Flow Control: Looping With while / until & for

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.

```
#!/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 -r -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.

break and continue

- 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.

select

- This would be a good time to mention the select shell builtin which is used to create looping menus. It has a syntax that looks like this:

```
select var in [string.. ;] do commands; done
```

- where var is a variable and string is the text of a menu choice.
- When select executes, it displays the string(s) followed by the contents of the PS3 (prompt string 3) variable as a prompt for the user's input. Once a choice is made, it sets the REPLY variable with the user's input (just like with read) and returns the string associated with the choice in the variable var. Once the values are set, commands are performed and the prompt is displayed again for another choice.

- This sounds a little confusing but we can demonstrate it with this tiny script:

```
#!/bin/bash

# select-demo: select builtin demo

PS3="
Your choice: "

select my_choice in First Second Third Fourth Quit; do
    echo "REPLY= $REPLY  my_choice= $my_choice"
    [[ "$my_choice" == "Quit" ]] && break
done
```

- First we set the contents of the PS3 variable with our desired prompt string.
- Next we execute select. In this example we have five strings and though we have used single words as our strings, we can use any kind of quoted text.
- For our commands, we simply echo the assignments made by select. We also test the contents of our variable my_choice to see if the user has chosen the “Quit” option and if so, we perform a break to exit the loop.

```
[me@linuxbox ~]$ select-demo
1) First
2) Second
3) Third
4) Fourth
5) Quit

Your choice:
```

- When select first executes it displays each of our strings preceded by a number followed by our prompt string. The user next enters the number representing the desired choice. The select command then sets the REPLY variable to contain whatever the user entered and the corresponding string if any.

```
Your choice: 1
REPLY= 1  my_choice= First

Your choice: 2
REPLY= 2  my_choice= Second
```

- Here we see the user entered “1” and the echo command displays the values of the RE- PLY and my_choice variables. select will repeat displaying the prompt string until the user enters a “5”.

- If the user enters an invalid value, select sets my_choice to an empty string. If the user simply types Enter, this will cause select to start over and redisplay the list of menu choices.

```
Your choice: 6
REPLY= 6  my_choice=

Your choice: abc
REPLY= abc  my_choice=
```

- The select loop will continue indefinitely until either a break command is encountered or the user types ctrl-d to signal end-of-file.

```
Your choice: 5
REPLY= 5  my_choice= Quit
[me@linuxbox ~]$
```

- One interesting feature of select is that it does not display its menu choices or prompt string on standard output, rather it uses standard error. This is actually handy because it allows the real work done by the commands within the loop to be redirected, for example:

```
[me@linuxbox ~]$ select-demo > choices.txt
```

- When we do this redirection the menu and prompt are still displayed but the output of the echo command is redirected.
- Let’s make an alternate version of our system information script replacing our previous while loop with select.

```
#!/bin/bash

# select-menu: a menu driven system information program

DELAY=3 # Number of seconds to display results
PS3="
Enter selection [1-4] > "

select str in \
    "Display System Information" \
    "Display Disk Space" \
    "Display Home Space Utilization" \
    "Quit"; do
    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 [[ $(id -u) -eq 0 ]]; then
            echo "Home Space Utilization (All Users)"
            du -sh /home/* 2> /dev/null
        else
            echo "Home Space Utilization ($USER)"
            du -sh "$HOME" 2> /dev/null
        fi
        sleep "$DELAY"
        continue
    fi
    if [[ "$REPLY" == "4" ]]; then
        break
    fi
     if [[ -z "$str" ]]; then
         echo "Invalid entry."
         sleep "$DELAY"
     fi
done
echo "Program terminated."
```

- In our alternate script, we set the PS3 variable and then invoke select with four strings. Though we could subsequently test the str variable set by select, it’s easier to test the REPLY variable and act accordingly. At the end of the loop we check if the str variable has a zero length indicating an invalid value.
- So which method should we use for constructing a menu? The select command is interesting, but besides its use of standard error for the menu display, it doesn't really save us much, if any, coding effort and it sharply limits the visual design of the menu display.

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 -r 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 -r 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.

Flow Control: Looping with for

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 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 example above) will be returned.

- To guard against this, we would code the example above 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.

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
```

- 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.