Advanced Unix Tools	**Lecture 4**
CS214 Spring 2004	**Monday, March 1, 2004**

While loops

While loops allow you to execute commands until essentially a given condition is met. The basic form of the statement is:

```
while cmd1
do
  cmd
  cmd
done
```

The interpretation is as follows. First, *cmd1* is executed. If it succeeds, the commands in the body of the loop (between `do` and `done`) are executed. Then, *cmd1* is executed again, and if it succeeds, the body of the loop is executed again, and so on until *cmd1* fails.

An alternate form allows you to loop until a given condition is met:

```
until cmd1
do
  cmd
  cmd
done
```

The interpretation is as in the `while` case, except that the body is executed as long as *cmd1* fails.

Here's an example of testing in loops. It also serves to introduce an interesting builtin `bash` command. The command `read` is used to read input from the user, something that may be useful in a shell script. The command `read` *`var`* will read a line of input from the user, and put the line in variable *`var`*. If more than one variable is supplied, i.e., `read` *`var1 var2 var3`*, then the first word of the line input from the user is put in *`var1`*, the second in *`var2`*, and the rest of the line is put in *`var3`*. A prompt may be supplied by using a `-p` option, as in `read -p 'some test' var` (notice the quotes to give a prompt which may contain spaces...) The following lines will repeatedly query the user for a `yes` or `no` until he gets it right:

```
read -p "yes or no? " answer
```

```
while [ "$answer" != "yes" -a "$answer" != "no" ]
do
  echo "Please enter yes or no"
  read -p "yes or no? " answer
done
echo "the answer was $answer"
```

For loops

A final form of looping is available, that does not rely on exit codes. The `for` statement has the form:

```
for var in word1 word2 ... wordn
do
  cmd
  cmd
done
```

The interpretation is as follows. The variable *`var`* is assigned the first token *`word1`* in the list, and the commands between `do` and `done` are executed. Then, the variable *`var`* gets the second token in the list, and the commands are executed. So on until all the tokens in the list have been processed. Note that I'm talking about tokens, that is, what you get after the shell performs word splitting.[1]

Consider a simple example, to mail a file `letter` to a list of people:

```
for name in alice bob trudy
do
  mail $name < letter
  echo "letter mailed to $name"
done
```

Of course, the list of words can be generated by substitution, as in:

```
names="alice bob"
junk="abc"
morenames="oscar trudy"
for name in $names $junk $morenames
```

[1]Recall that you can disable word splitting for a particular string by putting it in quotes. Thus, the following are different: `for name in alice bob trudy` and `for name in 'alice bob' trudy`. What's the difference?

```
do
  echo -n "Processing $name: "
  if finger $name@cornell.edu | grep 'no matches' > /dev/null
  then
    echo "match not found"
  else
    echo "match found"
  fi
done
```

Some things to notice in the above example. First, after substitution in the `for` loop, the list of words contains five words. Second, the option `-n` to `echo` suppresses the newline that gets added at the end of what's printed. Third, recall that `grep` returns an exit code of 0 if a match is found, and an error code of 1 if no match is found (and no error occurred). Finally, the redirection to `/dev/null` is used to suppress output by `grep`, which by defaults sends the matching lines to `stdout`. (`/dev/null` is the so-called Unix bit-bucket; it is a black holes that just swallows input.)

Break and Continue

A few more notes about loops, as we saw them last time. Two builtin are available to somewhat control a loop. The command *`break`* gets you out of the current loop. If it is used with a parameter, *`break n`* breaks you out of the *`n`*-th enclosing loop. If *`n`* is greater than the number of enclosing loops, you break out of all the loops. Alternatively, the command *`continue`* aborts the current iteration of the current loop, and attempts the next iteration. Again, if it is used with a parameter, *`continue n`* attempts the next iteration of the *`n`*-th enclosing loop. If *`n`* is greater than the number of enclosing loops, the last enclosing loop is resumed.