FIT2100 Practical #1
Introduction to Unix/Linux, Running Shell Scripts and C Programs
Week 2 Semester 2 2017

Dr Jojo Wong
Lecturer, Faculty of IT
Email: `Jojo.Wong@monash.edu`

© 2016-2017, Monash University

July 21, 2017

© 2016-2017, Faculty of IT, Monash University

Revision Status:

```
$Id: FIT2100-Practical-01.tex, Version 1.0 2017/07/21 17:30 Jojo $
```

© 2016-2017, Faculty of IT, Monash University

Contents

© 2016-2017, Faculty of IT, Monash University

© 2016-2017, Faculty of IT, Monash University

1 Background

The purpose of this first practical is to provide you with some basic experience on the Unix/Linux system and some general-purpose Unix/Linux commands. In addition, you will get to learn how to run Shell scripts and C programs.

There are some pre-lab preparation (Section 2) that you should complete before attending the lab. The practical tasks specificed in Section 3 are to be completed and assessed in the lab.

2 Pre-lab Preparation (3 marks)

2.1 A Brief History of Unix/Linux

A brief history of Unix/Linux is available on the FIT2100 Moodle site. Please have a good read yourself.

Let's get started with a Unix/Linux system by learning the basics of how to drive it from a command shell (i.e. a command line).

2.2 Manual Pages for Unix

The Unix manual pages (also called **man** pages) provide information about most of the commands, programs and libraries on Unix/Linux systems.

To find and display manual pages, the `man` command is used. For instance, to display the `man` page for the `man` command, type the following command:

```
$ man man
```

(Note: The $ at the start of a command line is the *shell prompt* in Unix/Linux systems. Please do not type $ as part of your command. However, your shell prompt can be different depending on the shell that you use.)

It is important to realise that the `man` pages do not contain information about all of the programs installed on the system. You should also realise that the contents of many of the `man` pages is aimed at experienced users, and if you are new to Unix/Linux you may find this a little overwhelming.

© 2016-2017, Faculty of IT, Monash University

Navigating the manual pages The `man` command enters into the so-called a *pager* mode, and will only display the first 20 (or so) lines of the `man` page. You can use a number of commands in the pager to scroll the text:

Command	**Description**
Press `[enter]`	to scroll forward to the next line of text
Press `[space]`	to scroll forward to the next page of text
Press `h`	to display a list of commands that man pager accepts
Press `q`	to quit and return back to the shell prompt

Try it yourself:

- Find out how to scroll backwards a page and then scroll back up to the top of the `man` page.
- Use the `man` command to find more information about the following commands:
  (a) `ls`
  (b) `more`
  (c) `cat`
  (d) `vim`

2.3 Unix Commands

Unix commands are essentially executable files representing programs — mainly written in the C language. These program files are stored in certain directories such as `/bin`. (We will explore the Unix file system in the next subsection.)

Unix commands are in lowercase (remember that case is significant in Unix). The syntax for Unix commands is invariably of the form:

```
$ command -option1 -option2 ... other-arguments
```

Note: You can press the UP arrow key to repeat a previous command, which is especially handy for repeating long command lines.

Before trying out some commands, create a directory (or folder) named `FIT2100` under your home directory and then change to this `FIT2100` directory, using the following commands:

```
$ mkdir FIT2100
$ cd FIT2100
```

Now, we are going to try out some UNIX commands. Before that, if you would like to clear the contents on your screen, the `clear` command does the job.

```
$ clear
```

2.3.1 Basic Commands

Try to understand the following commands. These commands can be used to find out who the users are since Unix is a system used by multiple users. What are the differences between these commands (check their `man` pages)?

```
$ who
$ w
$ users
$ finger
```

Sometimes, you may forget about your username, especially when you have a number of accounts on a particular system. The additional words used with `who` are known as *arguments*.

```
$ who am i
```

To find out your terminal name, the following command is used. (Note that a hardware device is also a file in the Unix/Linux file system.)

```
$ tty
```

Sometimes, you would like to know just what a command does and not get into its syntax. For example, what does the `cp` command do? The `whatis` command provides a one-line answer:

```
$ whatis cp
cp     -- copy files and directories
```

Often times, once you have identified the command you need, you can use `man` command to get further details.

The `whereis` command can be used to locate the executable file represented by a command. As mentioned above, all Unix commands are essentially executable files representing programs. Try the following command and see what is does.

```
$ whereis pwd
```

The `apropos` command performs a *keyword* look-up to locate commands. For example, if you wonder what the command to copy a file is, the following command could be used:

```
$ apropos "copy files"
```

The `man` command together with the option `-k` is an alternative to `apropos`. You can try this to verify:

```
$ man -k "copy files"
```

The `script` command allows you "record" your login session in a file. All the commands, their outputs and the error messages (if any) are stored in the file for later viewing. If you are doing some important work, and wish to keep a log of all your activities, then you should invoke this command immediately after you log in:

```
$ script
Script started, file is typescript
```

The (possibly new) prompt returns, and from now on all your keystrokes that you enter here as well as the outputs of the commands that you entered, get recorded in the log file named *typescript*. After your recording is over, you can terminate the session by entering:

```
$ exit
Script done, file is typescript
```

You can now view this file with either of the following commands.

```
$ cat typescript
$ more typescript
```

Please note that the `script` command overwrites any previous typescript that may exist. If you want to append to it, or use a different log file name, use the commands as shown below.

```
$ script -a
$ script logfile
```

The `uname` command is useful for finding out the type of operating system (OS) running on the machine that you are using. The option `-n` provides you the machine name, while the option `-r` shows the version number of the OS.

```
$ uname
$ uname -n
$ uname -r
```

To display the system date and the calendar, try the following:

```
$ date
$ cal
$ cal 7 2001
$ cal 7 2017
$ cal 7 2999
```

The `wc` is the command for counting the number of lines, words, and characters in a file. Try the following command and explain the output:

```
$ wc /etc/passwd
```

Generally the `passwd` file, which is a text file, contains information about all the users registered on the system. You can use the following command to read it.

```
$ cat /etc/passwd
```

(Note: For security reasons, the password file may be stored some where else too depending on the version of Unix/Linux. Some systems may store the `passwd` file in `/var/db`.)

Often times, you may want to search a file for a pattern and displays all the lines (of the file) that contain the given pattern. Try the following command line with the `grep` command:

```
$ grep <your_username> /etc/passwd
$ grep <your_username> /var/db/passwd
```

For example, if your username is "jojowong", then the command line should be:

```
$ grep jojowong /etc/passwd
```

Are you able to find out how many registered users on the system who have your first name as part of their name? (Hint: Check out the `-c` option of `grep`.)

Try it yourself:

- Which Unix command converts an image to `jpeg` format? (Note: jpeg is a popular data compression method.)
- Which Unix command displays the status of disk space (i.e. the number of free disk blocks) on a file system?
- Which Unix command displays the type of a file (e.g. ascii text, executable, etc.)?

2.3.2 Combining Commands

So far, you have been executing individual commands separately. In fact, Unix allows you to specify more than one command in the same command line. Each command has to be separated from the other by a semicolon (`;`).

```
$ who; date; cal
```

You can even *redirect* the output of these commands to a single file. For example:

```
$ who; date; cal > newfile
```

You can then view the contents of this file with the `cat` command:

```
$ cat newfile
```

Suppose that you inadvertently pressed the `[enter]` key just after entering `cat`:

```
$ cat [enter]
```

There is no action here; the command simply waits for you to enter something. You can use the [ctrl-D] key to get back to the shell prompt.

(Note: To *interrupt* a running program, you can also use the `[ctrl-C]` key. This will then cause the running program to terminate.)

© 2016-2017, Faculty of IT, Monash University

2.3.3 More Useful Commands

In addition to the cat command, there are a number of commands available in Unix for displaying the contents of files on the screen.

The more command is to display the contents of a file in one screenful at a time. (Press the [space] key to get the next screen.)

```
$ man man > test.txt
$ more test.txt
```

A similar command to more is called less. Can you find the difference between more and less?

The touch commands allows you to create a new empty file.

```
$ touch test2.txt
$ cat test2.txt
```

(Note that there is another command called zcat for displaying the contents of a compressed file on the screen.)

The echo command is used for printing the text on the screen. It will be useful in *shell programming* (also known as scripting). Try out the following command:

```
$ echo "Hello World"
```

To display the first few lines of a text file (10 lines by default), the head command is available. A similar command is tail, which displays the last few lines of a text file (10 lines by default).

```
$ head test.txt
$ tail test.txt
```

The command history is used for displaying all of the stored commands in the history list.

```
$ history
```

Another command which is similar to a man page — info — can be used to display the information page for a given command. For example:

```
$ info pwd
```

To find out a list of jobs (processes) started in the current shell environment, use the `jobs` command:

```
$ jobs
```

The `free` command is useful for finding out the amount of free and used memory (both physical and virtual), with basic information about how that memory is being used.

```
$ free
```

Finally, you can cause the shell to “sleep” for a specified number of seconds.

```
$ sleep 5
```

2.4 The Unix File System

In a Unix/Linux file system, everything is treated as *file* — every objects in the system can be accessed in a file-like way.

Some terminologies A *file* in the system contains whatever information a user places in it. There is no format imposed on a regular file; it is just a sequence of bytes.

A *directory* contains a number of files; or it may also contain subdirectories which in turn contain more files. There is only one *root* directory. All files in the system can be traced through a path as a chain of directories starting from the root directory.

When a file is specified to the system, it may be in the form of a *path name*, which is a sequence of filenames separated by slashes. In Unix/Linux systems, a forward slash (/) is used (rather than a backslash). Any filename except the one following the last slash must be the name of a directory.

Unix files All files in a Unix/Linux file system are treated equally — i.e. the system does not distinguish between a text file, a directory file or other file types. It is up to the user to know the type of file they are using, which can result in operations being attempted on incompatible file types (e.g. printing executable output files to printers).

The command `file` can be used to give a fairly reliable description of the contents of a file.

```
$ file /bin/tcsh
$ file /bin
$ file /etc/passwd
```

2.4.1 Handling Files and Directories

The Unix/Linux operating system provides a number of commands to create, modify, traverse and view the file systems. Make sure you know what each of the following commands does and how to use each command.

Command	Description
cd	change to another directory
pwd	print the present (working) directory
ls	list a directory's contents
mkdir	make a directory
rmdir	remove a directory
rm	remove a file or a number of files
mv	rename/move a directory or a file
cp	copy a file
df	display information about free space on the available file systems
du	display disk usage information (for files and directories)

The `cd` command without any argument takes you back to your home directory from anywhere.

The `ls` with the option `-l` will display more information about files. For example, you can see the file size with this command.

```
$ ls -l
```

Special characters Make sure you know how to use the special characters, such as '*' and '?' with the above commands. In addition, you should also know how to use the '..', '.', and '~' shortcut symbols with these commands.

Symbol	Description
* (asterisk)	represents all ordinary files in a directory
?	represents a single character
~ (tilde)	represents your home directory

© 2016-2017, Faculty of IT, Monash University

Try it yourself: Make sure you understand what is happening as well as the output after the execution of each of the following commands.

```
$ man man > f01.txt
$ cp f01.txt f02.txt
$ cp f01.txt f001.txt
$ cp f01.txt f002.txt
$ cp f01.txt test_f01.txt
$ cp f01.txt test_f02.txt
$ cp f01.txt f1.txt
$ cp f01.txt f2.txt
```

```
$ ls
$ ls f*.txt
$ ls f?.txt
$ ls f??.txt
$ ls f???.txt
$ ls test*.txt
$ ls ??.txt
$ ls ???.txt
$ ls ~
```

After you have a good understanding of the above, remove all of these text files using the following command:

```
$ rm f*.txt t*.txt
```

2.4.2 Unix File Attributes

Each file in the Unix/Linux file system has a number of *attributes* associated with it. Some attributes that are associated with a file include:

- name
- size
- permissions/protection
- time/date of modification
- owner
- group

To investigate file attributes, change your working directory to your home directory. Get a full directory listing by typing the command: `ls -al`. Make sure you know what each piece of information means.

Try it yourself:

(a) Under the directory FIT2100 make a subdirectory called `test_dir`.

(b) Create an ordinary file called `file.txt` under `test_dir`.

(c) Check the current file attributes (especially the permissions) of `file.txt`.

(d) Use the following commands to change the permissions of `file.txt`. Verify the effect of each command using `ls -l file.txt`. (Make sure you understand the *permissions* represented by each *octal* number.)

```
$ chmod 000 file.txt
$ chmod 777 file.txt
$ chmod 666 file.txt
$ chmod 444 file.txt
$ chmod 664 file.txt
$ chmod 600 file.txt
$ chmod 466 file.txt
$ chmod 251 file.txt
$ chmod 111 file.txt
$ chmod 700 file.txt
```

2.4.3 Locating Files

One of the powerful tools of the Unix/Linux system — the `find` command — which recursively examines a directory tree to look for files either by name or by matching one or more file attributes.

Use the `man` page to find out the options available for the `find` command.

Try it yourself:

(a) Under the `root` directory, find all files that are more than 2 months (60 days) old.

(b) Under the `root` directory, find all files that are of 2-character file names.

(c) Under the `root` directory, find all files that are of 2-uppercase character file names (e.g. `AB`, `BG`).

2.5 The Shell for Unix/Linux

The shell in Unix/Linux systems provides a command-line user interface that interprets and executes Unix-based commands from users.

There are several Unix shells available which largely fall within two classes — the 'Bourne' shells (`sh`, `ksh`) and the 'C' shells (`csh`, `tcsh`). The `bash` shell (known as the Bourne Again Shell) is the default shell on most Linux distributions.

2.5.1 Pre-processing of Unix commands

The life cycle of pre-processing Unix commands by a shell can be described as follows:

- The shell places the prompt on the user terminal and goes to sleep.
- The user types a command line consisting of one or more commands. These commands may be separated by the following symbols: `;` (sequentially execute), `||` (otherwise execute), `&&` (if ok then execute).
- When the user presses `[enter]`, the shell begins processing the command line.
- First step in this pre-processing is to *parse* the first command. If there are more than one command in a line they will be processed and executed after the first command has finished execution. However, the decision will be based on the separator (`;`, `||`, or `&&`) you use between the commands.
- The command is broken into its constituent words. The end of the words is usually identified by the spaces, tabs and special symbols. The command line parser is often not as nice as those used by the programming languages. As a result, sometimes your command may not be understood if there is an extra space or you miss a space between the words.
- Next, the shell replaces variables by their values. These variables are shown in the command by preceding them with the symbol `$`.
- *Command substitution* is done next. A command substitution is indicated by enclosing the command in a pair of back-quotes (``` `` ```). (Note: this quote is usually found on the left end of the top row on your keyboard.)
- The shell then performs redirection of the standard input, standard output and standard error output, if requested.
- Wild-cards are expanded next.

- Finally the command is ready to execute. The shell searches for an executable file whose name matches the command name.
- While the command executes, the shell waits.
- When the execution finishes, the shell displays next prompt on the terminal. A new cycle begins.

2.5.2 Writing C Programs with Unix Commands

In your home directory, do the following:

- First, try invoking the Bourne shell by typing `sh` at the command prompt.
- Next, use the following command to create a file: `$cat > prac01`.

```
$ cat > prac01

#include <stdio.h>

 int main (void)
{
     printf("This is the first FIT2100 practical.\n")

     return 0;
}
```

- Finally, use `[ctrl-D]` to finish the session with the `cat` command.

Do you notice an error in the C program above? There should be a semicolon '`;`' at the end of the longest line. This error is deliberate.

(Note: Some of you may wish to use a text editor, such as `pico`, `vi`, `vim` or `joe`, to create text files.)

2.6 Compilation and Execution of C Programs

In this section, we will have an oveview on how the C compiler and linker work in compiling executable C programs. Before that, let's have a quick recap on what we have learned in the Week 1 Tutorial.

How to create a C program? You can use any text editors of your choice to type in the source code. The source file should be saved with the `.c` extension, for example `simple.c`.

How to compile a C program? Before you are able to execute a C program, the program needs to be first compiled using the C compiler, such as `gcc`. To compile with `gcc`, one of the following commands can be used:

- `gcc simple.c`
- `gcc -o simple simple.c`

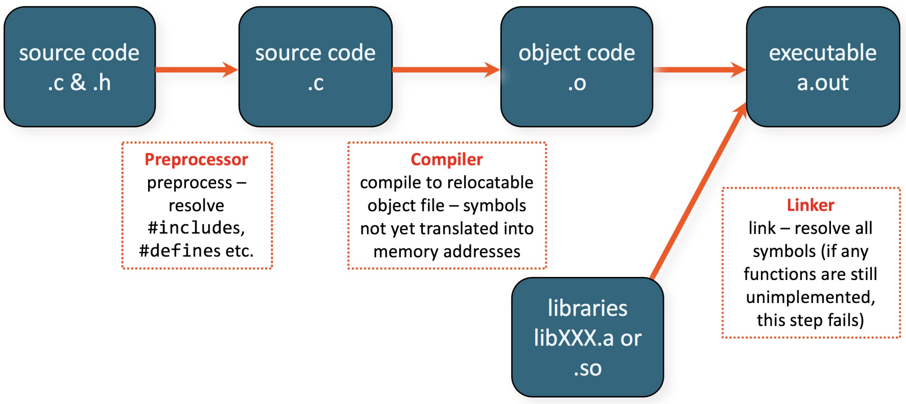
Note: The first command produces the *executable* file named `a.out` from the source file. However, the executable file can be re-named using the `-o` option; the second command sets the name of the executable file to `simple`.

How to run a C program? You can now run the C program with one of the following commands based on the name of the executable file:

- `./a.out`
- `./simple`

2.6.1 How Do The Compiler and Linker Work?

There are three main steps involved in compiling a C source code into an executable program:


source code .c & .h
source code .c
object code .o
executable a.out
Preprocessor
preprocess – resolve #includes, #defines etc.
Compiler
compile to relocatable object file – symbols not yet translated into memory addresses
libraries libXXX.a or .so
Linker
link – resolve all symbols (if any functions are still unimplemented, this step fails)

(Adopted from FIT3042 Courseware by Robyn McNamara)

Preprocessing The C source code is first given to a *preprocessor*, which looks for the *directives* (that begin with the # symbol). Directives such as `#include` and `#define` are resovled in this step.

For `#include`, the given header file is opened and its contents are copied into the current source file. For `#define`, the defined identifiers are searched are replaced with the specific values.

Compiling The modified source code now goes to the compiler, which translates it into machine instructions, also known as *object code*. The program at this step is not yet ready for exeuction; since symbols (such as variables and function names) are not yet translated into memory addresses.

Linking In the final step, the linker combines the object code produced by the compiler with any additional code that needed to produce a complete executable program. The linker attempts to resolve all the symbols and library functions at this last step.

2.6.2 GCC Compiler Options

A summary of various `gcc` options for quick reference.

Option	Meaning
`-c`	compile source but do not link
`-S`	stop after the compilation stage; do not assemble
`-E`	stop after the preprocessing stage; do not compile
`-g`	include debugging information
`-O`	optimise the code to a specific level (e.g. `-O3`)
`-W`	turn on or off particular warnings (e.g. `-Wall` for all warnings)
`-I`	specify a directory to look for include files (e.g. `-I/usr/lib/somelib`)
`-L`	specify a directory to look for library files (e.g. `-L/usr/lib/somelib`)
`-o outfile`	place the output in `outfile`

© 2016-2017, Faculty of IT, Monash University

3 Practical Tasks (7 marks)

3.1 Task 1 (1 mark)

The shell script shown below, is from a file called `helpme`. When executed, the script asks the user to specify the topic (a keyword) on which the help is required. Internally, it uses a standard Unix command to list each manual page that has the specified keyword occurring in it. Recall that the command `man` with the `-k` option is an alternative to the command `apropos`.

```
#!/bin/sh

echo "\nWhat do you need help on: \c"
read topic
man -k $topic | tee file1
```

Enter the above script in a file called `helpme`. First, use the command `chmod` to assign the "execute" permission to yourself; then, use the command `ls -l` to verify the permissions.

Run the shell script by entering the following keywords, respectively:

```
jpeg
telnet
login
chat
```

Also try other keywords made up by yourself on which you might require help.

Find out why the script begins with the line — `#!/bin/sh` — and what does the command `tee` do in this script.

(Note: The use of a shell variable, `topic`, is to assign a value to it and later to retrieve the value from it. How will you change the script if the `topic` variable can accept a phrase made of multiple words such as those suggested below?)

```
remote login
disk usage
file permission
```

© 2016-2017, Faculty of IT, Monash University

3.2 Task 2 (1 mark)

Experience shows that the list generated for most key topics is long and unmanageable, in particularly when you only provide one word instead of a phrase. Fortunately, the two commands — `apropos` and `man -k` — are able to list each command with a 1-line description of the command. We can refine the list by searching for lines containing (or not containing) secondary keywords and clues.

In this task, we will try to seek one more keyword to reduce the size of the list. Take note of the use of the `case` construct in Shell scripts.

Append the following script to the `helpme` file created in Task 1 (Section 3.1) after the last statement.

```
echo "\n\nIs the list of man pages too long? (y/n)"
read YN

case $YN in
[yY]*)
    echo "Please suggest another keyword: \c"
    read topic
    cat file1 | grep $topic | tee file2
    rm file1
    mv file2 file1
    ;;
[nN]*)
    echo "We are done for now.\n"
    ;;
esac
```

Make sure that you understand the purpose of each line in this script. Run this script by first entering the keyword remote and then the keyword login. Enter y to answer the question "Is the list of man pages too long?"

Run this script again to work on the following input pairs:

```
jpeg (first keyword), compress (second keyword)
memory (first keyword), statistics (second keyword)
cpu (first keyword), time (second keyword)
```

Has the content of the file `file1` changed (compared it with that from Task 1)? Run this script by entering other keywords made up by you.

3.3 Task 3 (1 mark)

In this third task, we will repeat the list shortening step as was completed in Task 2 (Section 3.2) for two times. If the list is not adequately short by the end of the second step, the user will be asked to remove some `man` pages based on a keyword that they do not want to read about.

The script presented below is a major re-write of the original `helpme` file. Some of you would like to copy your current version of `helpme` to another file before replacing the whole text of the file `helpme` with the text (script) below.

(Note: The command `grep` has been used with some options specified. How do they help?)

```
#!/bin/sh

 echo "\nWhat do you need help on: \c"
 read topic
 man -k $topic | tee file1

 RepeatsLeft="2"

 while [ $RepeatsLeft -ne "0" ]
 do
     RepeatsLeft=`expr $RepeatsLeft - 1`
                 echo "\n\nIs the list of man pages too long (y/n)?"
                 read YN

                 case $YN in
                 [yY]*)
                         echo "Please suggest another keyword: \c"
                         read topic
                         cat file1 | grep -i $topic | tee file2
                         rm file1
                         mv file2 file1
                         ;;
                 [nN]*)
                         echo "We are done for now.\n"
                         exit 0
                         ;;
                 esac
      done

 echo "Provide a keyword that you wish to exclude: \c"
 read AntiKey

 cat file1 | grep -iv $AntiKey > file2
 more file2
```

Explain the functionality of the following statements in the code given above:

- `` RepeatsLeft=`expr $RepeatsLeft - 1` ``
- `cat file1 | grep -iv $AntiKey > file2`

Run the script by entering the following keywords in turn. (Enter 'y' to answer the question: `"Is the list of man pages too long?"`)

```
file (first keyword)
compress (second keyword)
zip (third keyword)
bzip2 (last keyword)
```

Do you notice a shorter list is produced after each extra keyword has been entered?

3.4 Task 4 (2 marks)

In this task, we will develop a new script program to display the `man` pages selected in the previous task. We will name this script file as `showme`. But, first you need to modify the last statement in the script `helpme` from Task 3 (Section 3.3) — `more file2` — so that it now becomes:

```
./showme file2
```

Also, you have to replace the command line — `echo "We are done for now.\n"` — with the following command:

```
./showme file1
```

The following script `showme` uses the command `cut` to extract the first column of a formatted outpute generated by the command: `man -k $topic`. You should not run the script `showme` independently; it is launched when you run the script `helpme` from Task 3. The script for `showme` is given below.

```
#!/bin/sh

# Get the command names ---
# they are in the first column before a tab
# or a blank character of a formatted ouput

LIST=`cut -f1 $1 | cut -d " " -f1`

# You can use echo $LIST to view the value of LIST here

for R in $LIST
do
    echo Show $R?
        read YN

        case $YN in
            [Yy]*)
                        man $R
                        ;;
              *)
                        ;;
        esac
done
```

Research on the `cut` command. How could you improve the script above in order to display the `man` pages of the selected keywords?

3.5 Task 5 (2 marks)

In this last task, you are required to write a Shell script for compiling and executing C programs. The script should be able to perform two levels of functionality based on the following options:

- `compile-only`: compile a C source code without executing the program after the compilation
- `compile-and-execute`: compile a C source code and execute the program after the compilation

Noted that this script accepts two arguments: the first argument is the file name of a C source code to be compiled and/or for execution; and the second argument is the option of either `compile-only` or `compile-and-execute`.

You may use any of the C programs that you have attempted in the Week 1 Tutorial to test this script.

© 2016-2017, Faculty of IT, Monash University