CS1101: Lecture 8
UNIX Shell Scripts

Dr. Barry O'Sullivan
`b.osullivan@cs.ucc.ie`

Course Homepage
`http://www.cs.ucc.ie/˜osullb/cs1101`

Lecture Outline

- Shell Scripts
- Simple Shell Scripts
  - Running the Script
  - Permissions – `chmod`
  - Subshells
  - Defining our Subshell
- The Shell as a Programming Language
  - Variables
  - Environment Variables
  - User-created Variables
  - Positional Parameters
- Taken from: Anderson – Just Enough UNIX

Shell Scripts

- Until now we have used the UNIX shell as a **command-line interpreter**.
- The shell can also be used as a **high-level programming language**.
- Instead of entering commands one at a time in response to the shell prompt, you can put a number of commands in a file, to be executed all at once by the shell.
- A program consisting of shell commands is called a **shell script**.
- This lecture will introduce you to shell scripts for the Bourne Shell.

Simple Shell Scripts

- Suppose you were to make up a file named `commands` containing the following lines:

```
# A simple shell script
cal
date
who
```

- The first line in this file begins with a # symbol, which indicates a comment line.
- Anything following the # is ignored by the shell.
- The remaining three lines are shell commands:
  - the first produces a calendar for the current month,
  - the second gives the current date and time
  - the third lists the users currently logged onto your system.

Running the Script

- One way to get the Bourne Shell (sh) to run these commands is to type:

```
$ sh < commands
```

- The redirection operator (<) tells the shell to read from the file commands instead of from the standard input.
- It turns out, however, that the redirection symbol is not really needed in this case.
- Thus, you can also run the commands file by typing

```
sh commands
```

Permissions

- Is there any way to set up commands so that you can run it without explicitly invoking the shell?
- In other words, can you run commands without first typing sh, csh, or ksh?
- The answer is yes, but you first have to make the file executable.
- The chmod utility does this:

```
$ chmod u+x commands
```

- Now all you need do is type the file name:

```
$ commands
```

  and the shell will run the commands in the file.

Subshells

- When you tell the shell to run a script such as the commands file, your login shell actually calls up another shell process to run the script. (Remember, the shell is just another program, and UNIX can run more than one program at a time.)
- The parent shell waits for its child to finish, then takes over and gives you a prompt:

```
$
```

Subshells

- Incidentally, a subshell can be different from its parent shell.
- For example, you can have csh or ksh as your login shell, but use sh to run your shell scripts.
- Many users in fact do this.
- When it comes time to run a script, the csh or ksh simply calls up sh as a subshell to do the job.

Defining our Subshell

- We will always use sh for running shell scripts.
- To make sure that sh is used, we will include the following line at the top of each shell script file:

```
#!/bin/sh
```

- In this case # does not mark a comment.
- Thus, our commands file would look something like this:

```
#!/bin/sh
# A simple shell script
cal
date
who
```

The Shell as a Programming Language

- The sample script commands is almost trivial – it does nothing more than execute three simple commands that you could just as easily type into the standard input.
- The shell is actually is, in fact, a sophisticated programming language, with many of the features found in other programming languages:
  - Variable
  - Input-Output functions
  - Arithmetic operations
  - Conditional expressions
  - Selection structures
  - Repetition structures

Variables

- There are three types Of variables commonly used in Bourne Shell scripts:
  - **Environment variables:** Sometimes called special shell variables, keyword variables, predefined shell variables, or standard shell variables, they are used to tailor the operating environment to suit Your needs. Examples include TERM, HOME, MAIL
  - **User-created varables:** These are variables that you create yourself.
  - **Positional Parameters:** These are used by the shell to store the values of command-line arguments.

Environment Variables

- Some standard shell variables (such as HOME, SHELL) are set automatically for you when you log in.
- Others (such as TERM) you may set yourself – usually in one of your startup configuration files (`.profile`, for example).
- To define an environment variable called TERM, setting it equal to vt100:

```
TERM=vt100
export TERM
```

- To list the environment variables defined on your system type `set` at the command prompt.

User-created Variables

- You can specify these yourself, give them whatever names you wish.
- Example: create a synonym for a directory:

```
stuff=/user/local/users/allsorts
export stuff
```

- To refer to this directory you can type:

```
cd $stuff
```

Positional Parameters

- The positional parameters are very useful in shell programming.
- The positional parameters are also called **read-only variables**, or automatic variables, because the shell sets them for you automatically.
- They ”capture” the values of the command-line arguments that are to be used by a shell script. The positional parameters are numbered 0, 1, 2, 3, . . ., 9.
- To illustrate their use, consider the following shell script, and assume that it is contained in an executable file named `echo.args`:

```
#!/bin/sh

# Illustrate the use of positional param
echo $0 $1 $2 $3 $4 $5 $6 $7 $8 $9
```

Positional Parameters

- Suppose you run the script by typing the command line:

```
echo.args We like UNIX.
```

- The shell stores the name of the command (“`echo.args`”) in the parameter $0; it puts the argument “We” in the parameter $1; it puts “like” in the parameter $2, and “UNIX.” in parameter $3.
- Since that takes care of all the arguments, the rest of the parameters are left empty.
- Then the script prints the contents of the variables:

```
echo.args We like UNIX.
```

Positional Parameters

- What if the user types in more than nine arguments?
- The positional parameter $* contains all of the arguments $1, $2, $3, ... $9, and any arguments beyond these nine.
- Thus, we can rewrite echo. args to handle any number of arguments:

```
#!/bin/sh
# Illustrate the use of positional param
echo $*
```