NWEN 241 Exercise 1

(Basic C Program Development)

Release Date: **28 February 2023**

Submission Deadline: **10 March 2023, 23:59**

1 Objective

The objective of this exercise is to familiarise you with the C development tools used in the course, write and compile your first C program, and learn how to debug C programs.

At the end of this exercise, you should submit the required files to the Assessment System (`https://apps.ecs.vuw.ac.nz/submit/NWEN241/Exercise_1`) on or before the submission deadline. You may submit as many times as you like in order to improve your mark before the final deadline. Submissions beyond the deadline will not be marked and will receive 0 marks.

2 Exercise Requirements

For NWEN 241, it is highly recommended that you undertake all development using the computers in CO246. The computers in this lab use the Linux operating system. *This guide is written with the assumption that you are in CO246 lab.*

If you are not able to go the lab, you can remotely access similar computers via secure shell (ssh). Consult one of the remote study guides (see `https://ecs.wgtn.ac.nz/Courses/NWEN241_2023T1/RemoteStudyGuide`) and follow one that suits you the most.

3 Development Workflow

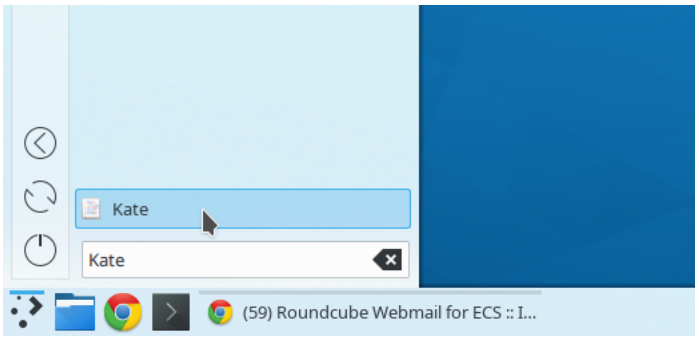
The development process is pretty simple and can be summarised into four major steps:

1. Edit source file using text editor of choice.
2. Compile source file in command line using `gcc`.
3. If bugs or warnings are present, debug and go back to step 1.
4. Run executable output file. If bugs are present, debug and go back to step 1.

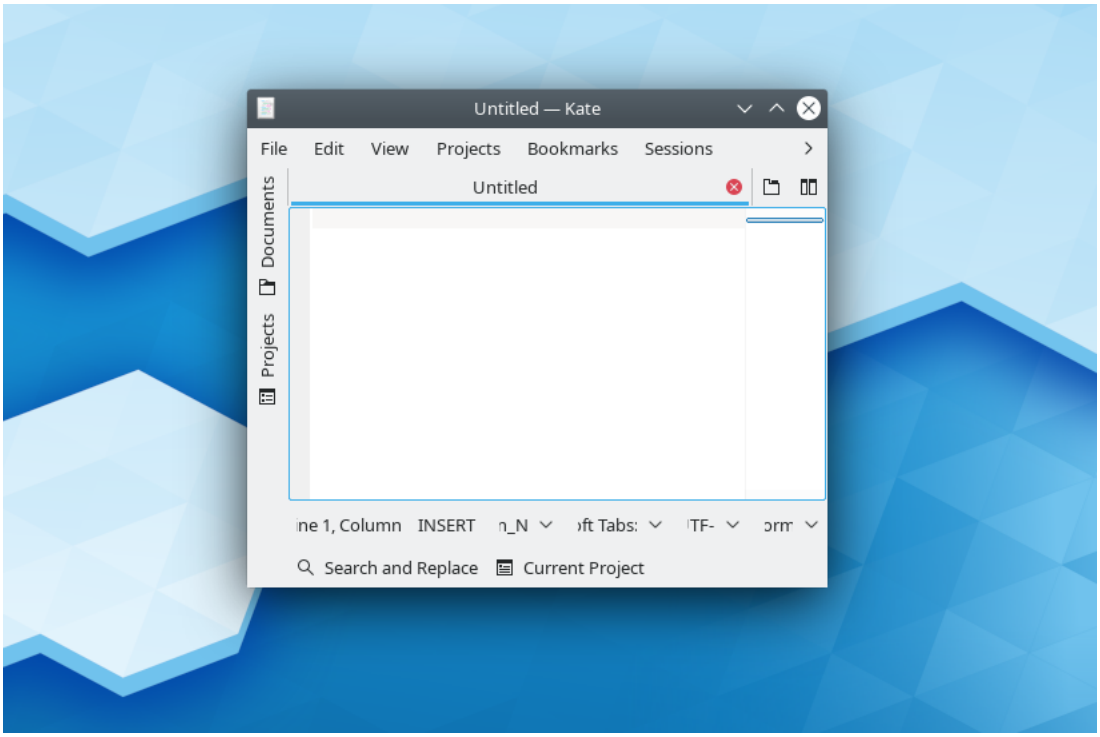
In Sections 4–7, we will go through each and every step of the workflow to get yourself ready to work on the first set of programming exercises.

4 Editing Source Code

You may use *any* text editor to edit your source code. In this guide, we will use `kate`, a no-frills text editor that is already installed in the CO246 lab computers. To open `kate`, click on the Application Menu and type `kate` in the search bar. You should be able to see Kate as an option in the menu.



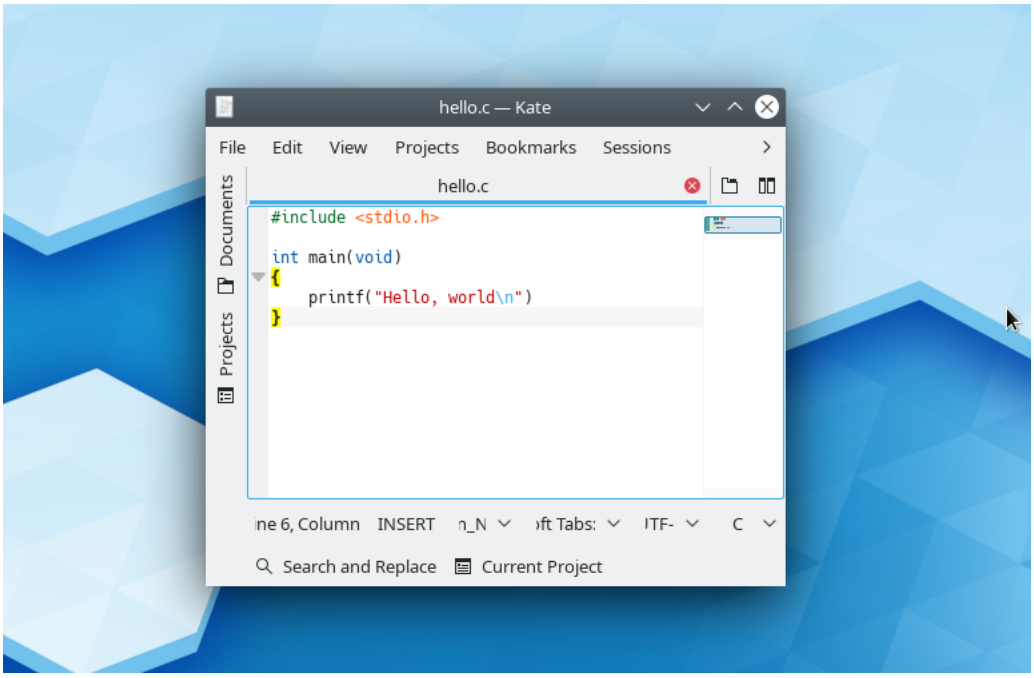
Click on Kate. This should launch the text editor.



Type the following simple C program in the text editor as is. (The above source code has intentional errors. Do not fix them at this time.)

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     printf("Hello, world\n")
6 }
```

Save the file as `hello.c`. Your text editor window should look like this:



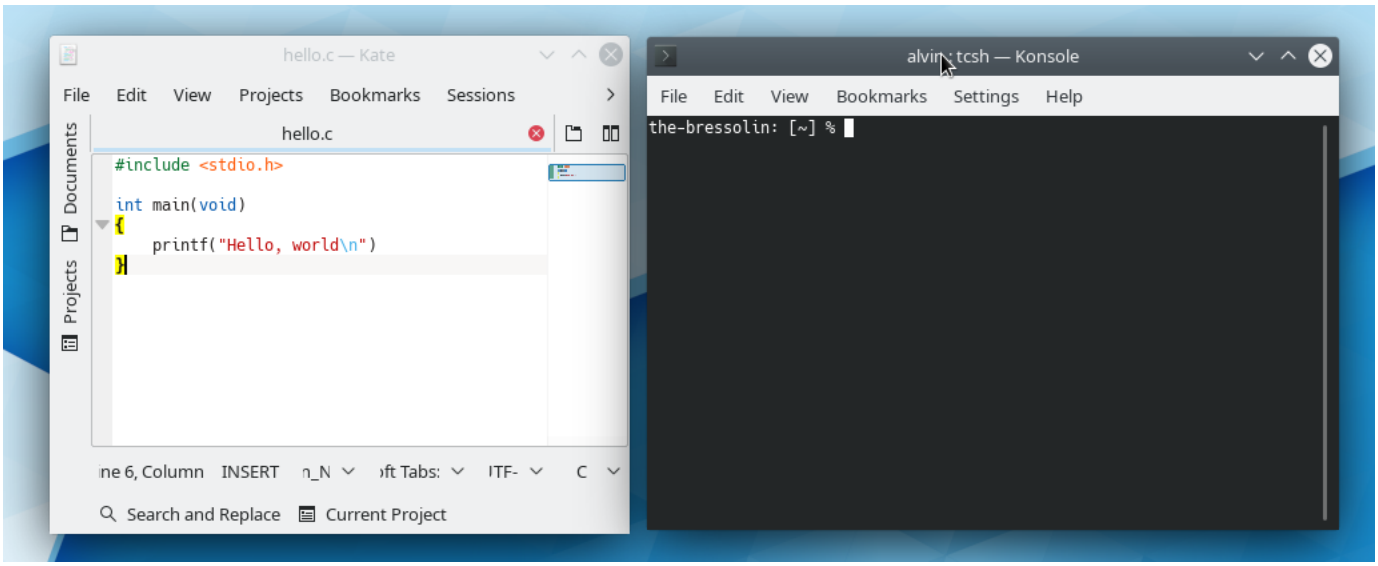
Note that after you saved the file to have the `.c` extension, `kate` automatically performed syntax highlighting. This should make your coding and debugging easier.

5 Compiling on Command Line

You will now attempt to compile the `hello.c` program using `gcc`.

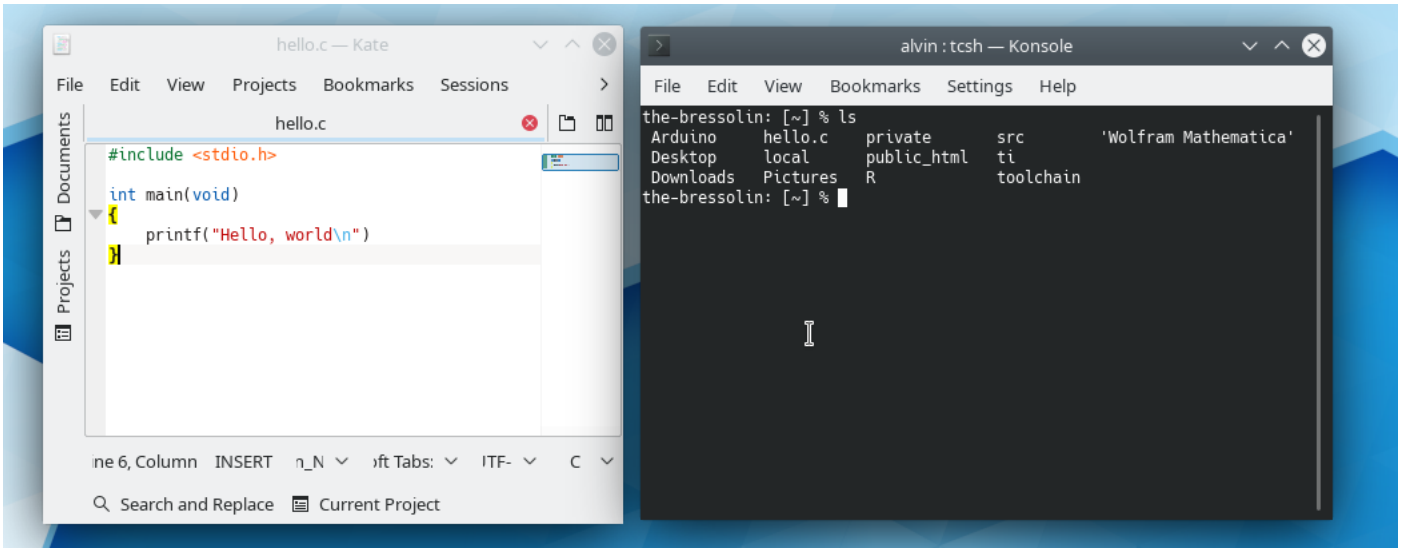
To do this, you will need to open a terminal that will enable you to invoke `gcc`. To open a terminal, click on the `Konsole` button on the taskbar.

This should launch a terminal window. You should now see something like this on your desktop:



The terminal can be pretty scary at first, but once you become familiar with some commands, you should be able to do your programming tasks easily. Section 9 lists down some of the most important commands that you should know.

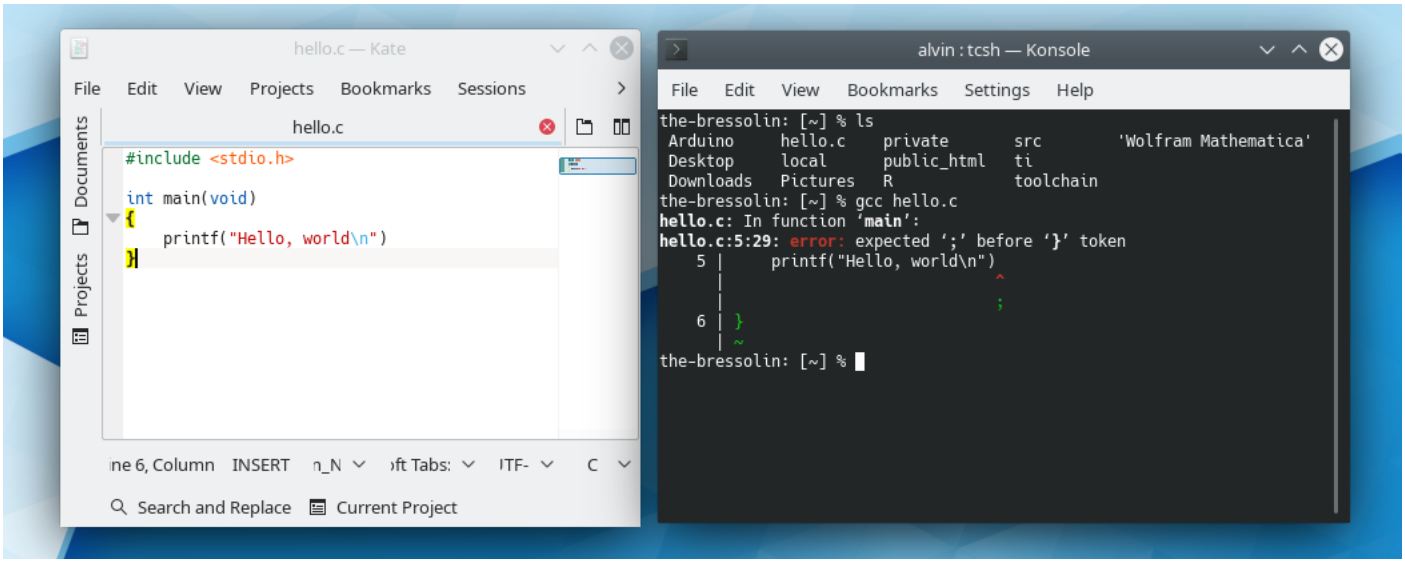
Let's go back to what we want to do which is to compile `hello.c` using `gcc`. Before doing this, ensure that the terminal is in the directory (folder) where the file `hello.c` is located. By default, the terminal will open under your home directory. This is also where you saved `hello.c` using kate (since you did not specify any directory to save it). To confirm that the file is really in the current directory, type the command `ls`. This command lists down all the files in the current directory.



You should see `hello.c` in the listing, hence, we are now certain that the file is indeed under the current directory. We can now proceed to compile it. To do this, type

```
gcc hello.c
```

in the terminal. You should see the following messages from the compiler:



When the compiler spits out error messages, it means that your source code has bugs that need to be fixed.

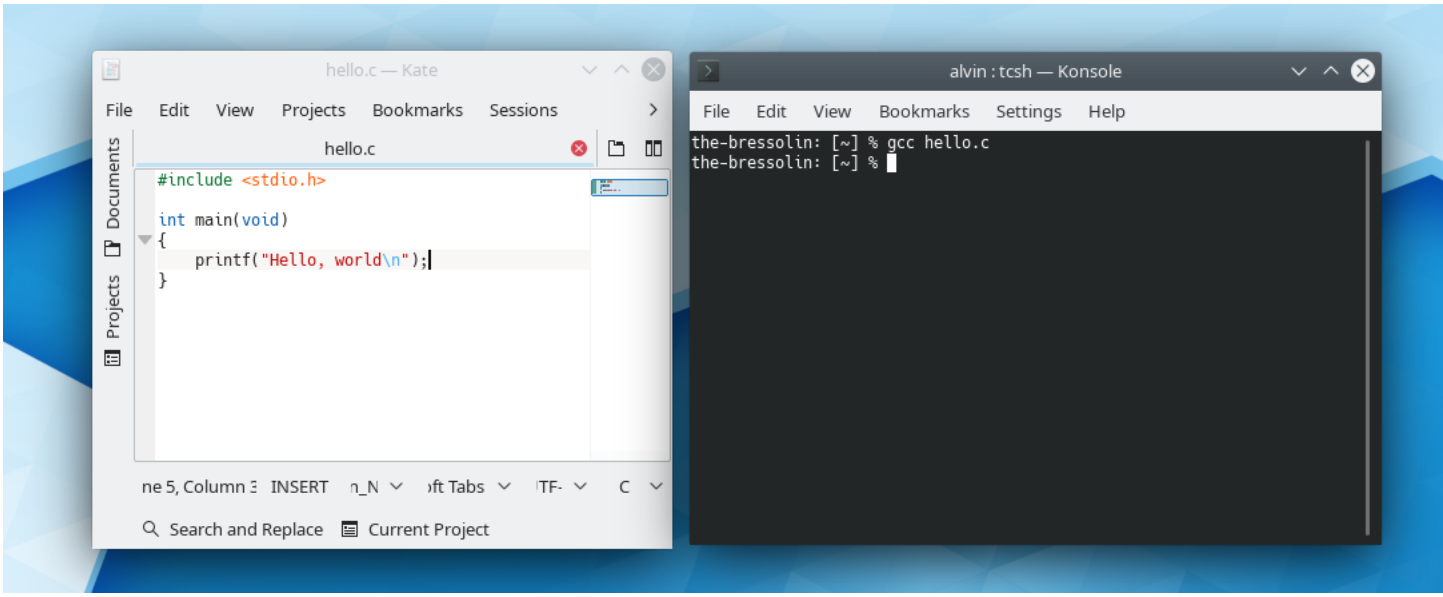
6 Debugging

If the compiler generates error messages, that means that your source code has bugs that you need to fix. To debug your source code, read and understand the error messages carefully.

In the above example, the error says that there is a missing semi-colon (`;`) before the parenthesis. It even suggests where the semi-colon should be inserted. This gives you a good clue on how to fix the code. Basically, you just have to insert a semi-colon at the end of line 5 as follows:

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     printf("Hello, world\n");
6 }
```

Edit your source code to fix the problem and compile again.



As you can see, the compiler did not generate any error or warning message. That means that it was successful and it generated an executable file.

7 Running the Executable File

When you invoke `gcc` as in

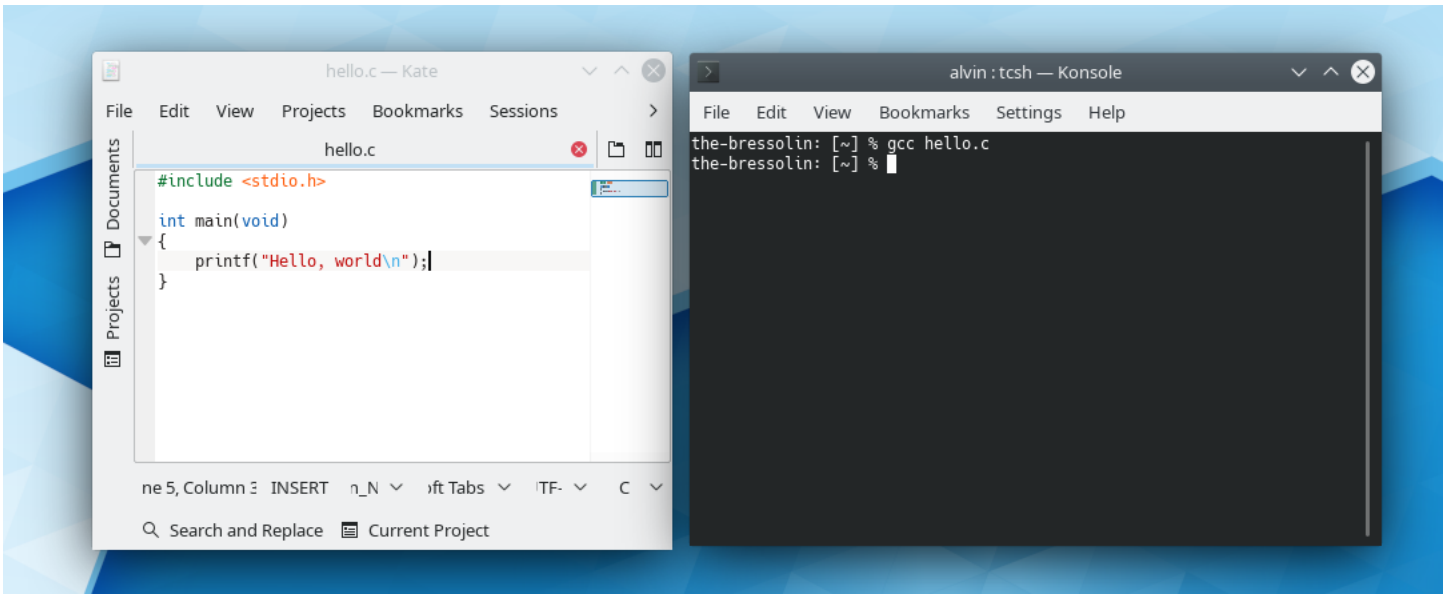
```
gcc hello.c
```

it automatically names the executable file as `a.out`. If you run `ls` again in the terminal, you should find the file `a.out`.

To run the program, type

```
./a.out
```

in the terminal.



It is good practice to explicitly name the executable file. You can do this by using the `-o` option in `gcc`. To compile `hello.c` and generate `hello` as executable file, just type

```
gcc hello.c -o hello
```

in the terminal. To know more about the common `gcc` options, please refer to Section 9.

That completes the basic C development guide in Linux. You will now perform some programming exercises that will require you to submit some files to the Assessment System.

8 Exercises

Activity 1: Basic I/O [10 Marks]

C uses the functions `printf()` and `scanf()` for basic output and input, respectively. These functions can actually perform complicated formatting, but for this activity, you will use them to acquire some input from the keyboard (referred to as standard input or `stdin` in technical manuals) and display something to the monitor (referred to as standard output or `stdout` in technical manuals).

We will talk about `printf()` and `scanf()` in the first tutorial-style lecture, but if you want to start this exercise, you can refer to our reference materials for help on how to use these functions.

For this activity, write a C program that performs the following:

1. Display the message `"Enter an integer:␣"` to `stdout` (␣ denotes a single space character). Do not modify the message, otherwise, the automated marking might not mark your work correctly.
2. Use `scanf()` to read input integer from `stdin`.
3. Display the input integer to `stdout`. Do not add any other string in the output, otherwise, the automated marking might not mark your work correctly.

Save the program as `activity1.c`. Compile and run the program. If you are happy with the program, submit it to the Assessment System for marking. Click on **Run checks** to initiate auto-marking.

Activity 2: Basic I/O [10 Marks]

This is similar to Activity 1, except that you will now handle a string input without whitespaces.

Write a C program that performs the following:

1. Display the message `"Enter a string without whitespaces:␣"` to `stdout` (␣ denotes a single space character). Do not modify the message, otherwise, the automated marking might not mark your work correctly.
2. Use `scanf()` to read input string from `stdin`.
3. Display the input string to `stdout`. (Do not add any other string in the output, otherwise, the automated marking might not mark your work correctly.)

Save the program as `activity2.c`. Compile and run the program. If you are happy with the program, submit it to the Assessment System for marking. Click on **Run checks** to initiate auto-marking.

Activity 3: Basic I/O [20 Marks]

This is similar to Activity 2, except that you will now handle any string input, even those with whitespaces.

Write a C program that performs the following:

1. Display the message `"Enter a string:␣"` to `stdout` (␣ denotes a single space character). Do not modify the message, otherwise, the automated marking might not mark your work correctly.
2. Use `scanf()` to read input from `stdin`. Make sure that strings with whitespaces are wholly captured.
3. Display the input string to `stdout`. (Do not add any other string in the output, otherwise, the automated marking might fail.)

Save the program as `activity3.c`. Compile and run the program. If you are happy with the program, submit it to the Assessment System for marking. Click on **Run checks** to initiate auto-marking.

Activity 4: Basic I/O and Operators [20 Marks]

In this activity, you will use some C operators to perform a computation, and format a floating point output using `printf()`.

Write a C program that will convert an input temperature from Celsius to Fahrenheit. The entire functionality should be implemented within the `main()` function. Follow this guide to implement the program.

1. Declare a float variable `ctemp` which will hold the temperature in Celsius.
2. Declare another float variable `ftemp` which will hold the temperature in Fahrenheit.
3. Display the message `"Enter temperature:␣"` to `stdout` (␣ denotes a single space character). Do not modify the message, otherwise, the automated marking might not mark your work correctly.
4. Use `scanf()` to read input from `stdin`. Store the input in `ctemp`.

5. Perform the actual conversion computation and assign the result to `ftemp`. Hint: The formula for converting Celius to Fahrenheit is given by

$$t_F = \frac{9}{5}t_C + 32,$$

where t_C is the temperature in Celsius, and t_F is the temperature in Fahrenheit.

6. Display `ftemp`, *to three decimal places*, using the `printf()` function. (Do not add any other string in the output, otherwise, the automated marking might fail.)

To verify your code is correct, you can try the following conversions:

- -20 Celsius is -4.000 Fahrenheit.
- 0 Celsius is 32.000 Fahrenheit.
- 100 Celsius is 212.000 Fahrenheit.

Save the program as `activity4.c`. Compile and run the program. If you are happy with the program, submit it to the Assessment System for marking. Click on **Run checks** to initiate auto-marking.

Activity 5: Functions [20 Marks]

In this activity, you will learn to write a C function.

Consider the following C program:

```
1  #include <stdio.h>
2
3  /* function implementation of calculate_area(length, width) */
4  // write your implementation here (you can use multiple lines)
5  /* end function implementation */
6
7  int main(void)
8  {
9      float l, w, a;
10     scanf("%f %f", &l, &w);
11     a = calculate_area(l, w);
12     printf("%f", a);
13 }
```

Complete the above program, providing an implementation of `calculate_area()` which returns the product of the input parameters `length` and `width`. (Do not change the other parts of the program.)

Save the program as `activity5.c`. Compile and run the program. If you are happy with the program, submit it to the Assessment System for marking. Click on **Run checks** to initiate auto-marking.

Activity 6: Function-Like Macro [20 Marks]

In this activity, you will learn to write a C function-like macro.

Consider the following C program:

```
1  #include <stdio.h>
2
3  /* function-like macro implementation of SUM(A,B) */
4  // write your implementation here
5  /* end function-like macro implementation */
6
7  int main(void)
8  {
9      int a, b, s;
10     scanf("%d %d", &a, &b);
11     s = SUM(a, b);
12     printf("%d", s);
13 }
```

Complete the above program, providing an function-like macro implementation of `SUM()` which returns the sum of the input parameters `A` and `B`. (Do not change the other parts of the program.)

Save the program as `activity6.c`. Compile and run the program. If you are happy with the program, submit it to the Assessment System for marking. Click on **Run checks** to initiate auto-marking.

9 Important Terminal Commands

The table below lists down the key commands that you should know to easily navigate the Linux file system using the command line interface or terminal:

Command	Purpose
`ls`	List the contents of the current directory
`pwd`	Print current working directory
`cd`	Change to home directory
`cd <dir>`	Change to directory `<dir>`
`cd ..`	Change to parent directory
`mkdir <dir>`	Create new directory `<dir>`
`rm <file>`	Delete `<file>`
`rm -r <dir>`	Delete directory `<dir>`
`cp <f1> <f2>`	Make a copy of file `<f1>` called `<f2>`
`mv <f1> <f2>`	Rename file `<f1>` to `<f2>`

10 Getting Help Using Manual Pages

If you want to know more about a particular command, you can use the `man` command (which stands for manual). For instance, to know more about the command `ls`, just type

```
man ls
```

in the terminal.

Many of the functions in the standard C library also have their own manual pages. The manual pages of C functions are usually in section 3. For instance, to view the manual page of `printf`, just type

```
man 3 printf
```

in the terminal. The parameter 3 in the above command tells the terminal that you want to see section 3 of the manual page.

11 GCC Options

The `gcc` compiler accepts many command-line options. The important options that you have to know are as follows:

Option	Purpose and Example
`-o <output-file-name>`	Specify the filename of the output executable file Example: `gcc hello.c -o hello` compiles `hello.c` and generates the executable file `hello` if successful.

`-Wall`	Show all warnings
	
	Examples:
	`gcc -Wall hello.c`
	`gcc -Wall hello.c -o hello`

`-Werror`	Changes warnings to errors (compilation will not complete, no executable file will be generated).
	This is a good option to use to force yourself to fix warnings.
	
	Examples:
	`gcc -Werror hello.c`
	`gcc -Werror hello.c -o hello`

`-g`	Compile with debugging flags enabled. This is needed for debugging the executable file using `gdb`.
	
	Examples:
	`gcc -g hello.c`
	`gcc -g hello.c -o hello`