

CHAPTER 28

Introduction to Shell Programming

Shell Programming

There is much more to a shell than meets the eye. The shell can do much more than the command-line interpreter everyone is used to using. UNIX shells actually provide a powerful interpretive programming language as well.

In this chapter, we'll cover **ksh** shell programming. I chose **ksh** because most **ksh** programming techniques work with the Bourne shell as well. There is a follow-on to **ksh** programming at the end of the chapter for **csk** because the **csk** employs some different programming techniques than the **ksh**.

We'll cover the most important **ksh** programming techniques. The climax of this chapter is a fairly sophisticated shell program to remove files and place them in a directory, rather than just permanently removing files from the system with **rm**. This program employs all the shell programming techniques covered in the chapter. This shell program, called **trash**, with some minor modifications, can be run in the Bourne shell as well.

The shell is one of the most powerful and versatile features on any UNIX system. If you can't find the right command to accomplish a task, you can probably build it quite easily using a shell script.

The best way to learn shell programming is by example. There are many examples given in this chapter. Some serve no purpose other than to demonstrate the current topic. Most, however, are useful tools or parts of tools that you can easily expand and adapt into your environment. The examples provide easy-to-understand prompts and output messages. Most examples show what is needed to provide the functionality we are after. They do not do a great deal of error checking. From my experience, however, it only takes a few minutes to get a shell program to do what you want; it can take hours to handle every situation and every error condition. Therefore, these programs are not very dressed up (perhaps a sport coat versus a tuxedo). I'm giving you what you need to know to build some useful tools for your environment. I hope that you will have enough knowledge and interest by the time we get to the end of this chapter to learn and do more.

Most of the examples in this chapter were performed on a Solaris system; however, shells are nearly identical going from one system to another, so you should be able to get the programs in this chapter running on your system quickly.

Steps to Create Shell Programs

When you craft a shell program, there are some initial steps you want to perform so that you have consistency among all of your programs. The following is a list of some important concepts to employ with your shell programs:

1. **Names of shell programs and their output** - You should give your shell programs and their output a meaningful name. If you call your shell programs *script1*, *script2*, and so on, these will not have any meaning for you and other potential users of the programs. If your shell program finds files of a particular type, then you can name the program *filefind* or some other descriptive name. Do the same with the output of shell programs. Don't call the output of a shell program *output1* if there is some descriptive name you can give it such as *filefind.out*. Also avoid naming shell programs and their output names that already exist. You may use commands such as **read** and **test**, so you may create confusion and conflicts if you were to give your program and their output the same names. The first shell pro-

gram in this chapter shows today's date. If we were to name the program **date**, we would actually run the system **date** command as opposed to our shell program **date**. The reason is that in most cases, the system **date** command would be found before our shell program **date** file.

2. **Repository for shell programs** - If you plan to produce numerous shell programs, you may want to place them in a directory. You can have a directory in your home directory called **bin** or **shellprogs**, in which all programs are located. You can then update your path to include the directory name where your shell programs are located.
3. **Specify shell to execute program** - The very first line of the program specifies the shell to be used to execute the program. The path of the shell is preceeded by **#!**, which is a "magic number" that indicates that the path of the shell is about to follow. The shell programs in this chapter use the **ksh** path.
4. **Formatting of shell programs** - Do not underestimate the importance of proper formatting to enhance the readability of shell programs. You should indent areas of the program to indicate that commands are part of a group. Indenting only three or four spaces is fine for groups of commands. You may also want to set autoindent in **vi** so that the next line starts at the same point as the previous line (try **:set ai** in **vi** to see how this technique works). You can break up long lines by placing a **** (backslash) at the end of one line and continue the command on the next line.
5. **Comments** - Include detailed comments in your shell program. You can write paragraphs of comments to describe what an upcoming section of your program will accomplish. These will help you understand your program when you look at it months or years after having originally created it as well as help others understand the program. No limit exists on the number of comments you can include. Start comment lines with a **#** (pound sign).
6. **Make the shell program executable** - The **chmod** command covered in the previous chapters introducing various shells is used to make your program executable.

Following the previous list of recommendations will make creating and using shell programs more efficient. Now we need to cover the types of shell programs.

A shell program is simply a file containing a set of commands you wish to execute sequentially. The file needs to be set with execute permissions so that you can execute it just by typing the name of the script.

There are two basic forms of shell programs exist:

1. **Simple command files** - When you have a command line or a set of command lines that you use over and over, you can use one simple command to execute them all.
2. **Structured programs** - The shell provides much more than the ability to “batch” together a series of commands. It has many of the features that any higher-level programming language contains:
 - Variables for storing data
 - Decision-making controls (the **if** and **case** commands)
 - Looping abilities (the **for** and **while** loops)
 - Function calls for modularity

Given these two basic forms you can build everything from simple command replacements to much larger and more complex data manipulation and system administration tools.

ksh Programming

I have created a directory in my home directory called **shellprogs** to serve as a repository for my shell programs. In order to execute these programs without having to specify an absolute path, I have updated my path in **.profile** to include the following line:

```
export PATH=${PATH} :~shellprogs
```

This adds **/home/martyp/shellprogs** to my path assuming colon is the delimiter, which is usually the case. After updating **.profile** you can log out and log in to update your path. You can issue the command below

to confirm that your path has indeed been updated to include the directory in which you keep your shell programs:

```
martyp $ echo $PATH
/usr/bin:/usr/ucb:/etc:/home/martyp/shellprogs
martyp $
```

Let's now go to the **shellprogs** directory and type a simple command file called **today**:

```
#!/bin/ksh
# This is a simple command file to display today's date.
echo "Today's date is"
date +%x
```

Before we look at each line in the program, let's run it to see the output:

```
martyp $ today
ksh: today: cannot execute
martyp $ ls -al
total 6
drwxrwxr-x  2 martyp  staff    512 May 21 09:53 .
drwxrwx---  4 martyp  staff    512 May 21 09:25 ..
-rw-rw-r--  1 martyp  staff    100 May 21 09:54 today
martyp $ chmod +x today
martyp $ ls -al
total 6
drwxrwxr-x  2 martyp  staff    512 May 21 09:53 .
drwxrwx---  4 martyp  staff    512 May 21 09:25 ..
-rwxrwxr-x  1 martyp  staff    100 May 21 09:54 today
martyp $ today
Today's date is
05/21/99
martyp $
```

We could not execute **today** because the file was created without execute permissions, which we confirm by performing a long listing. We then add execute permission to the file with **chmod +x**, which is confirmed with the next long listing. We are then able to execute **today** and view its results.

The **umask** discussion in each of the earlier shell chapters describes the defaults for new files. Almost invariably, new files are created without execute permission; therefore, you will have to update the permissions on the file to include execute.

Let's now walk through this simple shell program and analyze each line.

```
#!/bin/ksh
```

The first line specifies that the **ksh** will be used. If you are using Bash, C shell, or any other shell, you would specify its location in this manner. Some systems have multiple versions of shells running on them. It may be that a shell has been updated since the last release and some users may want to maintain the old version of the shell. For this reason, you want to be sure that the absolute path you specify is indeed that of the shell you wish to use. Note that the **#!** must be the very first two characters in the file.

Normally, when you run a shell program, the system tries to execute commands using the same shell that you are using for your interactive command lines. If we don't include this line, someone running a shell other than the **ksh** might have unexpected results when trying to run one of our programs.

As a good practice, you should include **#!shellname** as the first line of every shell program you write.

Let's now view the next line of our program:

```
# This is a simple command file to display today's date.
```

These are comments. Everything after a **#** in a command line is considered a comment (**#!** on the first line is the one very big exception). Keep in mind my early remarks about including comments liberally. It is a great pleasure to share a well commented shell program with a friend, knowing that you have adequately documented the program with comments.

Here is the next command:

```
echo "Today's date is"
```

The **echo** command generates prompts and messages in shell programs. See the **echo** manual entry to see all the options available with **echo** for formatting your output. We commonly enclose the string to be

displayed in double quotes. In this case, we did because we needed to let the shell know that the apostrophe was part of the string and not a single quote that needs a match.

Next is the last command in our program:

```
date +%x
```

This executes the **date** command. There are indeed many options to the **date** command, some of which are not intuitive. In this case, we use one of the simplest forms of the command, which simply produces today's date.

Let's cover one more example of a command file that you may find useful. This program informs you of the present working directory and then produces a long listing of the files in the directory. The following is a listing of the shell program **myll**:

```
#!/bin/ksh
# This is a simple shell program that displays the current
# directory name before a long file listing (ll) of that
# directory.
# The script name is myll
echo "Long listing of directory:"
pwd
echo
ll -l
```

This program uses **ll**; you may need to use **ls -al**. The following is what **myll** looks like when it runs:

```
martyp $ myll
Long listing of directory:
/home/martyp/shellprogs

total 4
-rwxrwxr-x  1 martyp  staff    220 May 24 05:28 myll
-rwxrwxr-x  1 martyp  staff    100 May 21 09:54 today
martyp $
```

This name of the present working directory is **/home/martyp/shell-progs**. A long listing of the contents of this directory shows the two programs we have covered so far in this chapter.

Before we can produce more complex shell programs, we need to learn more about some of the programming features built into the shell. Let's start with shell variables.

Shell Variables

A shell variable is similar to a variable in any programming language. A variable is simply a name you give to a storage location. Unlike most languages, however, you never have to declare or initialize your variables; you just use them.

Shell variables can have just about any name that starts with a letter (uppercase or lowercase). To avoid confusion with special shell characters (like file name generation characters), keep the names simple and use just letters, numbers, and underscore (_).

To assign values to shell variables, you simply type the following:

```
name=value
```

Note that there are no spaces before and after the = character.

Here are some examples of setting shell variables from the command line. These examples work correctly:

```
$ myname=ralph
$ HerName=mary
```

This one does not work because of the space after "his":

```
$ his name=norton
his: not found
```

The shell assumes that “his” is a command and tries to execute it. The rest of the line is ignored.

This example contains an illegal character (+) in the name:

```
$ one+one=two
one+one=two: not found
```

A variable must start with a letter. A common mistake is to give a variable a name that makes perfect sense to you but does not start with a letter. The following example uses a variable that starts with a number:

```
$ 3word=hi
3word=hi: not found
```

The "3" causes a "not found" to be produced when we attempt to assign this variable.

Now that we can store values in our variables, we need to know how to use those values. The dollar sign (\$) is used to get the value of a variable. Any time the shell sees a \$ in the command line, it assumes that the characters immediately following it are a variable name. It replaces the *\$variable* with its value. Here are some simple examples using variables at the command line:

```
$ myname=ralph
$ echo myname
myname
$ echo $myname
ralph
$ echo $abc123
```

In the first **echo** command, there is no \$, so the shell ignores **myname**, and **echo** gets **myname** as an argument to be echoed. In the second **echo**, however, the shell sees the \$, looks up the value of **myname**, and puts it on the command line. Now **echo** sees **ralph** as its argument (not **myname** or **\$myname**). The final **echo** statement is similar, except that we have not given a value to **abc123** so the shell assumes that it has no value and replaces **\$abc123** with nothing. Therefore, **echo** has no arguments and echos a blank line.

There may be times when you want to concatenate variables and strings. This is very easy to do in the shell:

```
$ myname=ralph
$ echo "My name is $myname"
My name is ralph
```

There may be times when the shell can become confused if the variable name is not easily identified in the command line:

```
$ string=dobeedobee
$ echo "$stringdoo"
```

We wanted to display “dobeedobee,” but the shell thought the variable name was stringdoo, which had no value. To accomplish this we can use curly braces around the variable name to separate it from surrounding characters:

```
$ echo "${string}doo"
dobeedobeedoo
```

You can set variables in shell programs in the same way, but you might also like to do things such as save the output of a command in a variable so that we can use it later. You may want to ask users a question and read their response into a variable so that you can examine it.

Command Substitution

Command substitution allows us to save the output from a command (**stdout**) into a shell variable. To demonstrate this, let's take another look at how our "today" example can be done using command substitution.

```
#!/bin/ksh
d=`date +%x`
echo "Today's date is $d"
```

The back quotes (‘) around the **date** command tell the shell to execute **date** and place its output on the command line. The output will then be assigned to the variable **d**. We'll name this updated script **today1** and run it:

```
$ today1
Today's date is 05/24/00
```

We could also have done this task without using the variable **d**. We could have just included the **date** command in the echo string, as shown in the **today2** script shown in the following example:

```
#!/bin/ksh
echo "Today's date is `date +%x`"
```

When we run this program, we see exactly the same output as we did with **today1**:

```
$ today2
Today's date is 05/24/00
```

We'll use shell variables and command substitution extensively in some of the upcoming examples. Let's now cover reading user input.