29 – 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 executes only 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 chapter.

Looping

Daily life is full of repeated activities. Going to work each day, walking the dog, and 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 1 to 5. 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 as follows:

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 previous script, the variable count is created and assigned an initial value of 1. The while command evaluates the exit status of the [[]] compound command. As long as the [[]] command returns an exit status of zero, the commands within the loop are executed. At the end of each cycle, the [[]] command is repeated. After five iterations of the loop, the value of count has increased to 6, the [[]] 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 of 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 an 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 5. 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