FIT2100 Laboratory #1
Introduction to Linux and C Programming Basics
Week 2 - Semester 2 - 2023

July 21, 2023

Revision Status

Written and first maintained by Dr Jojo Wong, 2018.
Updated by Dr. Charith Jayasekara, July 2023.

Acknowledgement

The majority of the content presented in this lab was adapted by Dr Jojo Wong from the courseware of FIT3042 prepared by Robyn McNamara.

Contents

1 Background

Knowing how to program in C is essential for implementing the various concepts of operating systems that we are going learn in this unit. In this first lab, you will be introduced with the fundamental elements of the C programming language, as well as how to compile and execute C programs.

Before attending the lab, you should:

- complete the following reading (Section 2)
- attempt assessed tasks given in Section 3

Note:

This week's lab is assessed and you will be assessed both on your preparation prior to class and work demonstrated in class.

2 Pre-lab Reading

2.1 A Brief History of Unix/Linux

A brief history of Unix/Linux is available on the FIT2100 Moodle site here. 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 itself, type the following command:

```
1 $ 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.)

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.

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 `/`	followed by a search query to find a word in the man page
Press `n`	after a search to find the next search result
Press `N`	after a search to find the previous search result
Press `q`	to quit and return back to the shell prompt

Try using the `man` pages 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:
  1. `ls`
  2. `more`
  3. `cat`
  4. `vim`

2.3 C Programming Manual

If desired, you can also install additional *sections* into the man page system, including a reference guide for each of the standard C library functions (section 3) and OS-specific functions in C (section 2). To install these, enter the following command into the Unix prompt:

```
$ sudo apt install manpages-dev manpages-posix-dev
```

You will be prompted to enter your login password for the Linux system. *The password will not be displayed*. When asked to continue, enter `Y` and press Enter.[1] A summary of the `man` sections can be found below. Generally, you will find the first three sections of more relevance to you:

Section	Description
1	Shell commands (i.e. ls)
2	System calls (low-level OS/kernel functions)
3	Library functions (C standard library)
4	Special files
5	File formats and conventions
6	Games
7	Miscellaneous
8	System administration commands

Try `man printf`. Is this helpful? Ooops! This is the man page for the Linux command `printf`. We want the man page for the C function `printf` instead!

So, press `q` to quit, then try `man 3 printf` to specify that you want a man page for the C standard library (section 3). While man pages are always highly technical, this man page is more complicated than most! There is a whole family of different `printf` functions in C, such as `fprintf`.

Beware! The synopsis section of the `printf(3)` man page shows how the function prototypes are *declared* for `printf` in C (e.g. the data type of each argument to be provided), not literally how to call them. The very bottom of the man page usually contains an example or two on how to actually call one of the functions described. This is reference material: you generally won't want to read the whole man page at once! These man pages are very useful for quickly looking up details such as what #include statements are required for the function you need, what arguments are required, and the data type of the function's return value.

[1] If you get an error, run: `sudo apt-get update` first, and leave it to run until it completes. You will require internet access, and cannot install software while other updates are running.

2.4 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
```

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

You can also *auto-complete* commands, filenames, and directory names by typing the start of the name and pressing TAB. If nothing happens or the filename is not completed fully, it means there are multiple files with similar names available; just press TAB a second time for a list of possibilities. Learning to use the TAB key will save you a lot of typing when navigating the command environment. You may even find it quicker than using the desktop file manager!

Before trying out some commands, create a directory (or folder) named `FIT2100` under the Documents directory. Then, create another directory named `LAB01` under the FIT2100 directory (`~/Documents/FIT2100`), then change to this `LAB01` directory, using the following commands:

```
$ cd Documents/
$ mkdir FIT2100
$ cd FIT2100/
$ mkdir LAB01
$ cd LAB01/
```

After completing the above commands, confirm which directory you are in with the following command:

```
$ pwd
```

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
```

2.4.1 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
```

Sometimes, you may forget about your username, especially when you have a number of accounts on a particular system. Try the following command:

```
$ whoami
```

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 (1)    - 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 it 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"
```

Try finding the appropriate commands/utilities yourself:

- Which Unix command converts a PDF file 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.)?

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 2018
$ 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, with each line referring to a different user. Even if you're the only user on the system, a lot of background processes running on the operating system run under their own dedicated user accounts for security reasons. Let's take a look:

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

Often times, you may want to search a file for a pattern and display all the lines (of the file) that contain the given pattern. `grep` is a useful utility for locating words (or other patterns) in files. If your username on the system is 'student', let's try to find information about your account inside the `passwd` file. Try the following command with the `grep` command:

```
$ grep student /etc/passwd
```

2.4.2 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. The `[Ctrl-D]` key is used to signify the "end of file" (EOF).

Note: To *interrupt* a running program, you can use the `[Ctrl-C]` key. This will then cause the running program to terminate. This is *generally* a good way to terminate a program that is stuck in an infinite loop.

2.4.3 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
```

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
...
40  whatis history
41  man history
42  history
```

Notice that each stored command in the history list has a sequence number which actually allows you to easily repeat commands. To repeat a command, type `!` followed by its sequence number. This especially useful when you would like to re-run a complicated command that you have run some time ago, without having to retype it.

```
$ !40
whatis history
history (3readline) - GNU History Library
```

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
```

2.5 The Unix File System

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

Terminology 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 content of a file.

```
1  $ file /bin
2  $ file /bin/bash
3  $ file /etc/passwd
```

2.5.1 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 an empty 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.

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

Try using commands to work with files and directories 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
```

2.5.2 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 -l`. Make sure you know what each piece of information means. In the left column is a list of permission bit flags for each file or a directory. For example, if a file has permission bits: `-rwxr-xr-x`, it means (reading from left to right) that the *owner* of the file can read `r`, write `w` and execute `x` the file[2], while the group of users the file belongs to can read `r` the file, cannot write `-` to it, but can execute it `x`. Everyone else (last 3 flags) can also read, cannot write, but can execute this file.

[2] only executable programs and directories should normally have executable permission set

Try working with file permissions yourself:

1. Under the directory `~/Documents/FIT2100/LAB01` make a subdirectory called `test_dir`.
2. Create an ordinary file called `file.txt` under `test_dir`.
3. Check the current file attributes (especially the permissions) of `file.txt`.
4. 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. When you convert each octal digit into binary, it matches the three permission bits mentioned above. The three octal digits correspond to permission bits for the owner of the file, user group and everyone else respectively)

```
$ 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
```

Another way to use `chmod` is with the + (add permission) and – (remove permission) operators. For example, `chmod +x file.txt` will make the file executable for all three categories of users.

2.6 The Shell for Unix/Linux

The command-line interface you have been using inside your terminal is more correctly called a 'shell' utility.

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 (short for `Bourne Again SHell`) is the default shell in your Linux environment. The shell is actually a primitive kind of programming environment (not to be confused with the C Programming Language), used to interact with the system.

2.6.1 Aside: Pre-processing of Unix commands

Let's look more closely at what happens when you type a command into the shell prompt in a Linux terminal...

- 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 (the shell can make use of special system variables known as 'environment variables'). 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, above your TAB key.)
- 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.

If you want to know what error value the last program returned when it terminated, you can use the following command. By convention, any piece of software on the machine that completes running successfully should return a value of 0.

```
$ echo $?
```

2.7 Shutting Down

Like any modern computing environment, your virtual machine environment must be shut down safely when you are done using it. Always shut down your virtual machine at the end of your session, and make regular backups of any important work!

2.8 A Simple C Program

2.8.1 Writing C Programs with Unix Commands

Under the directory ~/Documents/FIT2100/LAB01, do the following:

- Use the following command to create a file: $cat > prog01.c

```
$ cat > prog01.c

#include <stdio.h>
int main (void)
{
    printf("This is the first FIT2100 practical.\n")
    return 0;
}
```

- Finally, use [Ctrl-D] (which again, indicates an 'end of file') 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: This is a difficult way to work with files! You will probably wish to use a text editor, either console-based such as pico, vim or joe, or the graphical editors pluma, or subl (non-free) to work with text files.

2.8.2 A Simple C Program

This C program simply displays a statement "`Welcome to C Programming!`" to the standard output device (i.e. the console or the terminal).

```c
/* simple.c: a simple C program */

#include <stdio.h>

int main()
{
  printf("Welcome to C Programming!\n");
  return 0;
}
```

Each C program is defined with the **main** function since it is always the first function to be executed.

The `main` function often returns an integer value (`int`) indicating the status of its execution. Typically, a return value of `0` implies normal program termination (i.e. no errors); non-zero values indicate unusual or erroneous execution.

The `main` function often accepts two *arguments* or *parameters* (which we will see later in Section 2.16). However, in the program given above, no arguments were specified in the `main` function, as indicated by the empty parentheses `()` in the function heading.

Note: Each statement in C programs is terminated with a semicolon (`;`). Blocks of statements are enclosed in a pair of braces (`{}`).

How to create a C program? C source code files are simple text files. You can use any text editors of your choice to type in or edit 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 a 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` argument followed immediately by

the desired output file name. The second command sets the name of the executable file to `simple`.

How to run the program? You can now run the executable file with one of the following commands, based on the name of the executable file: (Note: the `./` in front of the executable file name. This specifies that the file to be run is located in the current directory. If it weren't, a different path could be specified.)

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

What is #include <stdio.h>? C provides a standard I/O library of functions for input and output. The library header `<stdio.h>` needs to be included and brought into the source file that intends to perform any input and output by using the directive in C — **#include**.

As for the `simple` program seen earlier, the `#include` statement is important as the compiler needs to know that the `printf` function takes a string as an argument.

2.8.3 Aside: How Do The Compiler and Linker Work?

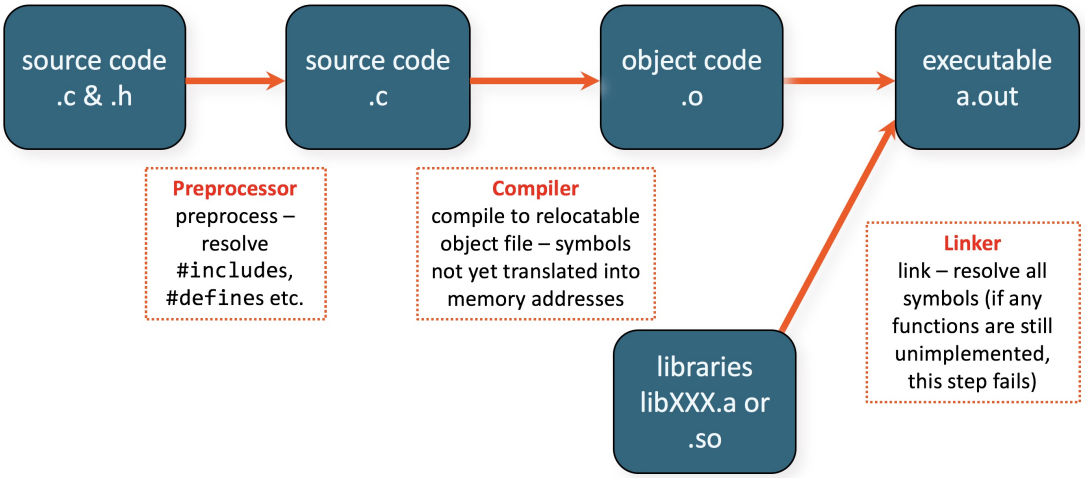
There are three main steps involved in compiling a C source code into an executable program:

Pre-processing 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 resolved 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 and 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 execution; 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.



(Adopted from FIT3042 Courseware by Robyn McNamara)

2.8.4 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`	embed debugging information inside the executable file
`-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`

2.9 Using valgrind to Find Memory Errors

In C programming, you may often experience undefined behaviour due to accessing memory incorrectly, or a *segmentation fault* where the operating system shuts down a program that has tried to access a forbidden part of memory. These errors can be frustrating and hard to pin down. For example, going past the end of an array in one part of your program might accidentally corrupt a different variable in a different part of your program!

The `valgrind` utility can help. This utility runs your program in a partly-simulated environment to check if your program is accessing memory safely. If your program has been compiled in `gcc` with the `-g` option, `valgrind` can even pull debugging information out of the executable file to tell you which line numbers in your original C code the problem might have come from. If your executable file is named `a.out`, you can run your program through `valgrind` like this:[3]

```
1 $ valgrind ./a.out
```

[3] Run `man valgrind` for the complete user manual.

2.10 Directives in C

Before the compilation process happens, C programs are first edited by a preprocessor. Directives (which always begin with the # symbol) are the commands intended for the preprocessor.

In addition to the `#include` statement that we have seen, *constants* in C programs can be defined with a specific value using the directive `#define`.

```
#define MAX_LENGTH 9
```

Any occurrence of the constant `MAX_LENGTH` within a program will be replaced by the corresponding value specified (such as 9 in the given example).

2.11 Variables in C

In C, variables must be *declared* before they are used. The common practice is to perform the variable declaration at the beginning of each function before any executable statements. (This is however not a rule that you must follow).

```
int i=0, j=1, k=2;
float pi = 3.1415926553f;
double e = 1.0e-32;
char c = 'a';
char newline = '\n';
```

All the variables have a specific data type which is set at the compile time, and it cannot be changed — because C is statically typed. However, implicit and explicit type conversions can be performed on C variables.

```
int i = 5;
float f = i;
int j;
j = (int) f;  //casting f into an int variable
```

Note: Variable names may contain letters, digits and underscores; but must not begin with a digit. Also note that variable names are case sensitive, thus `x` and `X` refer to two different variables.

2.12 Data Types in C

Data types in C are not guaranteed to be of a fixed size. The following are the primitive data types in C presented in the order of size.

- Integer: `char`, `short`, `int`, `long`, `long long`
- Floating point: `float`, `double`, `long double`

To determine the size of a particular type or variable, you can use the `sizeof` operator provided in C.

C does not have the built-in `string` type unlike other programming languages (such as Java and Python). A string in C is in fact an *array* of `char` terminated with the `null` character `'\0'`.[4]

2.12.1 Quotation marks in C

Note that there is an important difference between single and double quotation marks in C:

- **Single quotation marks** around a single character, like `'a'`, are used to represent the character code (`char` value) of that character.
- **Double quotation marks** are used for a string literal `"like this"`. When you use a string literal, C creates that sequence of characters, followed by a `null` character, as a hard-coded constant in program memory. (Technically, it is represented by the memory address of the first character) You can then access this just like an array of characters, assign it to an array, etc., but your program cannot modify the characters in this location of memory directly.

[4] In C, there's often no way to get the size of array data, since C does not waste memory by storing such information! Therefore the `null` character at the end is used as a way to determine the length of the string, and ensure the program does not keep reading undefined RAM past the end of the string! Functions like `printf()` look for this character to know when to stop printing.

2.13 Arrays in C

C supports both single-dimensional and multi-dimensional arrays. Array subscripts in C always start at 0.

```c
#define MAX_LENGTH 9

int a[10];     /* one-dimensional array of size 10 */
a[2] = 3;      /* initialise the third element */

char unit_code[MAX_LENGTH];    /* string with 9 chars */
char unit_name[] = "OS";       /* string of size 3 including '\0' */
```

```c
#define ROW 10
#define COL 10

int a[ROW][COL];     /* two-dimensional array of 10x10 */
a[9][9] = 100;       /* initialise the last element*/
```

Note: Multi-dimensional arrays can also be achieved by using arrays of **pointers** to other arrays (which we will cover in the next lesson). An example of this is shown below.

```c
char *argv[];        /* arrays of pointers*/
```

2.14 Enumerated Types in C

C supports another type of constant through *enumeration*. It is a convenient way to associate a list of constant values with names. The first name in an enumerated list is associated with the value of 0, the second name with 1, and so on; unless the values are explicitly specified.

```c
enum boolean {NO, YES};    /* NO is 0, YES is 1 */

enum months {JAN = 1, FEB, MAR, APR, MAY, JUN,
                  JUL, AUG, SEP, OCT, NOV, DEC};    /* FEB is 2, MAR is 3 */
```

Note: The names must always be distinct, but the associated values need not be.

2.15 Operators in C

Operator	Description
=	assignment
+ - * /	add, subtract, multiply, divide
-	unary minus (negation) — e.g. x = -y
%	modulus (remainder)
++ --	increment, decrement — e.g. x++; --y;
== !=	equals, not equals
< > <= >=	less than, greater than, less than or equal, greater than or equal
&& || !	logical AND, OR, NOT
& | ^ ~	bitwise logical AND, OR, XOR, NOT
<< >>	bitwise left shift, right shift

2.16 Functions in C

Typically, a C program may consist of multiple functions which can possibly be declared in multiple source files. Like variables, functions must be declared through function *prototypes* before they are used. The prototype of a function gives only the function name, its arguments, as well as the return type; without the function body (i.e. the statements of the function).

The function prototype must agree with the function definition in terms of the number of arguments and their types as well as the return type. However, the argument names need not agree given that they are in fact optional in the prototype.

```c
/* foo.c: the source file */

#include "foo.h"

int foo(int a, int b)   /* function definition */
{
  /* ...function code here... */
  return 0;
}

int main(int argc, char *argv[])
{
  int result = foo(1, 2);
  return 0;
}
```

```c
/* foo.h: the header file */

int foo(int a, int b);    /* function prototype */
```

As a common practice, function prototypes are often declared in the header file (the `.h` file), while the function definitions are placed in the source file (the `.c` file). If the prototypes are declared in the source file, they should be placed before all the functions (including the `main` function) — at the beginning of the source file.

Note: The header file needs to be included (e.g. `#include "foo.h"`) in the source file where the body of a function is defined. Likewise, whenever a function needs to be used (invoked) in another source file, the corresponding `.h` file must be included to provide the compiler with the necessary information about the function.

The main function Recall that the execution of a compiled C program always begins with the function named `main`. The main function often accepts two parameters which allow you to pass on a number of command-line arguments to a C program when it begins executing.

- `argc`: a count of the number of command-line arguments the program is invoked with;
- `argv`: an array of strings representing the arguments, with the program name as the first argument.

Running the following C program (`example.c`) with the command: `./example hello world` prints "`hello world`" as the output.[5]

```c
/* example.c: display command-line arguments */

#include <stdio.h>

int main(int argc, char *argv[])
{
   printf("%s %s\n", argv[1], argv[2]);  /* argv[0] is the program name */
   return 0;
}
```

Note: Arguments in C functions are passed "by value" — the called function is given the copies of the original arguments.

[5] As described in Section 2.8, you will need to first compile `example.c` (and all other source files in this lab sheet that follow).

2.17 Variable Scopes

Variables in C have different scopes (i.e. where they can be referenced) within a program, which are determined by *where* and *how* they are declared.

Global scope Variables declared outside all the functions have the global scope and are visible throughout the entire program; thus they are known as global variables.

Block scope Local variables are those declared within a function (i.e. the braces surrounding a function body). They are accessible from the point at which they are declared to the end of the enclosing function body; hence they are not visible to other functions.

Local variables that are defined with the `static` modifier still have the block scope within the function in which they were declared; however, their values *persist* between function invocations.

```c
/* scopes.c: global scopes and local scopes */

#include <stdio.h>

void func(void);

int main(void)
{
        func();   /* x = 1, y = 1 */
        func();   /* x = 1, y = 2 */
        func();   /* x = 1, y = 3 */
}

void func(void)
{
        int x = 0;
        static int y = 0;

        x++;
        y++;

        printf("%d %d\n", x, y);
}
```

Note: Global variables are often defined once in one source file. To make global variables visible in other source files, the `extern` modifier is used during the declaration in the header file.

2.18 Basic I/O in C

As we have seen in the Section 2.8, the standard I/O library (`stdio.h`) provides a set of functions that enable high-level input and output.

Three stardard I/O streams are defined in `stdio.h`:

- `stdin` (the standard input),
- `stdout` (the standard output), and
- `stderr` (the standard error output).

For most of the `stdio` functions, a reference (pointer) to an open I/O stream must be passed as an argument; however, there are a number of "shorthand" functions that assume one of these streams instead of requiring them to be specified.

The printf function The `printf` function is an example that we have seen, which is used to write to the `stdout` stream. It is a formatted output function that converts, formats, and prints arguments to the standard output.

```c
/* printing.c: sending output to stdout */

#include <stdio.h>

int main(void)
{
  int number = 1;
  char unit_code[] = "FIT2100";

  printf("This is Tutorial %d for %s\n", number, unit_code);
  return 0;
}
```

The first argument in `printf` is the format string which may contain two types of characters:

- literal characters — any ASCII characters;
- formatting characters (conversion specifiers), with each of which converts the next matching argument (variable) into the desired output format specified using the % directive.

Format Specifier	Description
%c	single characters
%d or %i	signed decimal integers
%e or %E	floating-point numbers in scientific format
%f	floating-point numbers in decimal notation
%d	double precision floating-point numbers in decimal notation
%g or %G	either %f or %e (%E) is used, whichever is shorter
%o	unsigned octal integers
%p	pointers
%s	character strings
%u	unsigned decimal integers
%x or %X	unsigned hexadecimal integers using a-f (A-F)
%%	the percent sign

The scanf function To read characters from the standard input, the `scanf` function is used. Each of the conversion specifiers (%) must have a matching reference (pointer) specified in the argument list — variables prefixed with &, which the `scanf` function is used to store the input values it interprets.

```c
/* scanning.c: receiving input from stdin */

#include <stdio.h>

int main(void)
{
  int day, month, year;

  printf("Enter the date: ");
  scanf("%d %d %d", &day, &month, &year);

  printf("%d-%d-%d\n", day, month, year);
  return 0;
}
```

Note: Variables where input values are assigned to usually require the & to appear before them. However, you will notice that this is not always the case, in particular when reading strings which are arrays. (This is related to the concept of **pointers** which we will explore in the next lab.)

2.19 The General Structure of C Programs

```
#include headers
#define constants

/* function prototypes */
void func_A (int a, int b);
. . .

/* declaration of global variables */

int main(int argc, char *argv[])
{
  /* declaration of local variables */
  /* statements of main*/
}

/*function definitions */
void func_A (int a, int b)
{
  /* declaration of local variables */
  /* statements of func_A*/
}

. . .
```

2.20 Additional References

– Kernighan & Ritchie, *The C Programming Language*: Chapter 1: 'A lab Introduction'

3 Assessed Tasks

Complete the following tasks, writing your answers (or what you did) in a plain text file, and get your answers marked off by your tutor before you leave. Also, do take note of the submission instructions in Section 3.5.

3.1 Task 1 (40%)

Experiment with navigating to the following paths using the `cd` command. After navigating to each directory, use the `ls` command to look at the files in that location. Are you able to navigate to all of these paths? Does it matter what working directory you are in before running the `cd` command?

- `/home/student/Documents`
- `FIT2100`
- `~/Documents/FIT2100`
- `.`
- `..`
- `.././.././home/student`
- `/`

After experimenting, answer the following questions in a plain .txt file:

1. Which of the paths above are absolute, and which of them are relative paths?
2. What are the advantages of using absolute paths over relative paths? Why and when, if ever, would you choose to use relative paths over absolute paths?
3. Write `cd` commands that allow you to achieve the following:
   - navigate to one directory above the user's home directory
   - no change to your current working directory
4. To run programs in your current directory, you will need to place a `./` in front of the name of a program– why is this required? Assuming you have an executable named `gcc` in your current directory, describe what might happen if we were able to run programs in the current directory without needing to prepend a `./` to the program name.

3.2 Task 2 (20%)

Write a C program that asks users to enter their first name and last name. The program then displays the *initial* of the users based on the first letter of the first and last names. You may assume the maximum number of characters for each string is 20.

3.3 Task 3 (20%)

Write a C program that reads the side lengths (length, width and height as integers) of a rectangular prism. The program then computes and prints the surface area and the volume of the prism, with two decimal places of precision.

3.4 Task 4 (20%)

Write a C program that reads two integer values and performs the four basic arithmetic operations on them: *addition*, *subtraction*, *multiplication*, and *division*. Each of these operations should be implemented as a separate function.

3.5 Wrapping Up

Please upload all your work to the Moodle submission link, which should include your .c source files and your responses to any non-coding tasks. Please note that non-code answers should be provided in plain text files. There is no need to include the compiled executable programs. Ensure to upload all the necessary files before the conclusion of your lab class. Any delay in submitting these files to Moodle will result in a late penalty.