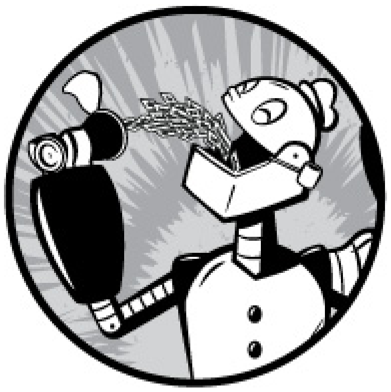
33

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 popular construct in `bash` scripting.

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

for: Traditional Shell Form

The original `for` command's syntax is as follows:

```
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, we can do it through brace expansion, like so:

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

or we could use a pathname expansion, as follows:

```
[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
```

Pathname expansion provides a nice, clean list of pathnames that can be processed in the loop. The one precaution needed is to check that the expansion actually matched something. By default, if the expansion fails to match any files, the wildcards themselves (`distros*.txt` in the preceding example) will be returned. To guard against this, we would code the preceding example in a script this way:

```
for i in distros*.txt; do
    if [[ -e "$i" ]]; then
        echo "$i"
    fi
done
```

By adding a test for file existence, we will ignore a failed expansion.

Another common method of word production is 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 -n "$i" | wc -c)"
            if (( len > max_len )); then
                max_len="$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 filenames 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 whether the current word is the longest found so far. When the loop concludes, the longest word is displayed.

One thing to note here is that, contrary to our usual practice, we do not surround the command substitution `$(strings "$1")` with double quotes. This is because we actually want word splitting to occur to give us our list. If we had surrounded the command substitution with quotes, it would produce only a single word containing every string in the file. That's not exactly what we are looking for.

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="$(echo -n "$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 previous `for` loop examples. 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 reals (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 the `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
```

Here, *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 5, and *expression3* increments the value of `i` by 1 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.

Support Sign Out