

Lecture 8

User Input, Temporary Files, and Functions

Reading Data

The primary mechanism for reading user input in the Bourne shell is the **read** command. A simple command, its sole purpose is to read text, and to assign that text to one or more variables. The **read** command is another shell built-in, and it reads its data from STDIN. It is most often used to prompt a user for input, or to read one or more values from a file. The syntax for the **read** command is as follows:

```
read var [ ... ]
```

where *var* is a variable that you name. There can be more than one variable, each uniquely named. When **read** is executed by the shell, it will wait for a line of user input, and then assign words from the line of text to the variable(s) that follows the keyword **read**. One word is assigned per variable, and any extra words are assigned to the last variable.

It is easy to use **read** to drive a loop. The command sequence below reads a user's input until the user enters the EOF (Control-D) character:

```
while read input ; do
    echo Input is $input
done
```

Each line of text is assigned to the variable *input*. When **read** reads an EOF character (user enters Control-D, or an actual end of file if STDIN is redirected to come from a file), it returns a non-zero exit status, forcing the loop above to terminate.

Recall that input to or output from a loop can be redirected. It is easy to write a simple loop to read some data from a file. This is useful in a shell program, for example, for retrieving previously stored data, such as per-user configuration data, or saved preferences. The while loop below demonstrates this:

```
while read x y ; do
    echo Coordinates are $x $y
done < coords
```

This loop reads one line at a time, until the end of the file. Each line is presumed to contain two values, which are assigned to *x* and to *y* respectively. If there are more than two values per line, then the last variable *y* will obtain all the remaining values that did not *fit*. Fewer values than variables will leave some variables empty.

You may be tempted to perform the loop above instead using **cat** and a pipe, as in:

```
cat coords |
while read x y ; do
    echo Coordinates are $x $y
done
```

This works fine, but as mentioned in a previous discussion, it is more expensive to run. Redirection using the shell is preferable because the shell does not need to start up an additional process (**cat**) and requires fewer system resources. Using **cat** in this fashion betrays one's inexperience as a shell program – perhaps avoiding this form is best.

The line Command

The **line** command can also be used to read a line of input from STDIN. It is somewhat similar to **read** but is not a shell built-in, and does not assign a value to a variable – instead, it simple outputs the read line on STDOUT. It also preserves leading space in the input, which **read** does not do by default. To assign the read data to a variable, you have to use command substitution. The following command reads on a line using the **line** command and assigns the text to the variable **inputline**:

```
inputline=`line`
```

It is more efficient to use **read** since it is a built-in and does not require any extra processes to be run.

The \$\$ Variable

The shell provides another variable for you to use - **\$\$**. The shell sets **\$\$** to be the process id (PID) of the current shell. If **\$\$** is used inside a shell program, it will be the PID of the process running that shell program. This is very useful if your program needs to know its own PID and even more so for creating temporary files.

Temporary Files

If you need to create a temporary file, the best place to do so is in the **/tmp** filesystem. There are several reasons for this. First, you may not have write permission in the current directory and **/tmp** is always writable by everyone. Second, on modern systems such as Solaris, the **/tmp** filesystem is a very fast RAM-based virtual filesystem – files created there only exist in RAM; there are no slow disks involved.

Of course with every summer comes a winter. And the problem with using the **/tmp** filesystem to create temporary files is that everyone on the system can use it! If your shell program created a file **/tmp/phonebook** as your temporary phonebook file, how would you guarantee that nobody else was also creating the same file? To prevent such occurrences, you must create *uniquely* named temporary files in **/tmp**. This is where **\$\$** comes in handy. Since your process is the only process on the system with that PID, it is sufficient to use **\$\$** as part of your temporary filename, as in **/tmp/phonebook.\$\$**. This creates a file with a name such as **/tmp/phonebook.4354**. The PIDs on the system are recycled, but this takes a fairly long time, and usually for most shell programs, this is not a concern. After you are done with your temporary files, you must remove them. It is considered very bad form for shell programs to leave temporary files lying about. Make sure that you remove every temporary file you create.

Finally, if at all possible, it is best to avoid creating temporary files. Using temporary files rather than variables for holding text means that you have to be sure to clean them up. And before you dismiss this argument as trivial, consider it more carefully. Removing temporary files can be problematic for a variety of reasons (consider what happens if a user types **^C**, killing your shell program!). But if using temporary files is the only way to accomplish your task, then by all means use them as necessary, and remove them as soon as possible afterwards (this minimizes the chance that they may not get removed).

Functions

The shell language provides support for functions. Functions are like mini-programs within your program, and are useful for grouping a series of commands together that performs some particular task. The syntax of creating a function in the Bourne shell is:

```
function_name() {  
    command-list  
}
```

The name of the function follows the same naming rules as do variables. In the body of the function is a list of one or more commands. When the function is *called* in your program, all the commands inside the body of the function run. Consider a function called **usage** which outputs a usage message and then exits with an error code.

```
usage() {  
    echo Usage: $prog [-xy ] file [...]  
    exit 1  
}
```

Later in your program, you can call your **usage** function just like any other command:

```
if [ $# -eq 0 ] ; then  
    usage  
fi
```

Using functions like this is very convenient for many reasons. First, functions allow you to write code once, but use the code many times. In the **usage** example above, you might have to write the **echo** and **exit** commands several times and duplication a) makes your programs larger and b) increases the probabilities for errors. Second, functions accept arguments just like shell programs do. In fact, the arguments are positional parameters just like the program's own positional parameters. This allows creation of more generally useful code. Third, functions allow you to create more *readable* and *usable* code by grouping commands into meaningful chunks.

To illustrate a function, the program **func** below adds and subtracts two numbers, which it receives as arguments:

```
#!/bin/sh  
  
subtract() {  
    echo $1 - $2 is `expr $1 - $2`  
}  
  
add() {  
    echo $1 + $2 is `expr $1 + $2`  
}  
  
add $1 $2  
subtract $1 $2
```

The first part of the program defines the functions **add** and **subtract**. Functions must be *defined* before they can be used. The function *definition* is not executed when the shell reads the definition – it only stores it for

later. The function is only used when called. A function is called simply by using its name as a command, and it can be supplied with arguments. These arguments become the function's own positional parameters. The positional parameters within a function are not the same as those outside of the function. Thus, \$1 refers to the function's first argument, not the program's first argument. In the program above, the calls to **add** and **subtract** each pass the program's first and second positional parameters to the functions. In turn, the function has access to these arguments via \$1, \$2, etc.

Running the program yields:

```
$ func 5 3
5 + 3 is 8
5 - 3 is 2
```

As you can see, functions allowed us to create more obvious and intuitive mini-commands for use in our program. Let's modify the program to make it even more useful. Since the **add** and **subtract** functions are identical, except for the + or - operator, we can re-write the functions to be more general and hence more useful. Consider this re-write:

```
#!/bin/sh

domath() {
    echo $1 "$2" $3 is `expr $1 "$2" $3`
}

domath $1 + $2
domath $1 - $2
```

Calling the program will generate the same results as above. Notice that with just a simple change to the one of the functions (passing in the operator as a parameter), we are able to eliminate the more specific **add** and **subtract** functions, replacing them with the single, more general **domath** function. The **domath** function can perform any calculation by passing in the desired operator as the second argument to **domath**. Notice below how easy it is to perform even more math just by calling **domath** with a different operator:

```
#!/bin/sh

domath() {
    echo $1 "$2" $3 is `expr $1 "$2" $3`
}

domath $1 + $2
domath $1 - $2
domath $1 / $2
domath $1 \* $2
```

You can see the results below of running the **func** program. I think you will agree the newer version is more useful.

```
$ func 5 3
5 + 3 is 8
5 - 3 is 2
5 / 3 is 1
5 * 3 is 15
```

When creating functions, you want to think of them as your servants. That is, they take care of something for you. When coded properly, a function can perform part of a larger job, allowing you to break down your large tasks into smaller tasks. The largest challenge in programming is breaking down problems into smaller, meaningful pieces, so that they can be coded more easily and reliably. As you start using functions, consider that a function performs a particular job – it receives certain input, performs its job, and produces certain output. Functions should be as general as possible to have the most utility. A function that adds any two numbers is more useful than one that adds just two specific numbers, and a function that allows any operator rather one that allows only additional is even more useful still.

Returning from Functions

Functions return a value or a function exit status. The keyword **return** causes a function to exit immediately. Adding an optional integer value after the **return** will result in this value being assigned to **\$?** immediately after the call to the function. This allows functions to behave very similarly to stand-alone shell programs. The code below demonstrates the return value of a function:

```
$ cat returntest
#!/bin/sh

somefunc() {
    echo in somefunc
    return 9
}

somefunc                                # this is the call to somefunc
echo somefunc returned $?              # $? after the call is the return value

$ returntest
in somefunc
somefunc returned 9
```

If no **return** statement is used to return an exit status from a function, then the exit status from the last executed command within the function is returned.