WEEK 2 LAB – BASIC C PROGRAMMING AND CONTROL FLOW

In this course, we will use **Code::Blocks** for developing your programs. Please refer to the uploaded document "Using Code::Blocks" for learning the creation of your first program in Code::Blocks. You do not need to submit your code for each lab session.

1. **(computeGrade)** Write a C program that prints the ID and grade of each student in a class. The input contains the student IDs and their marks. The range of the marks is from 0 to 100. The relationships of the marks and grades are given below:

Grade	Mark
A	100-75
B	74-65
C	64-55
D	54-45
F	44-0

Use the sentinel value –1 for student ID to indicate the end of user input.

The code using if-else if-else statements is given below for your reference:

```
#include <stdio.h>
int main()
{
   int studentNumber = 0, mark;

   printf("Enter Student ID: \n");
   scanf("%d", &studentNumber);
   while (studentNumber != -1)
   {
      printf("Enter Mark: ");
      scanf("%d", &mark);
      if (mark >= 75)
         printf("Grade = A\n");
      else if (mark >= 65)
        printf("Grade = B\n");
      else if (mark >= 55)
         printf("Grade = C\n");
      else if (mark >= 45)
         printf("Grade = D\n");
      else
         printf("Grade = F\n");
      printf("Enter Student ID: ");
      scanf("%d", &studentNumber);
   }
   return 0;
}
```

Write the program using the **switch** statement.

A sample program template is given below.

```
#include <stdio.h>
int main()
{
   /* insert variable declarations here */

   printf("Enter Student ID: \n");
   scanf("%d", &studentNumber);
   while (studentNumber != -1)
   {
```

```
        /* Write your program code here */
    }
    return 0;
}
```

Sample input and output sessions are given below:

(1) Test Case 1:

```
Enter Student ID:
11
Enter Mark:
56
Grade = C
Enter Student ID:
21
Enter Mark:
89
Grade = A
Enter Student ID:
31
Enter Mark:
34
Grade = F
Enter Student ID:
-1
```

(2) Test Case 2:

```
Enter Student ID:
-1
```

2. **(printAverage)** Write a C program that reads in several lines of non-negative integer numbers, computes the average for each line and prints out the average. The value –1 in each line of user input is used to indicate the end of input for that line.

A sample program template is given below.

```
#include <stdio.h>
int main()
{
    int total, count, lines, input;
    double average;
    int i;

    printf("Enter number of lines: \n");
    scanf("%d", &lines);

    /* Write your program code here */

    return 0;
}
```

Sample input and output sessions are given below:

(1) Test Case 1:

```
Enter number of lines:
2
Enter line 1 (end with -1):
2 4 6 8 -1
Average = 5.00
Enter line 2 (end with -1):
1 3 5 7 9 -1
Average = 5.00
```

(2) Test Case 2:

```
Enter number of lines:
1
Enter line 1 (end with -1):
1 2 3 4 -1
Average = 2.50
```

3. Using Program Debugger

In addition, you should also learn how to use Program Debugger in this lab. To develop a program to solve a problem, the program development process generally consists of 6 steps. They are Problem Definition, Problem Analysis, Program Design, Implementation, Program Testing and Documentation.

- **Step 1: Problem Definition**. This step determines the objective of the program, and writes a problem statement or paragraph describing the purpose of the program. The problem statement is a broad statement of the requirements of the program, in user terms.
- **Step 2: Problem Analysis**. This step analyzes the problem and produces a set of clear statements about the way the program is to work. This requires a clear understanding of the underlying concepts and principles of the problem. These statements define how the user uses the program (*program input*), what output the program will generate (*program output*), and the *functionality* of the program. The functionality may be expressed in terms of mathematical formulas or equations for specifying the transformation from input to output.
- **Step 3: Program Design**. This step is to formulate the program logic or *algorithm*. An algorithm is a series of actions in a specific order for solving a problem. There are two basic methods that can be used to design the program logic: *pseudocode* and *flowcharts*. The technique that is used for developing a program is the top-down stepwise refinement technique.
- **Step 4: Implementation**. This step is to convert the program logic into C statements forming the program. If the program has been designed properly, then it is a straightforward task to map the program design into the corresponding C code.
- **Step 5: Program Testing**. This step is to test the program code by running the program. This aims to determine whether the program does carry out the intended functionality. Program testing involves the use of test data that the correct answers are known beforehand, and the use of different test data to test the different computational paths of a program. Therefore, it is important to design test cases such that all conditions that can occur in program inputs are tested. However, it is also necessary to ensure that the tests are not too exhaustive.
- **Step 6: Documentation**. This step is to document the programs. Documentation of computer code is useful for the understanding of the program's design and logic. This is important for the maintenance and future modification of the programs. In addition, documentation such as user manuals can also help users to understand on how to use the program.

Programs are bound to contain errors when they are first written. There are mainly three types of programming errors: *syntax errors*, *run-time errors* and *logic errors*.

- **Syntax errors**. These errors are due to violations of syntax rules of the programming language. They are detected by the compiler during the program compilation process. They are also called compilation errors. The compiler will generate diagnostic error messages to inform the programmer about the locations in the program where the errors have occurred. There are two types of diagnostic messages: warning diagnostic messages and error diagnostic messages. A warning diagnostic message indicates a minor error that might cause problems in program execution. However, the compilation is not terminated. Error diagnostic messages are serious syntax errors, which stop the compilation process. The nature and locations of the errors are given in the messages. If the programmer cannot locate the error according to the location given by the compiler error message, then the programmer should also look at the statements preceding the stated error. If this is unsuccessful, the programmer should also check all the statements related to the stated error statement. Examples of this type of errors include illegal variable names, unmatched parentheses, undefined variable names, etc.
- **Run-time errors**. These errors are detected during the execution of the program. They are caused by incorrect instructions that perform illegal operations on the computer. Examples of such illegal

operations include division by zero, storing an inappropriate data value, etc. When run-time errors are detected, run-time error messages are generated and program execution is terminated.

- **Logic errors**. These errors occur due to incorrect design of the algorithm to the problem. Such errors are usually difficult to detect and correct since no error messages are given out. Debugging of logic errors requires a thorough review of the program development process on problem analysis, algorithm design and implementation. Tracing of program logic and testing of individual modules of the program are some of the techniques, which can be used to fix logic errors.

Since errors can occur during program development, program debugging is necessary to locate and correct errors. A number of techniques are available for program debugging. The most common approach is to trace the progress of the program. This approach is especially useful for debugging *logic errors*. Another approach is to use program debugger to help debug your logica errors.

- **Program tracing** is to write code during program development to produce extra printout statements to monitor the progress of a program during program execution. This can give us an idea how the program works during the course of execution. Intermediate results can be printed and we can check to see whether the results are the intended ones. It is quite frustrating when a program runs but nothing is on the screen. The program could be running in a loop repeatedly, or it could be just waiting for user input. If printout statements are displayed on the screen, we should then know what the program is currently doing.

  To do this, debugging printout statements such as **printf()** may be placed at several strategic locations inside the program. They can be used to print the input data read from the user, or to print computation results at different stages during program execution. Debugging printout statements are also very useful for printing data values inside a loop to show changes of these values in the loop. In addition, printout statements may also be inserted at the beginning of each function, so that the exact execution path of the program can be traced.

  Apart from tracing the program, program code can also be separated into different sections, and then each section can be tested separately on its correctness. Once the errors are isolated, you may correct the errors or rewrite the code for the errorneous section. Hand simulation can also be used by pretending that we run the program in sequence on the computer. All the values for variables and program outputs are recorded. However, this method is only suitable for small programs.

- **Program debugger** can also enable programmers to inspect a program during its execution. A debugger can run with a program. We can interact with the debugger to interpret the program instructions in a step-by-step manner. We may also see the results of computations, and monitors the progress of the program. As each compiler's debugger is different, it is necessary to consult the compiler's documentation on how to use it. Learning how to use the debugger will help to speed up your program development process. Debugger is especially useful for debugging programs using data structures such as linked lists, stacks, queues anfd queues. Some important features and functions of debuggers are listed below:

  – Setting breakpoints and stepping program pxecution. Breakpoints are the places where you want a program execution to stop and provide you a chance to enter debugger instructions, for example, examine the values of some variables or step the program execution. There are two ways of stepping a target program execution. Stepping means advancing target program execution one source code statement at a time.
  – Examining Program Variables. You may also inspect the values of program variables including pointer variables.
  – **The More You Explore, the More You Find**.

For this lab, you may practice on how to perform debugging in Code::Blocks. Please follow the following steps:

(1) For Lab Question 2, run the code and check whether the program is correct. Then, you may add some debugging statements at various locations in the program code in order to understand the logic behind the execution of the program. Every program is executed in

sequential manner, statement by statement. We may use the **printf()** statement to print out the values of important program variables to inspect the correctnesss of the program logic.

(2) Remove all the debugging statements in step (1).

(3) Watch the youtube video: https://www.youtube.com/watch?v=Jab1qj_QR8s on how to use Debugger in Code::Blocks.

(4) Try to learn how to use the following three important features: stepping through the program execution, examining the values of program variables and setting up breakpoints. Explore them in the program in step (2) to check whether the logic of the program execution is correct or not.