CAB202 Tutorial 1

Introduction to compiling and structured programming

The C programming language is designed to map efficiently to the machine code of the embedded microcontroller you will be using later in the semester, but provides the capabilities of a high level algorithmic language. By the completion of this tutorial, you will be able to write, compile, and execute basic C programs that utilise both control structures and types. This tutorial counts for 3% of your grade.

NOTE: Before starting this tutorial, you must have completed the software installation process outlined in the "Before your first tutorial!" item on Blackboard

Tutorial structure

The C programming tutorials will follow the same general structure. They will start with a small section detailing any specifics necessary for this tutorial. These tips do not provide a complete explanation – concepts are covered in full detail, with guided examples, in the corresponding lecture. The second section is a series of non-assessable exercises that your tutor will explain solutions to in your tutorial (you should attempt to complete the exercises **before** the tutorial). The last non-assessed question will always have no provided template code to help develop your ability to break complex problems down into code-based blocks. Also, some of these exercises have extra challenge exercises – if you are aiming for anything more than a pass in this unit it's highly recommended you attempt to complete these exercises! The final section links to the Automatic Marking System (AMS) where you are to submit your solutions to the assessed exercises.

Introduction to C programming ('Hello World')

The general purpose of programming is to create machine code that carries out a set of desired tasks. This machine code (a binary or executable file) is essentially a sequence of 1's and 0's used to control the CPU (central processing unit) of a computer. To create a piece of machine code, there are three distinct steps:

1. Creation of C source code
2. Compilation of the code
3. Execution of the code

1. Creating source code

To start creating a C program, like any program, you first write some form of source code (a file with ***.c** file extension). This extension tells the system how the information in the file is expected to be structured. For example, a 'hello world' program might exist in a source file called:

```
hello_world.c
```

Once this file is created, it must be edited to include the source code. This can be done in any text editor (a list of these is available on Blackboard under "Software Resources").

A C program requires one essential component to run, a **main** function. While functions will be introduced later, think of the **`main`** function as an entry point. When running your program, this function provides the entry point into your code (open '**{**' bracket) and exit point when it is completed (the **`return`** statement or close '**}**' bracket). If the following code were compiled and run for example, the computer would enter your code and then return:

```
int main() {
     return 0;
}
```

This working program currently does nothing, so let's change that by writing **Hello World!** to the console. To write any text to the console, a function called **`printf()`** is used. This function writes whatever **String** is supplied in the first argument. In C, **Strings** can be specified by enclosing quotation marks. For example, the follow code would print **Hello World!** to the console:

```
printf("Hello World!");
```

The **`printf()`** function also allows much more complicated behaviours. The following examples show only a few of the capabilities of the **`printf()`** function. A more complete list can be found at http://www.cplusplus.com/reference/cstdio/printf/ .

1. Print a string and then move to the next line ('**\n**' character does this):
   ```
   printf("Hello World!\n");
   ```
2. Print the value of a **integer** type variable called 'countVar':
   ```
   printf("The count is %d", countVar);
   ```
3. Print the value of a **float** type variable called 'piVar':
   ```
   printf("The value of pi is %f", piVar);
   ```
4. Print the values of two **integer** type variables called 'firstVar' and 'secondVar':
   ```
   printf("The values are %d and %d", firstVar, secondVar);
   ```

Before we can use this function, we first need to include the library (**`stdio.h`** – standard input and output) in which it resides. To include a library we use the **`#include`** directive at the start of the source code file (before the **`main`** function). Consequently, the following code would compile to make a complete program that writes **Hello World!** to the console:

```
#include <stdio.h>

int main() {
     printf("Hello World!");

     return 0;
}
```

This is now a complete C program. Now, before the program can be run, we need to compile the program and turn it into a language the computer hardware can understand.

2. Compiling the source code

A computer does not understand a program in its high level form, such as C, C++, MATLAB or Python source code. The code must first be changed it into a binary file that the computer can read and execute. This is done through the process known as code compilation. To compile C code, we use the **gcc** compiler which came out of the GNU project that started in 1984.

To compile a simple program with **gcc** open a console/terminal window and navigate to the directory where the program source code (**hello_world.c** in this example) is located. This is done with the **cd** command. Then run the **gcc** compiler on the source code file to produce a **hello_world** executable. For example, to compile source code that is located in **/home/you/hello_world/**:

```
cd /home/you/hello_world/
gcc hello_world.c –std=gnu99 –o hello_world
```

The second command is telling the compiler to compile `hello_world.c` into an executable. The **-o** flag tells `gcc` where to store the output of the compilation process. The **-std** flag specifies the C standard used by the compiler. In CAB202, we will be using the **gnu99** standard. This standard is similar to ISO standard C99, but it has small differences that make our programs more portable between Unix-like environments (and is directly compatible with the microcontroller we use later in the semester). If there were no errors, in either your C program, or in the compile process, you should end up with a **hello_world.exe** or **hello_world** binary file in the same directory as the **hello_world.c** file (you can check this by using the **ls** command after compiling).

3. Executing the compiled code

To run this executable, you have to execute it from the terminal. The following executes a file in the terminal:

```
./hello_world
```

Using the "cab202_graphics" from the ZDK

We will be using a support library called the ZDK to create game-like programs. The ZDK uses a library called ncurses. You don't need to know the details of ncurses, but you do need to include it in your compilation process. To compile a program that uses the ZDK, you need to add further arguments to the command line for gcc:

```
cd /home/you/hello_world/
gcc hello_world.c –std=gnu99 –I<zdk_folder> \
     -L<zdk_folder> -lzdk –lncurses –o hello_world
./hello_world
```

To use text-based graphics, you need to include "cab202_graphics.h", initialise the window, and restore normal terminal behaviour after the program:

```
#include <cab202_graphics.h>

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

```
    setup_screen();
    // your code goes here

    cleanup_screen();
}
```

In the middle is where you put your code. For example, if you want to draw a ball at the top left of the screen, you would add:

```
draw_char( 0, 0, 'o' );
show_screen();
```

If all you include is the code above, you won't see anything happen, because as soon as the ball is drawn the window is closed and normal terminal behaviour is returned. There are a variety of different methods you can use to see what you have drawn, the simplest being to add a never-ending loop into your program:

```
while (1);
```

To exit a program that never ends, you must send an interrupt signal by pressing **`Ctrl + c`**.

Non-Assessable Exercises

NOTE: These exercises are not assessable, and the tutor can help you with them. Assessable exercises are in the following section.

1. Compiling crash course

Download and unzip the **`question_1.zip`** from Blackboard. The folder contains 5 programs and a basic library, within a mess of folders and subfolders. Provide the 5 individual compile commands with `gcc` to compile each of the 5 programs. You **do not** need to modify any of the code, nor move any files! It is crucial that you can compile up to `program_4.c` (at least this level of knowledge will be required for the rest of the semester).

While completing this exercise, you should become familiar with the:

- Basics of using a bash console (https://www.gnu.org/software/bash/manual/bashref.html for more details)
- **`make`** and **`make clean`** commands
- Use of wildcards in compile and make commands

Note: *when trying to get a broad picture of the directory structure, you may find recursive file listing useful (the command is "**`ls -R`**").*

Challenge exercises:

- *Successfully compile **`program_5.c`***

2. Write counting programs for the command console

The **`question_2.c`** template file on Blackboard is set up to step through 5 counting operations – provide the implementation of each of these operations. You can use any elements of structured

programming that you find useful. Complete the following (the limits should be included – i.e. 0 to 3 would be 0, 1, 2, 3):

1. Count from 0 to 12
2. Count from 5 to 10
3. Count from 5 to 0
4. Count in 2's, from 10 to 30
5. Count in 0.25's, from 0 to 2

3. Complete a ZDK program that draws words on the screen based on a character press

Complete the template provided in **`question_3.c`** from Blackboard. The program starts with a screen informing the user that it is waiting for a key press. Provide an implementation after the key is received that does the following:

- If the provided character starts with 'a' to 'e', display a word starting with this character.
- Otherwise, display a message saying that the character is not supported.

Challenge exercises:

- *Adjust the implementation, so that it also accepts the capital letters 'A' to 'E'*
- *Wrap the code in a loop, so that it can be run over and over again*
- *If any numbers are provided, print a string explaining that no numbers are supported (**without** using 10 different cases for each possible digit)*

4. Create a program that divides the screen into 4 sections

Create a program from scratch that divides the screen into 4 sections using one vertical and one horizontal line (where these lines are placed is up to you). The vertical line should be created with "|" characters, and the horizontal line with "-" characters. Where the two lines intersect, a "+" character should be used instead. This program should work on **any screen size**!

Challenge exercises:

- *Adjust the implementation, so that the lines become dashed (i.e. a character is drawn at every second place)*
- *Overhaul the implementation, such that the corresponding arrow key presses move the vertical line left and right, and the horizontal line up and down (**hard**).*

Exercises

Complete the assessed exercises via the AMS (available at http://bio.mquter.qut.edu.au/CAB202).