CAB202 Tutorial 1

Introduction to compiling and structured programming

Background

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 providing 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.

NOTE: Before starting this tutorial, you must have completed the software installation process outlined on Blackboard.

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.

This tutorial sheet will give you general instructions to create a couple 'hello world' programs. And will give you some tips you can use to troubleshoot your code.

Create a file called **`hello_world.c`** by opening a text editor of your choice and naming it as such.

A C program requires one essential component to run, a main function. While functions will be introduced fully later in the coming weeks. When running your program, the main function provides the entry point into your code.

Type the following main function into the .c file you just created.

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

If you compile and run your code so far, the computer will enter your main function and then return. Although this is a working program, you will be unable to see anything happen. So let's change that

by writing “Hello World!” to the console, using a function called **`printf()`**. This function writes whatever **String** is supplied in the first argument. In C, **Strings** can be specified by being enclosed in double quotation marks.

To use **`printf()`**, 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 your source code file (before the main function) type the following:

```
#include <stdio.h>
```

And then inside your main function (i.e. between the “{…}” and before the “return 0”) type:

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

If you compile your code now and execute it, you will see “Hello World!” (minus the quotation marks) printed to the console.

The **`printf()`** function also allows much more complicated behaviours. We will go through some examples but a more complete list can be found at:
http://www.cplusplus.com/reference/cstdio/printf/

Basic C Program Structure

The code below is what you have been instructed to write, so far, as well as an example of how to write comments in C.

Comments are used solely to explain the code to humans and do not affect the running of you program (i.e. computers ignore them). Using comments is good programming practice and may be useful to you in CAB202.

Another thing to note about C is that spaces and indentation do not affect the running of your program. The code in the examples is spaced out and indented for human readability only. It is also good practice to layout your code neatly and have the closing “}” on a line by itself. Indentation is a good way to visually highlight code inside your functions.

```
#include <stdio.h>

int main() {

    /* This is a comment that spans multiple lines. Comments
    are used by programmers to explain their code to
    themselves and to others. It is good practice to comment
    your code. Computers will ignore comments when compiling
    your code. */

    //This is a comment that spans one line.

    printf("Hello World!");

    return 0;

}
```

You have now completed your first C program in CAB202. Before you can run the program, you need to compile the program and turn it into a language the computer hardware can understand.

2. Compiling the source code

Your 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, you will 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/Cygwin window and navigate to the directory where the program source code (**`hello_world.c`** in this example) is located. This is done with the **change directory command (cd)**. 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/ type:

```
cd /home/you/hello_world/
```

After changing directory with **cd**, you can use the **list command (ls)** to check that your file is in the location you have navigated.

```
ls
```

If your file is there, you are in the correct folder and can now compile your code by calling the **gcc** compiler as follows:

```
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 (i.e. the executable file). The output does not have to have the same name as the source file. 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 again after compiling).

3. Executing the compiled code

To run this executable, you have to execute it from the terminal/Cygwin. To execute the file use the name you specified for the output file in your compile command. For example, typing the following executes the file compiled above in the terminal:

```
./hello_world
```

Well done! You have now written, compiled and executed your first C program ☺

Using the "cab202_graphics" from the ZDK

We will be using a support library called the Zombie Development Kit (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**. You can use a Makefile (provided on Blackboard) to compile your code in future, however this tutorial sheet will explain how to write a "hello world" program using the ZDK in a similar manner to the previous section.

The ZDK allows us to create games with text-based graphics. To use this, you need to include the **cab202_graphics** library file and **`setup_screen()`** to initialise the window. To restore normal terminal behaviour after the program, you can use **`cleanup_screen()`**.

```
#include <cab202_graphics.h>
int main(){
	setup_screen();
	//the drawing part of your code goes here
	cleanup_screen();

	return 0;
}
```

If you want to see what functions are available in the ZDK, you can open any of the .h files in a text editor. The .h files contain the function declarations and are commented, with the comments above the function name explaining what the function does. The .c files contain the implementation of the functions so if you are interested in the inner workings of the ZDK, this is the place to look.

To draw "hello world!" to the screen you will need to use a function called **`draw_string()`** and specify an **x- and y-coordinate** for the computer to start drawing at. In this example, the x-coordinate is 10 and the y-coordinate is 20. The coordinate system in the ZDK is a little different to what you're probably used because the **top left hand corner is the (0,0) position**.

```
draw_string(10, 20, "Hello World!");
```

The **`draw_string()`** function saves the String to the screen buffer so your text still won't be visible. To make actually show it on the screen, you need to call the **show_screen()** function.

```
show_screen();
```

Now your program will write your text to the screen. However, you will be unable to see it because it the time between when it is shown on the screen and when the screen is returned to the normal terminal functionality is too small for humans to see the image. So, we have a couple of options to make the text visible.

1. Create an **infinite loop (covered in Topic 2)** so that the **`cleanup_screen()`** function is never called and the program stays in the section of code where you are displaying your text. This option means **your program will never exit** unless you **kill it forcefully**.

To create an infinite loop you can put **`while(1)`** after **`show_screen()`**. We will explain loops in more detail in the coming weeks.

```
while(1);
```

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

2. Change the amount of time spent between showing your text and cleaning up the screen by using a **delay**. This option allows you to choose a time interval that allows humans to read the text and also **allows your program to continue** and **exit gracefully**.

   This option requires you to include another ZDK library file, this time called **`cab202_timers.h`** and use the function **`timer_pause()`** which takes **milliseconds** as input. The example below pauses the program for 3 seconds.

```
//put this at the top of your code before main
#include cab202_timers.h

//put this directly after show_screen()
timer_pause(3000);
```

Now you have written your program, it's time to compile it. You should have already compiled the ZDK as per the set up instructions in the "before your tutorial" section on Blackboard. To check this, look in your ZDK folder and you should see a file called **libzdk.a**. If you do not have this, you will need to compile the ZDK first. The compile command this time includes some more flags because you need to tell the computer to reference the ZDK library and where to find the files. In the command below **"placeholder" is the path to the folder where you have installed the ZDK** (i.e. the libzdk.a file should be in folder "placeholder"). You can specify your folder path using absolute paths or relative paths.

```
cd /home/you/hello_zdk/

gcc hello_zdk.c -std=gnu99 -Iplaceholder -Lplaceholder -lzdk -
lncurses –o hello_zdk

./hello_zdk
```

Well done! You have now successfully compiled a program referencing the ZDK.

Practice and more information

This section contains things that will help you familiarise yourself with C and the types of things we will expect you to be able to do fairly early on in the semester. Don't worry if you find it difficult or confusing at first. Practice makes perfect and your tutors and lecturers are always here to help!

1. Compiling crash course exercises

After completing the compiling crash course you will be able to compile your own programs confidently and be familiar with the basics of using a bash console.

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 exercise:

Successfully compile **`program_5.c`**

2. Printing examples

This section contains some examples on how you can print different data types. This will be useful to you throughout CAB202.

- Print a string and then move to the next line (‘\n’character does this):

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

- Print the value of a integer type variable called ‘countVar’:

```
int countVar = 10;

printf("The count is %d", countVar);
```

- Print the value of a double precision floating point type variable called ‘piVar’:

```
double piVar = 3.14;

printf("The value of pi is %f", piVar);
```

- Print the values of two integer type variables called ‘firstVar’ and ‘secondVar’:

```
int firstVar = 1;

int secondVar = 2;

printf("The values are %d and %d", firstVar, secondVar);
```

3. Troubleshooting tips

Troubleshooting is a skill that is important and is best learnt by doing. As you complete this unit, you will come across error messages, learn to interpret them, and find solutions to the errors. Make full use of your resources including Google searches, StackOverflow and your tutors.

Now that you know how to print values to the console, you can also use printing to troubleshoot. For example, if you want to check the position of an object, you could print its x- and y-coordinates to the console instead of checking visually.

The list below contains general troubleshooting tips based on common mistakes CAB202 students have made in the past.

- **`no such file or directory`**
  - Are you looking / compiling in the correct folder?
  - Is the file named exactly the same in your compile command?
  - Does the file have the correct file extension?
  - Have you saved your file?
- **`undefined reference to 'WinMain@16'`**
  - Have you got a main function?
- **`implicit declaration of function 'function_name'`**
  - Have you included all the ZDK files you need?
  - Have you spelt all your ZDK includes correctly?
  - Have you created your own function after main without declaring it first?
- **`too few arguments in function 'function_name'`**
  - Have you included all the arguments?
  - Have you got all the commas in the right places (i.e. between all function parameters)?
- **`expected ')' before ';'`**
  - Have you missed a closing bracket somewhere?
  - Have you got an extra opening bracket somewhere?
  - Have you got the brackets in the wrong places?
  - Can you highlight things to check your brackets?
- **`expected ';' before 'function_name'`**
  - Have you missed a semicolon at any point before function_name?
- **`expected ';' before '}'`**
  - Have you missed a semicolon at any point in your program?
  - Can you comment out your code, uncomment small sections at a time, and logically go through your program to find which section the error is in?