

# Part III

## Bash scripts

# Shells

- A shell is the program that interprets what you type at the command-line.
- There are many shells.
- The most common (default) is the Bourne-Again Shell (`bash`).
- Others include `ksh`, `csch`, `tcsh`.
- These can look deceptively similar, but have subtle differences.
- I shall focus on `bash`.
- Bash includes a complete programming language, with loops, if-else statements, functions, etc.
- For far more detail than you will ever need, see <http://www.tldp.org/LDP/abs/html/>

# Shell script

- Instead of manually repeating a set of instructions at the command-line, you can put the commands into a single file **compile**:

```
#!/bin/bash
gcc -c main.c -o main.o
gcc -c ODE.c -o ODE.o
gcc main.o ODE.o -lm -o ODEsolver
```

- Once this is defined to be executable with `chmod u+x ./compile` you can run the command `./compile` which will run these commands.
- The first line indicates the interpreter for the file.
- You can equally use another shell or even `/usr/bin/python` or similar.

# Environment variables

- There are many special environment variables.
- To see a full list as defined in your shell, type **export**.
- To set an existing or new environment variable for use in the whole shell, use

```
export MYNAME=pmb39
```

- This is then defined for the remainder of the shell session.
- To set a variable for the remainder of a script only, use

```
allFiles="file1.txt file2.txt"
```

- To access a variable's value, use **\$MYNAME**:

```
$ echo "My name is $MYNAME"
```

```
My name is pmb39
```

```
$
```

This is called variable *expansion*.

# Special environment variables - PATH

- PATH is the set of directories that are searched for executables:

```
PATH=/bin:/usr/bin:.
```

- If a command is not found in any of these directories:

```
$ castep.mpi
castep.mpi: command not found
```

- so you would need to give its full path:  
/lsc/opt/castep-18.1/castep.mpi or put the path into PATH:

```
export PATH=/lsc/opt/castep-18.1:$PATH
```

- The directories are searched in order from the beginning, and the first one to contain the required executable is used.

```
echo $PATH
/lsc/opt/bin:/usr/local/cuda/bin:/usr/local/sbin:
/usr/local/bin:/usr/bin:/bin
```

# Making use of PATH

- It can be useful to create a directory `~/bin` and add this to your `PATH`:

```
export PATH=~/bin:$PATH
```

- Then, you can make various useful scripts available everywhere.
- Note the existence of the `which` command:

```
$ which g++
```

```
/usr/bin/g++
```

```
$ which spellCheck
```

```
/home/pmb39/bin/spellCheck
```

- You can add extra lines into the file `~/.bashrc` to automatically add this directory for every new shell/terminal you start.

# Special environment variables

- SHELL - the current shell e.g. `/bin/bash`
- HOSTNAME - the current computer name e.g. `cdt-laptop-01`
- PWD - the current working directory
- OLDPWD - the previous working directory - use `cd -` to go there.
- HOME - your home directory on this computer - use `cd` to go there

# Regular Expressions

- We may wish to operate on a set of files whose names conform to a particular format:
- `ls *.C` will list all files ending in “.C”
- `ls *[0-9].txt` will list all files ending in a digit followed by `.txt`
- `ls Water*[02468].txt` will list all files starting with `Water` and ending with an even digit followed by `.txt`.
- Strictly, Bash uses the POSIX Extended Regular Expression (ERE) dialect of regular expressions.



# Basic quoting and escaping

- To preserve the literal meaning of characters, enclose them in “ ”
- So `ls "*.C"` will list all files called, literally, `*.C`, of which there will probably be none.
- `ls "My Documents"` will list the directory called `My Documents`.
- `ls My Documents` will try to list the files/directories called `My` and `Documents`.
- Or, `ls My\ Documents` will list the directory `My Documents`
- The backslash “escapes” the following space, i.e. interprets it as an ordinary character, rather than as a word separator.
- (You are strongly advised to avoid having spaces in directory names, to avoid problems in scripts and commands that fail to deal with them correctly.)

# Command substitution

- If you want the result of one command to be used later on, you can use command substitution:

```
files=`find -name "*.pdf"`
```

- The backticks ` ` can contain piped or other commands:

```
words=`detex Thesis.tex | spell | sort | uniq`
```

# Exit codes

- All Linux programs return an integer between 0 and 255.
- C/C++ programmers will recognise this from `int main(void)`
- Usually, 0 indicates the program exited correctly with no errors.
- Other exit codes depend on the application, and should be listed in the `man` page.
- For example, for `grep`:

## EXIT STATUS

The exit status is 0 if selected lines are found,  
and 1 if not found.  
If an error occurred the exit status is 2.

# Special variables

There are variables whose value changes depending on the current shell script, etc.

Variable	Description	Example
<code>\$?</code>	Exit code of last process run	0
<code>\$#</code>	Number of command-line arguments passed to a script	2
<code>\$*</code>	All command-line parameters, as a single word	<code>myFile.txt myFile2.txt</code>
<code>@</code>	As <code>\$*</code> but with words separately quoted	<code>myFile.txt myFile2.txt</code>
<code>\$0</code> , <code>\$1</code> , <code>\$2</code> etc.	Successive command-line parameters passed to script, starting with command-name	<code>ls, ./</code>

# Conditional statements

- In **bash**, the **test** command is capable of comparing strings, integers, and testing whether files exist.
- For example:

```
if test $? -ne 0; then
    echo "Error"
    exit 1
fi
```

will only print “Error” (and exit the script) if the previous command exited with exit code not equal to zero.

- Other options include:

```
if test -f output.txt; then
    echo "output.txt exists. Will not overwrite."
    exit 1
fi
```

- See **man test** for more details.

# Logical operations

- **test** can deal with AND (**-a**) and OR (**-o**).
- So can the shell, with **&&** and **||**
- These employ short-circuiting, i.e. work from left to right and stop as soon as the result is known.
- Here 0 is true (success), and anything else is false (error).

```
mkdir Pictures && mv *.png ./Pictures
```

will only move files if the directory has been successfully made.

- If there had already been a *file* called **Pictures**, we might otherwise have overwritten it.

# For Looping

- There are two main forms of for loop in bash:

```
for f in $myFiles; do cp $f $f.bak; done
```

would copy all files given in the `myFiles` variable to backup versions of same.

```
for ((i=0 ; i < 10 ; i++)); do  
    mv "Data$i.txt" ./FirstPass/  
done
```

would move 10 files into the directory `FirstPass`.

- You can write everything on one line with `;` or on multiple lines, either in a script or at the command line.
- Braces may be needed around the variable being expanded:

```
mv "Data_${i}_coarse.txt" ./FirstPass/  
mv "Data_${i}_coarse.txt" ./FirstPass/
```

- In the first version the shell would attempt to expand the variable `i_coarse`, resulting in `Data_.txt`

# While looping

- There is also the do-while loop:

```
while true; do  
    echo "Hello"  
done
```

would print “Hello” for ever.

- Use **break** to exit a loop early.
- Use **continue** to go immediately to the next iteration of a loop.



# Reading user input

- You may want user input during a shell script.
- `read a b` will read two words from the user into variables `a` and `b`
- Then to read repeatedly from stdin:

```
while true; do
    read a b || break
    echo "Received pair of inputs ${a} and ${b}"
done
```

- If `read` fails (i.e. no input left), then it will return exit-code 1, corresponding to failure/false, therefore `break` will be evaluated, so that execution of the loop stops.

- Basic text-substitution on one or more text-files is sometimes necessary.
  - If you have mis-capitalized an acronym, for example:
- ```
sed 's/Muscl/MUSCL/' Thesis.tex > Thesis_new.tex
```
- This will replace (almost) all occurrences of “Muscl” with “MUSCL” throughout `Thesis.tex` and put the result into `Thesis_new.tex`
  - This actually only replaces the first instance of “Muscl” on each line.
  - To replace all occurrences:

```
sed 's/Muscl/MUSCL/g' Thesis.tex > Thesis_new.tex
```

where the extra ‘g’ stands for ‘global’.

- If you are certain that your `sed` script is working properly, you can modify the files as they are processed:

```
sed -i 's/Muscl/MUSCL/g' Thesis.tex
```

- This means you can also do:

```
for f in Chapter*.tex; do  
    sed -i 's/Muscl/MUSCL/g' $f;  
done
```

to replace all occurrences throughout all chapters in your thesis.

- You will need to take care if you refer to “Muscles” anywhere in your thesis...

# Escaping characters

- You may need to modify a script which contains paths:

```
sed 's/\/home\/pmb39\/\/\/home\/raid\/pmb39\/\/g'  
myScript.sh
```

which replaces `/home/pmb39/` with `/home/raid/pmb39/`

- The `/` character needs to be escaped as otherwise it would be interpreted as the delimiter between separate parts of the replacement command.
- Other characters which need to be escaped are:  
`$ . [ ] ^ ? +`
- Another way of writing the above is:

```
sed 's%/home/pmb39/%/home/raid/pmb39/%g' myScript.sh
```

where the `%` character is now the delimiter as it is the first character after the `'s'`.

# Matching certain lines

- What if you only want to replace Muscl with MUSCL in a list:
  - 1) Use the Muscl method (see Toro for details of Muscl)
  - 2) Another line
  - 3) A line with MusclReference to Muscl.
- Here we can force sed only to make the replacement if the line starts with a number:

```
sed '/^[0-9]*)/s/Muscl/MUSCL/g' Thesis.tex
```

- The regular expression `^[0-9]*)` matches the list indicators.
- The result will be:
  - 1) Use the MUSCL method (see Toro for details of MUSCL)
  - 2) Another line
  - 3) A line with MUSCLReference to Muscl.

# More regular expressions

- Regular expressions in **sed** are used for pattern-matching text.
- Examples are:
  - `.` match any character
  - `*` match any number (including zero) of the preceding character
  - `[0-9]` match any digit
  - `^` match the beginning of a line
  - `$` match the end of a line
- These are *not* exactly the same as what your **bash** command-line will recognize.
- **sed** supports POSIX.2 Basic Regular Expressions (BRE)

## Extended examples

- The regular expression:  
`wh*it*ch`  
will match all of  
`which witch whitch wich`
- The regular expression:  
`wh*it.`  
will match all of  
`white wits with` but not `which` or `wit`  
although `'wit '` would be matched as the `'.'` matches the space.
- To avoid matching `'wit '`:  
`wh*it[^ ]`  
where `[^ ]` means to match everything except a `' '`

- More details of `sed` are available at <http://www.gnu.org/software/sed/manual/>
- An easier to read and more powerful alternative to `sed` is `awk`.
- Regular expressions are covered in detail at <http://www.zytrax.com/tech/web/regex.htm>
- Knowing about regular expressions can speed things up at the command-line as well as in text-replacement.
- Text-editors (such as `emacs` and `vi`) can also deal with regular-expression search/replace.



# Extended #!

- Earlier, you may have wondered about the `#!/bin/bash` at the start of a bash-script.
- The `#!` (hash-bang) are special characters indicating that the next thing on the line is a separate executable that will be used to parse the file.
- This could be any parser, such as:

```
#!/usr/bin/sed -f
#!/usr/bin/python
...
```

Note the `-f` for `sed` because what actually happens is that the filename is appended to the `#!` line, and `sed` requires the `-f` flag in this case.

- (If you start writing `sed` scripts into a file, stop and reconsider.)