executable/searchable."
      fi
else
      echo "$FILE does not exist"
      exit 1
fi

exit
```

The script evaluates the file assigned to the constant `FILE` and displays its results as the evaluation is performed. There are two interesting things to note about this script. First, notice how the parameter `$FILE` is quoted within the expressions. This is not required, but is a defense against the parameter being empty. If the parameter expansion of `$FILE` were to result in an empty value, it would cause an error (the operators would be interpreted as non-null strings rather than operators). Using the quotes around the parameter insures that the operator is always followed by a string, even if the string is empty. Second, notice the presence of the `exit` commands near the end of the script. The `exit` command accepts a single, optional argument, which becomes the script's exit status. When no argument is passed, the exit status defaults to zero. Using `exit` in this way allows the script to indicate failure if `$FILE` expands to the name of a nonexistent file. The `exit` command appearing on the last line of the script is there as a formality. When a script "runs off the end" (reaches end of file), it terminates with an exit status of zero by default, anyway.

Similarly, shell functions can return an exit status by including an integer argument to the `return` command. If we were to convert the script above to a shell function to include it in a larger program, we could replace the `exit` commands with `return` statements and get the desired behavior:

```
test_file () {

      # test-file: Evaluate the status of a file

      FILE=~/.bashrc

      if [ -e "$FILE" ]; then
            if [ -f "$FILE" ]; then
                  echo "$FILE is a regular file."
            fi
            if [ -d "$FILE" ]; then
                  echo "$FILE is a directory."
            fi
            if [ -r "$FILE" ]; then
                  echo "$FILE is readable."
            fi
            if [ -w "$FILE" ]; then
                  echo "$FILE is writable."
            fi
            if [ -x "$FILE" ]; then
                  echo "$FILE is executable/searchable."
            fi
      else
            echo "$FILE does not exist"
            return 1
```

```
      fi

}
```

String Expressions

The following expressions are used to evaluate strings:

Table 28-2: test String Expressions

Expression	Is True If...
`string`	*string* is not null.
`-n string`	The length of *string* is greater than zero.
`-z string`	The length of *string* is zero.
`string1 = string2` `string1 == string2`	*string1* and *string2* are equal. Single or double equal signs may be used, but the use of double equal signs is greatly preferred.
`string1 != string2`	*string1* and *string2* are not equal.
`string1 > string2`	*string1* sorts after *string2*.
`string1 < string2`	*string1* sorts before *string2*.

Warning: the `>` and `<` expression operators must be quoted (or escaped with a backslash) when used with `test`. If they are not, they will be interpreted by the shell as redirection operators, with potentially destructive results. Also note that while the `bash` documentation states that the sorting order conforms to the collation order of the current locale, it does not. ASCII (POSIX) order is used in versions of `bash` up to and including 4.0.

Here is a script that incorporates string expressions:

```
#!/bin/bash

# test-string: evaluate the value of a string

ANSWER=maybe

if [ -z "$ANSWER" ]; then
```

```
      echo "There is no answer." >&2
      exit 1
fi

if [ "$ANSWER" = "yes" ]; then
      echo "The answer is YES."
elif [ "$ANSWER" = "no" ]; then
      echo "The answer is NO."
elif [ "$ANSWER" = "maybe" ]; then
      echo "The answer is MAYBE."
else
      echo "The answer is UNKNOWN."
fi
```

In this script, we evaluate the constant `ANSWER`. We first determine if the string is empty. If it is, we terminate the script and set the exit status to one. Notice the redirection that is applied to the echo command. This redirects the error message "There is no answer." to standard error, which is the "proper" thing to do with error messages. If the string is not empty, we evaluate the value of the string to see if it is equal to either "yes," "no," or "maybe." We do this by using `elif`, which is short for "else if." By using `elif`, we are able to construct a more complex logical test.

Integer Expressions

The following expressions are used with integers:

Table 28-3: test Integer Expressions

Expression	Is True If...
integer1 `-eq` *integer2*	*integer1* is equal to *integer2*.
integer1 `-ne` *integer2*	*integer1* is not equal to *integer2*.
integer1 `-le` *integer2*	*integer1* is less than or equal to *integer2*.
integer1 `-lt` *integer2*	*integer1* is less than *integer2*.
integer1 `-ge` *integer2*	*integer1* is greater than or equal to *integer2*.
integer1 `-gt` *integer2*	*integer1* is greater than *integer2*.

Here is a script that demonstrates them:

```
#!/bin/bash
```

```
# test-integer: evaluate the value of an integer.

INT=-5

if [ -z "$INT" ]; then
      echo "INT is empty." >&2
      exit 1
fi

if [ $INT -eq 0 ]; then
      echo "INT is zero."
else
      if [ $INT -lt 0 ]; then
            echo "INT is negative."
      else
            echo "INT is positive."
      fi
      if [ $((INT % 2)) -eq 0 ]; then
            echo "INT is even."
      else
            echo "INT is odd."
      fi
fi
```

The interesting part of the script is how it determines whether an integer is even or odd. By performing a modulo 2 operation on the number, which divides the number by two and returns the remainder, it can tell if the number is odd or even.

A More Modern Version Of `test`

Recent versions of `bash` include a compound command that acts as an enhanced replacement for `test`. It uses the following syntax:

`[[ expression ]]`

where, like `test`, *expression* is an expression that evaluates to either a true or false result. The `[[ ]]` command is very similar to `test` (it supports all of its expressions), but adds an important new string expression:

`string1 =~ regex`

which returns true if *string1* is matched by the extended regular expression *regex*. This opens up a lot of possibilities for performing such tasks as data validation. In our earlier example of the integer expressions, the script would fail if the constant `INT` contained anything except an integer. The script needs a way to verify that the constant contains an integer. Using `[[ ]]` with the `=~` string expression operator, we could improve the

script this way:

```
#!/bin/bash

# test-integer2: evaluate the value of an integer.

INT=-5

if [[ "$INT" =~ ^-?[0-9]+$ ]]; then
      if [ $INT -eq 0 ]; then
            echo "INT is zero."
      else
            if [ $INT -lt 0 ]; then
                  echo "INT is negative."
            else
                  echo "INT is positive."
            fi
            if [ $((INT % 2)) -eq 0 ]; then
                  echo "INT is even."
            else
                  echo "INT is odd."
            fi
      fi
else
      echo "INT is not an integer." >&2
      exit 1
fi
```

By applying the regular expression, we are able to limit the value of `INT` to only strings that begin with an optional minus sign, followed by one or more numerals. This expression also eliminates the possibility of empty values.

Another added feature of `[[ ]]` is that the `==` operator supports pattern matching the same way pathname expansion does. For example:

```
[me@linuxbox ~]$ FILE=foo.bar
[me@linuxbox ~]$ if [[ $FILE == foo.* ]]; then
> echo "$FILE matches pattern 'foo.*'"
> fi
foo.bar matches pattern 'foo.*'
```

This makes `[[ ]]` useful for evaluating file and path names.

(( )) - Designed For Integers

In addition to the [[ ]] compound command, bash also provides the (( )) compound command, which is useful for operating on integers. It supports a full set of arithmetic evaluations, a subject we will cover fully in Chapter 35.

(( )) is used to perform *arithmetic truth tests*. An arithmetic truth test results in true if the result of the arithmetic evaluation is non-zero.

```
[me@linuxbox ~]$ if ((1)); then echo "It is true."; fi
It is true.
[me@linuxbox ~]$ if ((0)); then echo "It is true."; fi
[me@linuxbox ~]$
```

Using (( )), we can slightly simplify the test-integer2 script like this:

```
#!/bin/bash

# test-integer2a: evaluate the value of an integer.

INT=-5

if [[ "$INT" =~ ^-?[0-9]+$ ]]; then
      if ((INT == 0)); then
            echo "INT is zero."
      else
            if ((INT < 0)); then
                  echo "INT is negative."
            else
                  echo "INT is positive."
            fi
            if (( ((INT % 2)) == 0)); then
                  echo "INT is even."
            else
                  echo "INT is odd."
            fi
      fi
else
      echo "INT is not an integer." >&2
      exit 1
fi
```

Notice that we use less than and greater than signs and that == is used to test for equivalence. This is a more natural looking syntax for working with integers. Notice too, that because the compound command (( )) is part of the shell syntax rather than

an ordinary command, and it deals only with integers, it is able to recognize variables by name and does not require expansion to be performed. We'll discuss `(( ))` and the related arithmetic expansion further in Chapter 35.

Combining Expressions

It's also possible to combine expressions to create more complex evaluations. Expressions are combined by using logical operators. We saw these in Chapter 16, when we learned about the `find` command. There are three logical operations for `test` and `[[ ]]`. They are AND, OR and NOT. `test` and `[[ ]]`use different operators to represent these operations :

Table 28-4: Logical Operators

Operation	test	[[ ]] and (( ))
AND	-a	&&
OR	-o	||
NOT	!	!

Here's an example of an AND operation. The following script determines if an integer is within a range of values:

```
#!/bin/bash

# test-integer3: determine if an integer is within a
# specified range of values.

MIN_VAL=1
MAX_VAL=100

INT=50

if [[ "$INT" =~ ^-?[0-9]+$ ]]; then
      if [[ INT -ge MIN_VAL && INT -le MAX_VAL ]]; then
            echo "$INT is within $MIN_VAL to $MAX_VAL."
      else
            echo "$INT is out of range."
      fi
else
      echo "INT is not an integer." >&2
      exit 1
fi
```

In this script, we determine if the value of integer `INT` lies between the values of `MIN_VAL` and `MAX_VAL`. This is performed by a single use of `[[ ]]`, which includes two expressions separated by the && operator. We could have also coded this using `test`:

```
      if [ $INT -ge $MIN_VAL -a $INT -le $MAX_VAL ]; then
            echo "$INT is within $MIN_VAL to $MAX_VAL."
      else
            echo "$INT is out of range."
      fi
```

The ! negation operator reverses the outcome of an expression. It returns true if an expression is false, and it returns false if an expression is true. In the following script, we modify the logic of our evaluation to find values of `INT` that are outside the specified range:

```
#!/bin/bash

# test-integer4: determine if an integer is outside a
# specified range of values.

MIN_VAL=1
MAX_VAL=100

INT=50

if [[ "$INT" =~ ^-?[0-9]+$ ]]; then
      if [[ ! (INT -ge MIN_VAL && INT -le MAX_VAL) ]]; then
            echo "$INT is outside $MIN_VAL to $MAX_VAL."
      else
            echo "$INT is in range."
      fi
else
        echo "INT is not an integer." >&2
        exit 1
fi
```

We also include parentheses around the expression, for grouping. If these were not included, the negation would only apply to the first expression and not the combination of the two. Coding this with `test` would be done this way:

```
      if [ ! \( $INT -ge $MIN_VAL -a $INT -le $MAX_VAL \) ]; then
```

```
        echo "$INT is outside $MIN_VAL to $MAX_VAL."
    else
        echo "$INT is in range."
    fi
```

Since all expressions and operators used by `test` are treated as command arguments by the shell (unlike `[[ ]]` and `(( ))`), characters which have special meaning to `bash`, such as `<`, `>`, `(`, and `)`, must be quoted or escaped.

Seeing that `test` and `[[ ]]` do roughly the same thing, which is preferable? `test` is traditional (and part of POSIX), whereas `[[ ]]` is specific to `bash`. It's important to know how to use `test`, since it is very widely used, but `[[ ]]` is clearly more useful and is easier to code.

Portability Is The Hobgoblin Of Little Minds

If you talk to "real" Unix people, you quickly discover that many of them don't like Linux very much. They regard it as impure and unclean. One tenet of Unix followers is that everything should be "portable." This means that any script you write should be able to run, unchanged, on any Unix-like system.

Unix people have good reason to believe this. Having seen what proprietary extensions to commands and shells did to the Unix world before POSIX, they are naturally wary of the effect of Linux on their beloved OS.

But portability has a serious downside. It prevents progress. It requires that things are always done using "lowest common denominator" techniques. In the case of shell programming, it means making everything compatible with `sh`, the original Bourne shell.

This downside is the excuse that proprietary vendors use to justify their proprietary extensions. Only they call them "innovations." But they are really just lock-in devices for their customers.

The GNU tools, such as `bash`, have no such restrictions. They encourage portability by supporting standards and by being universally available. You can install `bash` and the other GNU tools on almost any kind of system, even Windows, without cost. So feel free to use all the features of `bash`. It's *really* portable.

Control Operators: Another Way To Branch

bash provides two control operators that can perform branching. The && (AND) and || (OR) operators work like the logical operators in the [[ ]] compound command. This is the syntax:

command1 && *command2*

and

command1 || *command2*

It is important to understand the behavior of these. With the && operator, *command1* is executed and *command2* is executed if, *and only if*, *command1* is successful. With the || operator, *command1* is executed and *command2* is executed if, *and only if*, *command1* is unsuccessful.

In practical terms, it means that we can do something like this:

```
[me@linuxbox ~]$ mkdir temp && cd temp
```

This will create a directory named temp, and if it succeeds, the current working directory will be changed to temp. The second command is attempted only if the mkdir command is successful. Likewise, a command like this:

```
[me@linuxbox ~]$ [ -d temp ] || mkdir temp
```

will test for the existence of the directory temp, and only if the test fails, will the directory be created. This type of construct is very handy for handling errors in scripts, a subject we will discuss more in later chapters. For example, we could do this in a script:

```
[ -d temp ] || exit 1
```

If the script requires the directory temp, and it does not exist, then the script will terminate with an exit status of one.

Summing Up

We started this chapter with a question. How could we make our sys_info_page script detect if the user had permission to read all the home directories? With our knowledge of if, we can solve the problem by adding this code to the

`report_home_space` function:

```
report_home_space () {
      if [[ $(id -u) -eq 0 ]]; then
            cat <<- _EOF_
                  <H2>Home Space Utilization (All Users)</H2>
                  <PRE>$(du -sh /home/*)</PRE>
                  _EOF_
      else
            cat <<- _EOF_
                  <H2>Home Space Utilization ($USER)</H2>
                  <PRE>$(du -sh $HOME)</PRE>
                  _EOF_
      fi
      return
}
```

We evaluate the output of the `id` command. With the `-u` option, `id` outputs the numeric user ID number of the effective user. The superuser is always zero and every other user is a number greater than zero. Knowing this, we can construct two different here documents, one taking advantage of superuser privileges, and the other, restricted to the user's own home directory.

We are going to take a break from the `sys_info_page` program, but don't worry. It will be back. In the meantime, we'll cover some topics that we'll need when we resume our work.

Further Reading

There are several sections of the `bash` man page that provide further detail on the topics covered in this chapter:

- Lists (covers the control operators `||` and &&)
- Compound Commands (covers `[[ ]]`, `(( ))` and `if`)
- CONDITIONAL EXPRESSIONS
- SHELL BUILTIN COMMANDS (covers `test`)

Further, the Wikipedia has a good article on the concept of pseudocode:

http://en.wikipedia.org/wiki/Pseudocode

29 – Reading Keyboard Input

The scripts we have written so far lack a feature common in most computer programs—*interactivity*. That is, the ability of the program to interact with the user. While many programs don't need to be interactive, some programs benefit from being able to accept input directly from the user. Take, for example, this script from the previous chapter:

```
#!/bin/bash

# test-integer2: evaluate the value of an integer.

INT=-5

if [[ "$INT" =~ ^-?[0-9]+$ ]]; then
      if [ $INT -eq 0 ]; then
            echo "INT is zero."
      else
            if [ $INT -lt 0 ]; then
                  echo "INT is negative."
            else
                  echo "INT is positive."
            fi
            if [ $(( INT % 2)) -eq 0 ]; then
                  echo "INT is even."
            else
                  echo "INT is odd."
            fi
      fi
else
      echo "INT is not an integer." >&2
      exit 1
fi
```

Each time we want to change the value of `INT`, we have to edit the script. It would be much more useful if the script could ask the user for a value. In this chapter, we will begin to look at how we can add interactivity to our programs.

`read` – Read Values From Standard Input

The `read` builtin command is used to read a single line of standard input. This command can be used to read keyboard input or, when redirection is employed, a line of data from a file. The command has the following syntax:

`read [-options] [variable...]`

where *options* is one or more of the available options listed below and *variable* is the name of one or more variables used to hold the input value. If no variable name is supplied, the shell variable `REPLY` contains the line of data.

Basically, `read` assigns fields from standard input to the specified variables. If we modify our integer evaluation script to use `read`, it might look like this:

```
#!/bin/bash

# read-integer: evaluate the value of an integer.

echo -n "Please enter an integer -> "
read int

if [[ "$int" =~ ^-?[0-9]+$ ]]; then
      if [ $int -eq 0 ]; then
            echo "$int is zero."
      else
            if [ $int -lt 0 ]; then
                  echo "$int is negative."
            else
                  echo "$int is positive."
            fi
            if [ $((int % 2)) -eq 0 ]; then
                  echo "$int is even."
            else
                  echo "$int is odd."
            fi
      fi
else
      echo "Input value is not an integer." >&2
      exit 1
fi
```

We use `echo` with the `-n` option (which suppresses the trailing newline on output) to display a prompt, then use `read` to input a value for the variable `int`. Running this script results in this:

```
[me@linuxbox ~]$ read-integer
Please enter an integer -> 5
5 is positive.
5 is odd.
```

`read` can assign input to multiple variables, as shown in this script:

```
#!/bin/bash

# read-multiple: read multiple values from keyboard

echo -n "Enter one or more values > "
read var1 var2 var3 var4 var5

echo "var1 = '$var1'"
echo "var2 = '$var2'"
echo "var3 = '$var3'"
echo "var4 = '$var4'"
echo "var5 = '$var5'"
```

In this script, we assign and display up to five values. Notice how `read` behaves when given different numbers of values:

```
[me@linuxbox ~]$  read-multiple
Enter one or more values > a b c d e
var1 = 'a'
var2 = 'b'
var3 = 'c'
var4 = 'd'
var5 = 'e'
[me@linuxbox ~]$ read-multiple
Enter one or more values > a
var1 = 'a'
var2 = ''
var3 = ''
var4 = ''
var5 = ''
[me@linuxbox ~]$ read-multiple
Enter one or more values > a b c d e f g
var1 = 'a'
var2 = 'b'
var3 = 'c'
var4 = 'd'
var5 = 'e f g'
```

If `read` receives fewer than the expected number, the extra variables are empty, while an excessive amount of input results in the final variable containing all of the extra input.

If no variables are listed after the `read` command, a shell variable, `REPLY`, will be assigned all the input:

```
#!/bin/bash

# read-single: read multiple values into default variable

echo -n "Enter one or more values > "
read

echo "REPLY = '$REPLY'"
```

Running this script results in this:

```
[me@linuxbox ~]$ read-single
Enter one or more values > a b c d
REPLY = 'a b c d'
```

Options

`read` supports the following options:

Table 29-1: read Options

Option	Description
`-a` *`array`*	Assign the input to *array*, starting with index zero. We will cover arrays in Chapter 36.
`-d` *`delimiter`*	The first character in the string *delimiter* is used to indicate end of input, rather than a newline character.
`-e`	Use Readline to handle input. This permits input editing in the same manner as the command line.
`-n` *`num`*	Read *num* characters of input, rather than an entire line.
`-p` *`prompt`*	Display a prompt for input using the string *prompt*.
`-r`	Raw mode. Do not interpret backslash characters as escapes.

`-s`	Silent mode. Do not echo characters to the display as they are typed. This is useful when inputting passwords and other confidential information.
`-t` *`seconds`*	Timeout. Terminate input after *seconds*. `read` returns a non-zero exit status if an input times out.
`-u` *`fd`*	Use input from file descriptor *fd*, rather than standard input.

Using the various options, we can do interesting things with `read`. For example, with the `-p` option, we can provide a prompt string:

```
#!/bin/bash

# read-single: read multiple values into default variable

read -p "Enter one or more values > "

echo "REPLY = '$REPLY'"
```

With the `-t` and `-s` options we can write a script that reads "secret" input and times out if the input is not completed in a specified time:

```
#!/bin/bash

# read-secret: input a secret pass phrase

if read -t 10 -sp "Enter secret pass phrase > " secret_pass; then
      echo -e "\nSecret pass phrase = '$secret_pass'"
else
      echo -e "\nInput timed out" >&2
      exit 1
fi
```

The script prompts the user for a secret pass phrase and waits ten seconds for input. If the entry is not completed within the specified time, the script exits with an error. Since the `-s` option is included, the characters of the pass phrase are not echoed to the display as they are typed.

IFS

Normally, the shell performs word splitting on the input provided to `read`. As we have seen, this means that multiple words separated by one or more spaces become separate items on the input line, and are assigned to separate variables by `read`. This behavior is configured by a shell variable named `IFS` (for Internal Field Separator). The default value of `IFS` contains a space, a tab, and a newline character, each of which will separate items from one another.

We can adjust the value of `IFS` to control the separation of fields input to `read`. For example, the `/etc/passwd` file contains lines of data that use the colon character as a field separator. By changing the value of `IFS` to a single colon, we can use `read` to input the contents of `/etc/passwd` and successfully separate fields into different variables. Here we have a script that does just that:

```
#!/bin/bash

# read-ifs: read fields from a file

FILE=/etc/passwd

read -p "Enter a user name > " user_name

file_info=$(grep "^$user_name:" $FILE)

if [ -n "$file_info" ]; then
      IFS=":" read user pw uid gid name home shell <<< "$file_info"
      echo "User =      '$user'"
      echo "UID =       '$uid'"
      echo "GID =       '$gid'"
      echo "Full Name = '$name'"
      echo "Home Dir. = '$home'"
      echo "Shell =     '$shell'"
else
      echo "No such user '$user_name'" >&2
      exit 1
fi
```

This script prompts the user to enter the user name of an account on the system, then displays the different fields found in the user's record in the `/etc/passwd` file. The script contains two interesting lines. The first is:

```
file_info=$(grep "^$user_name:" $FILE)
```

This line assigns the results of a `grep` command to the variable `file_info`. The regular expression used by `grep` assures that the user name will only match a single line

in the /etc/passwd file.

The second interesting line is this one:

```
IFS=":" read user pw uid gid name home shell <<< "$file_info"
```

The line consists of three parts: a variable assignment, a read command with a list of variable names as arguments, and a strange new redirection operator. We'll look at the variable assignment first.

The shell allows one or more variable assignments to take place immediately before a command. These assignments alter the environment for the command that follows. The effect of the assignment is temporary; only changing the environment for the duration of the command. In our case, the value of IFS is changed to a colon character. Alternately, we could have coded it this way:

```
OLD_IFS="$IFS"
IFS=":"
read user pw uid gid name home shell <<< "$file_info"
IFS="$OLD_IFS"
```

where we store the value of IFS, assign a new value, perform the read command, then restore IFS to its original value. Clearly, placing the variable assignment in front of the command is a more concise way of doing the same thing.

The <<< operator indicates a *here string*. A here string is like a here document, only shorter, consisting of a single string. In our example, the line of data from the /etc/passwd file is fed to the standard input of the read command. We might wonder why this rather oblique method was chosen rather than:

```
echo "$file_info" | IFS=":" read user pw uid gid name home shell
```

Well, there's a reason...

You Can't Pipe read

While the read command normally takes input from standard input, you cannot do this:

```
echo "foo" | read
```

We would expect this to work, but it does not. The command will appear to succeed but the REPLY variable will always be empty. Why is this?

The explanation has to do with the way the shell handles pipelines. In bash (and other shells such as sh), pipelines create *subshells*. These are copies of the shell

and its environment which are used to execute the command in the pipeline. In our example above, `read` is executed in a subshell.

Subshells in Unix-like systems create copies of the environment for the processes to use while they execute. When the processes finishes the copy of the environment is destroyed. This means that a *subshell can never alter the environment of its parent process*. `read` assigns variables, which then become part of the environment. In the example above, `read` assigns the value "foo" to the variable `REPLY` in its subshell's environment, but when the command exits, the subshell and its environment are destroyed, and the effect of the assignment is lost.

Using here strings is one way to work around this behavior. Another method is discussed in Chapter 37.

Validating Input

With our new ability to have keyboard input comes an additional programming challenge, validating input. Very often the difference between a well-written program and a poorly written one is in the program's ability to deal with the unexpected. Frequently, the unexpected appears in the form of bad input. We've done a little of this with our evaluation programs in the previous chapter, where we checked the value of integers and screened out empty values and non-numeric characters. It is important to perform these kinds of programming checks every time a program receives input, to guard against invalid data. This is especially important for programs that are shared by multiple users. Omitting these safeguards in the interests of economy might be excused if a program is to be used once and only by the author to perform some special task. Even then, if the program performs dangerous tasks such as deleting files, it would be wise to include data validation, just in case.

Here we have an example program that validates various kinds of input:

```
#!/bin/bash

# read-validate: validate input

invalid_input () {
      echo "Invalid input '$REPLY'" >&2
      exit 1
}
```

```
read -p "Enter a single item > "

# input is empty (invalid)
[[ -z $REPLY ]] && invalid_input

# input is multiple items (invalid)
(( $(echo $REPLY | wc -w) > 1 )) && invalid_input

# is input a valid filename?
if [[ $REPLY =~ ^[-[:alnum:]\._]+$ ]]; then
      echo "'$REPLY' is a valid filename."
      if [[ -e $REPLY ]]; then
            echo "And file '$REPLY' exists."
      else
            echo "However, file '$REPLY' does not exist."
      fi

      # is input a floating point number?
      if [[ $REPLY =~ ^-?[[:digit:]]*\.[[:digit:]]+$ ]]; then
            echo "'$REPLY' is a floating point number."
      else
            echo "'$REPLY' is not a floating point number."
      fi

      # is input an integer?
      if [[ $REPLY =~ ^-?[[:digit:]]+$ ]]; then
            echo "'$REPLY' is an integer."
      else
            echo "'$REPLY' is not an integer."
      fi
else
      echo "The string '$REPLY' is not a valid filename."
fi
```

This script prompts the user to enter an item. The item is subsequently analyzed to determine its contents. As we can see, the script makes use of many of the concepts that we have covered this far, including shell functions, `[[ ]]`, `(( ))`, the control operator &&, and `if`, as well as a healthy dose of regular expressions.

Menus

A common type of interactivity is called *menu-driven*. In menu-driven programs, the user is presented with a list of choices and is asked to choose one. For example, we could imagine a program that presented the following:

```
Please Select:
```

```
1. Display System Information
2. Display Disk Space
3. Display Home Space Utilization
0. Quit

Enter selection [0-3] >
```

Using what we learned from writing our `sys_info_page` program, we can construct a menu-driven program to perform the tasks on the above menu:

```
#!/bin/bash

# read-menu: a menu driven system information program

clear
echo "
Please Select:

1. Display System Information
2. Display Disk Space
3. Display Home Space Utilization
0. Quit
"
read -p "Enter selection [0-3] > "

if [[ $REPLY =~ ^[0-3]$ ]]; then
      if [[ $REPLY == 0 ]]; then
            echo "Program terminated."
            exit
      fi
      if [[ $REPLY == 1 ]]; then
            echo "Hostname: $HOSTNAME"
            uptime
            exit
      fi
      if [[ $REPLY == 2 ]]; then
            df -h
            exit
      fi
      if [[ $REPLY == 3 ]]; then
            if [[ $(id -u) -eq 0 ]]; then
                  echo "Home Space Utilization (All Users)"
                  du -sh /home/*
            else
                  echo "Home Space Utilization ($USER)"
                  du -sh $HOME
```

```
            fi
            exit
      fi
else
      echo "Invalid entry." >&2
      exit 1
fi
```

This script is logically divided into two parts. The first part displays the menu and inputs the response from the user. The second part identifies the response and carries out the selected action. Notice the use of the `exit` command in this script. It is used here to prevent the script from executing unnecessary code after an action has been carried out. The presence of multiple exit points in a program is generally a bad idea (it makes program logic harder to understand), but it works in this script.

Summing Up

In this chapter, we took our first steps toward interactivity; allowing users to input data into our programs via the keyboard. Using the techniques presented thus far, it is possible to write many useful programs, such as specialized calculation programs and easy-to-use front ends for arcane command line tools. In the next chapter, we will build on the menu-driven program concept to make it even better.

Extra Credit

It is important to study the programs in this chapter carefully and have a complete understanding of the way they are logically structured, as the programs to come will be increasingly complex. As an exercise, rewrite the programs in this chapter using the `test` command rather than the `[[ ]]` compound command. Hint: use `grep` to evaluate the regular expressions and evaluate its exit status. This will be good practice.

Further Reading

- The *Bash Reference Manual* contains a chapter on builtins, which includes the `read` command:
  http://www.gnu.org/software/bash/manual/bashref.html#Bash-Builtins

30 – Flow Control: Looping With while / until

In the previous chapter, we developed a menu-driven program to produce various kinds of system information. The program works, but it still has a significant usability problem. It only executes a single choice and then terminates. Even worse, if an invalid selection is made, the program terminates with an error, without giving the user an opportunity to try again. It would be better if we could somehow construct the program so that it could repeat the menu display and selection over and over, until the user chooses to exit the program.

In this chapter, we will look at a programming concept called *looping*, which can be used to make portions of programs repeat. The shell provides three compound commands for looping. We will look at two of them in this chapter, and the third in a later one.

Looping

Daily life is full of repeated activities. Going to work each day, walking the dog, slicing a carrot are all tasks that involve repeating a series of steps. Let's consider slicing a carrot. If we express this activity in pseudocode, it might look something like this:

1. get cutting board
2. get knife
3. place carrot on cutting board
4. lift knife
5. advance carrot
6. slice carrot
7. if entire carrot sliced, then quit, else go to step 4

Steps 4 through 7 form a *loop*. The actions within the loop are repeated until the condition, "entire carrot sliced," is reached.

`while`

`bash` can express a similar idea. Let's say we wanted to display five numbers in

sequential order from one to five. a `bash` script could be constructed as follows:

```
#!/bin/bash

# while-count: display a series of numbers

count=1

while [ $count -le 5 ]; do
      echo $count
      count=$((count + 1))
done
echo "Finished."
```

When executed, this script displays the following:

```
[me@linuxbox ~]$ while-count
1
2
3
4
5
Finished.
```

The syntax of the `while` command is:

`while` *`commands`*`; do` *`commands`*`; done`

Like `if`, `while` evaluates the exit status of a list of commands. As long as the exit status is zero, it performs the commands inside the loop. In the script above, the variable `count` is created and assigned an initial value of 1. The `while` command evaluates the exit status of the `test` command. As long as the `test` command returns an exit status of zero, the commands within the loop are executed. At the end of each cycle, the `test` command is repeated. After six iterations of the loop, the value of `count` has increased to six, the `test` command no longer returns an exit status of zero and the loop terminates. The program continues with the next statement following the loop.

We can use a *while loop* to improve the `read-menu` program from the previous chapter:

```
#!/bin/bash

# while-menu: a menu driven system information program
```

```
DELAY=3 # Number of seconds to display results

while [[ $REPLY != 0 ]]; do
      clear
      cat <<- _EOF_
            Please Select:

            1. Display System Information
            2. Display Disk Space
            3. Display Home Space Utilization
            0. Quit

      _EOF_
      read -p "Enter selection [0-3] > "

      if [[ $REPLY =~ ^[0-3]$ ]]; then
            if [[ $REPLY == 1 ]]; then
                  echo "Hostname: $HOSTNAME"
                  uptime
                  sleep $DELAY
            fi
            if [[ $REPLY == 2 ]]; then
                  df -h
                  sleep $DELAY
            fi
            if [[ $REPLY == 3 ]]; then
                  if [[ $(id -u) -eq 0 ]]; then
                        echo "Home Space Utilization (All Users)"
                        du -sh /home/*
                  else
                        echo "Home Space Utilization ($USER)"
                        du -sh $HOME
                  fi
                  sleep $DELAY
            fi
      else
            echo "Invalid entry."
            sleep $DELAY
      fi
done
echo "Program terminated."
```


By enclosing the menu in a while loop, we are able to have the program repeat the menu display after each selection. The loop continues as long as REPLY is not equal to “0” and the menu is displayed again, giving the user the opportunity to make another selection. At the end of each action, a `sleep` command is executed so the program will pause for a few seconds to allow the results of the selection to be seen before the screen is cleared and the menu is redisplayed. Once REPLY is equal to “0,” indicating the “quit” selection,

the loop terminates and execution continues with the line following `done`.

Breaking Out Of A Loop

`bash` provides two builtin commands that can be used to control program flow inside loops. The `break` command immediately terminates a loop, and program control resumes with the next statement following the loop. The `continue` command causes the remainder to the loop to be skipped, and program control resumes with the next iteration of the loop. Here we see a version of the `while-menu` program incorporating both `break` and `continue`:

```
#!/bin/bash

# while-menu2: a menu driven system information program

DELAY=3 # Number of seconds to display results

while true; do
      clear
      cat <<- _EOF_
            Please Select:

            1. Display System Information
            2. Display Disk Space
            3. Display Home Space Utilization
            0. Quit

      _EOF_
      read -p "Enter selection [0-3] > "

      if [[ $REPLY =~ ^[0-3]$ ]]; then
            if [[ $REPLY == 1 ]]; then
                  echo "Hostname: $HOSTNAME"
                  uptime
                  sleep $DELAY
                  continue
            fi
            if [[ $REPLY == 2 ]]; then
                  df -h
                  sleep $DELAY
                  continue
            fi
            if [[ $REPLY == 3 ]]; then
                  if [[ $(id -u) -eq 0 ]]; then
                        echo "Home Space Utilization (All Users)"
                        du -sh /home/*
                  else
```

```
                        echo "Home Space Utilization ($USER)"
                        du -sh $HOME
                  fi
                  sleep $DELAY
                  continue
            fi
            if [[ $REPLY == 0 ]]; then
                  break
            fi
      else
            echo "Invalid entry."
            sleep $DELAY
      fi
done
echo "Program terminated."
```

In this version of the script, we set up an *endless loop* (one that never terminates on its own) by using the `true` command to supply an exit status to `while`. Since `true` will always exit with a exit status of zero, the loop will never end. This is a surprisingly common scripting technique. Since the loop will never end on its own, it's up to the programmer to provide some way to break out of the loop when the time is right. In this script, the `break` command is used to exit the loop when the "0" selection is chosen. The `continue` command has been included at the end of the other script choices to allow for more efficient execution. By using `continue`, the script will skip over code that is not needed when a selection is identified. For example, if the "1" selection is chosen and identified, there is no reason to test for the other selections.

until

The `until` command is much like `while`, except instead of exiting a loop when a non-zero exit status is encountered, it does the opposite. An *until loop* continues until it receives a zero exit status. In our `while-count` script, we continued the loop as long as the value of the `count` variable was less than or equal to five. We could get the same result by coding the script with `until`:

```
#!/bin/bash

# until-count: display a series of numbers

count=1

until [ $count -gt 5 ]; do
      echo $count
```

```
      count=$((count + 1))
done
echo "Finished."
```

By changing the test expression to `$count -gt 5`, `until` will terminate the loop at the correct time. The decision of whether to use the `while` or `until` loop is usually a matter of choosing the one that allows the clearest test to be written.

Reading Files With Loops

`while` and `until` can process standard input. This allows files to be processed with while and until loops. In the following example, we will display the contents of the `distros.txt` file used in earlier chapters:

```
#!/bin/bash

# while-read: read lines from a file

while read distro version release; do
      printf "Distro: %s\tVersion: %s\tReleased: %s\n" \
            $distro \
            $version \
            $release
done < distros.txt
```

To redirect a file to the loop, we place the redirection operator after the `done` statement. The loop will use `read` to input the fields from the redirected file. The `read` command will exit after each line is read, with a zero exit status until the end-of-file is reached. At that point, it will exit with a non-zero exit status, thereby terminating the loop. It is also possible to pipe standard input into a loop:

```
#!/bin/bash

# while-read2: read lines from a file

sort -k 1,1 -k 2n distros.txt | while read distro version release; do

      printf "Distro: %s\tVersion: %s\tReleased: %s\n" \
            $distro \
            $version \
            $release
done
```

Here we take the output of the `sort` command and display the stream of text. However, it is important to remember that since a pipe will execute the loop in a subshell, any variables created or assigned within the loop will be lost when the loop terminates.

Summing Up

With the introduction of loops, and our previous encounters with branching, subroutines and sequences, we have covered the major types of flow control used in programs. `bash` has some more tricks up its sleeve, but they are refinements on these basic concepts.

Further Reading

- The *Bash Guide for Beginners* from the Linux Documentation Project has some more examples of while loops: http://tldp.org/LDP/Bash-Beginners-Guide/html/sect_09_02.html
- The Wikipedia has an article on loops, which is part of a larger article on flow control: http://en.wikipedia.org/wiki/Control_flow#Loops

31 – Troubleshooting

As our scripts become more complex, it's time to take a look at what happens when things go wrong and they don't do what what we want. In this chapter, we'll look at some of the common kinds of errors that occur in scripts, and describe a few useful techniques that can be used to track down and eradicate problems.

Syntactic Errors

One general class of errors is *syntactic*. Syntactic errors involve mis-typing some element of shell syntax. In most cases, these kinds of errors will lead to the shell refusing to execute the script.

In the following the discussions, we will use this script to demonstrate common types of errors:

```
#!/bin/bash

# trouble: script to demonstrate common errors

number=1

if [ $number = 1 ]; then
      echo "Number is equal to 1."
else
      echo "Number is not equal to 1."
fi
```

As written, this script runs successfully:

```
[me@linuxbox ~]$ trouble
Number is equal to 1.
```

Missing Quotes

If we edit our script and remove the trailing quote from the argument following the first `echo` command:

```
#!/bin/bash

# trouble: script to demonstrate common errors

number=1

if [ $number = 1 ]; then
      echo "Number is equal to 1.
else
      echo "Number is not equal to 1."
fi
```

watch what happens:

```
[me@linuxbox ~]$ trouble
/home/me/bin/trouble: line 10: unexpected EOF while looking for
matching `"'
/home/me/bin/trouble: line 13: syntax error: unexpected end of file
```

It generates two errors. Interestingly, the line numbers reported are not where the missing quote was removed, but rather much later in the program. We can see why, if we follow the program after the missing quote. `bash` will continue looking for the closing quote until it finds one, which it does immediately after the second `echo` command. `bash` becomes very confused after that, and the syntax of the `if` command is broken because the `fi` statement is now inside a quoted (but open) string.

In long scripts, this kind of error can be quite hard to find. Using an editor with syntax highlighting will help. If a complete version of `vim` is installed, syntax highlighting can be enabled by entering the command:

```
:syntax on
```


Missing Or Unexpected Tokens

Another common mistake is forgetting to complete a compound command, such as `if` or

`while`. Let's look at what happens if we remove the semicolon after the `test` in the `if` command:

```
#!/bin/bash

# trouble: script to demonstrate common errors

number=1

if [ $number = 1 ] then
      echo "Number is equal to 1."
else
      echo "Number is not equal to 1."
fi
```

The result is this:

```
[me@linuxbox ~]$ trouble
/home/me/bin/trouble: line 9: syntax error near unexpected token
`else'
/home/me/bin/trouble: line 9: `else'
```

Again, the error message points to a error that occurs later than the actual problem. What happens is really pretty interesting. As we recall, `if` accepts a list of commands and evaluates the exit code of the last command in the list. In our program, we intend this list to consist of a single command, `[`, a synonym for test. The `[` command takes what follows it as a list of arguments. In our case, three arguments: `$number`, `=`, and `]`. With the semicolon removed, the word `then` is added to the list of arguments, which is syntactically legal. The following `echo` command is legal, too. It's interpreted as another command in the list of commands that `if` will evaluate for an exit code. The `else` is encountered next, but it's out of place, since the shell recognizes it as a *reserved word* (a word that has special meaning to the shell) and not the name of a command, hence the error message.

Unanticipated Expansions

It's possible to have errors that only occur intermittently in a script. Sometimes the script will run fine and other times it will fail because of results of an expansion. If we return our missing semicolon and change the value of `number` to an empty variable, we can demonstrate:

```
#!/bin/bash

# trouble: script to demonstrate common errors

number=

if [ $number = 1 ]; then
      echo "Number is equal to 1."
else
      echo "Number is not equal to 1."
fi
```

Running the script with this change results in the output:

```
[me@linuxbox ~]$ trouble
/home/me/bin/trouble: line 7: [: =: unary operator expected
 Number is not equal to 1.
```

We get this rather cryptic error message, followed by the output of the second `echo` command. The problem is the expansion of the `number` variable within the `test` command. When the command:

```
[ $number = 1 ]
```

undergoes expansion with `number` being empty, the result is this:

```
[  = 1 ]
```

which is invalid and the error is generated. The `=` operator is a binary operator (it requires a value on each side), but the first value is missing, so the `test` command expects a unary operator (such as `-z`) instead. Further, since the `test` failed (because of the error), the `if` command receives a non-zero exit code and acts accordingly, and the second `echo` command is executed.

This problem can be corrected by adding quotes around the first argument in the `test` command:

```
[ "$number" = 1 ]
```

Then when expansion occurs, the result will be this:

```
[ "" = 1 ]
```

which yields the correct number of arguments. In addition to empty strings, quotes should be used in cases where a value could expand into multi-word strings, as with filenames containing embedded spaces.

Logical Errors

Unlike syntactic errors, *logical errors* do not prevent a script from running. The script will run, but it will not produce the desired result, due to a problem with its logic. There are countless numbers of possible logical errors, but here are a few of the most common kinds found in scripts:

1. **Incorrect conditional expressions.** It's easy to incorrectly code an if/then/else and have the wrong logic carried out. Sometimes the logic will be reversed or it will be incomplete.
2. **"Off by one" errors.** When coding loops that employ counters, it is possible to overlook that the loop may require the counting start with zero, rather than one, for the count to conclude at the correct point. These kinds of errors result in either a loop "going off the end" by counting too far, or else missing the last iteration of the loop by terminating one iteration too soon.
3. **Unanticipated situations.** Most logic errors result from a program encountering data or situations that were unforeseen by the programmer. This can also include unanticipated expansions, such as a filename that contains embedded spaces that expands into multiple command arguments rather than a single filename.

Defensive Programming

It is important to verify assumptions when programming. This means a careful evaluation of the exit status of programs and commands that are used by a script. Here is an example based on a true story. An unfortunate system administrator wrote a script to perform a maintenance task on an important server. The script contained the following two lines of code:

```
cd $dir_name
rm *
```

There is nothing intrinsically wrong with these two lines, as long as the directory named in the variable, `dir_name`, exists. But what happens if it does not? In that case, the `cd` command fails, the script continues to the next line and deletes the files in the current working directory. Not the desired outcome at all! The hapless administrator destroyed an important part of the server because of this design decision.

Let's look at some ways this design could be improved. First, it might be wise to make the execution of rm contingent on the success of cd:

```
cd $dir_name && rm *
```

This way, if the `cd` command fails, the `rm` command is not carried out. This is better, but still leaves open the possibility that the variable, `dir_name`, is unset or empty, which would result in the files in the user's home directory being deleted. This could also be avoided by checking to see that `dir_name` actually contains the name of an existing directory:

```
[[ -d $dir_name ]] && cd $dir_name && rm *
```

Often, it is best to terminate the script with an error when an situation such as the one above occurs:

```
if [[ -d $dir_name ]]; then
      if cd $dir_name; then
            rm *
      else
            echo "cannot cd to '$dir_name'" >&2
            exit 1
      fi
else
      echo "no such directory: '$dir_name'" >&2
      exit 1
fi
```

Here, we check both the name, to see that it is that of an existing directory, and the success of the cd command. If either fails, a descriptive error message is sent to standard error and the script terminates with an exit status of one to indicate a failure.

Verifying Input

A general rule of good programming is that if a program accepts input, it must be able to deal with anything it receives. This usually means that input must be carefully screened, to ensure that only valid input is accepted for further processing. We saw an example of this in the previous chapter when we studied the `read` command. One script contained the following test to verify a menu selection:

```
[[ $REPLY =~ ^[0-3]$ ]]
```

This test is very specific. It will only return a zero exit status if the string returned by the user is a numeral in the range of zero to three. Nothing else will be accepted. Sometimes these sorts of tests can be very challenging to write, but the effort is necessary to produce a high quality script.

Design Is A Function Of Time

When I was a college student studying industrial design, a wise professor stated that the degree of design on a project was determined by the amount of time given to the designer. If you were given five minutes to design a device "that kills flies," you designed a flyswatter. If you were given five months, you might come up with a laser-guided "anti-fly system" instead.

The same principle applies to programming. Sometimes a "quick and dirty" script will do if it's only going to be used once and only used by the programmer. That kind of script is common and should be developed quickly to make the effort economical. Such scripts don't need a lot of comments and defensive checks. On the other hand, if a script is intended for *production use*, that is, a script that will be used over and over for an important task or by multiple users, it needs much more careful development.

Testing

Testing is an important step in every kind of software development, including scripts. There is a saying in the open source world, "release early, release often," which reflects this fact. By releasing early and often, software gets more exposure to use and testing. Experience has shown that bugs are much easier to find, and much less expensive to fix, if they are found early in the development cycle.

In a previous discussion, we saw how stubs can be used to verify program flow. From

the earliest stages of script development, they are a valuable technique to check the progress of our work.

Let's look at the file deletion problem above and see how this could be coded for easy testing. Testing the original fragment of code would be dangerous, since its purpose is to delete files, but we could modify the code to make the test safe:

```
if [[ -d $dir_name ]]; then
      if cd $dir_name; then
            echo rm * # TESTING
      else
            echo "cannot cd to '$dir_name'" >&2
            exit 1
      fi
else
      echo "no such directory: '$dir_name'" >&2
      exit 1
fi
exit # TESTING
```

Since the error conditions already output useful messages, we don't have to add any. The most important change is placing an `echo` command just before the `rm` command to allow the command and its expanded argument list to be displayed, rather than the command actually being executed. This change allows safe execution of the code. At the end of the code fragment, we place an `exit` command to conclude the test and prevent any other part of the script from being carried out. The need for this will vary according to the design of the script.

We also include some comments that act as "markers" for our test-related changes. These can be used to help find and remove the changes when testing is complete.

Test Cases

To perform useful testing, it's important to develop and apply good *test cases*. This is done by carefully choosing input data or operating conditions that reflect *edge* and *corner* cases. In our code fragment (which is very simple), we want to know how the code performs under three specific conditions:

1. `dir_name` contains the name of an existing directory
2. `dir_name` contains the name of a non-existent directory
3. `dir_name` is empty

By performing the test with each of these conditions, good *test coverage* is achieved.

Just as with design, testing is a function of time, as well. Not every script feature needs to be extensively tested. It's really a matter of determining what is most important. Since it could be so potentially destructive if it malfunctioned, our code fragment deserves careful consideration during both its design and testing.

Debugging

If testing reveals a problem with a script, the next step is debugging. "A problem" usually means that the script is, in some way, not performing to the programmers expectations. If this is the case, we need to carefully determine exactly what the script is actually doing and why. Finding bugs can sometimes involve a lot of detective work.

A well designed script will try to help. It should be programmed defensively, to detect abnormal conditions and provide useful feedback to the user. Sometimes, however, problems are quite strange and unexpected and more involved techniques are required.

Finding The Problem Area

In some scripts, particularly long ones, it is sometimes useful to isolate the area of the script that is related to the problem. This won't always be the actual error, but isolation will often provide insights into the actual cause. One technique that can be used to isolate code is "commenting out" sections a script. For example, our file deletion fragment could be modified to determine if the removed section was related to an error:

```
if [[ -d $dir_name ]]; then
      if cd $dir_name; then
            rm *
      else
            echo "cannot cd to '$dir_name'" >&2
            exit 1
      fi
# else
#     echo "no such directory: '$dir_name'" >&2
#     exit 1
fi
```

By placing comment symbols at the beginning of each line in a logical section of a script, we prevent that section from being executed. Testing can then be performed again, to see if the removal of the code has any impact on the behavior of the bug.

Tracing

Bugs are often cases of unexpected logical flow within a script. That is, portions of the

script are either never being executed, or are being executed in the wrong order or at the wrong time. To view the actual flow of the program, we use a technique called *tracing*.

One tracing method involves placing informative messages in a script that display the location of execution. We can add messages to our code fragment:

```
echo "preparing to delete files" >&2
if [[ -d $dir_name ]]; then
      if cd $dir_name; then
echo "deleting files" >&2
            rm *
      else
            echo "cannot cd to '$dir_name'" >&2
            exit 1
      fi
else
      echo "no such directory: '$dir_name'" >&2
      exit 1
fi
echo "file deletion complete" >&2
```

We send the messages to standard error to separate them from normal output. We also do not indent the lines containing the messages, so it is easier to find when it's time to remove them.

Now when the script is executed, it's possible to see that the file deletion has been performed:

```
[me@linuxbox ~]$ deletion-script
preparing to delete files
deleting files
file deletion complete
[me@linuxbox ~]$
```

`bash` also provides a method of tracing, implemented by the `-x` option and the `set` command with the `-x` option. Using our earlier `trouble` script, we can activate tracing for the entire script by adding the `-x` option to the first line:

```
#!/bin/bash -x

# trouble: script to demonstrate common errors

number=1
```

```
if [ $number = 1 ]; then
      echo "Number is equal to 1."
else
      echo "Number is not equal to 1."
fi
```

When executed, the results look like this:

```
[me@linuxbox ~]$ trouble
+ number=1
+ '[' 1 = 1 ']'
+ echo 'Number is equal to 1.'
Number is equal to 1.
```

With tracing enabled, we see the commands performed with expansions applied. The leading plus signs indicate the display of the trace to distinguish them from lines of regular output. The plus sign is the default character for trace output. It is contained in the PS4 (prompt string 4) shell variable. The contents of this variable can be adjusted to make the prompt more useful. Here, we modify the contents of the variable to include the current line number in the script where the trace is performed. Note that single quotes are required to prevent expansion until the prompt is actually used:

```
[me@linuxbox ~]$ export PS4='$LINENO + '
[me@linuxbox ~]$ trouble
5 + number=1
7 + '[' 1 = 1 ']'
8 + echo 'Number is equal to 1.'
Number is equal to 1.
```

To perform a trace on a selected portion of a script, rather than the entire script, we can use the `set` command with the `-x` option:

```
#!/bin/bash

# trouble: script to demonstrate common errors

number=1

set -x # Turn on tracing
if [ $number = 1 ]; then
```

```
      echo "Number is equal to 1."
else
      echo "Number is not equal to 1."
fi
set +x # Turn off tracing
```

We use the `set` command with the `-x` option to activate tracing and the `+x` option to deactivate tracing. This technique can be used to examine multiple portions of a troublesome script.

Examining Values During Execution

It is often useful, along with tracing, to display the content of variables to see the internal workings of a script while it is being executed. Applying additional `echo` statements will usually do the trick:

```
#!/bin/bash

# trouble: script to demonstrate common errors

number=1

echo "number=$number" # DEBUG
set -x # Turn on tracing
if [ $number = 1 ]; then
      echo "Number is equal to 1."
else
      echo "Number is not equal to 1."
fi
set +x # Turn off tracing
```

In this trivial example, we simply display the value of the variable number and mark the added line with a comment to facilitate its later identification and removal. This technique is particularly useful when watching the behavior of loops and arithmetic within scripts.

Summing Up

In this chapter, we looked at just a few of the problems that can crop up during script development. Of course, there are many more. The techniques described here will enable finding most common bugs. Debugging is a fine art that can be developed through experience, both in knowing how to avoid bugs (testing constantly throughout development) and in finding bugs (effective use of tracing).

Further Reading

- The Wikipedia has a couple of short articles on syntactic and logical errors:
  http://en.wikipedia.org/wiki/Syntax_error
  http://en.wikipedia.org/wiki/Logic_error
- There are many online resources for the technical aspects of `bash` programming:
  http://wooledge.org:8000/BashPitfalls
  http://tldp.org/LDP/abs/html/gotchas.html
  http://www.gnu.org/software/bash/manual/html_node/Reserved-Word-Index.html
- Eric Raymond's *The Art of Unix Programming* is a great resource for learning the basic concepts found in well-written Unix programs. Many of these ideas apply to shell scripts:
  http://www.faqs.org/docs/artu/
  http://www.faqs.org/docs/artu/ch01s06.html
- For really heavy-duty debugging, there is the Bash Debugger:
  http://bashdb.sourceforge.net/

32 – Flow Control: Branching With case

In this chapter, we will continue to look at flow control. In Chapter 29, we constructed some simple menus and built the logic used to act on a user's selection. To do this, we used a series of `if` commands to identify which of the possible choices has been selected. This type of construct appears frequently in programs, so much so that many programming languages (including the shell) provide a flow control mechanism for multiple-choice decisions.

`case`

The `bash` multiple-choice compound command is called `case`. It has the following syntax:

```
case word in
      [pattern [| pattern]...) commands ;;]...
esac
```

If we look at the `read-menu` program from Chapter 29, we see the logic used to act on a user's selection:

```
#!/bin/bash

# read-menu: a menu driven system information program

clear
echo "
Please Select:

1. Display System Information
2. Display Disk Space
3. Display Home Space Utilization
0. Quit
"
read -p "Enter selection [0-3] > "

if [[ $REPLY =~ ^[0-3]$ ]]; then
      if [[ $REPLY == 0 ]]; then
```

```
            echo "Program terminated."
            exit
      fi
      if [[ $REPLY == 1 ]]; then
            echo "Hostname: $HOSTNAME"
            uptime
            exit
      fi
      if [[ $REPLY == 2 ]]; then
            df -h
            exit
      fi
      if [[ $REPLY == 3 ]]; then
            if [[ $(id -u) -eq 0 ]]; then
                  echo "Home Space Utilization (All Users)"
                  du -sh /home/*
            else
                  echo "Home Space Utilization ($USER)"
                  du -sh $HOME
            fi
            exit
      fi
else
      echo "Invalid entry." >&2
      exit 1
fi
```

Using `case`, we can replace this logic with something simpler:

```
#!/bin/bash

# case-menu: a menu driven system information program

clear
echo "
Please Select:

1. Display System Information
2. Display Disk Space
3. Display Home Space Utilization
0. Quit
"
read -p "Enter selection [0-3] > "

case $REPLY in
      0)    echo "Program terminated."
            exit
            ;;
```

```
      1)    echo "Hostname: $HOSTNAME"
            uptime
            ;;
      2)    df -h
            ;;
      3)    if [[ $(id -u) -eq 0 ]]; then
                  echo "Home Space Utilization (All Users)"
                  du -sh /home/*
            else
                  echo "Home Space Utilization ($USER)"
                  du -sh $HOME
            fi
            ;;
      *)    echo "Invalid entry" >&2
            exit 1
            ;;
esac
```

The `case` command looks at the value of *word*, in our example, the value of the `REPLY` variable, and then attempts to match it against one of the specified *patterns*. When a match is found, the *commands* associated with the specified pattern are executed. After a match is found, no further matches are attempted.

Patterns

The patterns used by `case` are the same as those used by pathname expansion. Patterns are terminated with a “)” character. Here are some valid patterns:

Table32- 1: case Pattern Examples

Pattern	Description
`a)`	Matches if *word* equals “a”.
`[[:alpha:]])`	Matches if *word* is a single alphabetic character.
`???)`	Matches if *word* is exactly three characters long.
`*.txt)`	Matches if *word* ends with the characters “.txt”.
`*)`	Matches any value of *word*. It is good practice to include this as the last pattern in a `case` command, to catch any values of *word* that did not match a previous pattern; that is, to catch any possible invalid values.

Here is an example of patterns at work:

```
#!/bin/bash

read -p "enter word > "

case $REPLY in
      [[:alpha:]])      echo "is a single alphabetic character." ;;
      [ABC][0-9])       echo "is A, B, or C followed by a digit." ;;
      ???)              echo "is three characters long." ;;
      *.txt)            echo "is a word ending in '.txt'" ;;
      *)                echo "is something else." ;;
esac
```

It is also possible to combine multiple patterns using the vertical bar character as a separator. This creates an “or” conditional pattern. This is useful for such things as handling both upper and lower case characters. For example:

```
#!/bin/bash

# case-menu: a menu driven system information program

clear
echo "
Please Select:

A. Display System Information
B. Display Disk Space
C. Display Home Space Utilization
Q. Quit
"
read -p "Enter selection [A, B, C or Q] > "

case $REPLY in
      q|Q)  echo "Program terminated."
            exit
            ;;
      a|A)  echo "Hostname: $HOSTNAME"
            uptime
            ;;
      b|B)  df -h
            ;;
      c|C)  if [[ $(id -u) -eq 0 ]]; then
                  echo "Home Space Utilization (All Users)"
                  du -sh /home/*
            else
                  echo "Home Space Utilization ($USER)"
                  du -sh $HOME
            fi
```

```
            ;;
      *)    echo "Invalid entry" >&2
            exit 1
            ;;
esac
```

Here, we modify the `case-menu` program to use letters instead of digits for menu selection. Notice how the new patterns allow for entry of both upper and lower case letters.

Summing Up

The `case` command is a handy addition to our bag of programming tricks. As we will see in the next chapter, it's the perfect tool for handling certain types of problems.

Further Reading

- The *Bash Reference Manual* section on Conditional Constructs describes the `case` command in detail:
  http://tiswww.case.edu/php/chet/bash/bashref.html#SEC21
- The *Advanced Bash-Scripting Guide* provides further examples of `case` applications:
  http://tldp.org/LDP/abs/html/testbranch.html

33 – Positional Parameters

One feature that has been missing from our programs is the ability to accept and process command line options and arguments. In this chapter, we will examine the shell features that allow our programs to get access to the contents of the command line.

Accessing The Command Line

The shell provides a set of variables called *positional parameters* that contain the individual words on the command line. The variables are named `0` through `9`. They can be demonstrated this way:

```
#!/bin/bash

# posit-param: script to view command line parameters

echo "
\$0 = $0
\$1 = $1
\$2 = $2
\$3 = $3
\$4 = $4
\$5 = $5
\$6 = $6
\$7 = $7
\$8 = $8
\$9 = $9
"
```

A very simple script that displays the values of the variables `$0-$9`. When executed with no command line arguments:

```
[me@linuxbox ~]$  posit-param

$0 = /home/me/bin/posit-param
```

```
$1 =
$2 =
$3 =
$4 =
$5 =
$6 =
$7 =
$8 =
$9 =
```

Even when no arguments are provided, `$0` will always contain the first item appearing on the command line, which is the pathname of the program being executed. When arguments are provided, we see the results:

```
[me@linuxbox ~]$ posit-param a b c d

$0 = /home/me/bin/posit-param
$1 = a
$2 = b
$3 = c
$4 = d
$5 =
$6 =
$7 =
$8 =
$9 =
```

Note: You can actually access more than nine parameters using parameter expansion. To specify a number greater than nine, surround the number in braces. For example ${10}, ${55}, ${211} and so on.

Determining The Number of Arguments

The shell also provides a variable, $#, that yields the number of arguments on the command line:

```
#!/bin/bash

# posit-param: script to view command line parameters

echo "
```

```
Number of arguments: $#
\$0 = $0
\$1 = $1
\$2 = $2
\$3 = $3
\$4 = $4
\$5 = $5
\$6 = $6
\$7 = $7
\$8 = $8
\$9 = $9
"
```

The result:

```
[me@linuxbox ~]$ posit-param a b c d

Number of arguments: 4
$0 = /home/me/bin/posit-param
$1 = a
$2 = b
$3 = c
$4 = d
$5 =
$6 =
$7 =
$8 =
$9 =
```

`shift` – Getting Access To Many Arguments

But what happens when we give the program a large number of arguments such as this:

```
[me@linuxbox ~]$ posit-param *

Number of arguments: 82
$0 = /home/me/bin/posit-param
$1 = addresses.ldif
$2 = bin
$3 = bookmarks.html
$4 = debian-500-i386-netinst.iso
$5 = debian-500-i386-netinst.jigdo
$6 = debian-500-i386-netinst.template
$7 = debian-cd_info.tar.gz
```

```
$8 = Desktop
$9 = dirlist-bin.txt
```

On this example system, the wildcard * expands into eighty-two arguments. How can we process that many? The shell provides a method, albeit clumsy, to do this. The `shift` command causes all of the parameters to "move down one" each time it is executed. In fact, by using `shift`, it is possible to get by with only one parameter (in addition to `$0`, which never changes):

```
#!/bin/bash

# posit-param2: script to display all arguments

count=1

while [[ $# -gt 0 ]]; do
      echo "Argument $count = $1"
      count=$((count + 1))
      shift
done
```

Each time `shift` is executed, the value of `$2` is moved to `$1`, the value of `$3` is moved to `$2` and so on. The value of $# is also reduced by one.

In the `posit-param2` program, we create a loop that evaluates the number of arguments remaining and continues as long as there is at least one. We display the current argument, increment the variable `count` with each iteration of the loop to provide a running count of the number of arguments processed and, finally, execute a `shift` to load `$1` with the next argument. Here is the program at work:

```
[me@linuxbox ~]$ posit-param2 a b c d
Argument 1 = a
Argument 2 = b
Argument 3 = c
Argument 4 = d
```

Simple Applications

Even without `shift`, it's possible to write useful applications using positional parameters. By way of example, here is a simple file information program:

```
#!/bin/bash

# file_info: simple file information program

PROGNAME=$(basename $0)

if [[ -e $1 ]]; then
      echo -e "\nFile Type:"
      file $1
      echo -e "\nFile Status:"
      stat $1
else
      echo "$PROGNAME: usage: $PROGNAME file" >&2
      exit 1
fi
```

This program displays the file type (determined by the `file` command) and the file status (from the `stat` command) of a specified file. One interesting feature of this program is the `PROGNAME` variable. It is given the value that results from the `basename $0` command. The `basename` command removes the leading portion of a pathname, leaving only the base name of a file. In our example, `basename` removes the leading portion of the pathname contained in the `$0` parameter, the full pathname of our example program. This value is useful when constructing messages such as the usage message at the end of the program. By coding it this way, the script can be renamed and the message automatically adjusts to contain the name of the program.

Using Positional Parameters With Shell Functions

Just as positional parameters are used to pass arguments to shell scripts, they can also be used to pass arguments to shell functions. To demonstrate, we will convert the `file_info` script into a shell function:

```
file_info () {

      # file_info: function to display file information

      if [[ -e $1 ]]; then
            echo -e "\nFile Type:"
            file $1
            echo -e "\nFile Status:"
            stat $1
      else
            echo "$FUNCNAME: usage: $FUNCNAME file" >&2
            return 1
```

```
        fi
}
```

Now, if a script that incorporates the `file_info` shell function calls the function with a filename argument, the argument will be passed to the function.

With this capability, we can write many useful shell functions that can not only be used in scripts, but also within the `.bashrc` file.

Notice that the `PROGNAME` variable was changed to the shell variable `FUNCNAME`. The shell automatically updates this variable to keep track of the currently executed shell function. Note that `$0` always contains the full pathname of the first item on the command line (i.e., the name of the program) and does not contain the name of the shell function as we might expect.

Handling Positional Parameters *En Masse*

It is sometimes useful to manage all the positional parameters as a group. For example, we might want to write a "wrapper" around another program. This means that we create a script or shell function that simplifies the execution of another program. The wrapper supplies a list of arcane command line options and then passes a list of arguments to the lower level program.

The shell provides two special parameters for this purpose. They both expand into the complete list of positional parameters, but differ in rather subtle ways. They are:

*Table 33-1: The * And $ Special Parameters*

Parameter	Description
`$*`	Expands into the list of positional parameters, starting with 1. When surrounded by double quotes, it expands into a double quoted string containing all of the positional parameters, each separated by the first character of the IFS shell variable (by default a space character).
`$@`	Expands into the list of positional parameters, starting with 1. When surrounded by double quotes, it expands each positional parameter into a separate word surrounded by double quotes.

Here is a script that shows them in action:

```
#!/bin/bash

# posit-params3 : script to demonstrate $* and $@

print_params () {
      echo "\$1 = $1"
      echo "\$2 = $2"
      echo "\$3 = $3"
      echo "\$4 = $4"
}

pass_params () {
      echo -e "\n" '$* :';   print_params $*
      echo -e "\n" '"$*" :'; print_params "$*"
      echo -e "\n" '$@ :';   print_params $@
      echo -e "\n" '"$@" :'; print_params "$@"
}

pass_params "word" "words with spaces"
```

In this rather convoluted program, we create two arguments: “word” and “words with spaces” and pass them to the `pass_params` function. That function, in turn, passes them on to the `print_params` function using each of the four methods available with the special parameters `$!` and `$@`. When executed, the script reveals the differences:

```
[me@linuxbox ~]$ posit-param3

 $* :
$1 = word
$2 = words
$3 = with
$4 = spaces

 "$*" :
$1 = word words with spaces
$2 =
$3 =
$4 =

 $@ :
$1 = word
$2 = words
$3 = with
$4 = spaces

 "$@" :
$1 = word
```

```
$2 = words with spaces
$3 =
$4 =
```

With our arguments, both `$!` and `$@` produce a four word result:

```
word words with spaces
```

`"$*"` produces a one word result:

```
"word words with spaces"
```

`"$@"` produces a two word result:

```
"word" "words with spaces"
```

which matches our actual intent. The lesson to take from this is that even though the shell provides four different ways of getting the list of positional parameters, `"$@"` is by far the most useful for most situations, because it preserves the integrity of each positional parameter.

A More Complete Application

After a long hiatus, we are going to resume work on our `sys_info_page` program. Our next addition will add several command line options to the program as follows:

- **Output file.** We will add an option to specify a name for a file to contain the program's output. It will be specified as either `-f` *`file`* or `--file` *`file`*.
- **Interactive mode.** This option will prompt the user for an output filename and will determine if the specified file already exists. If it does, the user will be prompted before the existing file is overwritten. This option will be specified by either `-i` or `--interactive`.
- **Help.** Either `-h` or `--help` may be specified to cause the program to output an informative usage message.

Here is the code needed to implement the command line processing:

```
usage () {
      echo "$PROGNAME: usage: $PROGNAME [-f file | -i]"
      return
}

# process command line options

interactive=
```

```
filename=

while [[ -n $1 ]]; do
      case $1 in
            -f | --file)            shift
                                    filename=$1
                                    ;;
            -i | --interactive)     interactive=1
                                    ;;
            -h | --help)            usage
                                    exit
                                    ;;
            *)                      usage >&2
                                    exit 1
                                    ;;
      esac
      shift
done
```

First, we add a shell function called `usage` to display a message when the help option is invoked or an unknown option is attempted.

Next, we begin the processing loop. This loop continues while the positional parameter `$1` is not empty. At the bottom of the loop, we have a `shift` command to advance the positional parameters to ensure that the loop will eventually terminate.

Within the loop, we have a `case` statement that examines the current positional parameter to see if it matches any of the supported choices. If a supported parameter is found, it is acted upon. If not, the usage message is displayed and the script terminates with an error.

The `-f` parameter is handled in an interesting way. When detected, it causes an additional `shift` to occur, which advances the positional parameter `$1` to the filename argument supplied to the `-f` option.

We next add the code to implement the interactive mode:

```
# interactive mode

if [[ -n $interactive ]]; then
      while true; do
            read -p "Enter name of output file: " filename
            if [[ -e $filename ]]; then
                  read -p "'$filename' exists. Overwrite? [y/n/q] > "

                  case $REPLY in
```

```
                        Y|y)  break
                              ;;
                        Q|q)  echo "Program terminated."
                              exit
                              ;;
                        *)    continue
                              ;;
                  esac
            elif [[ -z $filename ]]; then
                  continue
            else
                  break
            fi
      done
fi
```

If the `interactive` variable is not empty, an endless loop is started, which contains the filename prompt and subsequent existing file handling code. If the desired output file already exists, the user is prompted to overwrite, choose another filename, or quit the program. If the user chooses to overwrite an existing file, a `break` is executed to terminate the loop. Notice how the `case` statement only detects if the user chooses to overwrite or quit. Any other choice causes the loop to continue and prompts the user again.

In order to implement the output filename feature, we must first convert the existing page writing code into a shell function, for reasons that will become clear in a moment:

```
write_html_page () {
      cat <<- _EOF_
      <HTML>
            <HEAD>
                  <TITLE>$TITLE</TITLE>
            </HEAD>
            <BODY>
                  <H1>$TITLE</H1>
                  <P>$TIME_STAMP</P>
                  $(report_uptime)
                  $(report_disk_space)
                  $(report_home_space)
            </BODY>
      </HTML>
      _EOF_
      return
}

# output html page
```

```
if [[ -n $filename ]]; then
      if touch $filename && [[ -f $filename ]]; then
            write_html_page > $filename
      else
            echo "$PROGNAME: Cannot write file '$filename'" >&2
            exit 1
      fi
else
      write_html_page
fi
```

The code that handles the logic of the `-f` option appears at the end of the listing shown above. In it, we test for the existence of a filename and, if found, a test is performed to see if the file is indeed writable. To do this, a `touch` is performed, followed by a test to determine if the resulting file is a regular file. These two tests take care of situations where an invalid pathname is input (`touch` will fail) and if the file already exists, that it's a regular file.

As we can see, the `write_html_page` function is called to perform the actual generation of the page. Its output is either directed to standard output (if the variable `filename` is empty) or redirected to the specified file.

Summing Up

With the addition of positional parameters, we can now write fairly functional scripts. For simple, repetitive tasks, positional parameters make it possible to write very useful shell functions that can be placed in a user's `.bashrc` file.

Our `sys_info_page` program has grown in complexity and sophistication. Here is a complete listing, with the most recent changes highlighted:

```
#!/bin/bash

# sys_info_page: program to output a system information page

PROGNAME=$(basename $0)
TITLE="System Information Report For $HOSTNAME"
CURRENT_TIME=$(date +"%x %r %Z")
TIME_STAMP="Generated $CURRENT_TIME, by $USER"

report_uptime () {
      cat <<- _EOF_
            <H2>System Uptime</H2>
            <PRE>$(uptime)</PRE>
```

```
            _EOF_
      return
}

report_disk_space () {
      cat <<- _EOF_
            <H2>Disk Space Utilization</H2>
            <PRE>$(df -h)</PRE>
            _EOF_
      return
}

report_home_space () {
      if [[ $(id -u) -eq 0 ]]; then
            cat <<- _EOF_
                  <H2>Home Space Utilization (All Users)</H2>
                  <PRE>$(du -sh /home/*)</PRE>
                  _EOF_
      else
            cat <<- _EOF_
                  <H2>Home Space Utilization ($USER)</H2>
                  <PRE>$(du -sh $HOME)</PRE>
                  _EOF_
      fi
      return
}

usage () {
      echo "$PROGNAME: usage: $PROGNAME [-f file | -i]"
      return
}

write_html_page () {
      cat <<- _EOF_
      <HTML>
            <HEAD>
                  <TITLE>$TITLE</TITLE>
            </HEAD>
            <BODY>
                  <H1>$TITLE</H1>
                  <P>$TIME_STAMP</P>
                  $(report_uptime)
                  $(report_disk_space)
                  $(report_home_space)
            </BODY>
      </HTML>
      _EOF_
      return
}
```

```
# process command line options

interactive=
filename=

while [[ -n $1 ]]; do
      case $1 in
            -f | --file)            shift
                                    filename=$1
                                    ;;
            -i | --interactive)     interactive=1
                                    ;;
            -h | --help)            usage
                                    exit
                                    ;;
            *)                      usage >&2
                                    exit 1
                                    ;;
      esac
      shift
done

# interactive mode

if [[ -n $interactive ]]; then
      while true; do
            read -p "Enter name of output file: " filename
            if [[ -e $filename ]]; then
                  read -p "'$filename' exists. Overwrite? [y/n/q] > "

                  case $REPLY in
                        Y|y)  break
                              ;;
                        Q|q)  echo "Program terminated."
                              exit
                              ;;
                        *)    continue
                              ;;
                  esac
            fi
      done
fi

# output html page

if [[ -n $filename ]]; then
      if touch $filename && [[ -f $filename ]]; then
            write_html_page > $filename
      else
            echo "$PROGNAME: Cannot write file '$filename'" >&2
```

```
            exit 1
      fi
else
      write_html_page
fi
```

We're not done yet. There are still more things we can do and improvements we can make.

Further Reading

- The *Bash Hackers Wiki* has a good article on positional parameters:
  http://bash-hackers.org/wiki/doku.php/scripting/posparams
- The *Bash Reference Manual* has an article on the special parameters, including `$*` and `$@`:
  http://www.gnu.org/software/bash/manual/bashref.html#Special-Parameters
- In addition to the techniques discussed in this chapter, `bash` includes a builtin command called `getopts`, which can also be used for process command line arguments. It is described in the SHELL BUILTIN COMMANDS section of the `bash` man page and at the *Bash Hackers Wiki*:
  http://bash-hackers.org/wiki/doku.php/howto/getopts_tutorial

34 – Flow Control: Looping With for

In this final chapter on flow control, we will look at another of the shell's looping constructs. The *for loop* differs from the while and until loops in that it provides a means of processing sequences during a loop. This turns out to be very useful when programming. Accordingly, the for loop is a very popular construct in `bash` scripting.

A for loop is implemented, naturally enough, with the `for` command. In modern versions of `bash`, `for` is available in two forms.

`for`: Traditional Shell Form

The original `for` command's syntax is:

```
for variable [in words]; do
      commands
done
```

Where *variable* is the name of a variable that will increment during the execution of the loop, *words* is an optional list of items that will be sequentially assigned to *variable*, and *commands* are the commands that are to be executed on each iteration of the loop.

The `for` command is useful on the command line. We can easily demonstrate how it works:

```
[me@linuxbox ~]$ for i in A B C D; do echo $i; done
A
B
C
D
```

In this example, `for` is given a list of four words: "A," "B," "C," and "D." With a list of four words, the loop is executed four times. Each time the loop is executed, a word is assigned to the variable `i`. Inside the loop, we have an `echo` command that displays the value of `i` to show the assignment. As with the `while` and `until` loops, the `done` keyword closes the loop.

The really powerful feature of `for` is the number of interesting ways we can create the list of words. For example, through brace expansion:

```
[me@linuxbox ~]$ for i in {A..D}; do echo $i; done
A
B
C
D
```

Or pathname expansion:

```
[me@linuxbox ~]$ for i in distros*.txt; do echo $i; done
distros-by-date.txt
distros-dates.txt
distros-key-names.txt
distros-key-vernums.txt
distros-names.txt
distros.txt
distros-vernums.txt
distros-versions.txt
```

Or command substitution:

```
#!/bin/bash

# longest-word : find longest string in a file

while [[ -n $1 ]]; do
      if [[ -r $1 ]]; then
            max_word=
            max_len=0
            for i in $(strings $1); do
                  len=$(echo $i | wc -c)
                  if (( len > max_len )); then
                        max_len=$len
                        max_word=$i
                  fi
            done
            echo "$1: '$max_word' ($max_len characters)"
      fi
      shift
done
```

In this example, we look for the longest string found within a file. When given one or more files names on the command line, this program uses the `strings` program (which is included in the GNU binutils package) to generate a list of readable text "words" in each file. The `for` loop processes each word in turn and determines if the current word is the longest found so far. When the loop concludes, the longest word is displayed.

If the optional `in` *`words`* portion of the `for` command is omitted, `for` defaults to processing the positional parameters. We will modify our `longest-word` script to use this method:

```
#!/bin/bash

# longest-word2 : find longest string in a file

for i; do
      if [[ -r $i ]]; then
            max_word=
            max_len=0
            for j in $(strings $i); do
                  len=$(echo $j | wc -c)
                  if (( len > max_len )); then
                        max_len=$len
                        max_word=$j
                  fi
            done
            echo "$i: '$max_word' ($max_len characters)"
      fi
done
```

As we can see, we have changed the outermost loop to use `for` in place of `while`. By omitting the list of words in the `for` command, the positional parameters are used instead. Inside the loop, previous instances of the variable `i` have been changed to the variable `j`. The use of `shift` has also been eliminated.

Why i?

You may have noticed that the variable `i` was chosen for each of the `for` loop examples above. Why? No specific reason actually, besides tradition. The variable used with `for` can be any valid variable, but `i` is the most common, followed by `j` and `k`.

The basis of this tradition comes from the Fortran programming language. In Fortran, undeclared variables starting with the letters I, J, K, L, and M are automatically typed as integers while variables beginning with any other letter are typed as real (numbers with decimal fractions). This behavior led programmers to use the variables I, J, and K for loop variables, since it was less work to use them when a temporary variable (as loop variables often are) was needed.

It also led to the following Fortran-based witticism:

"GOD is real, unless declared integer."

`for`: C Language Form

Recent versions of `bash` have added a second form of `for` command syntax, one that resembles the form found in the C programming language. Many other languages support this form, as well:

```
for (( expression1; expression2; expression3 )); do
      commands
done
```

where *expression1*, *expression2*, and *expression3* are arithmetic expressions and *commands* are the commands to be performed during each iteration of the loop.

In terms of behavior, this form is equivalent to the following construct:

```
(( expression1 ))
while (( expression2 )); do
      commands
      (( expression3 ))
done
```

expression1 is used to initialize conditions for the loop, *expression2* is used to determine when the loop is finished, and *expression3* is carried out at the end of each iteration of the loop.

Here is a typical application:

```
#!/bin/bash

# simple_counter : demo of C style for command

for (( i=0; i<5; i=i+1 )); do
      echo $i
done
```

When executed, it produces the following output:

```
[me@linuxbox ~]$ simple_counter
0
1
2
3
4
```

In this example, *expression1* initializes the variable `i` with the value of zero, *expression2* allows the loop to continue as long as the value of `i` remains less than five, and *expression3* increments the value of `i` by one each time the loop repeats.

The C language form of `for` is useful anytime a numeric sequence is needed. We will see several applications for this in the next two chapters.

Summing Up

With our knowledge of the `for` command, we will now apply the final improvements to our `sys_info_page` script. Currently, the `report_home_space` function looks like this:

```
report_home_space () {
      if [[ $(id -u) -eq 0 ]]; then
            cat <<- _EOF_
                  <H2>Home Space Utilization (All Users)</H2>
                  <PRE>$(du -sh /home/*)</PRE>
                  _EOF_
      else
            cat <<- _EOF_
                  <H2>Home Space Utilization ($USER)</H2>
                  <PRE>$(du -sh $HOME)</PRE>
                  _EOF_
      fi
      return
}
```

Next, we will rewrite it to provide more detail for each user's home directory, and include the total number of files and subdirectories in each:

```
report_home_space () {

```

```
      local format="%8s%10s%10s\n"
      local i dir_list total_files total_dirs total_size user_name

      if [[ $(id -u) -eq 0 ]]; then
            dir_list=/home/*
            user_name="All Users"
      else
            dir_list=$HOME
            user_name=$USER
      fi

      echo "<H2>Home Space Utilization ($user_name)</H2>"

      for i in $dir_list; do

            total_files=$(find $i -type f | wc -l)
            total_dirs=$(find $i -type d | wc -l)
            total_size=$(du -sh $i | cut -f 1)

            echo "<H3>$i</H3>"
            echo "<PRE>"
            printf "$format" "Dirs" "Files" "Size"
            printf "$format" "----" "-----" "----"
            printf "$format" $total_dirs $total_files $total_size
            echo "</PRE>"
      done
      return
}
```

This rewrite applies much of what we have learned so far. We still test for the superuser, but instead of performing the complete set of actions as part of the `if`, we set some variables used later in a for loop. We have added several local variables to the function and made use of `printf` to format some of the output.

Further Reading

- The *Advanced Bash-Scripting Guide* has a chapter on loops, with a variety of examples using `for`:
  http://tldp.org/LDP/abs/html/loops1.html
- The *Bash Reference Manual* describes the looping compound commands, including `for`:
  http://www.gnu.org/software/bash/manual/bashref.html#Looping-Constructs

35 – Strings And Numbers

Computer programs are all about working with data. In past chapters, we have focused on processing data at the file level. However, there are many programming problems that need to be solved using smaller units of data such as strings and numbers.

In this chapter, we will look at several shell features that are used to manipulate strings and numbers. The shell provides a variety of parameter expansions that perform string operations. In addition to arithmetic expansion (which we touched upon in Chapter 8), there is a common command line program called bc, which performs higher level math.

Parameter Expansion

Though parameter expansion came up in Chapter 8, we did not cover it in detail because most parameter expansions are used in scripts rather than on the command line. We have already worked with some forms of parameter expansion; for example, shell variables. The shell provides many more.

Basic Parameters

The simplest form of parameter expansion is reflected in the ordinary use of variables. For example:

```
$a
```

when expanded, becomes whatever the variable a contains. Simple parameters may also be surrounded by braces:

```
${a}
```

This has no effect on the expansion, but is required if the variable is adjacent to other text, which may confuse the shell. In this example, we attempt to create a filename by appending the string "_file" to the contents of the variable a.

```
[me@linuxbox ~]$ a="foo"
[me@linuxbox ~]$ echo "$a_file"
```

If we perform this sequence, the result will be nothing, because the shell will try to expand a variable named `a_file` rather than `a`. This problem can be solved by adding braces:

```
[me@linuxbox ~]$ echo "${a}_file"
foo_file
```

We have also seen that positional parameters greater than 9 can be accessed by surrounding the number in braces. For example, to access the eleventh positional parameter, we can do this:

```
${11}
```

Expansions To Manage Empty Variables

Several parameter expansions deal with nonexistent and empty variables. These expansions are handy for handling missing positional parameters and assigning default values to parameters.

${*parameter*:-*word*}

If *parameter* is unset (i.e., does not exist) or is empty, this expansion results in the value of *word*. If *parameter* is not empty, the expansion results in the value of *parameter*.

```
[me@linuxbox ~]$ foo=
[me@linuxbox ~]$ echo ${foo:-"substitute value if unset"}
substitute value if unset
[me@linuxbox ~]$ echo $foo

[me@linuxbox ~]$ foo=bar
[me@linuxbox ~]$ echo ${foo:-"substitute value if unset"}
bar
[me@linuxbox ~]$ echo $foo
bar
```

${*parameter*:=*word*}

If *parameter* is unset or empty, this expansion results in the value of *word*. In addition, the value of *word* is assigned to *parameter*. If *parameter* is not empty, the expansion results in the value of *parameter*.

```
[me@linuxbox ~]$ foo=
[me@linuxbox ~]$ echo ${foo:="default value if unset"}
```

```
default value if unset
[me@linuxbox ~]$ echo $foo
default value if unset
[me@linuxbox ~]$ foo=bar
[me@linuxbox ~]$ echo ${foo:="default value if unset"}
bar
[me@linuxbox ~]$ echo $foo
bar
```

Note: Positional and other special parameters cannot be assigned this way.

${*parameter*:?*word*}

If *parameter* is unset or empty, this expansion causes the script to exit with an error, and the contents of *word* are sent to standard error. If *parameter* is not empty, the expansion results in the value of *parameter*.

```
[me@linuxbox ~]$ foo=
[me@linuxbox ~]$ echo ${foo:?"parameter is empty"}
bash: foo: parameter is empty
[me@linuxbox ~]$ echo $?
1
[me@linuxbox ~]$ foo=bar
[me@linuxbox ~]$ echo ${foo:?"parameter is empty"}
bar
[me@linuxbox ~]$ echo $?
0
```

${*parameter*:+*word*}

If *parameter* is unset or empty, the expansion results in nothing. If *parameter* is not empty, the value of *word* is substituted for *parameter*; however, the value of *parameter* is not changed.

```
[me@linuxbox ~]$ foo=
[me@linuxbox ~]$ echo ${foo:+"substitute value if set"}

[me@linuxbox ~]$ foo=bar
[me@linuxbox ~]$ echo ${foo:+"substitute value if set"}
substitute value if set
```

Expansions That Return Variable Names

The shell has the ability to return the names of variables. This is used in some rather exotic situations.

`${!prefix*}`
`${!prefix@}`

This expansion returns the names of existing variables with names beginning with *prefix*. According to the `bash` documentation, both forms of the expansion perform identically. Here, we list all the variables in the environment with names that begin with BASH:

```
[me@linuxbox ~]$ echo ${!BASH*}
BASH BASH_ARGC BASH_ARGV BASH_COMMAND BASH_COMPLETION
BASH_COMPLETION_DIR BASH_LINENO BASH_SOURCE BASH_SUBSHELL
BASH_VERSINFO BASH_VERSION
```

String Operations

There is a large set of expansions that can be used to operate on strings. Many of these expansions are particularly well-suited for operations on pathnames.

`${#parameter}`

Expands into the length of the string contained by *parameter*. Normally, *parameter* is a string; however, if *parameter* is either `@` or `*`, then the expansion results in the number of positional parameters.

```
[me@linuxbox ~]$ foo="This string is long."
[me@linuxbox ~]$ echo "'$foo' is ${#foo} characters long."
'This string is long.' is 20 characters long.
```

`${parameter:offset}`
`${parameter:offset:length}`

This expansion is used to extract a portion of the string contained in *parameter*. The extraction begins at *offset* characters from the beginning of the string and continues until the end of the string unless the *length* is specified.

```
[me@linuxbox ~]$ foo="This string is long."
[me@linuxbox ~]$ echo ${foo:5}
string is long.
```

```
[me@linuxbox ~]$ echo ${foo:5:6}
string
```

If the value of *offset* is negative, it is taken to mean it starts from the end of the string rather than the beginning. Note that negative values must be preceded by a space to prevent confusion with the ${*parameter*:-*word*} expansion. *length*, if present, must not be less than zero.

If *parameter* is @, the result of the expansion is *length* positional parameters, starting at *offset*.

```
[me@linuxbox ~]$ foo="This string is long."
[me@linuxbox ~]$ echo ${foo: -5}
long.
[me@linuxbox ~]$ echo ${foo: -5:2}
lo
```

${*parameter*#*pattern*}
${*parameter*##*pattern*}

This expansion removes a leading portion of the string contained in *parameter* defined by *pattern*. *pattern* is a wildcard pattern like those used in pathname expansion. The difference in the two forms is that the # form removes the shortest match, while the ## form removes the longest match.

```
[me@linuxbox ~]$ foo=file.txt.zip
[me@linuxbox ~]$ echo ${foo#*.}
txt.zip
[me@linuxbox ~]$ echo ${foo##*.}
zip
```

${*parameter*%*pattern*}
${*parameter*%%*pattern*}

These expansions are the same as the # and ## expansions above, except they remove text from the end of the string contained in *parameter* rather than from the beginning.

```
[me@linuxbox ~]$ foo=file.txt.zip
[me@linuxbox ~]$ echo ${foo%.*}
file.txt
[me@linuxbox ~]$ echo ${foo%%.*}
```

```
file
```

```
${parameter/pattern/string}
${parameter//pattern/string}
${parameter/#pattern/string}
${parameter/%pattern/string}
```

This expansion performs a search and replace upon the contents of *parameter*. If text is found matching wildcard *pattern*, it is replaced with the contents of *string*. In the normal form, only the first occurrence of *pattern* is replaced. In the `//` form, all occurrences are replaced. The `/#` form requires that the match occur at the beginning of the string, and the `/%` form requires the match to occur at the end of the string. */string* may be omitted, which causes the text matched by *pattern* to be deleted.

```
[me@linuxbox ~]$ foo=JPG.JPG
[me@linuxbox ~]$ echo ${foo/JPG/jpg}
jpg.JPG
[me@linuxbox ~]$ echo ${foo//JPG/jpg}
jpg.jpg
[me@linuxbox ~]$ echo ${foo/#JPG/jpg}
jpg.JPG
[me@linuxbox ~]$ echo ${foo/%JPG/jpg}
JPG.jpg
```

Parameter expansion is a good thing to know. The string manipulation expansions can be used as substitutes for other common commands such as `sed` and `cut`. Expansions improve the efficiency of scripts by eliminating the use of external programs. As an example, we will modify the `longest-word` program discussed in the previous chapter to use the parameter expansion `${#j}` in place of the command substitution `$(echo $j | wc -c)` and its resulting subshell like so:

```
#!/bin/bash

# longest-word3 : find longest string in a file

for i; do
      if [[ -r $i ]]; then
            max_word=
            max_len=
            for j in $(strings $i); do
                  len=${#j}
                  if (( len > max_len )); then
```

```
                        max_len=$len
                        max_word=$j
                fi
            done
            echo "$i: '$max_word' ($max_len characters)"
        fi
        shift
done
```

Next, we will compare the efficiency of the two versions by using the `time` command:

```
[me@linuxbox ~]$ time longest-word2 dirlist-usr-bin.txt
dirlist-usr-bin.txt: 'scrollkeeper-get-extended-content-list' (38
characters)

real  0m3.618s
user  0m1.544s
sys   0m1.768s
[me@linuxbox ~]$ time longest-word3 dirlist-usr-bin.txt
dirlist-usr-bin.txt: 'scrollkeeper-get-extended-content-list' (38
characters)

real  0m0.060s
user  0m0.056s
sys   0m0.008s
```

The original version of the script takes 3.618 seconds to scan the text file, while the new version, using parameter expansion, takes only 0.06 seconds—a very significant improvement.

Arithmetic Evaluation And Expansion

We looked at arithmetic expansion in Chapter 8. It is used to perform various arithmetic operations on integers. Its basic form is:

`$((`*`expression`*`))`

where *expression* is a valid arithmetic expression.

This is related to the compound command `(( ))` used for arithmetic evaluation (truth tests) we encountered in Chapter 28.

In previous chapters, we saw some of the common types of expressions and operators. Here, we will look at a more complete list.

Number Bases

Back in Chapter 10, we got a look at octal (base 8) and hexadecimal (base 16) numbers. In arithmetic expressions, the shell supports integer constants in any base.

Table 35-1: Specifying Different Number Bases

Notation	Description
number	By default, numbers without any notation are treated as decimal (base 10) integers.
0*number*	In arithmetic expressions, numbers with a leading zero are considered octal.
0x*number*	Hexadecimal notation
base#*number*	*number* is in *base*

Some examples:

```
[me@linuxbox ~]$ echo $((0xff))
255
[me@linuxbox ~]$ echo $((2#11111111))
255
```

In the examples above, we print the value of the hexadecimal number `ff` (the largest two- digit number) and the largest eight-digit binary (base 2) number.

Unary Operators

There are two unary operators, the `+` and `-`, which are used to indicate if a number is positive or negative, respectively. For example, `-5`.

Simple Arithmetic

The ordinary arithmetic operators are listed in the table below:

Table 35-2: Arithmetic Operators

Operator	Description
+	Addition
-	Subtraction

*	Multiplication
/	Integer division
**	Exponentiation
%	Modulo (remainder)

Most of these are self-explanatory, but integer division and modulo require further discussion.

Since the shell's arithmetic only operates on integers, the results of division are always whole numbers:

```
[me@linuxbox ~]$ echo $(( 5 / 2 ))
2
```

This makes the determination of a remainder in a division operation more important:

```
[me@linuxbox ~]$ echo $(( 5 % 2 ))
1
```

By using the division and modulo operators, we can determine that five divided by two results in two, with a remainder of one.

Calculating the remainder is useful in loops. It allows an operation to be performed at specified intervals during the loop's execution. In the example below, we display a line of numbers, highlighting each multiple of five:

```
#!/bin/bash

# modulo : demonstrate the modulo operator

for ((i = 0; i <= 20; i = i + 1)); do
    remainder=$((i % 5))
    if (( remainder == 0 )); then
        printf "<%d> " $i
    else
        printf "%d " $i
    fi
done
printf "\n"
```

When executed, the results look like this:

```
[me@linuxbox ~]$ modulo
<0> 1 2 3 4 <5> 6 7 8 9 <10> 11 12 13 14 <15> 16 17 18 19 <20>
```

Assignment

Although its uses may not be immediately apparent, arithmetic expressions may perform assignment. We have performed assignment many times, though in a different context. Each time we give a variable a value, we are performing assignment. We can also do it within arithmetic expressions:

```
[me@linuxbox ~]$ foo=
[me@linuxbox ~]$ echo $foo

[me@linuxbox ~]$ if (( foo = 5 ));then echo "It is true."; fi
It is true.
[me@linuxbox ~]$ echo $foo
5
```

In the example above, we first assign an empty value to the variable `foo` and verify that it is indeed empty. Next, we perform an `if` with the compound command `(( foo = 5 ))`. This process does two interesting things: 1) it assigns the value of five to the variable `foo` and 2) it evaluates to true because the assignment was successful.

Note: It is important to remember the exact meaning of the `=` in the expression above. A single `=` performs assignment. `foo = 5` says "make `foo` equal to five," while `==` evaluates equivalence. `foo == 5` says "does `foo` equal five?" This can be very confusing because the `test` command accepts a single `=` for string equivalence. This is yet another reason to use the more modern `[[ ]]` and `(( ))` compound commands in place of `test`.

In addition to the `=`, the shell also provides notations that perform some very useful assignments:

Table 35-3: Assignment Operators

Notation	Description

parameter = value	Simple assignment. Assigns *value* to *parameter*.
parameter += value	Addition. Equivalent to *parameter* = *parameter* + *value*
parameter -= value	Subtraction. Equivalent to *parameter* = *parameter* - *value*
*parameter *= value*	Multiplication. Equivalent to *parameter* = *parameter* * *value*
parameter /= value	Integer division. Equivalent to *parameter* = *parameter* / *value*
parameter %= value	Modulo. Equivalent to *parameter* = *parameter* % *value*
parameter++	Variable post-increment. Equivalent to *parameter* = *parameter* + 1 (however, see discussion below)
parameter−−	Variable post-decrement. Equivalent to *parameter* = *parameter* − 1
++parameter	Variable pre-increment. Equivalent to *parameter* = *parameter* + 1
--parameter	Variable pre-decrement. Equivalent to *parameter* = *parameter* − 1

These assignment operators provide a convenient shorthand for many common arithmetic tasks. Of special interest are the increment (++) and decrement (−−) operators, which increase or decrease the value of their parameters by one. This style of notation is taken from the C programming language and has been incorporated by several other programming languages, including `bash`.

The operators may appear either at the front of a parameter or at the end. While they both either increment or decrement the parameter by one, the two placements have a subtle difference. If placed at the front of the parameter, the parameter is incremented (or decremented) before the parameter is returned. If placed after, the operation is performed *after* the parameter is returned. This is rather strange. but it is the intended behavior. Here is a demonstration:

```
[me@linuxbox ~]$ foo=1
[me@linuxbox ~]$ echo $((foo++))
1
[me@linuxbox ~]$ echo $foo
```

```
2
```

If we assign the value of one to the variable `foo` and then increment it with the `++` operator placed after the parameter name, `foo` is returned with the value of one. However, if we look at the value of the variable a second time, we see the incremented value. If we place the `++` operator in front of the parameter, we get the more expected behavior:

```
[me@linuxbox ~]$ foo=1
[me@linuxbox ~]$ echo $((++foo))
2
[me@linuxbox ~]$ echo $foo
2
```

For most shell applications, prefixing the operator will be the most useful.

The ++ and -- operators are often used in conjunction with loops. We will make some improvements to our modulo script to tighten it up a bit:

```
#!/bin/bash

# modulo2 : demonstrate the modulo operator

for ((i = 0; i <= 20; ++i )); do
      if (((i % 5) == 0 )); then
            printf "<%d> " $i
      else
            printf "%d " $i
      fi
done
printf "\n"
```

Bit Operations

One class of operators manipulates numbers in an unusual way. These operators work at the bit level. They are used for certain kinds of low level tasks, often involving setting or reading bit-flags.

Table 35-4: Bit Operators

Operator	Description

~	Bitwise negation. Negate all the bits in a number.
<<	Left bitwise shift. Shift all the bits in a number to the left.
>>	Right bitwise shift. Shift all the bits in a number to the right.
&	Bitwise AND. Perform an AND operation on all the bits in two numbers.
|	Bitwise OR. Perform an OR operation on all the bits in two numbers.
^	Bitwise XOR. Perform an exclusive OR operation on all the bits in two numbers.

Note that there are also corresponding assignment operators (for example, <<=) for all but bitwise negation.

Here we will demonstrate producing a list of powers of two, using the left bitwise shift operator:

```
[me@linuxbox ~]$ for ((i=0;i<8;++i)); do echo $((1<<i)); done
1
2
4
8
16
32
64
128
```

Logic

As we discovered in Chapter 28, the (( )) compound command supports a variety of comparison operators. There are a few more that can be used to evaluate logic. Here is the complete list:

Table 35.5: Comparison Operators

Operator	Description
<=	Less than or equal to
>=	Greater than or equal to
<	Less than

`>`	Greater than
`==`	Equal to
`!=`	Not equal to
`&&`	Logical AND
`||`	Logical OR
`expr1?expr2:expr3`	Comparison (ternary) operator. If expression *expr1* evaluates to be non-zero (arithmetic true) then *expr2*, else *expr3*.

When used for logical operations, expressions follow the rules of arithmetic logic; that is, expressions that evaluate as zero are considered false, while non-zero expressions are considered true. The `(( ))` compound command maps the results into the shell's normal exit codes:

```
[me@linuxbox ~]$ if ((1)); then echo "true"; else echo "false"; fi
true
[me@linuxbox ~]$ if ((0)); then echo "true"; else echo "false"; fi
false
```

The strangest of the logical operators is the *ternary operator*. This operator (which is modeled after the one in the C programming language) performs a standalone logical test. It can be used as a kind of if/then/else statement. It acts on three arithmetic expressions (strings won't work), and if the first expression is true (or non-zero) the second expression is performed. Otherwise, the third expression is performed. We can try this on the command line:

```
[me@linuxbox ~]$ a=0
[me@linuxbox ~]$ ((a<1?++a:--a))
[me@linuxbox ~]$ echo $a
1
[me@linuxbox ~]$ ((a<1?++a:--a))
[me@linuxbox ~]$ echo $a
0
```

Here we see a ternary operator in action. This example implements a toggle. Each time the operator is performed, the value of the variable `a` switches from zero to one or vice versa.

Please note that performing assignment within the expressions is not straightforward. When attempted, `bash` will declare an error:

```
[me@linuxbox ~]$ a=0
[me@linuxbox ~]$ ((a<1?a+=1:a-=1))
bash: ((: a<1?a+=1:a-=1: attempted assignment to non-variable (error
token is "-=1")
```

This problem can be mitigated by surrounding the assignment expression with parentheses:

```
[me@linuxbox ~]$ ((a<1?(a+=1):(a-=1)))
```

Next, we see a more complete example of using arithmetic operators in a script that produces a simple table of numbers:

```
#!/bin/bash

# arith-loop: script to demonstrate arithmetic operators

finished=0
a=0
printf "a\ta**2\ta**3\n"
printf "=\t====\t====\n"

until ((finished)); do
      b=$((a**2))
      c=$((a**3))
      printf "%d\t%d\t%d\n" $a $b $c
      ((a<10?++a:(finished=1)))
done
```

In this script, we implement an until loop based on the value of the `finished` variable. Initially, the variable is set to zero (arithmetic false) and we continue the loop until it becomes non-zero. Within the loop, we calculate the square and cube of the counter variable `a`. At the end of the loop, the value of the counter variable is evaluated. If it is less than ten (the maximum number of iterations), it is incremented by one, else the variable `finished` is given the value of one, making `finished` arithmetically true, thereby terminating the loop. Running the script gives this result:

```
[me@linuxbox ~]$ arith-loop
a     a**2  a**3
=     ====  ====
0     0     0
1     1     1
2     4     8
3     9     27
4     16    64
5     25    125
6     36    216
7     49    343
8     64    512
9     81    729
10    100   1000
```

bc – An Arbitrary Precision Calculator Language

We have seen how the shell can handle all manner of integer arithmetic, but what if we need to perform higher math or even just use floating point numbers? The answer is, we can't. At least not directly with the shell. To do this, we need to use an external program. There are several approaches we can take. Embedding Perl or AWK programs is one possible solution, but unfortunately, outside the scope of this book.

Another approach is to use a specialized calculator program. One such program found on most Linux systems is called bc.

The bc program reads a file written in its own C-like language and executes it. A bc script may be a separate file or it may be read from standard input. The bc language supports quite a few features including variables, loops, and programmer-defined functions. We won't cover bc entirely here, just enough to get a taste. bc is well-documented by its man page.

Let's start with a simple example. We'll write a bc script to add two plus two:

```
/* A very simple bc script */

2 + 2
```

The first line of the script is a comment. bc uses the same syntax for comments as the C programming language. Comments, which may span multiple lines, begin with /* and end with */.

Using bc

If we save the bc script above as `foo.bc`, we can run it this way:

```
[me@linuxbox ~]$ bc foo.bc
bc 1.06.94
Copyright 1991-1994, 1997, 1998, 2000, 2004, 2006 Free Software
Foundation, Inc.
This is free software with ABSOLUTELY NO WARRANTY.
For details type `warranty'.
4
```

If we look carefully, we can see the result at the very bottom, after the copyright message. This message can be suppressed with the `-q` (quiet) option.

bc can also be used interactively:

```
[me@linuxbox ~]$ bc -q
2 + 2
4
quit
```

When used interactively, we simply type the calculations we wish to perform, and the results are immediately displayed. The bc command `quit` ends the interactive session.

It is also possible to pass a script to bc via standard input:

```
[me@linuxbox ~]$ bc < foo.bc
4
```

The ability to take standard input means that we can use here documents, here strings, and pipes to pass scripts. This is a here string example:

```
[me@linuxbox ~]$ bc <<< "2+2"
4
```


An Example Script

As a real-world example, we will construct a script that performs a common calculation,

monthly loan payments. In the script below, we use a here document to pass a script to bc:

```
#!/bin/bash

# loan-calc : script to calculate monthly loan payments

PROGNAME=$(basename $0)

usage () {
      cat <<- EOF
      Usage: $PROGNAME PRINCIPAL INTEREST MONTHS

      Where:

      PRINCIPAL is the amount of the loan.
      INTEREST is the APR as a number (7% = 0.07).
      MONTHS is the length of the loan's term.

      EOF
}

if (($# != 3)); then
      usage
      exit 1
fi

principal=$1
interest=$2
months=$3

bc <<- EOF
      scale = 10
      i = $interest / 12
      p = $principal
      n = $months
      a = p * ((i * ((1 + i) ^ n)) / (((1 + i) ^ n) - 1))
      print a, "\n"
EOF
```

When executed, the results look like this:

```
[me@linuxbox ~]$ loan-calc 135000 0.0775 180
1270.7222490000
```

This example calculates the monthly payment for a $135,000 loan at 7.75% APR for 180

months (15 years). Notice the precision of the answer. This is determined by the value given to the special `scale` variable in the `bc` script. A full description of the `bc` scripting language is provided by the `bc` man page. While its mathematical notation is slightly different from that of the shell (`bc` more closely resembles C), most of it will be quite familiar, based on what we have learned so far.

Summing Up

In this chapter, we have learned about many of the little things that can be used to get the "real work" done in scripts. As our experience with scripting grows, the ability to effectively manipulate strings and numbers will prove extremely valuable. Our `loan-calc` script demonstrates that even simple scripts can be created to do some really useful things.

Extra Credit

While the basic functionality of the `loan-calc` script is in place, the script is far from complete. For extra credit, try improving the `loan-calc` script with the following features:

- Full verification of the command line arguments
- A command line option to implement an "interactive" mode that will prompt the user to input the principal, interest rate, and term of the loan.
- A better format for the output.

Further Reading

- The *Bash Hackers Wiki* has a good discussion of parameter expansion: http://bash-hackers.org/wiki/doku.php/syntax/pe
- The *Bash Reference Manual* covers this, too: http://www.gnu.org/software/bash/manual/bashref.html#Shell-Parameter-Expansion
- The *Wikipedia* has a good article describing bit operations: http://en.wikipedia.org/wiki/Bit_operation
- and an article on ternary operations: http://en.wikipedia.org/wiki/Ternary_operation
- as well as a description of the formula for calculating loan payments used in our `loan-calc` script: http://en.wikipedia.org/wiki/Amortization_calculator

36 – Arrays

In the last chapter, we looked at how the shell can manipulate string and numbers. The data types we have looked at so far are known in computer science circles as *scalar variables;* that is, variables that contain a single value.

In this chapter, we will look at another kind of data structure called an *array*, which holds multiple values. Arrays are a feature of virtually every programming language. The shell supports them, too, though in a rather limited fashion. Even so, they can be very useful for solving programming problems.

What Are Arrays?

Arrays are variables that hold more than one value at a time. Arrays are organized like a table. Let's consider a spreadsheet as an example. A spreadsheet acts like a *two-dimensional array*. It has both rows and columns, and an individual cell in the spreadsheet can be located according to its row and column address. An array behaves the same way. An array has cells, which are called *elements*, and each element contains data. An individual array element is accessed using an address called an *index* or *subscript*.

Most programming languages support *multi-dimensional arrays*. A spreadsheet is an example of a multi-dimensional array with two dimensions, width and height. Many languages support arrays with an arbitrary number of dimensions, though two and three dimensional arrays are probably the most commonly used.

Arrays in `bash` are limited to a single dimension. We can think of them as a spreadsheet with a single column. Even with this limitation, there are many applications for them. Array support first appeared in `bash` version 2. The original Unix shell program, `sh`, did not support arrays at all.

Creating An Array

Array variables are named just like other `bash` variables, and are created automatically when they are accessed. Here is an example:

```
[me@linuxbox ~]$ a[1]=foo
[me@linuxbox ~]$ echo ${a[1]}
foo
```

Here we see an example of both the assignment and access of an array element. With the first command, element 1 of array `a` is assigned the value "foo". The second command displays the stored value of element 1. The use of braces in the second command is required to prevent the shell from attempting pathname expansion on the name of the array element.

An array can also be created with the `declare` command:

```
[me@linuxbox ~]$ declare -a a
```

Using the `-a` option, this example of `declare` creates the array `a`.

Assigning Values To An Array

Values may be assigned in one of two ways. Single values may be assigned using the following syntax:

name[*subscript*]=*value*

where *name* is the name of the array and *subscript* is an integer (or arithmetic expression) greater than or equal to zero. Note that the first element of an array is subscript zero, not one. *value* is a string or integer assigned to the array element.

Multiple values may be assigned using the following syntax:

name=(*value1 value2* ...)

where *name* is the name of the array and *value*... are values assigned sequentially to elements of the array, starting with element zero. For example, if we wanted to assign abbreviated days of the week to the array `days`, we could do this:

```
[me@linuxbox ~]$ days=(Sun Mon Tue Wed Thu Fri Sat)
```

It is also possible to assign values to a specific element by specifying a subscript for each value:

```
[me@linuxbox ~]$ days=([0]=Sun [1]=Mon [2]=Tue [3]=Wed [4]=Thu
```

```
[5]=Fri [6]=Sat)
```

Accessing Array Elements

So what are arrays good for? Just as many data management tasks can be performed with a spreadsheet program, many programming tasks can be performed with arrays.

Let's consider a simple data gathering and presentation example. We will construct a script that examines the modification times of the files in a specified directory. From this data, our script will output a table showing at what hour of the day the files were last modified. Such a script could be used to determine when a system is most active. This script, called `hours`, produces this result:

```
[me@linuxbox ~]$ hours .
Hour  Files Hour  Files
----  ----- ----  -----
00    0     12    11
01    1     13    7
02    0     14    1
03    0     15    7
04    1     16    6
05    1     17    5
06    6     18    4
07    3     19    4
08    1     20    1
09    14    21    0
10    2     22    0
11    5     23    0

Total files = 80
```

We execute the `hours` program, specifying the current directory as the target. It produces a table showing, for each hour of the day (0-23), how many files were last modified. The code to produce this is as follows:

```
#!/bin/bash

# hours : script to count files by modification time

usage () {
      echo "usage: $(basename $0) directory" >&2
}
```

```
# Check that argument is a directory
if [[ ! -d $1 ]]; then
      usage
      exit 1
fi

# Initialize array
for i in {0..23}; do hours[i]=0; done

# Collect data
for i in $(stat -c %y "$1"/* | cut -c 12-13); do
      j=${i/#0}
      ((++hours[j]))
      ((++count))
done

# Display data
echo -e "Hour\tFiles\tHour\tFiles"
echo -e "----\t-----\t----\t-----"
for i in {0..11}; do
      j=$((i + 12))
      printf "%02d\t%d\t%02d\t%d\n" $i ${hours[i]} $j ${hours[j]}
done
printf "\nTotal files = %d\n" $count
```

The script consists of one function (`usage`) and a main body with four sections. In the first section, we check that there is a command line argument and that it is a directory. If it is not, we display the usage message and exit.

The second section initializes the array `hours`. It does this by assigning each element a value of zero. There is no special requirement to prepare arrays prior to use, but our script needs to ensure that no element is empty. Note the interesting way the loop is constructed. By employing brace expansion (`{0..23}`), we are able to easily generate a sequence of words for the `for` command.

The next section gathers the data by running the `stat` program on each file in the directory. We use `cut` to extract the two-digit hour from the result. Inside the loop, we need to remove leading zeros from the hour field, since the shell will try (and ultimately fail) to interpret values “00” through “09” as octal numbers (see Table 35-1). Next, we increment the value of the array element corresponding with the hour of the day. Finally, we increment a counter (`count`) to track the total number of files in the directory.

The last section of the script displays the contents of the array. We first output a couple of header lines and then enter a loop that produces two columns of output. Lastly, we output the final tally of files.

Array Operations

There are many common array operations. Such things as deleting arrays, determining their size, sorting, etc. have many applications in scripting.

Outputting The Entire Contents Of An Array

The subscripts * and @ can be used to access every element in an array. As with positional parameters, the @ notation is the more useful of the two. Here is a demonstration:

```
[me@linuxbox ~]$ animals=("a dog" "a cat" "a fish")
[me@linuxbox ~]$ for i in ${animals[*]}; do echo $i; done
a
dog
a
cat
a
fish
[me@linuxbox ~]$ for i in ${animals[@]}; do echo $i; done
a
dog
a
cat
a
fish
[me@linuxbox ~]$ for i in "${animals[*]}"; do echo $i; done
a dog a cat a fish
[me@linuxbox ~]$ for i in "${animals[@]}"; do echo $i; done
a dog
a cat
a fish
```

We create the array `animals` and assign it three two-word strings. We then execute four loops to see the affect of word-splitting on the array contents. The behavior of notations `${animals[*]}` and `${animals[@]}` are identical until they are quoted. The * notation results in a single word containing the array's contents, while the @ notation results in three words, which matches the arrays "real" contents.

Determining The Number Of Array Elements

Using parameter expansion, we can determine the number of elements in an array in much the same way as finding the length of a string. Here is an example:

```
[me@linuxbox ~]$ a[100]=foo
[me@linuxbox ~]$ echo ${#a[@]}  # number of array elements
1
[me@linuxbox ~]$ echo ${#a[100]}  # length of element 100
3
```

We create array `a` and assign the string "foo" to element 100. Next, we use parameter expansion to examine the length of the array, using the `@` notation. Finally, we look at the length of element 100 which contains the string "foo". It is interesting to note that while we assigned our string to element 100, bash only reports one element in the array. This differs from the behavior of some other languages in which the unused elements of the array (elements 0-99) would be initialized with empty values and counted.

Finding The Subscripts Used By An Array

As `bash` allows arrays to contain "gaps" in the assignment of subscripts, it is sometimes useful to determine which elements actually exist. This can be done with a parameter expansion using the following forms:

`${!`*`array`*`[*]}`

`${!`*`array`*`[@]}`

where *array* is the name of an array variable. Like the other expansions that use `*` and `@`, the `@` form enclosed in quotes is the most useful, as it expands into separate words:

```
[me@linuxbox ~]$ foo=([2]=a [4]=b [6]=c)
[me@linuxbox ~]$ for i in "${foo[@]}"; do echo $i; done
a
b
c
[me@linuxbox ~]$ for i in "${!foo[@]}"; do echo $i; done
2
4
6
```

Adding Elements To The End Of An Array

Knowing the number of elements in an array is no help if we need to append values to the end of an array, since the values returned by the `*` and `@` notations do not tell us the maximum array index in use. Fortunately, the shell provides us with a solution. By using the `+=` assignment operator, we can automatically append values to the end of an array. Here, we assign three values to the array `foo` and then append three more.

```
[me@linuxbox ~]$ foo=(a b c)
[me@linuxbox ~]$ echo ${foo[@]}
a b c
[me@linuxbox ~]$ foo+=(d e f)
[me@linuxbox ~]$ echo ${foo[@]}
a b c d e f
```

Sorting An Array

Just as with spreadsheets, it is often necessary to sort the values in a column of data. The shell has no direct way of doing this, but it's not hard to do with a little coding:

```
#!/bin/bash

# array-sort : Sort an array

a=(f e d c b a)

echo "Original array: ${a[@]}"
a_sorted=($(for i in "${a[@]}"; do echo $i; done | sort))
echo "Sorted array:   ${a_sorted[@]}"
```

When executed, the script produces this:

```
[me@linuxbox ~]$ array-sort
Original array: f e d c b a
Sorted array:   a b c d e f
```

The script operates by copying the contents of the original array (`a`) into a second array (`a_sorted`) with a tricky piece of command substitution. This basic technique can be used to perform many kinds of operations on the array by changing the design of the pipeline.

Deleting An Array

To delete an array, use the `unset` command:

```
[me@linuxbox ~]$ foo=(a b c d e f)
[me@linuxbox ~]$ echo ${foo[@]}
a b c d e f
```

```
[me@linuxbox ~]$ unset foo
[me@linuxbox ~]$ echo ${foo[@]}

[me@linuxbox ~]$
```

`unset` may also be used to delete single array elements:

```
[me@linuxbox ~]$ foo=(a b c d e f)
[me@linuxbox ~]$ echo ${foo[@]}
a b c d e f
[me@linuxbox ~]$ unset 'foo[2]'
[me@linuxbox ~]$ echo ${foo[@]}
a b d e f
```

In this example, we delete the third element of the array, subscript 2. Remember, arrays start with subscript zero, not one! Notice also that the array element must be quoted to prevent the shell from performing pathname expansion.

Interestingly, the assignment of an empty value to an array does not empty its contents:

```
[me@linuxbox ~]$ foo=(a b c d e f)
[me@linuxbox ~]$ foo=
[me@linuxbox ~]$ echo ${foo[@]}
b c d e f
```

Any reference to an array variable without a subscript refers to element zero of the array:

```
[me@linuxbox ~]$ foo=(a b c d e f)
[me@linuxbox ~]$ echo ${foo[@]}
a b c d e f
[me@linuxbox ~]$ foo=A
[me@linuxbox ~]$ echo ${foo[@]}
A b c d e f
```


Summing Up

If we search the `bash` man page for the word "array," we find many instances of where `bash` makes use of array variables. Most of these are rather obscure, but they may provide occasional utility in some special circumstances. In fact, the entire topic of arrays is rather under-utilized in shell programming owing largely to the fact that the

traditional Unix shell programs (such as `sh`) lacked any support for arrays. This lack of popularity is unfortunate because arrays are widely used in other programming languages and provide a powerful tool for solving many kinds of programming problems.

Arrays and loops have a natural affinity and are often used together. The

```
for ((expr; expr; expr))
```

form of loop is particularly well-suited to calculating array subscripts.

Further Reading

http://en.wikipedia.org/wiki/Scalar_(computing)

37 – Exotica

In this, the final chapter of our journey, we will look at some odds and ends. While we have certainly covered a lot of ground in the previous chapters, there are many bash features that we have not covered. Most are fairly obscure, and useful mainly to those integrating bash into a Linux distribution. However, there are a few that, while not in common use, are helpful for certain programming problems. We will cover them here.

Group Commands And Subshells

bash allows commands to be grouped together. This can be done in one of two ways; either with a *group command* or with a *subshell*. Here are examples of the syntax of each:

Group Command:

```
{ command1; command2; [command3; ...] }
```

Subshell:

```
(command1; command2; [command3;...])
```

The two forms differ in that a group command surrounds its commands with braces and a subshell uses parentheses. It is important to note that, due to the way bash implements group commands, the braces must be separated from the commands by a space and the last command must be terminated with either a semicolon or a newline prior to the closing brace.

So what are group commands and subshells good for? While they have an important difference (which we will get to in a moment), they are both used to manage redirection. Let's consider a script segment that performs redirections on multiple commands:

```
ls -l > output.txt
echo "Listing of foo.txt" >> output.txt
cat foo.txt >> output.txt
```

This is pretty straightforward. Three commands with their output redirected to a file

named `output.txt`. Using a group command, we could code this as follows:

```
{ ls -l; echo "Listing of foo.txt"; cat foo.txt; } > output.txt
```

Using a subshell is similar:

```
(ls -l; echo "Listing of foo.txt"; cat foo.txt) > output.txt
```

Using this technique, we have saved ourselves some typing, but where a group command or subshell really shines is with pipelines. When constructing a pipeline of commands, it is often useful to combine the results of several commands into a single stream. Group commands and subshells make this easy:

```
{ ls -l; echo "Listing of foo.txt"; cat foo.txt; } | lpr
```

Here we have combined the output of our three commands and piped them into the input of `lpr` to produce a printed report.

Process Substitution

While they look similar and can both be used to combine streams for redirection, there is an important difference between group commands and subshells. Whereas a group command executes all of its commands in the current shell, a subshell (as the name suggests) executes its commands in a child copy of the current shell. This means that the environment is copied and given to a new instance of the shell. When the subshell exits, the copy of the environment is lost, so any changes made to the subshell's environment (including variable assignment) is lost as well. Therefore, in most cases, unless a script requires a subshell, group commands are preferable to subshells. Group commands are both faster and require less memory.

We saw an example of the subshell environment problem in Chapter 29, when we discovered that a `read` command in a pipeline does not work as we might intuitively expect. To recap, if we construct a pipeline like this:

```
echo "foo" | read
echo $REPLY
```

The content of the REPLY variable is always empty because the read command is executed in a subshell, and its copy of REPLY is destroyed when the subshell terminates.

Because commands in pipelines are always executed in subshells, any command that assigns variables will encounter this issue. Fortunately, the shell provides an exotic form of expansion called *process substitution* that can be used to work around this problem.

Process substitution is expressed in two ways:

For processes that produce standard output:

`<(list)`

or, for processes that intake standard input:

`>(list)`

where *list* is a list of commands.

To solve our problem with read, we can employ process substitution like this:

```
read < <(echo "foo")
echo $REPLY
```

Process substitution allows us to treat the output of a subshell as an ordinary file for purposes of redirection. In fact, since it is a form of expansion, we can examine its real value:

```
[me@linuxbox ~]$ echo <(echo "foo")
/dev/fd/63
```

By using echo to view the result of the expansion, we see that the output of the subshell is being provided by a file named /dev/fd/63.

Process substitution is often used with loops containing read. Here is an example of a read loop that processes the contents of a directory listing created by a subshell:

```
#!/bin/bash

# pro-sub : demo of process substitution

while read attr links owner group size date time filename; do
      cat <<- EOF
            Filename:   $filename
```

```
            Size:       $size
            Owner:      $owner
            Group:      $group
            Modified:   $date $time
            Links:      $links
            Attributes: $attr

      EOF
done < <(ls -l | tail -n +2)
```

The loop executes `read` for each line of a directory listing. The listing itself is produced on the final line of the script. This line redirects the output of the process substitution into the standard input of the loop. The `tail` command is included in the process substitution pipeline to eliminate the first line of the listing, which is not needed.

When executed, the script produces output like this:

```
[me@linuxbox ~]$ pro_sub | head -n 20
Filename:   addresses.ldif
Size:       14540
Owner:      me
Group:      me
Modified:   2009-04-02 11:12
Links:      1
Attributes: -rw-r--r--

Filename:   bin
Size:       4096
Owner:      me
Group:      me
Modified:   2009-07-10 07:31
Links:      2
Attributes: drwxr-xr-x

Filename:   bookmarks.html
Size:       394213
Owner:      me
Group:      me
```


Traps

In Chapter 11, we saw how programs can respond to signals. We can add this capability to our scripts, too. While the scripts we have written so far have not needed this capability (owing to the fact that they have very short execution times, and do not create temporary files), larger and more complicated scripts may benefit from having a signal

handling routine.

When we design a large, complicated script, it is important to consider what happens if the user logs off or shuts down the computer while the script is running. When such an event occurs, a signal will be sent to all affected processes and, in turn, the programs representing those processes can perform actions to ensure a proper and orderly termination of the program. Let's say, for example, that we wrote a script that created a temporary file during its execution. In the course of good design, we would have the script delete the file when the script finishes its work. It would also be smart to have the script delete the file if a signal is received indicating that the program was going to be terminated prematurely.

`bash` provides a mechanism for this purpose known as a *trap*. Traps are implemented with the appropriately named builtin command, `trap`. `trap` uses the following syntax:

`trap` *`argument signal`* [*`signal...`*]

where *argument* is a string which will be read and treated as a command and *signal* is the specification of a signal that will trigger the execution of the interpreted command.

Here is a simple example:

```
#!/bin/bash

# trap-demo : simple signal handling demo

trap "echo 'I am ignoring you.'" SIGINT SIGTERM

for i in {1..5}; do
      echo "Iteration $i of 5"
      sleep 5
done
```

This script defines a trap that will execute an `echo` command each time either the SIGINT or SIGTERM signal is received while the script is running. Execution of the program looks like this when the user attempts to stop the script by typing `Ctrl-c`:

```
[me@linuxbox ~]$ trap-demo
Iteration 1 of 5
Iteration 2 of 5
I am ignoring you.
Iteration 3 of 5
I am ignoring you.
Iteration 4 of 5
Iteration 5 of 5
```

As we can see, each time the user attempts to interrupt the program, the message is printed instead.

Constructing a string to form a useful sequence of commands can be awkward, so it is common practice to specify a shell function as the command. In this example, a separate shell function is specified for each signal to be handled:

```
#!/bin/bash

# trap-demo2 : simple signal handling demo

exit_on_signal_SIGINT () {
      echo "Script interrupted." 2>&1
      exit 0
}

exit_on_signal_SIGTERM () {
      echo "Script terminated." 2>&1
      exit 0
}

trap exit_on_signal_SIGINT SIGINT
trap exit_on_signal_SIGTERM SIGTERM

for i in {1..5}; do
      echo "Iteration $i of 5"
      sleep 5
done
```

This script features two `trap` commands, one for each signal. Each trap, in turn, specifies a shell function to be executed when the particular signal is received. Note the inclusion of an `exit` command in each of the signal handling functions. Without an `exit`, the script would continue after completing the function.

When the user presses `Ctrl-C` during the execution of this script, the results look like this:

```
[me@linuxbox ~]$ trap-demo2
Iteration 1 of 5
Iteration 2 of 5
Script interrupted.
```

Temporary Files

One reason signal handlers are included in scripts is to remove temporary files that the script may create to hold intermediate results during execution. There is something of an art to naming temporary files. Traditionally, programs on Unix-like systems create their temporary files in the `/tmp` directory, a shared directory intended for such files. However, since the directory is shared, this poses certain security concerns, particularly for programs running with superuser privileges. Aside from the obvious step of setting proper permissions for files exposed to all users of the system, it is important to give temporary files non-predictable filenames. This avoids an exploit known as a *temp race attack*. One way to create a non-predictable (but still descriptive) name is to do something like this:

```
tempfile=/tmp/$(basename $0).$$.$RANDOM
```

This will create a filename consisting of the program's name, followed by its process ID (PID), followed by a random integer. Note, however, that the `$RANDOM` shell variable only returns a value in the range of 1-32767, which is not a very large range in computer terms, so a single instance of the variable is not sufficient to overcome a determined attacker.

A better way is to use the `mktemp` program (not to be confused with the `mktemp` standard library function) to both name and create the temporary file. The `mktemp` program accepts a template as an argument that is used to build the filename. The template should include a series of "X" characters which are replaced by a corresponding number of random letters and numbers. The longer the series of "X" characters, the longer the series of random characters. Here is an example:

```
tempfile=$(mktemp /tmp/foobar.$$.XXXXXXXXXX)
```

This creates a temporary file and assigns its name to the variable `tempfile`. The "X" characters in the template are replaced with random letters and numbers so that the final filename (which, in this example, also includes the expanded value of the special parameter `$$` to obtain the PID) might be something like:

```
/tmp/foobar.6593.UOZuvM6654
```

While the `mktemp` man page states that `mktemp` makes a temporary filename, `mktemp` also creates the file as well.

For scripts that are executed by regular users, it may be wise to avoid the use of the `/tmp` directory and create a directory for temporary files within the user's home directory, with a line of code such as this:

```
[[ -d $HOME/tmp ]] || mkdir $HOME/tmp
```

Asynchronous Execution

It is sometimes desirable to perform more than one task at the same time. We have seen how all modern operating systems are at least multitasking if not multi-user as well. Scripts can be constructed to behave in a multitasking fashion.

Usually this involves launching a script that, in turn, launches one or more child scripts that perform an additional task while the parent script continues to run. However, when a series of scripts runs this way, there can be problems keeping the parent and child coordinated. That is, what if the parent or child is dependent on the other, and one script must wait for the other to finish its task before finishing its own?

`bash` has a builtin command to help manage *asynchronous execution* such as this. The `wait` command causes a parent script to pause until a specified process (i.e., the child script) finishes.

`wait`

We will demonstrate the `wait` command first. To do this, we will need two scripts, a parent script:

```
#!/bin/bash

# async-parent : Asynchronous execution demo (parent)

echo "Parent: starting..."

echo "Parent: launching child script..."
async-child &
pid=$!
echo "Parent: child (PID= $pid) launched."

echo "Parent: continuing..."
sleep 2

echo "Parent: pausing to wait for child to finish..."
wait $pid

echo "Parent: child is finished.  Continuing..."
```

```
echo "Parent: parent is done.  Exiting."
```

and a child script:

```
#!/bin/bash

# async-child : Asynchronous execution demo (child)

echo "Child: child is running..."
sleep 5
echo "Child: child is done.  Exiting."
```

In this example, we see that the child script is very simple. The real action is being performed by the parent. In the parent script, the child script is launched and put into the background. The process ID of the child script is recorded by assigning the `pid` variable with the value of the `$!` shell parameter, which will always contain the process ID of the last job put into the background.

The parent script continues and then executes a `wait` command with the PID of the child process. This causes the parent script to pause until the child script exits, at which point the parent script concludes.

When executed, the parent and child scripts produce the following output:

```
[me@linuxbox ~]$ async-parent
Parent: starting...
Parent: launching child script...
Parent: child (PID= 6741) launched.
Parent: continuing...
Child: child is running...
Parent: pausing to wait for child to finish...
Child: child is done.  Exiting.
Parent: child is finished.  Continuing...
Parent: parent is done.  Exiting.
```


Named Pipes

In most Unix-like systems, it is possible to create a special type of file called a *named pipe*. Named pipes are used to create a connection between two processes and can be used just like other types of files. They are not that popular, but they're good to know about.

There is a common programming architecture called *client-server*, which can make use of a communication method such as named pipes, as well as other kinds of *interprocess communication* such as network connections.

The most widely used type of client-server system is, of course, is a web browser communicating with a web server. The web browser acts as the client, making requests to the server and the server responds to the browser with web pages.

Named pipes behave like files, but actually form first-in first-out (FIFO) buffers. As with ordinary (unnamed) pipes, data goes in one end and emerges out the other. With named pipes, it is possible to set up something like this:

process1 > named_pipe

and

process2 < named_pipe

and it will behave as if:

process1 | process2

Setting Up A Named Pipe

First, we must create a named pipe. This is done using the `mkfifo` command:

```
[me@linuxbox ~]$ mkfifo pipe1
[me@linuxbox ~]$ ls -l pipe1
prw-r--r-- 1 me   me    0 2009-07-17 06:41 pipe1
```

Here we use `mkfifo` to create a named pipe called `pipe1`. Using `ls`, we examine the file and see that the first letter in the attributes field is “p”, indicating that it is a named pipe.

Using Named Pipes

To demonstrate how the named pipe works, we will need two terminal windows (or alternately, two virtual consoles). In the first terminal, we enter a simple command and redirect its output to the named pipe:

```
[me@linuxbox ~]$ ls -l > pipe1
```

After pressing the `Enter` key, the command will appear to hang. This is because there is

nothing receiving data from the other end of the pipe yet. When this occurs, it is said that the pipe is *blocked*. This condition will clear once we attach a process to the other end and it begins to read input from the pipe. Using the second terminal window, we enter this command:

```
[me@linuxbox ~]$ cat < pipe1
```

and the directory listing produced from the first terminal window appears in the second terminal as the output from the `cat` command. The `ls` command in the first terminal successfully completes once it is no longer blocked.

Summing Up

Well, we have completed our journey. The only thing left to do now is practice, practice, practice. Even though we covered a lot of ground in our trek, we barely scratched the surface as far as the command line goes. There are still thousands of command line programs left to be discovered and enjoyed. Start digging around in `/usr/bin` and you'll see!

Further Reading

- The "Compound Commands" section of the bash man page contains a full description of group command and subshell notations.
- The EXPANSIONS section of the bash man page contains a subsection of process substitution.
- *The Advanced Bash Scripting Guide* also has a discussion of process substitution: http://tldp.org/LDP/abs/html/process-sub.html
- *Linux Journal* has two good articles on named pipes. The first, from September 1997: http://www.linuxjournal.com/article/2156
- and the second, from March 2009: http://www.linuxjournal.com/content/using-named-pipes-fifos-bash

Index

C

D

E

F

G

H

I

J

N

O

P

Q

R

T

$