Digital Design Verification

FYP/Internship Program

LAB # 01

GCC Compiler, GNU Debugger and Data Types,

Release: 1.0

Date: 27-April-2024

Contents

Objective

The objective of this lab is to enable students to answer following questions:

- How to debug a code using debugger?
- How are variables stored in Computer?
- Why are there multiple types of data types?
- What should be the decision criteria to choose a particular data type for a variable?
- How much memory is allocated to a variable of a particular data type?
- Where in memory these variables are stored?
- How to represent real numbers?
- What is the decimal point accuracy for real number representation in a computer?
- How to solve real-world problems using programming?
  - Mathematical problems
  - Linear circuits analysis
  - Calculus

Tools

- GNU debugger
- GC compiler

Introduction

GCC:

GCC is a toolchain that compiles code, links it with any library dependencies, converts that code to assembly, and then prepares executable files.

Key Features of GCC:

1. **Multi-Language Support**: gcc can compile code written in several languages, including C, C++, Objective-C, Fortran, Ada, Go, and D.
2. **Cross-Platform**: It is available on many platforms, including Unix, Linux, Windows (via MinGW or Cygwin), and macOS.
3. **Optimization**: gcc provides various optimization levels to improve the performance of the generated code, ranging from basic optimizations to aggressive transformations.
4. **Warnings and Errors**: It provides detailed warnings and error messages to help developers identify and fix issues in their code.
5. **Open Source**: As part of the GNU Project, gcc is free and open-source software, allowing users to modify and distribute it.

GDB

GDB, the GNU Project debugger, allows you to see what is going on `inside' another program while it executes -- or what another program was doing at the moment it crashed.

GDB can do four main kinds of things (plus other things in support of these) to help you catch bugs in the act:

- Start your program, specifying anything that might affect its behavior.
- Make your program stop on specified conditions.
- Examine what happened when your program has stopped.

Change things in your program, so you can experiment with correcting the effects of one bug and go on to learn about another.

Install GCC using the following command in linux red hat:
```
sudo yum install gcc
```

Check the version of GCC
```
gcc –version
```

Tutorial:

Compiling and Running a C Program

In this lab, we will be using the command line program gcc to compile programs in C.

```
#include <stdio.h>
int main() {
    int num1, num2, modulus;
    // Input two numbers
    printf("Enter the first number: ");
    scanf("%d", &num1);
    printf("Enter the second number: ");
    scanf("%d", &num2);
    // Compute the modulus
    modulus = num1 % num2;
    // Display the result
    printf("The modulus of %d and %d is: %d\n", num1, num2, modulus);
    return 0;
    printf("\n");
    return 0;
}
```

Use the following command to compile the code:
```
gcc ex1.c
```

This compiles ex1.c into an executable file named a.out. This file can be run with the following command:
```
./a.out
```
The executable file is a.out, so what is the ./ for?

Answer: When you want to execute an executable, you need to prepend the path to the name of the executable. The dot refers to the "current directory." Double dots (..) would refer to the directory one level up. gcc has various command line options which you are encouraged to explore. In this lab, however, we will only be using -o, which is used to specify the name of the executable file that gcc creates.

By default, the name of the executable generated by gcc is a.out. You can use the following commands to compile ex1.c into a program named ex1, and then run it. This is helpful if you don't want all of your executable files to be named a.out.

```
gcc -o ex1 ex1.c

./ex1
```

Assert

The Linux man pages serve as a manual for various standard library and operating system features. You can access the man pages through your terminal.

Type the following line into your terminal to learn about assert.

```
man assert
```

to exit the man pages, press 'q'.

Task 1

Part 1-Compiler warnings

1. Read over the code in pwd_checker.c file attached.
2. Learn about compiler warnings and the importance of resolving them. This section will resolve bug(s) along the way. Make sure to fix the bug(s) before moving on to the next section.
   - Compiler warnings are generated to help you find potential bugs in your code. Make sure that you fix all of your compiler warnings before you attempt to run your code. This will save you a lot of time debugging in the future because fixing the compiler warnings is much faster than trying to find the bug on your own.
     - Compile your code
       ```
       gcc pwd_checker.c test_pwd_checker.c -o pwd_checker
       ```
       The -o flag names your executable pwd_checker.
   - You should see 4 warnings. When you read the warning, you can see that the first warning is in the function check_upper. The warning states that we are doing a comparison between a pointer and a zero-character constant and prints out the line where the warning occurs.
   - The next line gives us a suggestion for how to fix our warning. The

compiler will not always give us a suggestion for how to fix warnings. When it does give us a suggestion, it might not be accurate, but it is a good place to start.

- Take a look at the code where the warning is occurring. It is trying to compare a const char * to a char. The compiler has pointed this out as a potential error because we should not be performing a comparison between a pointer and a character type.
  - This line of code is trying to check if the current character in the password is the null terminator. The code is currently comparing the pointer-to-the-character and the null terminator. We need to compare the pointed-to-character with the null terminator. To do this, we need to dereference the pointer. Change the line to this:
    ```
    while (*password != '\0') {…
    ```
- Recompile your code. You can now see that this warning does not appear and there are three warnings left.

3. Fix the remaining compiler warnings in pwd_checker.c

Part 2: Assert Statements

1. Learn about how you can use assert statements to debug your code. Edit the code in pwd_checker according to the directions in this section.
   - Compile and run your code
     ```
     Gcc pwd_checker.c test_pwd_checker.c -o pwd_checker
     ```
     ```
     ./pwd_checker
     ```
   - The program says that Assertion 'test1' failed. However, we can see that qrtv?,mp!ltrA0b13rab4ham password fits all of the requirements. It looks like there is a bug in our code.
   - The function check_password makes several function calls to verify that the password meets each of the requirements. To find the location of our bug, we can use assert statements to figure out which function is not returning the expected value. For example, you can add the following line after the function call to check_lower to verify that the function returns the correct value. We expect check_lower to return true because the password contains a lower case letter.
     ```
     assert(lower)
     ```

2. Add the remaining assert statements after each function call in the function check_password. This will help you determine which functions are not working as expected.
3. Compile your code.
4. Oh no! We just created a new compiler warning! Learn how to fix this warning using the man pages.
   - The warning states that we have an implicit declaration of the function assert. This means that we do not have a definition for assert. This often

means that you have forgotten to include a header file or that we spelled something wrong. The only thing that we have added since the last time our code was compiling without warnings is assert statements. This must mean that we need to include the definition of the function assert. assert is a library macro, so we can use the man pages to figure out which header file we need to include. Type the following into your terminal to pull up the man pages

```
man assert
```

- The synopsis section tells us which header file to include. We can see that in order to use assert we need to include assert.h
- Add the following line to the top of pwd_checker.c

```
#include <assert.h>
```

  Note that it is best practice to put your include statements in alphabetical order to make it easier for someone else reading your code to see which header files you have included. System header files should come before your own header files.
- Compile your code. There should be no warnings.
- Run your code. We can see that the assertion length failed. Look back at the function check_password. It looks like the function check_lower is working properly because assert(length) comes after assert(lower).

- This failed assertion is telling us that check_length is not working properly for this test case. We will investigate this in Part 3.

Part 3: Intro to GDB: start, step, next, finish, print, quit

What is GDB?

Here is an excerpt from the GDB website:

Install GDB on your machines if not already installed using the command

```
sudo yum update
```

```
sudo yum install gdb
```

The remainder of the document uses gdb and gdb interchangeably. GDB reference card

1. In this section, you will learn the GDB commands start, step, next, finish, print, and quit. This section will resolve bug(s) along the way. Make sure to fix the bug(s) in the code before moving on.
   - Before we can run our code through GDB, we need to include some additional debugging information in the executable. To do this, you will compile your code with the -g flag

```
gcc -g pwd_checker.c test_pwd_checker.c -o pwd_checker
```

   - To start gdb, run the following command. Note that you should be using the executable (pwd_checker) as the argument, not the source file

(pwd_checker.c)

```
gdb pwd_checker
```

- You should now see gdb open.

  Use command for a good split interface like in cgdb:

  ```
  Gdb -tui my_program
  ```

  Or

  ```
  (gdb) tui enable
  ```

  Now, the top window displays our code and the bottom window displays the console
- Start running your program at the first line in main by typing the following command into the console. This will set a breakpoint at line 1 and begin running the program.

  ```
  start
  ```

  The first line in main is a call to printf. We do not want to step into this function. To step over it, you can use the following command

  ```
  next
  ```

  or

  ```
  n
  ```

- When the cursor reaches test 1 line, it stops there and prints signal aborted.
- To solve the bugs, we have to go in our main.c file i.e. pwd_checker.c. For this follow following steps:
- Recompile and run your code.
- Start gdb

  ```
  gdb pwd_checker
  ```

- Let's set a breakpoint in our code to jump straight to the the function check_password using the following command:

  ```
  break pwd_checker.c:check_password
  ```

  or

  ```
  b pwd_checker.c:check_password
  ```

- Use the following command to begin running the program

  ```
  Run
  ```

- Step into check_lower.

  ```
  step
  ```

  or

  ```
  s
  ```

- We have already seen that check_lower behaves properly with the given test case, so there is nothing that we need to look at here. To get out of this function, type the following command into the console:

  ```
  finish
  ```

  Alternatively, you could have stepped until you reached the end of the function, and it would have returned.
- Step to the next line. We do not want to step into the assert function call because this is a library function, so we know that our error isn't going to be in there. Step over this line.
- Step into check_length.
- Step over strlen because this is a library function.
- Step to the last line of the function.

- Let's print out the return value

```
print meets_len_req
```

or

```
p meets_len_req
```

- Hmmm... it's false. That's odd. Let's print out length.
- The value of length looks correct, so there must be some logic error on line 24
- Ahah, the code is checking if length is less than or equal to 10, not greater than or equal. Update this line to

```
bool meets_len_req = (length >= 10);
```

- Let's run our code to see if this works. First, we need to quit out of gdb which you can do with the following commands

```
quit
```

or

```
q
```

- GDB will ask you to make sure that you want to quit. Type

```
y
```

- Compile and run your code.
- Yay, it worked! We can see that we are failing the assertion name meaning that check_name is not behaving as expected.

2. Debug check_name on your own using the commands you just learned. Make sure you use the -g flag when compiling. Make sure that you are using cgdb to debug so that you practice using it because it will help you with your projects.

3. check_number is now failing, we will address this in the next part.

Part 4: GDB: break, conditional break, run, continue

1. In this section, you will learn the gdb commands break, conditional break, run, and continue. This section will resolve bug(s) along the way. Make sure to fix the bug(s) in the code before moving on.
   - Recompile and run your code. You should see that the assertion number is failing
   - Start gdb

   ```
   gdb pwd_checker
   ```

   - Let's set a breakpoint in our code to jump straight to the the function check_number using the following command

   ```
   break pwd_checker.c:check_number
   ```

   or

   ```
   b pwd_checker.c:check_number
   ```

   - Use the following command to begin running the program

   ```
   Run
   ```

   Or

   ```
   r
   ```

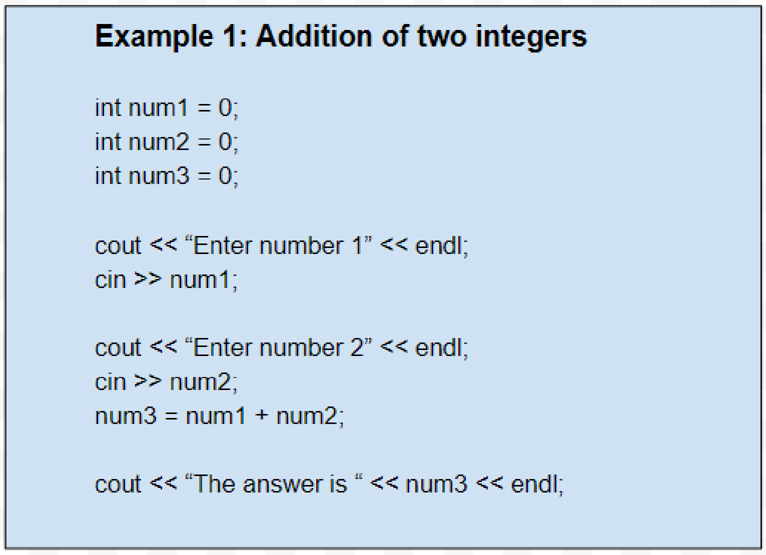
   - Your code should run until it gets to the breakpoint that we just set.
   - Step into check_range.
   - Let's look at the upper and lower bounds of the range. Print lower and upper.
   - Ahah! The ASCII representation of lower is ‘U’ and the ASCII representation of upper is \’240’. It looks like we passed in the numbers 0 and 9 instead of the characters '0' and '9'! Let's fix that
     ```
     if (check_range(*password, '0', '9')) {
     ```
   - Quit gdb and compile and run your code.

2. Debug check_upper on your own using the commands you just learn.

3. After correcting all the bugs, you should pass all the tests. It should display “Congrats! you have passed all of the test cases!”

Data Types

Programming is all about problem solving. In almost all problems, there are some independent variables whose values are usually specified by the user and we need to figure out some solution based on those variables. Before solving the problem, we need to store the variable's value in some data type based on the possible values the variable can acquire.



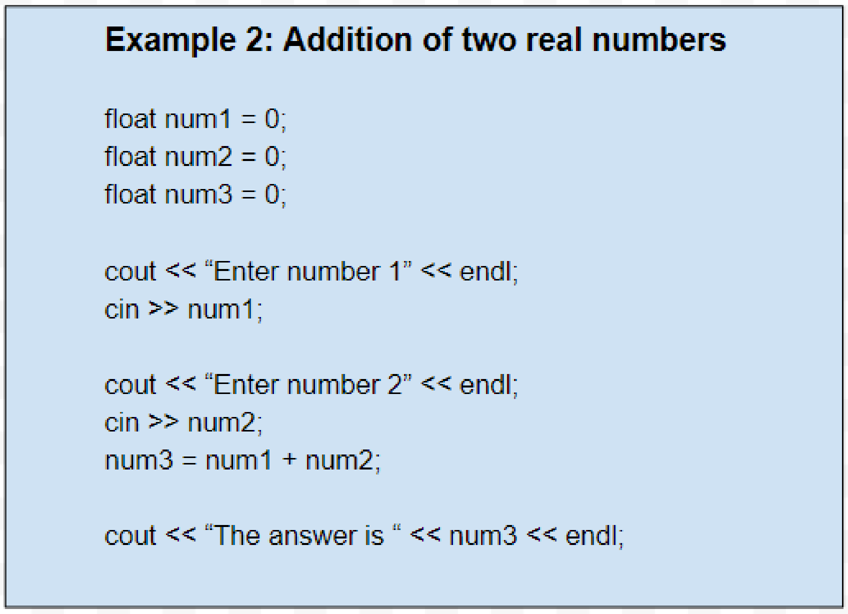
Example 1: Addition of two integers

```
int num1 = 0;
int num2 = 0;
int num3 = 0;

cout << “Enter number 1” << endl;
cin >> num1;

cout << “Enter number 2” << endl;
cin >> num2;
num3 = num1 + num2;

cout << “The answer is “ << num3 << endl;
```



Example 2: Addition of two real numbers

```
float num1 = 0;
float num2 = 0;
float num3 = 0;

cout << “Enter number 1” << endl;
cin >> num1;

cout << “Enter number 2” << endl;
cin >> num2;
num3 = num1 + num2;

cout << “The answer is “ << num3 << endl;
```

Categories of Data Types

S.No.	Type of Variables	Data Type	Memory Size	Range
1.	Binary State a. Status of Door - Open/Close b. Status of Light - On/Off c. Value of Bit - 1/0	**bool**	1-bit	0 or 1
2.	Alphabets, Names, Description	**char**	1 byte	-128 to +127
		unsigned char		0 to 255
3.	Integers (Short)	**short**	2 bytes	-32768 to 32765
		unsigned short		0 to 65535
4.	Integers	**int**	4 bytes	-2,147,483,648 to +2,147,483,647
		unsigned int		0 to 4,294,967,295
5.	Real Numbers **(6 digit accuracy)**	**float**	4 bytes	1.17549×10^{-38} to 3.40282×10^{38}

6.	Real Numbers **(15 digit accuracy)**	**double**	8 bytes	2.22507 x 10^{-308} to 1.79769 x 10^{308}

Make a flow chart on paper before going to the code.

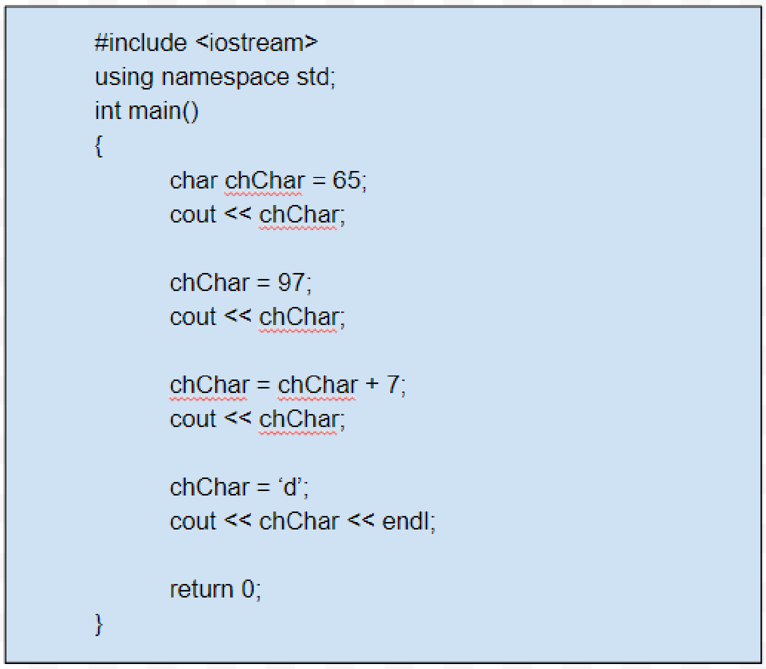
Task 2:

Let's do some real stuff with data types. What do you think is suitable for these variables? Try using data types with minimum storage requirements yet capturing a full range of values.

S.No.	**Variable**	**Data Type**
1.	Distance between home and SEECS in km	
2.	Your GPA	
3.	Your total credit hours	
4.	Your Age in YY-MM-DD	
5.	Petrol Price (PKR)	
6.	University Open/Close Status	
7.	Origin of Universe in years	
8.	Origin of human in years	
9.	Current in Ampere	
10.	Voltage in Volts	
11.	Radius of earth in meters	

Working with Characters

Let's work with characters. Characters are used for storing names.



```
#include <iostream>
using namespace std;
int main()
{
        char chChar = 65;
        cout << chChar;

        chChar = 97;
        cout << chChar;

        chChar = chChar + 7;
        cout << chChar;

        chChar = ‘d’;
        cout << chChar << endl;

        return 0;
}
```

What will get printed? It will print-------------. But how?

Task 3:

Using char print your name and CMS ID. Like in above example you are only allowed to store 1 character in the variable *i.e.* the last character of your name and likewise last digit of your CMS ID. All other characters should be printed using numbers.

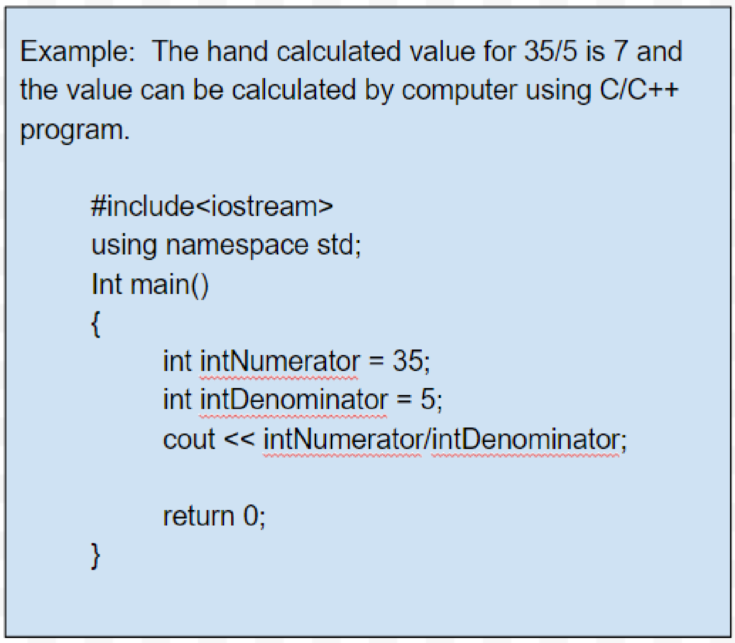
Working with Arithmetic Operators

C/C++ provides a number of arithmetic operators to work with different data types. These include:

S.No.	Arithmetic Operator	Operation
1.	+, - , *, /	Addition, Subtraction, Multiplication, Division
2.	%	Modulo

Task 4:

Evaluate following expressions by hand and as well as using on computer through programming.


Example: The hand calculated value for 35/5 is 7 and the value can be calculated by computer using C/C++ program.

```
#include<iostream>
using namespace std;
Int main()
{
        int intNumerator = 35;
        int intDenominator = 5;
        cout << intNumerator/intDenominator;

        return 0;
}
```

S.No.	Operation	Value by Hand	Value by Computer
1.	35/5	7	7
2.	36/7		
3.	18 - 32/6 * 3		
4.	220/5		
5.	27 - 7%3 + 8/3		

Working with integers and floats

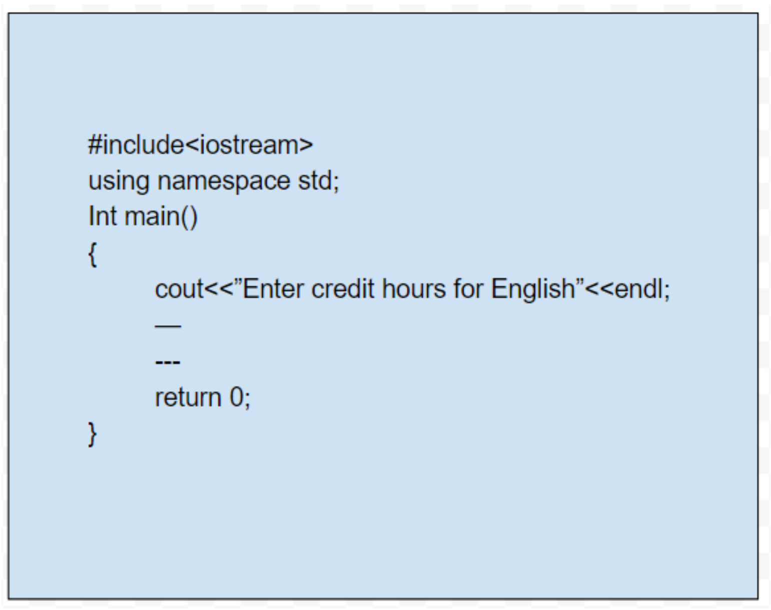
Nowadays, universities are working on a semester system as compared to an annual system. In each semester, you will be studying certain credit hours. At the end of each semester, your GPA will be calculated based on your earned scores and credit hours.

Task 5:

Take input from the user about credit hours of your first semester courses along with the grade received in that course. Write a program to calculate the GPA at the end of your semester.

-

Note: You can take earned grades as numerical values such as: 4.0 (A), 3.5 (B+), 3.0 (B), 2.5 (C+), 2.0 (C) , 1.5 (D+), 1.0 (D)



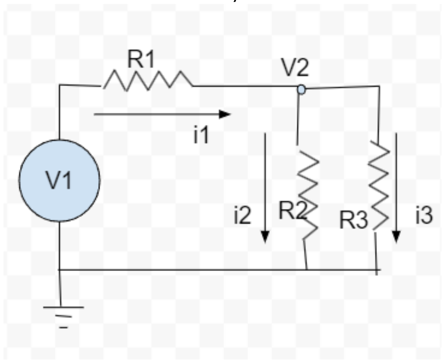
```
#include<iostream>
using namespace std;
Int main()
{
        cout<<"Enter credit hours for English"<<endl;
        —
        ---
        return 0;
}
```

Linear Circuits Analysis

Kirchoff's Law is one of the important tool in Linear Circuit Analysis which dictates that currents entering a particular node are equal to the currents exiting at that node. This law is usually used to calculate currents passing through the resistor or voltage across a resistor.

Task 6:

Consider the following circuit. You can take input from the user the value of V1, R1, R2 and R3. Using Kirchoff's Law, you need to find out i1, i2 and i3.



Hint: Kirchoff's Law at Node V2 i1 = i2 + i3

Calculus

The extrema points of a function are usually computed to figure out the maximum or minimum value of that function. Consider a quadratic function.

$$ax^2 + bx + c = 0.$$

Task 7:

Take the coefficients of the above polynomial (a, b, c) from the user and figure out the extrema points.

Submission:

Make flow chart on paper first for task 5, 6 and 7. Add scanned images of all the flow charts along with the screenshots of outputs on LMS in a proper report. Submit .c files of all the tasks along with the report.