

## Lecture 6

### Iteration

#### Overview of Iteration

Conditionals give your programs the ability to make decisions; iteration gives your programs the ability to perform actions and decisions over and over again. Iteration is a fancy combination of a conditional and an invisible **goto** statement. Iteration constructs test a condition, run some commands, and then go back to the top to test the condition all over again. In the shell, there are three forms of iteration: a **for** loop, a **while** loop, and an **until** loop. Each of these is described well enough in the textbook. I've added some more supplementary material on each below. It might be best to read these notes and the textbook side by side, comparing the examples and information from each.

#### The for Loop

The **for** loop is used to loop repeatedly, each time assigning a value to a variable, the value taken from a list you specify. The syntax of a **for** loop is:

```
for variable in word1 word2 ... wordn
do
    commands
done
```

As with the **if** statement, it is common practice to place the uninteresting **do** keyword on the same line as the **for** after the last word *word<sub>n</sub>*. Here is the same loop syntax written in the alternate form:

```
for variable in word1 word2 ... wordn ; do
    commands
done
```

If this is your first exposure to programming and loops, you may find it helpful to think about **for** loops in a simpler way. Consider the code below. Notice that the same **echo** statement is run three times, and the variable **num** is set to **A**, **B**, and then **C** before the **echo** statement is run.

```
num=A
echo The value num is $num
num=B
echo The value num is $num
num=C
echo The value num is $num
```

Since essentially the exact same code is repeated three times, the only difference being that the value of **num** is changed each time, it would be nice if this could be written such that the duplication was removed. Also, what would you do if you could not predetermine how many times you wanted to assign and print out **num**? The code above only works for a fixed number of assignments to **num**. The **for** loop solves these problems. Here is the above code written using a simple **for** loop:

```
for num in A B C ; do
    echo The value num is $num
done
```

This loop iterates (*spins*) three times, each time with the variable `num` taking one of the values `A`, `B`, and then `C`. Clearly the code using the loop is not only more succinct, it is clearer and less prone to programming errors. And most importantly, it allows creation of loops that can iterate *any* number of times. Here is a **for** loop that iterates some unknown number of times, based on how many words are in the file `namesfile.txt`:

```
for name in `cat namesfile.txt` ; do
    echo Name is $name
done
```

The shell first performs the *command substitution* on ``cat namesfile.txt`` and obtains the output, creating a list of words much like our previously hard-coded `A B C`. The **for** loop then iterates over each of these names, performing the commands in the *body* of the **for** loop (in this case, another trusty **echo** statement). There is no way to hard-code the equivalent code without using a loop, since we do not know ahead of time how many words are in the file `namesfile.txt`.

It is important to understand that the **for** statement wants a *list of words* - it does *not* use an **exit status** like either the **if** or **while** statements (the **while** statement will be discussed shortly). If you need to obtain this list of words at run-time (meaning, while your program is running), you need to use command substitution to obtain this list, as the example above demonstrated.

Inexperienced shell programmers often use the form of a **for** loop below to iterate over all the files in the current directory:

```
for currentFile in `ls` ; do
    echo The current file is $currentFile
done
```

While this works perfectly, it is more expensive on the system - it requires two additional processes to run (a sub-shell and `ls` itself). You already know there is an easier way to get all the files in the current directory - use the `*` wildcard. A much better and cheaper form of the loop is:

```
for currentFile in * ; do
    echo The current file is $currentFile
done
```

While we are on the topic of iterating over the files in the current directory using a **for** loop, I'll point out a very common mistake made by beginners. Examine the **for** loop below:

```
for currentFile in `ls -l` ; do
    echo The current file is $currentFile
done
```

Can you spot the problem? If not, think real hard about what the `ls -l` command will actually output. Hint: it does not output a simple list of files! No, instead there is a whole bunch of other stuff in the output. The output will look something like this:

```
total 2
```

```
-rw----- 1 cappella staff      5 Feb 10 17:17 firstfile
-rw----- 1 cappella staff    414 Feb 10 17:17 afile
-rw----- 1 cappella staff      0 Feb 10 17:17 lastfile
```

The *list of words* in the **for** loop above would actually iterate 29 times! And the variable `currentFile` would take on every single word in all of the output of the `ls -l` command, including "total", "2", "-rw-----", "1", "cappella", "staff", ..., "lastfile"! Remember, *command substitution* replaces newlines with spaces to make a single line of text - and each of **for**'s *list of words* are separated by white space. Count the words in the output above - if you count like I do, you'll find there are 29 of them. Make sure you understand this, or the **for** loop will at times frustrate you.

The textbook shows an alternate form of the **for** loop that omits both the keyword **in** and the word list:

```
for arg ; do
    echo arg is: $arg
done
```

This form of the **for** loop iterates over all the arguments passed on the command line; there is an implicit **in "\$@"** after the loop variable. While you and Perl programmers may enjoy this shortcut, personally I find these language idiosyncrasies a bit gratuitous and basically pointless. I suppose the savings of not having to type seven more characters (the missing **in "\$@"**) really makes programmers go wild<sup>1</sup>. Let them live their dream; I'd recommend avoiding the shortcut, just to be clearer.

## The while Loop

The **while** statement is like a **for** statement in that it also loops, but this is where the similarity ends. There are several fundamental differences. First, there is no loop variable assigned a value each time through the loop. Second, the **while** loop does not iterate over a *list of words*; it iterates as long as some *condition* remains **true**. This form of looping is used to perform some commands *while* some testing command *command<sub>t</sub>* returns a 0 exit status. This is like mom telling you: "Stay in your room until I tell you to come out!" And every minute you ask: "Can I come out now?" And every minute mom says: "In five more minutes!"<sup>2</sup> As far as the terminating condition, it is very similar to the **if** statement, which only performs some action *conditional upon* some command *command<sub>t</sub>* returning a 0 exit status. Likewise, the **while** loop only iterates while the exit status of the conditional command *command<sub>t</sub>* returns a 0 exit status.

```
while commandt
do
    commands
done
```

As with the **if** statement, it is common practice to place the uninteresting **do** keyword on the same line as the **while** after the *command<sub>t</sub>*. Here is the same **while** loop written in the alternate form:

```
while commandt ; do
    commands
done
```

<sup>1</sup> Don't miss out on their fun; compare the two versions of the **args** programs at the top of page 194 and the middle of page 195.

<sup>2</sup> This was probably your first encounter with what is known as an infinite loop!

Beginners commonly confuse what type of *thingy* goes with a **for** and what goes with a **while**. After you use these for a while (no pun intended), it seems obvious - but while (ditto) you are learning, remember that **for** wants a *list of words* and **while** wants a *condition* command.

## The until Loop

The **until** statement is exactly like the **while** statement, except that the condition is the negated. Remember mom telling you to stay in your room? The **until** statement is like you continuing to ask: "*Can I come out now?*" until mom finally says: "*Ok!*" Your mom's approval is your felt-like-an-eternity terminating condition - she finally gives you a **true** exit status. The **until** loop iterates until some conditional test returns a **true** exit status, just the opposite of the **while** loop. Use either form:

```
until commandi
do
    commands
done
```

or

```
until commandi ; do
    commands
done
```

## Various Forms?

You have seen two forms of **for**, **while**, and **until**. Why are there two forms of these loops, one with the **do** keyword on a separate line, and the other on the same line following a semi-colon. Truth be told, there really is only one form of each, and they look like:

```
for variable in word_list CS do command_list CS done
```

```
while commandi CS do command_list CS done
```

```
until commandi CS do command_list CS done
```

The only difference between the two forms, and the actual syntax above, is the choice of using either a semicolon or a newline for the command separator *CS* above. Recall that the shell uses either a semicolon or a newline as a command separator. Use a newline for the first *CS* above and you have the first form with the **do** on a separate line. Use a semicolon instead, and you have the second form. Use a semicolon for both *CS* command separators above, and you can place the entire loop on a single line! Notice that there is no command separator after the **do**! Why? Because the shell *knows* that after the *CS* following the *word\_list* or the *command<sub>i</sub>*, the keyword **do** *must* follow, and after that *must* be the *command\_list*. But it cannot know when either the *word\_list* or *command<sub>i</sub>* ends until you tell it<sup>3</sup>.

## Choose Your Loop

Selecting which loop construct to use depends on the circumstances, and requires some practice. Like in flying where any landing you walk away from is a good landing, any loop construct that gets the job done is a good loop. In fact, often any of the three types of loops can be used to get the job done. For example, you

<sup>3</sup> The shell actually could do better at finding the keyword **do**, but this creates all sorts of other quoting, parsing and problems. It is best in this case to have the programmer place the **do** where it belongs.

can iterate over the command line arguments with a **for** loop, a **while** loop, or an **until** loop. Here are all three versions:

```
for arg in "$@" ; do
    echo Arg is $arg
done

while [ $# -ne 0 ] ; do
    echo Arg is $1
    shift
done

until [ $# -eq 0 ] ; do
    echo Arg is $1
    shift
done
```

The **for** loop above preserves the `$@`, `$*`, and `$#` while both the **while** and **until** loops shift and essentially destroy them. But argument list preservation is not the primary criteria for selecting a type of loop. Another much more important criterion is *speed* and *expense* on the system as a whole. Before the **test** command was a shell built-in command, the **while** and **until** loops were more expensive than **for** loops because they required a new **test** process to be created for each iteration. If the loop iterated 100 times, that caused 100 extra processes to be created! If the loop conditional command is not built-in, the number of extra processes run is a major factor for consideration. Now that **test** is built into the shell, the performance and system impact of using **test** in **while** and **until** loops is no longer an issue.

In many circumstances, you either have a list of items (perhaps how many is unknown at the time the program is written) or you have some condition that would terminate a loop. Select the **for** loop when you have a list of items to process, and select a **while** or **until** loop when you have a condition to test. I would choose the **for** loop above to process arguments because it is simpler, more direct, and feels more natural<sup>4</sup>. Ultimately it comes down to what *feels* correct, and this *feeling* is only gained through experience.

## The Infinite Loop

One of the most common bugs when programming loops is the dreaded *infinite loop*. This is a loop that never terminates because the terminating condition is never met, often due to a problem with the logic of the loop. The example loop below will never exit:

```
while true ; do
    echo System Error! Reboot Now!
done
```

The **true** command always returns a 0 exit status, and so this loop will just spin forever. Type the loop into your shell and try it<sup>5</sup>. Don't worry, you can interrupt and terminate the loop with `^C`.

While most infinite loops are unintentional, believe it or not, there are times when you intentionally program a seemingly infinite loop. Recall that **while** and **until** loops terminate when a *single* condition is met. But what if you wanted the loop to terminate for any number of reasons? Sometimes the clearest way to solve this problem is to use an infinite loop structure like the one above, and then **break** out of the loop when one

<sup>4</sup> But then, I didn't think much of the Internet Browser when I saw and used the first one, so what do I know?

<sup>5</sup> It was great fun as a kid when visiting the local Radio Shack to enter such an infinite loop into a TRS-80 using Basic.

or more of the terminating conditions is met. The shell provides a means to break out of a loop with the **break** statement (did you guess?). The **break** statement will cause the loop to terminate, and execution of the program will continue at the first statement that follows the loop's **done** statement.

## Loop Pipes and Redirection

One of the interesting things about loops in the shell is that you can not only redirect or pipe the output of the commands that run inside the loop, but you can also pipe or redirect the loop itself. This may seem peculiar, but it will make sense if you understand that redirected or piped loops actually run in a sub-shell (a separate process)

The shell creates another shell process just to run the loop itself. Like any other process, the sub-shell'ed loop therefore has a STDIN, STDOUT, and STDERR. This means that redirection and piping works correctly on the loop itself. For performance reasons, loops that are not piped or redirected are run in the current shell process. The two examples below help to clarify this point.

```
$ cd /tmp
$ for arg in 1 ; do
>   cd /usr/local/bin
> done
$ echo The current directory is `pwd`
The current directory is /usr/local/bin
```

Notice in the example above that the current working directory was changed outside the loop, as reported by the shell variable PWD. Now look at what happens when we redirect the loop:

```
$ cd /tmp
$ for arg in 1 ; do
$   cd /usr/local/bin
$ done > /dev/null
$ echo The current directory is `pwd`
The current directory is /tmp
```

Notice this time how even though the directory was changed within the loop, outside the loop, the directory remained the same! This is easily explained when you understand sub-shells and the environment, which is a little more advanced, and we won't worry about it just yet. Just keep this oddity in mind if you redirect or pipe a loop.

## Using vi Effectively

### Abbreviations

One of **vi**'s nice functions is its abbreviation capabilities. That is, you can have **vi** type for you. If you wanted your name to be typed whenever you typed **name**, you could use the abbreviation setting below in your **.exrc** file:

```
ab name Sam Spade
```

Whenever you type **name** in insert mode, followed by a space, **vi** will replace the word **name** with **Sam Spade**, in essence, typing for you. You may not consider this very interesting, but sometimes it is useful. What if you could type **for** and have **vi** fill out a **for** loop template for you, saving you typing, and helping you memorize the syntax of the **for** loop? Or the same for a **while** loop? Well, the following **vi**

abbreviations do exactly that. Here are two more interesting and useful abbreviations to add to your **.exrc** file:

```
ab for for VAR in LIST ; do^M^TCOMMANDS^Mdone^[<<?VAR^Mh
ab while while CMD_TEST ; do^M^TCOMMANDS^Mdone^[<<?CMD_TEST^Mh
```

The above commands are two abbreviations, that when you type either **for** or **while** followed by a space when in insert mode, **vi** will replace the word with a complete **for** or **while** loop template for you. The **^M** and **^T** and **^[]** characters are actual control characters (Control-M, Control-T, and Escape). The only way to enter these characters in **vi** is to use the universal **vi** escape character **^V** (Control-V). These abbreviations will also move your cursor to the first thing you will need to change to fill-in your loop. Just type **cw** to start the **change word** command then type the variable or testing command you want to use. The above abbreviations work in both **vi** and **vim**. You can add these to your **.exrc** or **.vimrc** quickly - just edit your **.exrc** or **.vimrc** file and enter the command:

```
:r ~cappella/files/exrc
```

This will read the two abbreviations above into your editor startup file. Make sure there are no blank lines in the **.exrc** file - **vi** has a hard time with these.

Now test out your new abbreviations. Edit a new file and get into insert mode, and type **for** followed by a space. Wow! Type **cw** and then a new variable name; hit Escape to complete. Now go to the end of the file and type **while** and then a space. Enter **cw** again and type in your testing command, hitting Escape to finish.