Here be dragons!

This is not a substitute for the lectures and lecture notes!
Rather, it’s a ‘summary’ to help late starters up to speed,
and to provide more information for people curious about what’s going on,
which isn’t required knowledge for this course.
There will be examinable content in the lecture notes
which aren’t here, and things here which aren’t examinable.
The Shell

The shell allows you to command the OS; when launching a Terminal on Mac OS X and on Linux systems, you’ll arrive at a shell.

In this course, you’ll be interacting with a UNIX shell. UNIX was an operating system built in the early 1970s, and could only interact with a user through a text interface. It did not have a graphical user interface, or GUI: you couldn’t interact with it with a mouse, and in fact GUIs were only invented in the 1980s.

The original way to control UNIX was with sh, the shell (sometimes called the Bourne shell, or the Almquist shell). This was extremely limited, so others developed better shells, and in the lab, we’re using one of those, called bash. When you open a terminal in the lab, you’re interacting with bash.

Shell Tips

This is a small set of tips to speeding up shell usage and also to understand what is happening in the shell. It is not a requirement of the course to study this however if you understand why things are the way they are - you will find you will pick things up quite quickly. It will answer a lot of questions you may have.

Whitespace

A commonly made mistake is to have trouble with whitespace. The shell takes whitespace very literally, and it assumes, unless otherwise instructed, that each “word” is a separate piece of information to pass to the program it’s about to run. So, if you create a “New Folder”,

```
dillon@air:~$ cd New Folder
```

will not take you there! The shell instead sees this as cd, New, and Folder, and the cd command has no idea what to do with this.

However, you may wrap the string in quotes, or use the special backslash character, which ‘\’ tells the shell to treat the following whitespace as part of the previous word. So, these are correct and will work.

```
dillon@air:~$ cd New\ Folder

dillon@air:~$ cd "New Folder"
```

To avoid having to do this entirely, choose your file and directory names such that they don’t have any spaces in them. If you must, connect your words with underscores (_).

The opposite can occur when you’re missing whitespace: if you were to try to run

```
dillon@air:~$ gcc-Wall -Werror ...
```

the shell would try to run a program called gcc-Wall... which doesn’t exist.
Arguments and Options

The commands you run tend to have the shape:

```
dillon@air:~$
```

The program to run would be, for example, `dcc`, `gedit`, `cd`, or `rm`, but what are the options and arguments?

Options tend to begin with a ‘-’, a single dash, and will change the behaviour of a program. So, for example, you will have seen `-Wall`, or `-Werror`, or `-o`. Some options take another argument; for example, `–o` , which says that the following argument is the output file. Mostly, though, arguments are values to be passed into a program, and are not options.

```
dillon@air:~$ dcc -o helloworld helloworld.c
```

In this example, `dcc` is the command, `–o helloworld` is an option, and `helloworld.c` is an argument.

Shell Shortcuts

Up arrow
If you are in a terminal, the up arrow will show your previous command. Multiple presses will cycle through your entire history.

Pasting into XTerm
On the lab machines you can paste commands you have copied by clicking the wheel in the center of the mouse. Ctrl + V does not work.

Tab key
The tab key will attempt to complete partial words. For example, if your home directory contains `Documents` and `Downloads`, typing “cd Doc” would complete to the command “cd Documents”, but if you type “cd Do<tab>”, the shell will beep at you and not complete, as there’s no unique completion. If you press tab again, the shell will list the options, and you should add more characters to make the selection unique.

& (ampersand)
The ampersand is added to the end of a command, and makes that command run in the background, which means we can still use the terminal while the program continues to run.

&& (Double Ampersand)
Double-ampersand allows you to join commands in sequence, but only if the programs are successful:

```
dillon@air:~$ gcc -Wall -Werror -O -o hello hello.c && ./hello
```

What happens is the code will compile, and then it will execute only if your code compiled. Useful in combination with the up arrow so you only need to search history for the single line as we normally would compile then run the program.
The Path

There is a lot of confusion over why we need to execute our programs as ./prog whilst we can use commands like gcc or gedit without a './'. To understand this, a basic understanding of the path is required.

On all UNIX-like systems, there are several directories (including /bin and /usr/bin) that contain sets of pre-installed programs. If we were to run a command, the shell looks through its list of paths, checking if there’s a program matching that name in each of the directories in turn.

```
dillon@air:~$ gedit somefile.txt &
```

It happens that /usr/bin/gedit exists, so the shell will run it.

Usually, your path doesn’t include the current directory, so, if you were to run one of your own programs:

```
dillon@air:~$ helloworld
```

The shell would, again, go looking for helloworld in its path, and because your program isn’t there, the program won’t be found. Instead, you have to specify that this program however is in our current directory, not in the path, so we add the “./” before the program name to tell the shell to look right here for the program.

(By the same mechanism, you could type /usr/bin/gedit every time you wanted to edit a file... but that’s just annoying.)

Shell Commands

```
cd
```

```
cd <directory>
cd -
cd
```

This command allows you to change directory in the terminal. Let’s say in our home directory we have a directory called COMP1911, with two subdirectories lab01 and lab02; we could navigate around like this (noting that the prompt tells us where we are):

```
dillon@air:~$ cd COMP1911
              dillon@air:-/COMP1911$ cd lab01
              dillon@air:-/COMP1911/lab01$
```

There are some special directories that cd recognises:

```
~ (tilde)
Your very own home directory!
.
(a single dot)
The current directory.
```
The directory above the current directory. You can also chain directories together, using a slash as a separator:

```
dillon@air:~$ cd COMP1911/lab02
dillon@air:~/COMP1911/lab02$
```

You can move to the last directory you were in (useful if you've moved to some complex path) using a single dash:

```
dillon@air:~/COMP1911/lab02$ cd -
~
dillon@air:~$
```

And the 'cd' command by itself will take you back to your home directory, like 'cd ~'.

**ls**

```
ls <options>
ls [<directory>]
```

The list command will show you the contents of a directory. When called with no arguments it will list the contents of the current directory. Some useful options include “-a” to list all files - including hidden ones, and “-l” to display the contents as a list. Try "man ls" for more.

**mkdir**

```
mkdir <directory-name>
```

Makes a directory in the specified folder. You must specify the name of the folder as the argument.

**rm**

```
rm <files...>
rmm -rf <files-or-directories...>
```

Removes the specified file or if given the option "-rf" (recursive, force) is able to remove directories. **BE CAREFUL.** Only ever use the rm command with the exact name of what you wish to remove. You will not be able to recover the files from the trash can.

**cp**

```
cp <files...> <destination>
cp -r <files-or-directories...> <destination>
```

Copies the file or files from source to destination.

Option “-r” (recursive) allows directories to be copied, by copying the contents of the directory.
**mv**

```
mv <files-or-directories...> <destination>
```

Moves the file or files from source to destination. Can be used to rename files by moving the file to the current directory with the new name. For example:

```
dillon@air:~$ mv bad_pun.c badPun.c
```

**pwd**

```
pwd
```

Prints the working directory: shows where you are in the directory tree.

**man**

```
man <topic>
man -k <query>
```

Get an online manual page on a particular topic: commands, library functions, and much, much more. If you can’t remember what the topic you’re looking for is called, you can “search” using the -k option, which will list things that roughly match what you query for.

There are many, many other commands!
These are the barest of essentials...
What other commands can you find?
C Programming

In this course, our language of choice is C.

C is a very clean, very simple programming language developed in the early 1970s by Dennis Ritchie and Brian Kernighan, who also wrote “The C Programming Language”, a marvellous book introducing the language.

Editing Text

To write C, we need a text editor. We can’t use programs like Word or Pages, which don’t save plain text, but in their own (highly complex) file formats, storing metadata about the text you’re writing.

Instead, there are a wide range of text editors, such as Gedit, which also give us useful features like syntax highlighting and automatic indentation. In an exam, you’ll only be able to use software in the CSE environment, so make sure you’re familiar with one of the editors available at CSE. Some editors you might like to try

- Nano (simple, no GUI)
- Vim (very powerful, works with/without GUI)
- Emacs (very powerful, works with/without GUI)
- Sublime Text (GUI only; not on CSE)
- Atom (slow, GUI only; not on CSE)
- Notepad++ (Windows only; not on CSE)
- Visual Studio Code (GUI only; not on CSE)

Try a few of these and see which one you like. Some have substantial learning curves, and take a long time to become familiar with; others, less so.

The Style Guide

Style is crucially important in programming, and a useful habit to develop. Good style is a multifaceted set of rules about indentation and whitespace, variable and function naming, the use of brackets around expressions, and much, much more. Importantly, wherever you will write software, there will be a particular style guide that you should adhere to.

In this course (and, indeed, for all our C courses at CSE), we provide a style guide, at wiki.cse.unsw.edu.au/info/CoreCourses/StyleGuide and which is linked from the course website.

You should be familiar with, and adhere to, our style guide, to ensure your code is readable and debuggable, and to ensure you’re working in a common style with your lab partners and our supplied code.
A Simple Program

The easiest way to digest C is to pull apart an actual program, line by line:

1 // Print a friendly message. Whenever you begin a program, you should always have a comment briefly explaining what it does, who wrote it, and when it was written.

2 // Jashank Jeremy, 2017-03-08

3

4 #include <stdio.h> The ‘#include’ directive allows us to pull in pre-written libraries of code, defined in "header files". These allowing our programs to do more than they could otherwise. In this case, we’re pulling in the "standard input output" header, letting us use the printf function.

We specify the name of the header file in angle brackets, to tell the compiler to look in a particular location; by default, they live in /usr/include.

5 It's also a good habit to keep whitespace between each of the major sections of your program.

6 int main (void) { Every program needs a main function, to tell it where to begin from. It returns an integer, and takes no parameters, though you may see, instead of void, something like

int main (int argc, char *argv[]) int main (int argc, char **argv)

For now, don’t worry too much about these; they allow your program take arguments, like gcc and gedit do. void tells the compiler you’re not expecting any arguments.

7 printf ("Hello!\n"); The printf function comes in from stdio.h, and lets us print formatted output to the screen. It takes a string, and prints it to the screen.

Remember also that every statement in a program needs to be ended by a semicolon.

8

9 return 0; Finally, we need to return an integer at the end of the program. The value that main returns is your program's exit code, which indicates to the system whether your program completed successfully. 0 is a magic number that indicates your program completed successfully; a non-zero result indicates your program didn’t succeed.

10 }
Compiling

Without diving too deeply, computers are only able to execute machine code. It is hard and tedious to write this code by hand. To make life easier we make programming languages which are easy to write programs in and then can be translated into machine code. The process of translation is called compilation and that is why we require a compiler.

After we create a program in C, we turn it into a machine readable language with a compiler such as gcc or clang, and then we run the program that compiles. Therefore if we compile a program and make changes in the source code, these changes will not be seen when we run the program as we need to recompile it again. This turns our updated program into a new executable.

At CSE you will be able to use either dcc or gcc:

```bash
  dillon@air:~$ dcc -o myProgram myProgram.c
```

If you are working locally on your own computer you will only be able to use gcc:

```bash
  dillon@water:~$ gcc -Wall -Werror -O -o myProgram myProgram.c
```

The Standard Library

There are a number of functions in the C standard library that we'll see in this course.

```c
#include <stdio.h>

int printf (const char *fmt, ...);
int scanf (const char *fmt, ...);
```

`printf` and `scanf` are functions that allow you to interact with the user, by outputting (formatted) text to them, and reading (formatted) input back from them.

Strings in C are enclosed in double-quotes. In these strings, you may need to enter characters that are otherwise in use or you wouldn't be able to enter, like the double-quote itself: in this case, you need to use escape sequences: the double-quote can be input with `"` and the backslash with `\`. You may also like "control" characters like new-line (`\n`), or tab (`\t`). For more on escape sequences, see [Escape sequences in C](https://en.wikipedia.org/wiki/Escape_sequence) on Wikipedia.

We are also able to indirectly reference variables from strings using printf and scanf using format strings, which use the ‘%’ character followed by a format code, and which will print from or scan into the corresponding variable according to that code. There are many possible format codes, and some are quite complex; you won’t need many in the course, and here are those we’ve seen so far:

- `%d` denotes a signed decimal
- `%f` denotes a single-precision floating point number
- `%lf` denotes a double-precision floating point number
- `%.4lf` denotes a double to be printed to four decimal places (`printf` only)
- `%%` a literal ‘%’
For `printf`, you specify a format string, then some number of expressions, separated by commas, that match each of the format codes you’ve used, in order. The second argument is the first formatted item, and so on. `printf` will return the number of characters it printed, or some number less than zero if an error occurred. For example:

```c
printf ("There are %d students, and a capacity of %d.\n", nStudents, nTheatreSeats);
```

Similarly for `scanf`, you specify a format string, then some number of variables to store the read values into. Remember to specify an ‘&’ before each of the variable names: the reason we do so will become apparent when we cover pointers, later in the course. For example:

```c
int zId;
printf ("What is your zID? ");
scanf ("%d", &zId);
```

<`stdlib.h>`

```c
#define EXIT_SUCCESS 0
#define EXIT_FAILURE 1
```

Above, we saw that the `main` function return a particular value to indicate whether your program completed successfully. However, we know that the 0 is a magic number, and it so happens that in the `<stdlib.h>` header, there’s a pair of definitions: EXIT_SUCCESS and EXIT_FAILURE, that you can use in place of these magic numbers. On some systems, the success and failure codes returned by a program aren’t necessarily these, either.

By using these definitions, we can make the meaning of the return at the end more apparent, and to ensure our code is portable in time and space:

```c
return EXIT_SUCCESS;
```

---

1 Oh, all right: for `printf`, we pass variables in by their values, but we can’t do the same for `scanf` as this doesn’t let us change a variable’s value. Instead, we take a reference to the variable (read the ‘&’ as “reference to”), and `scanf` puts the value it reads in there.
Home Computing

CSE vlab is the simplest of these methods, and is well-supported by course staff. There are many other ways you can do so; though these are complex and prone to problems that course staff may not be able to help with.

CSE vlab

Internet required
Win WSL Lin Mac
difficulty: easy

A precis:

Install the required tools:

1. Install Cisco AnyConnect, from vpn.unsw.edu.au.
2. Install TigerVNC.

To connect:

1. Connect to the VPN; log in with your zID and zPass.
2. Open TigerVNC and connect to vlab.cse.unsw.edu.au

Secure Shell (SSH)

Internet required
Win Lin Mac
difficulty: easy

In a terminal, run

$ ssh zNNNNNNN@login.cse.unsw.edu.au
You are using a computer owned by:
School of Computer Science and Engineering, UNSW Australia

***** This service is for authorised clients only *****

******************************************************************************
*WARNING:     It is a criminal offence to:
*              i.  Obtain access to data without permission                     *
*                        (Penalty 2 years imprisonment)                    *
*              ii. Damage, delete, alter or insert data without permission *
*                        (Penalty 10 years imprisonment)                   *
******************************************************************************

zNNNNNNN@weill:$
You will only get a command-line interface, so graphical programs like *gedit* won’t work, but other editors like *nano*, *emacs*, and *vim* will, and you’ll be able to compile, run, and submit your programs in this environment.

Typing the whole zID-at-CSE address each time is extremely tedious, and programmers are proverbially lazy. Instead, you can set up an alias that means you don’t have to type as much, and I’ll use this below.

```
$ mkdir -p ~/.ssh
$ echo "Host cse" >> ~/.ssh/config
$ echo "HostName login.cse.unsw.edu.au" >> ~/.ssh/config
$ echo "User zNNNNNNN" >> ~/.ssh/config
```

Now, you can simply do

```
$ ssh cse
[... warning screen elided ...]
zNNNNNNN@weill:~$
```

To transfer files from your computer to CSE, use the *scp* command:

```
$ scp prism.c cse:~/1911/lab02/
```

And to go the other way:

```
$ scp cse:~/1911/lab02/prism.c .
```

A common workflow is to write your code on your own system, compile and test locally, then copy it to CSE to submit.

## Windows Subsystem for Linux (WSL)

*Internet not required*  

*Win*  

*Lin*  

*Mae*  

*difficulty: hard*

If you’re on Windows 10, the way I’d recommend doing home computing is using something known as Windows Subsystem for Linux. This will allow you to do all your programming locally without any need for any internet connection (useful for coding on the train). To install it follow the following instructions:

1. Go to Settings > System > About.  
   Ensure you have Windows 10 build **14393** or greater.  
   If not, update your computer.
2. Go to Settings > Updates and Security > For Developers.  
   Select “Developer Mode”.
3. In Settings, search for “Turn Windows features on and off”.  
   Enable “Windows Subsystem for Linux (Beta)”.  
4. Reboot.
5. Open the Command Prompt or PowerShell, and run *bash*, which will finish downloading and setting up the WSL environment.
6. Once WSL is installed, you can access it from Command Prompt or PowerShell by running *bash*, or by opening the application.
7. Install gcc by running the command ‘sudo apt-get install gcc’ in the WSL bash.

Now to work with the subsystem take note that there are two filesystems to work with- there’s the ‘linux root’ which you can access by typing in cd or opening through the bash application, and then there’s the Windows filesystem which is basically your normal C drive. If you see something like

:\/mnt/c/...

Then you are in the Windows file system (C drive). Otherwise, you’re probably running in the Linux root. I recommend working in your C drive because that allows you to view files using the file explorer and edit files using graphical text editors. In your Linux root you can’t actually access it through any other part of the operating system (that I’m aware). If you’re in the linux root, type

    cd /mnt/c/

And then navigate to your working directory.

In addition, you won’t be able to (you might but with great difficulty) open a graphical editor from the terminal if you’re in bash. However, you can always just open them with your preferred editor using the file explorer or from command prompt/ powershell if you add it to your path. Alternatively, you can use vim or nano from within Bash to edit files e.g.

    vim hello.c

Once you’ve created a C file, you can simply compile it using gcc and then execute it as you would in Linux. Other commands such as mkdir, cp, mv etc. will also work.

PuTTY

Internet required

Win Lin Mac difficulty: easy

This is the standard setup and is documented in the windows specific section of home computing so I will touch only lightly on it here. A combination of three programs are used to work remotely. PuTTY is an SSH client which will connect you to a CSE shell allowing you to type commands as if you were sitting at a lab computer. It cannot however by itself run GUI applications like gedit. To allow this you will need to install XMing. Once installed, in PuTTY go to the X11 options and select enable X11 forwarding. When you now connect you can use graphical applications. The third program WinSCP allows you to transfer files between your computer and CSE. It has a similar configuration to PuTTY and should be simple to set up.

Cygwin

Internet not required

Win Lin Mac difficulty: insane

In addition to the remote connection setup you will need a third program called Cygwin. It is difficult to install correctly and has a higher learning curve than the above. If you are willing to fiddle with it, during the install - search for and select Clang which is under devel. It will take a while to install completely so let it run. If you have a very big hard drive and are exceptionally keen you can install other/all packages Cygwin offers. You can add packages later by using the
same installer file and then selecting more packages, it will only add more packages rather than reinstall everything. Once installed you can now use the Bash commands like in the labs. To get to your own files you will need to use:

\[
\text{
\$ cd /cygdrive/c
}
\]

This will take you to your C drive folder and you can work as normal from there. Note that the tilde will take you to the special Cygwin home folder and not your actual home folder (it can be changed but you will have to research how by looking into .bashrc and $HOME variable). Also note that when you go to compile you will need to use clang instead of gcc:

\[
\text{
\$ clang -Wall -Werror -O -o output source.c
}
\]

It is possible to change a file in the Cygwin home directory called .bashrc and make your actually home directory work as you would expect by exporting the HOME variable.

### Mac OS X Developer Tools

*Internet not required*  
*Win Lin Mac*  
*difficulty: hard*

Mac OS X is UNIX-based, so most things you can do on a CSE workstation, you can also do natively on your Mac. However, Apple don’t ship a C compiler by default, so you’ll need to install it. There are two options:

1. Open a Terminal (:Applications:Utilities:Terminal), run
   \[
   \text{
   \$ xcode-select --install
   }
   \]
   and click ‘Yes’ when required, to install a C compiler and a range of other useful command-line tools.
2. Install Xcode from the App Store.

You can then code locally, but you’ll need to copy files to CSE to submit. See the section on Secure Shell (SSH) above.

### Cyberduck SFTP

*Internet required*  
*Win Lin Mac*  
*difficulty: medium*

Cyberduck is a graphical application to transfer files. To use Cyberduck, click on Open Connection, change the protocol to SFTP (Secure File Transfer Protocol), and set the server to “login.cse.unsw.edu.au”. The username and password will be your zID and password. Leave everything else as default and click connect. It should work if done correctly and you will get a Finder-like list of all your files on the CSE server. From here you can literally drag and drop files both ways between CSE and your home computers.