Bourne Shell Programming in One Hour

Ben Pfaff <pfaffben@msu.edu>

1 Aug 1999

1 Introduction

Programming with the Bourne shell is similar to programming in a conventional language. If you've ever written code in C or Pascal, or even BASIC or FORTRAN, you'll recognize many common features. For instance, the shell has variables, conditional and looping constructs, functions, and more.

Shell programming is also different from conventional programming languages. For example, the shell itself doesn't provide much useful functionality; instead, most work must be done by invoking external programs. As a result, the shell has powerful features for using programs together in sequence to get work done.

This article examines the features of the POSIX shell, more commonly known as the Bourne shell. The most common Bourne shell implementation on GNU/Linux systems is `bash`, the "Bourne again shell." `bash` incorporates several extensions to the standard Bourne functionality; none of these will be explored by this article. For a POSIX-compliant Bourne shell without extensions, I recommend `ash`.

This article is by no means comprehensive. It just skims the surface of many shell features. I recommend referring to a good reference book or manpage for more details on shell programming.

2 Shell command basics

You should already know how shell commands work at a basic level. To start out, the command line you typed is divided up into words. The first word is used as the command name, which is either understood by the shell itself, or used as the name of an external program to run. In either case, the rest of the words are used as arguments to the command.

This basic description is fairly accurate, but there is a little more going on behind the scenes. The following aims to provide a brief explanation of what goes on.

2.1 Word expansion

Before the shell executes a command, it performs "word expansion," which is a kind of macro processing. Word expansion has a number of steps, named in the list below. The steps are performed in order.

1. All of the following occur at the same time in a single pass across the line.
   - Variable substitution.
   - Arithmetic expansion.
   - Tilde expansion.
   - Command substitution.
2. Field splitting.
3. Filename expansion.
4. Quote removal.

Each step is explained in more detail below.

2.1.1 Variable substitution

The shell has variables that you can set. To set a shell variable, use the syntax `name=value`. Note that there may not be whitespace on either side of the equals sign. Names of variables defined this way may contain letters, digits, and underscore and may not begin with a digit.

To reference the value of a variable, use the syntax `$name` or `${name}`. The variable reference is expanded like a macro into the command contents.

There are more powerful ways to reference a variable; see Fig. 1 on page 2 for a few of the more useful.

The shell has a number of built-in variables. See Fig. 2 on page 2 for some of the most commonly used.

`${name:-value}` If *name* is an existing variable with a nonempty value, then its value is used. Otherwise, *value* is used as a default value.

`${name:=value}` If *name* is an existing variable with a nonempty value, then its value is used. Otherwise, *value* is used as a default value and variable *name* is assigned the specified *value*.

`${name:?[message]}` If *name* is an existing variable with a nonempty value, then its value is used. Otherwise, *message* is output on standard error and the shell program stops execution. If *message* is not given then a default error message is used.

Figure 1: Useful variable references.

`$0` The name under which this shell program was invoked.

`$1 ...$9` Command-line arguments passed to the shell program, numbered from left to right.

`$*` All the command-line arguments.

`$#` The number of command-line arguments.

`$?` The exit status of the last command executed. Typically, programs return an exit status of zero on successful execution, nonzero otherwise.

`$$` The process ID number of the executing shell.

Figure 2: Commonly used built-in shell variables.

2.1.2 Arithmetic expansion

Constructions of the form `$((expression))` are treated as arithmetic expressions. First, *expression* is subjected to variable subsitution, command substitution, and quote removal. The result is treated as an arithmetic expression and evaluated. The entire construction is replaced by the value of the result.

For example:

```
$ a=1
$ a=$(($a + 1))
$ echo $a
2
```

2.1.3 Tilde expansion

'`~/`' at the beginning of a word is replaced by the value of the `HOME` variable, which is usually the currently logged-in user's home directory.

The syntax `~username/` at the beginning of a word is replaced by the specified user's home directory.

You can disable this treatment by quoting the tilde (`~`); see section 2.2 on page 3 for more information on quoting.

2.1.4 Command substitution

Sometimes you want to execute a command and use its output as an argument for another command. For instance, you might want to view detailed information on all the files with a `.c` extension under the current directory. If you know about the `xargs` command, quoting, and pipes, you could do it this way:

```
find . -name \*.c -print | xargs ls -l
```

With command substituion, invoking `xargs` isn't necessary:[1]

```
ls -l `find . -name \*.c -print`
```

In command substitution, backquotes are paired up and their contents are treated as shell commands, which are run in a subshell. The output of the command is collected and substituted for the backquotes and their contents.

[1]However, if there are many, many `.c` files under the current directory, the first form is preferable because there is a (system-dependent) limit on the maximum number of arguments that can be passed to a single command, which the first form will avoid hitting.

2.1.5 Field splitting

After the substitutions above are performed, the shell scans the substitutions' results breaks them into words at whitespace (mostly spaces and tabs). Quoting (see below) can be used to prevent this.

2.1.6 Filename expansion

After field splitting, each word that contains wildcard characters is expanded in the usual way. For instance, `*a*` is replaced by all files in the current directory that have an "a" in their name. Quoting (see below) can be used to prevent filename expansion.

2.2 Quoting

Sometimes you want to disable some of the shell word expansion mechanisms above, or you want to group what would normally be multiple space-separated words into a single "word." Quoting takes care of both of these.

Quoting can be done with single quotes (`'`) or double quotes (`"`):

- When single quotes surround text, the contents are treated as a single literal word. No changes at all are made. Single quotes cannot be included in a word surrounded by single quotes.
- When double quotes surround text, the contents are subjected to variable substitution, arithmetic substitution, and command substitution. In addition, the sequences \\`$`, \\`` ` ``, \\`"`, and \\\\ are replaced by their second character.

In addition, single characters can be quoted by preceding them with a backslash (\\).

2.3 Pipelines and redirections

Pipelines are a key shell feature. They allow the output of one program to be used as the input for another. For instance,

```
find . -print | cut -b 3- | sort
```

causes the output of `find` to be the input for `cut`, whose output in turn supplies the input for `sort`.

You can also redirect input and output to a file with the redirection operators. The most common redirections are `<`, which redirects input, and `>`, which redirects output. See Fig. 3 on page 3 for a more complete list of redirections.

`>`*`file`* Redirect output to *file*. If *file* exists then its contents are truncated.

`<`*`file`* Supply input from *file*.

`>>`*`file`* Append output to *file*.

`2>&1` Redirect error output to standard output. Usually seen in a construction like '`>/dev/null 2>&1`' which causes both regular and error output to be redirected to `/dev/null`.

Figure 3: Common types of redirection.

3 Intermediate shell programming

3.1 The first line

A shell program should begin with a line like the one below.

```
#! /bin/sh
```

This line, which must be the first one in the file, means different things to the shell and to the kernel:

- To the shell, the octothorpe (`#`) character at the beginning of the line tells it that the line is a comment, which it ignores.
- To the kernel, the special combination `#!`[2], called sharp-bang, means that the file is a special executable to be interpreted by the program whose name appears on the line.

You can pass a single command-line argument to the shell by putting it after the shell's name. Many kernels truncate the sharp-bang line after the first 32 characters[3], so don't get too fancy.

To make full use of this feature, shell programs should have their executable bit set. You can do this from the shell prompt with the command "`chmod a+x` *`filename`*" or similar.

Shell programs should never be setuid or setgid. Such programs are a security risk with most Unix kernels, including Linux.

[2]On some kernels the entire sequence `#! /` is used. For this reason, never omit the space between `!` and `/`.

[3]The Linux limit is approximately 128.

3.2 Command return values

Every command returns a value between 0 and 255. This is separate from any output produced. The shell interprets a return value of zero as success and a return value of nonzero as failure.

This return value is used by several shell constructs described below.

The character `!` can be used as a command prefix to reverse the sense of a command's result; i.e., a nonzero return value is interpreted as zero, and vice versa.

3.3 Lists

Lists of commands can be formed with the `&&` and `||` operators:

- When a pair of commands is separated by `&&`, the first command is executed. If the command is successful (returns a zero result), the second command is executed.
- When a pair of commands is separated by `||`, the first command is executed. If the command is unsuccessful (returns a zero result), the second command is executed.

The value of a list is the value of the last command executed.

3.4 Grouping commands

Commands may be grouped together using the following syntaxes:

`(`*`commands`*`...)` Executes the specified *commands* in a subshell. Commands executed in this way, such as variable assignments, won't affect the current shell.

{*`commands`*`...`} Executes *commands* under the current shell. No subshell is invoked.

3.5 Testing conditions

Besides the list operators above, conditions can be tested with the `if` command, which has the following syntax:

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

If the first *condition*, which may be any command, is successful, then the corresponding *commands* are executed. Otherwise, each *condition* on the `elif` clauses is tested in turn, and if any is successful, then its *commands* are executed. If none of the conditions is met, then the `else` clause's *commands* are executed, if any.

For example:

```
$ echo
$ if test $? = 0
> then echo 'Success!'
> else echo 'Failure!'
> fi
Success!
$ asdf
asdf: not found
$ if test $? = 0
> then echo 'Success!'
> else echo 'Failure!'
> fi
Failure!
```

3.6 Repeating an action conditionally

The `while` command is used to repeat an action as long as a condition is true. It has the following syntax:

```
while condition
do commands...
done
```

When a `while` command is executed, the *condition* is first executed. If it is successful, then the *commands* are executed, then it starts over with another test of the *condition*, and so on.

3.7 Iterating over a set of words

To repeat an action for each word in a set, use the `for` command, which has the following syntax:

```
for variable in words...
do commands...
done
```

The *commands* specified are performed for each word in *words* in the order given. The example below shows how this could be used, along with `sed`, to rename each file in the current directory whose name ends in `.x` to the same name but ending in `.y`.

```
$ ls
a.x  b.x  c.x  d
$ for d in *.x
> do mv $d `echo $d | sed -e 's/\.x$/.y/;'`
> done
$ ls
a.y  b.y  c.y  d
```

3.8 Selecting one of several alternatives

The `case` statement can be used to select one alternative from several using wildcard pattern matching. It has the following syntax:

```
    case word in
    pattern) commands...;;
    ...
    esac
```

word is compared to each *pattern* in turn. The *commands* corresponding to the first matching *pattern* are executed. Multiple patterns may be specified for a single set of commands by separating the patterns with a vertical bar (`|`).

Each *pattern* may use shell wildcards for matching. To match all patterns as a final alternative, use the generic wildcard `*`, which matches any string.

3.9 Shell functions

You can define your own shell functions using a function definition command, which has the following syntax:

```
    name () {
    commands...
    }
```

After defining a function, it may be executed like any other command. Arguments are passed to the function in the built-in variables `$0` ... `$9`. Commands inside functions have the same syntax as those outside.

4 Built-in shell commands

The commands described below are built into the shell. This list is not comprehensive, but it describes the commands that are most important for shell programming.

4.1 :

This command does nothing and returns a value of zero. It is used as a placeholder.

4.2 `cd` *`directory`*

Changes the current working directory to *directory*.

4.3 `exec` *`program arguments`*...

Replaces the shell by the *program* (which must not be built-in), passing it the given *arguments*. *program* replaces the shell rather than running as a subprocess; control will never return to this shell.

4.4 `exit` *`value`*

Exits the shell, returning the specified *value* to the program that invoked it. `exit 0` is often the last line of a shell script. If a shell program doesn't end with an explicit `exit` command, it returns the value returned by the last command that it executed.

4.5 `export` *`names`*...

By default, shell variables are limited to the current shell. But when `export` is applied to a variable, it is passed in the environment to programs that are executed by the shell, including subshells.

4.6 `getopts` *`optstring name`*

Can be used to parse command-line arguments to a shell script. Refer to a shell reference manual for details.

4.7 `read [ -p` *`prompt`* `]` *`variables`*...

prompt is printed if given. Then a line is read from the shell's input. The line is split into words, and the words are assigned to the specified *variables* from left to right. If there are more words than variables, then all the remaining words, along with the whitespace that separates them, is assigned to the last variable in *variables*.

4.8 `set`

The `set` command can be used to modify the shell's execution options and set the values of the numeric variables `$1` ... `$9`. See a shell reference manual for details.

4.9 `shift`

Shifts the shell's built-in numeric variables to the left; i.e., `$2` becomes `$1`, `$3` becomes `$2`, and so on. The value of `$#` is decremented. If there are no (remaining) numeric variables, nothing happens.

5 Useful external commands

Most of what goes on in a shell program is actually performed by external programs. Some of the most important are listed below, along with their primary purposes. To achieve proficiency in shell programming you should learn to use each of these. Unfortunately, describing what each of them do in detail is far beyond the scope of this article.

Most shells implement at least some of the programs listed below as internal features.

5.1 Shell utilities

These programs are specifically for the use of shell programs.

basename Extracts the last component of a filename.

dirname Extracts the directory part of a filename.

echo Writes its command-line arguments on standard output, separated by spaces.

expr Performs mathematical operations.

false Always returns unsuccessfully.

printf Provided formatted output.

pwd Displays the current working directory.

sleep Waits for a specified number of seconds.

test Tests for the existence of files and other file properties.

true Always returns successfully.

yes Repeatedly writes a string to standard output.

[An alias for the `test` command.

5.2 Text utilities

These programs are for manipulation of text files.

awk Programming language for text manipulation.

cat Writes files to standard output.

cut Outputs selected columns of a file.

diff Compare text files.

grep Searches files for patterns.

head Outputs the first part of a file.

patch Applies patches produced by `diff`.

sed Stream EDitor for text manipulation.

sort Sorts lines of text based on specified fields.

tail Outputs the last part of a file.

tr Translates characters.

uniq Removes duplicate lines of text.

wc Counts words.

5.3 File utilities

These programs operate on files.

chgrp Changes the group associated with a file.

chmod Changes a file's permissions.

chown Changes the owner of a file.

du Calculates disk storage used by a file.

cp Copies files.

find Finds files having specified attributes.

ln Creates links to a file.

ls Lists files in a directory.

mkdir Creates a directory.

mv Moves or renames files.

rm Deletes files.

rmdir Deletes directories.

touch Updates file timestamps.