

Lab Session 1

Objective:

Getting Started – Familiarization with Environment

In this lab session, we shall cover the following objectives

- How to install Code::Blocks IDE on computer
- Use Code::Blocks IDE and built in MinGW GCC Compiler to run our first program
- Explore Command Prompt (cmd)
- Run our first program via cmd
- Run an existing program (GuessNumber.exe) via cmd

1.1 Installing Code::Blocks Integrated Development Environment (IDE)

C++ (pronounced cee plus plus) is a compiled language. In order to get started, two requirements are essential. First is the compiler and second is text editor (for typing the program). These requirements often come under a single packaged software application termed as Integrated Development Environment (IDE). For the lab sessions of this course we shall be using an open source and free of cost IDE called **Code::Blocks**. Getting Code::Blocks is just a matter of few clicks (provided you have an internet connection). In order to download the IDE follow these steps

- i. Access www.codeblocks.org/downloads from your favorite web browser.
- ii. Click **Download the binary release**
- iii. Download the Code::Blocks with Mingw setup file, at the time of writing this text codeblocks-16.01mingw-setup.exe was available.
- iv. You are ready to go now.

Note: These instructions are for Windows users. If you are running any other operating system then download the version for your operating system.

If you don't have internet access you can get a copy of the binary release from the Computer Lab. For now, it's only available for Windows users.

The screenshot shows the 'Downloads' page of the Code::Blocks website. It features a navigation bar with links to Home, Features, Downloads, Forums, and Wiki. The main content area is titled 'Main' and includes a sidebar with 'Quick links' such as FAQ, Wiki, Forums, and Ticket System. The primary section, 'Downloads', prompts users to select a setup package based on their platform. It lists three options: Windows XP / Vista / 7 / 8.x / 10, Linux 32-bit, and Linux 64-bit. Below these, it provides instructions for older OS versions and mentions nightly builds available in the forums. A table titled 'Windows XP / Vista / 7 / 8.x / 10:' lists six specific download files, all dated 28 Jan 2016, and sourced from Sourceforge.net or FossHub. The files include standard setup, non-admin setup, and mingw compiler setups with and without separate setup files.

File	Date	Download from
codeblocks-16.01-setup.exe	28 Jan 2016	Sourceforge.net or FossHub
codeblocks-16.01-setup-nonadmin.exe	28 Jan 2016	Sourceforge.net or FossHub
codeblocks-16.01-nosetup.zip	28 Jan 2016	Sourceforge.net or FossHub
codeblocks-16.01mingw-setup.exe	28 Jan 2016	Sourceforge.net or FossHub
codeblocks-16.01mingw-nosetup.zip	28 Jan 2016	Sourceforge.net or FossHub
codeblocks-16.01mingw_fortran-setup.exe	28 Jan 2016	Sourceforge.net or FossHub

Figure 1 Screenshot of Downloads Page for Binary Release

Installation process is simple. Run the executable file you just downloaded (or acquired from Computer Lab). The installation Wizard will guide you through the whole process.

Once you run the setup file, the Wizard will get started.

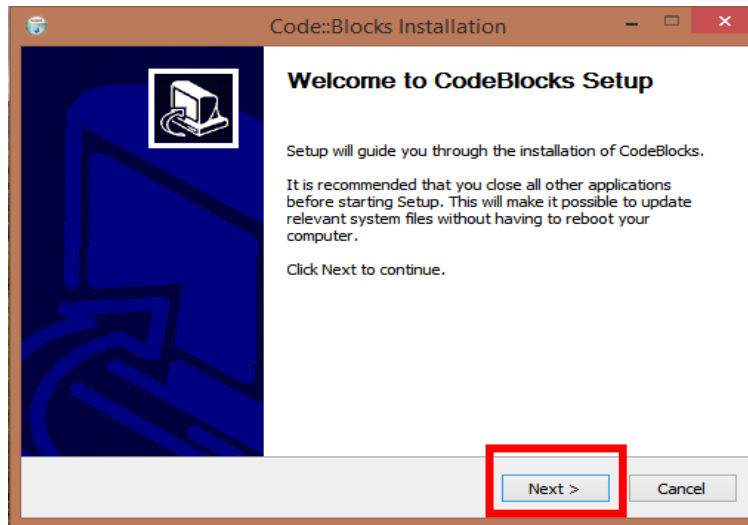


Figure 2 Step 1 of Installation Process Wizard Guide

Click Next to continue.

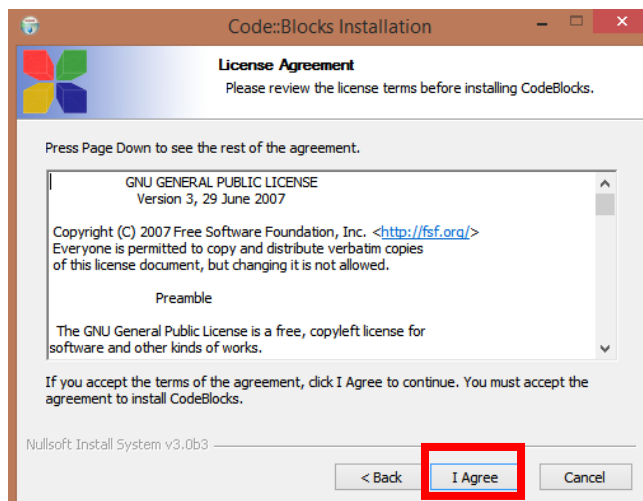


Figure 3 Step 2 of installation process License Agreement

You must agree with license terms to install and use Code::Blocks (read the terms provided and click I Agree). Once you are agreed with the terms, the installation wizard will now prompt to choose the components to install, check all components and click Next.

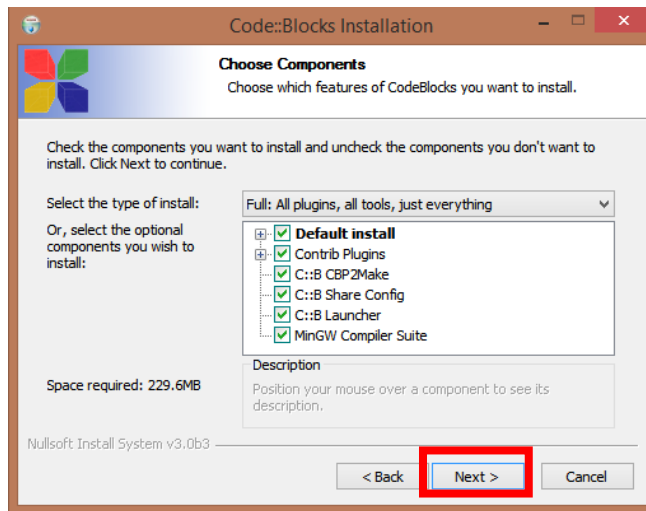


Figure 4 Step 3 of installation process, Components to install

Now select the hard disk location to install the Code::Blocks (using default is recommended)

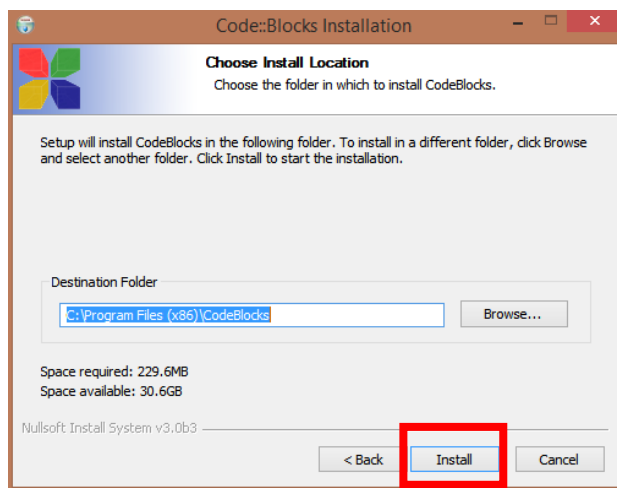
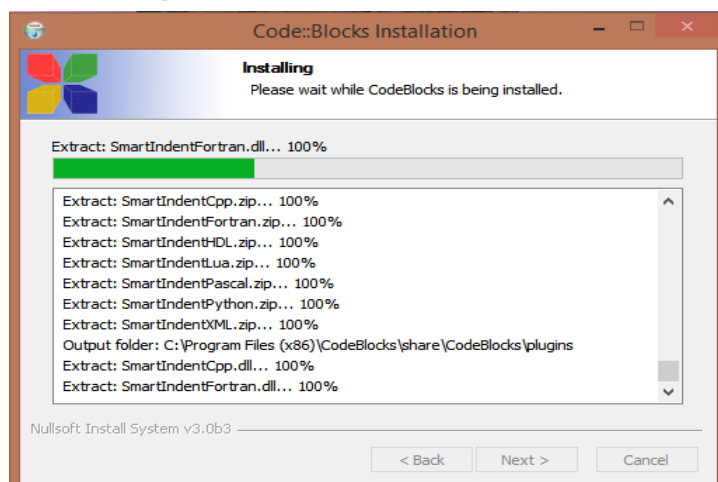


Figure 5 Step 4 of installation process, choose destination

Once you click the install button the installation will take place. Upon successful installation you will get the message.

Figure 6 Installation in process



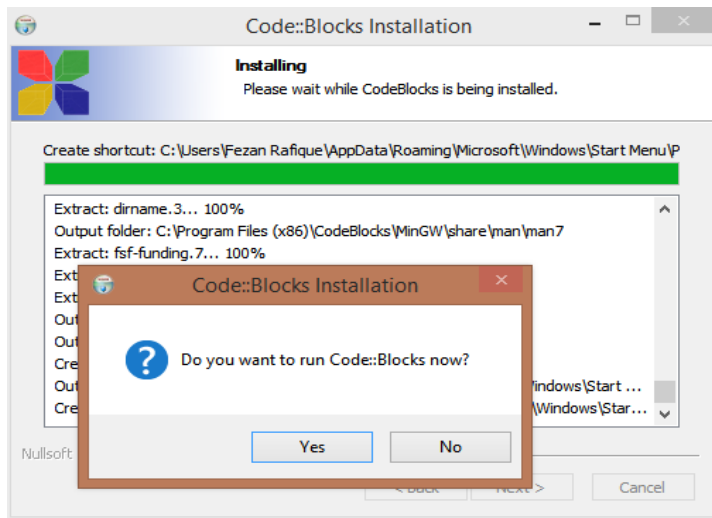


Figure 7 Installation successful

Once the installation process completed, click Finish button

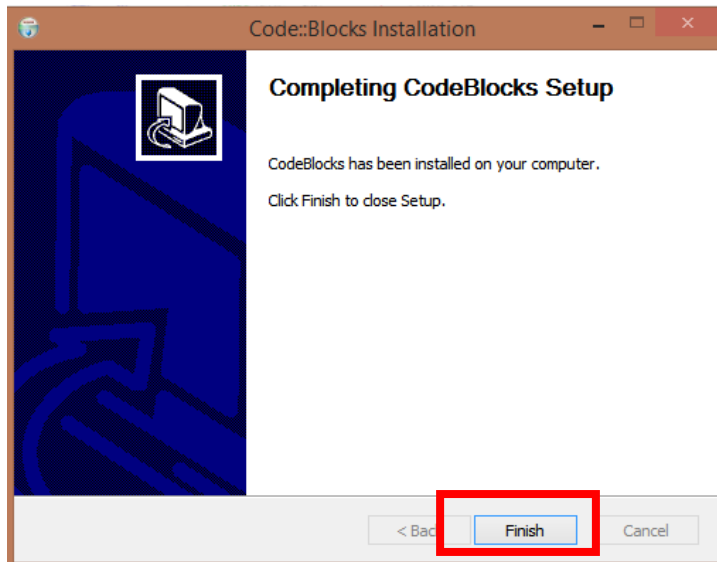


Figure 8 Installation process completed

1.2 Running the First Program

Once, the IDE is installed successfully we are now ready to develop our first C++ program. Follow the following steps

- Open Code::Blocks
- Create a new empty file (shortcut Ctrl + Shift + N)
- Save the file as lab_01_code_01.cpp
- Beware about the format .cpp

lab01_code_01.cpp

```
#include<iostream>
using namespace std;
int main(void)
{
    cout<<"Hello World";
    return 0;
}
```

- Type the code as shown (don't worry if you don't understand it for now)
- After typing the code Go to BUILD>>BUILD and RUN (shortcut F9)
- If your program was successfully written, it will be executed otherwise you will get an error

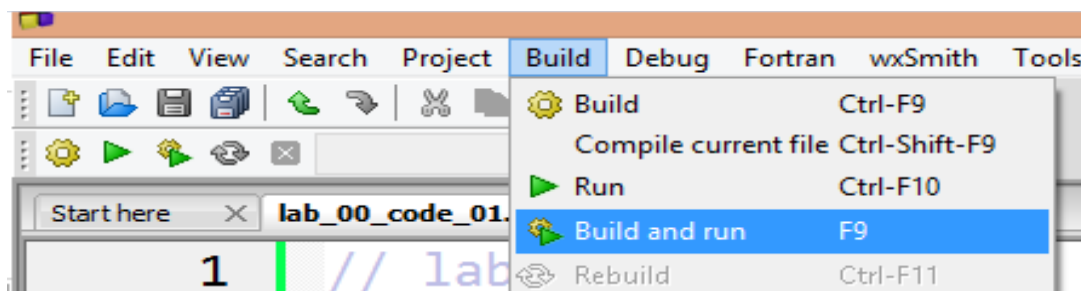


Figure 9 Step to Build Code

```
== Build finished: 0 error(s), 0 warning(s) (0 minute(s), 12 second(s)) ==
```

Figure 10 Console Log for Successful Build

```
Hello World
Process returned 0 (0x0)   execution time : 0.010 s
Press any key to continue.
```

Figure 11 Output for lab_01_code_01.cpp

1.3 Exploring Command Prompt

The target of our first program and all the other programs in this course is Console (command prompt or terminal). It is therefore necessary to have a brief introduction of command prompt.

To start command prompt, type “cmd” (without quotes) in Run.

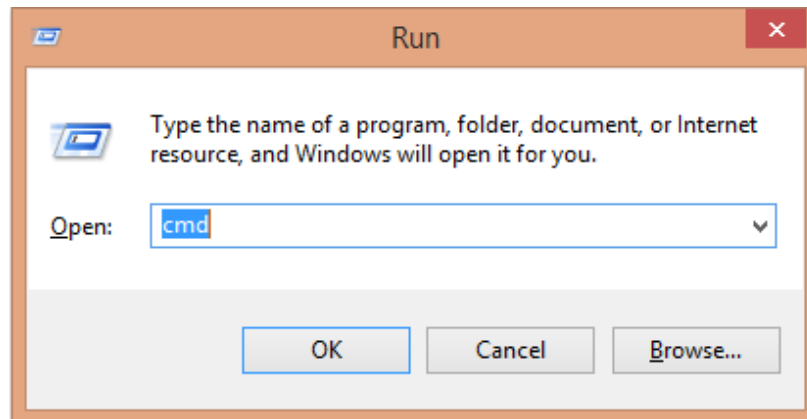


Figure 12 Run command for command prompt

This will open the command prompt window.

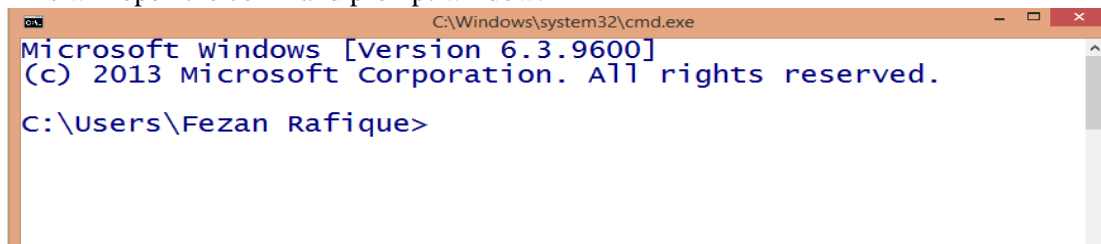


Figure 13 Command Prompt

To navigate through the directories, one can use **cd** command. A sample is shown in figure.

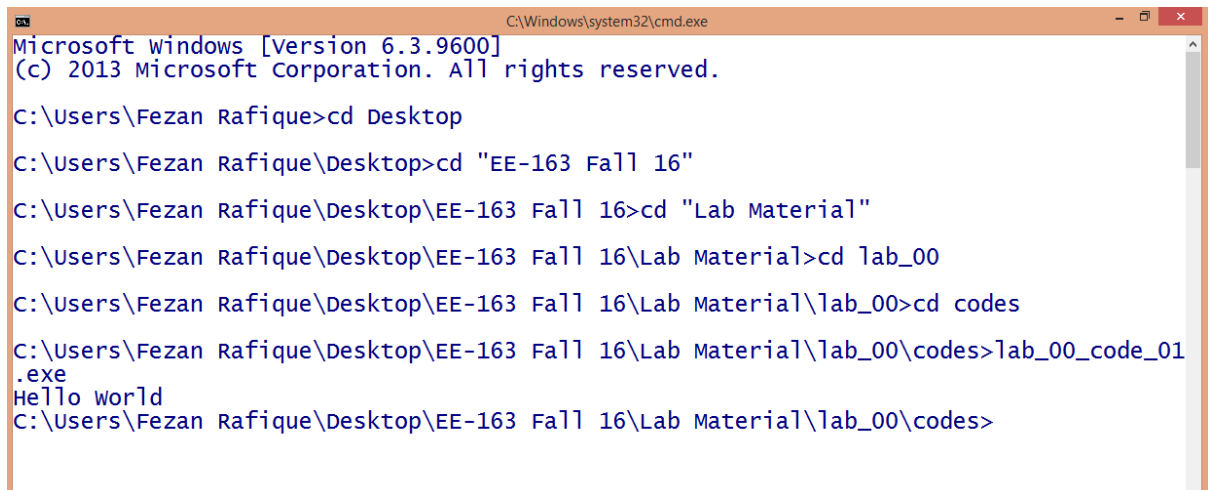


Figure 14 Navigating directories

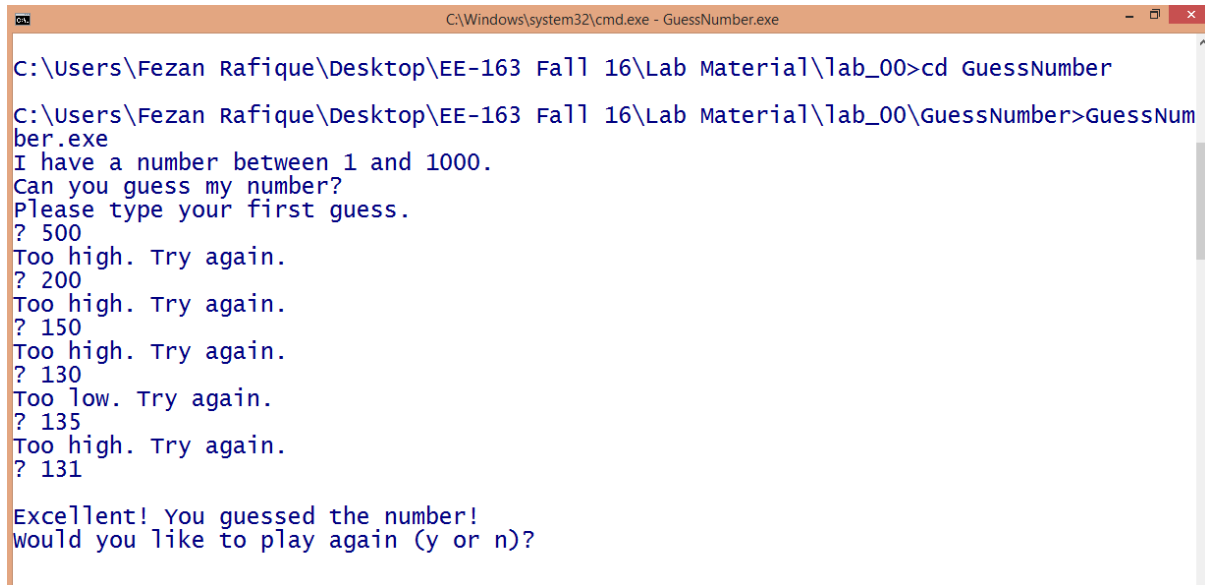
There are many useful commands for command prompt, following links are helpful to get started.

- <http://www.digitalcitizen.life/command-prompt-how-use-basic-commands>
- <http://www.computerhope.com/overview.htm>

1.4 Run GuessNumber.exe

As part of cmd exercise we shall now run an already developed program called **GuessNumber.exe** through cmd. This file is provided in the folder for Lab01

- GuessNumber.exe is already written program, the program asks the user to guess a number (which is in computer's mind)
- The user will respond by typing and can do so, until correct number is guessed
- In the meanwhile for any wrong guess computer will give a hint
- Let's try it



```
C:\Windows\system32\cmd.exe - GuessNumber.exe
C:\Users\Fezan Rafique\Desktop\EE-163 Fall 16\Lab Material\lab_00>cd GuessNumber
C:\Users\Fezan Rafique\Desktop\EE-163 Fall 16\Lab Material\lab_00\GuessNumber>GuessNumber.exe
I have a number between 1 and 1000.
Can you guess my number?
Please type your first guess.
? 500
Too high. Try again.
? 200
Too high. Try again.
? 150
Too high. Try again.
? 130
Too low. Try again.
? 135
Too high. Try again.
? 131
Excellent! you guessed the number!
would you like to play again (y or n)?
```

Figure 15 Running GuessNumber.exe

Exercise

Task 1:

Write a program to print text in following pattern,

```
Hello World
  Hello World
Hello World
```

Lab Session 2

Objective:

C++ Building Blocks

In this lab session, we shall cover the following objectives

- Basic data types in C++
- Declaring and using variables
- Comments in a C++ Program
- Printing variable values with `cout`
- Interactive computing with `cin`
- Escape sequences

2.1 Basic Data Types in C++

Fundamental to any computer program is the data associated with its use. Based on the nature of data it can be classified into various categories. Data types are important to understand, they define proper use of an identifier and expression. In C++ data types can be categorized as following.



Figure 16 Basic data types in C++

Numeric: This type contains the numbers including integers and floating point values. Following are the example of numeric data

- 100
- 895
- -237
- $6.022140857 \times 10^{23}$
- $6.62607004 \times 10^{-34}$
- $-1.60217662 \times 10^{-19}$

Character: Character data includes the alpha numeric characters and special symbols (enclosed in single quotes). Following are the examples

- 'a'
- 'F'
- '@'
- '%'
- '^'

Strings: Strings include all the text values (enclosed in double quotes). Following are the examples

- "Finland"
- "NED University"
- "PO Box No 341"
- "all along the watch tower"

Boolean: Boolean includes true and false values.

Following C++ statements show the possible use of these data types

```
cout<<100;
cout<<'~';
cout<<true;
cout<<"Mixing the stream "<<200<<'<<true<<" "<<false;
```

2.2 Declaring and Using Variables

- Variables are named objects with a specific type
- Variables can be used to store data of a certain type which can later be used, processed and/or updated in the program
- A variable must be declared using appropriate keyword
- There are some rules with variable naming

The following table shows the keyword and memory requirement of several data types

Type	Keyword	Memory
Boolean	bool	1 Byte
Character	char	1 Byte
Integer	int	4 Bytes
Floating point	float	4 Bytes
Double floating point	double	8 Bytes
String	string	?

lab_02_code_01.cpp

Following code can be used to check the memory requirements of various data types

```
#include<iostream>
using namespace std;
// sizeof() function calculates the Bytes
int main(void)
{
    cout<<"Integer Bytes="<<sizeof(int);
    cout<<"\nDouble Bytes="<<sizeof(double);
    cout<<"\nCharacter Bytes="<<sizeof(char);
```

```
    cout<<"\nBoolean Bytes="<<sizeof(bool);  
    return 0;  
}
```

Identify the data types for the following items

Item	Type	Item	Type
TRUE		@	
127		192.12	
Pakistan			

Variable Naming Rules: Following rules must be taken care while assigning a name to any variable.

- Variable name must start with a letter or _ (underscore)
- May contains letter, numbers and the underscore character only
- Uppercase and lower case are distinct
- Name should not be a reserved keyword

Good Examples

salary, new_name, myValue

Bad Examples

3name, my name, my-val, class, struct, while

A variable can be assigned a value with the assignment operator “=” . (Discussion about associativity will be the part of Lab03)

The following codes will be helpful to understand the use and role of variables in a C++ program

lab02_code_02.cpp

```
#include<iostream>  
using namespace std;  
int main(void)  
{  
    int age;    // declaring int variable  
    string name;// declaring string variable  
    float height_in_cms, weight_in_kg; // 2 float variables  
    age = 19;   // now assigning values to variables  
    name = "Ahmed Khan";  
    height_in_cms = 123.8;  
    weight_in_kg = 58.7;  
    cout<<"Name:"<<name<<"\t Age:"<<age<<endl;  
    cout<<"Height(cm):"<<height_in_cms<<"\t Weight(kg):"<<weight_in_kg;  
    return 0;  
}
```

```
}
```

lab02_code_03.cpp

```
#include<iostream>
using namespace std;
int main(void)
{
    int Roll_No = 123, salary = 40000;
    float CGPA = 3.2;
    double pi = 3.1214, x = 0.012, y;
    string enrolment_no = "ned/0145/14-15",name;
    char section = 'D';
    bool logical = 1;
    cout<< "My Roll No:" <<Roll_No<<"\t Pi="<<CGPA;
    cout<< endl<<"Value of y is:"<<y<<endl;
    cout<< "Name:"<<name<< endl;
    cout << "Enrolment:"<< enrolment_no;
    return 0; }
```

2.3 Comments in a C++ Program

Program comments are explanatory statements that you can include in the C++ code that you write and helps anyone reading it's source code. All programming languages allow for some form of comments. C++ supports single-line and multi-line comments. All characters available inside any comment are ignored by C++ compiler.

C++ comments start with /* and end with */. For example:

```
/* This is a comment */

/* C++ comments can also
 * span multiple lines
 */
```

A comment can also start with //, extending to the end of the line. For example:

```
#include <iostream>
using namespace std;

main() {
    cout << "Hello World"; // prints Hello World

    return 0;
}
```

When the above code is compiled, it will ignore // prints Hello World and final executable will produce the following result:

```
Hello World
```

Within a /* and */ comment, // characters have no special meaning. Within a // comment, /* and */ have no special meaning. Thus, you can "nest" one kind of comment within the other kind. For example:

```
/* Comment out printing of Hello World:

cout << "Hello World"; // prints Hello World

*/
```

2.4 Idea of Interactive Computing

In the above programs the value was directly assigned to the variable via assignment operator. This was done by the programmer. If it is needed to take input from the user and assign the user value to a particular variable. This is called interactive computing. C++ provides means to do so. One can use stream insertion via `cin` to assign value to a variable. This can be done like following

```
int value;
cout<<"Please enter the value ";
cin>>value;
```

The following code further illustrates the idea of interactive computing

lab_02_code_04.cpp

```
#include<iostream>
using namespace std;
int main(void)
{ // Starting braces of main
  /***Variable Declaration***/
  string name, year, department ;
  char section;
  int roll_no;
  float cgpa;
  /***Taking user input***/
  cout<<"Enter your name:";
  cin>>name;
  cout<<"Enter your Roll No.:";
  cin>>roll_no;
  cout<<"Enter your department:";
  cin>>department;
  cout<<"Enter year of study:";
  cin>>year;
  cout<<"Enter your section:";
  cin>>section;
  cout<<"What is your CGPA?";
```

```
cin>>cgpa;
cout<<endl<<endl;
//*****Printing Output*****
cout<<"\t  My Profile"<<endl;
cout<<"Name:"<<name<<"\tRoll No:"
    <<roll_no<<endl<<"Section:"
    <<section<<"\tYear:"<<year<<
    endl<<"Department:"<<
    department<<"\tCGPA:"<<cgpa;
return 0;
}
```

2.4 Escape Sequences

You must have observed some difference in the last code, e.g. using \t in cout statements. This is called escape sequence. Escape sequences are used to represent certain special characters within string literals and character literals. Following escape sequences are commonly used in C++.

Sequence	Purpose
\n	Next line
\r	Carriage return
\t	Horizontal tab
\b	Backspace
\a	Alert (beep)
\\	Print \
\'	Print '
\"	Print "

Taking help from your textbook and online resources, try to figure out the purpose of these escape sequences and explain with the help of an example program.

Exercise

Task 1:

How to insert single line and multiline comments in a C++ program.

Task 2:

Variable Declarations can appear almost anywhere in the body of C++ function (T/F).

If true, then discuss the situation in which variable declaration must be done prior to some specific task. Support your answer by giving example.

Task 3:

Calculate the maximum and minimum number that can be accommodated by *int* data type (calculate range).

Task 4:

What do you mean by *Variable Declaration* and *Variable Definition* in C/C++?

Task 5:

Check the output of the following *cout* functions and write your comments.

1. `cout << "I am a computer geek, \r its a \b lie."`
2. `cout << "a" << "\t" << "b" << "\t" << "c" << endl;`

Lab Session 3

Objective:

C++ Mathematics

In this lab session, we shall cover the following objectives

- Mathematical Operators in C++
- Operators Precedence and Associativity
- Special Mathematics Operators
 - Increment/ Decrement
 - Compound Assignments
- Type Conversion
- <cmath> Library

3.1 Mathematical Operators in C++

C++ can be used to perform basic mathematical operations. The following program can be used to illustrate this.

Code 01

```
1. #include<iostream>
2. using namespace std;
3. int main()
4. {
5.     int number1;
6.     int number2;
7.     int result;
8.     cout<<"Please enter number1 & number2";
9.     cin>>number1>>number2;
10.     result = number1 + number2; // addition
11.     result = number1 - number2; // subtraction
12.     result = number1 * number2; // multiplication
13.     result = number1 / number2; // division
14.     result = number1 % number2; // remainder division
15.     cout<<result;
16.     return 0;
17. }
```


S.No	number1	number2	operation	result
1	12	8	+	20
2	12	8	-	4
3	12	8	*	96
4	12	8	/	1
5	12	8	%	4

In the above program the variable `number1` and `number2` are called operands and they are connected via different operators in expressions given on line numbers 10 through 14. Response of each operation is stored in the variable `result`.

Keep in mind that modulus (%) operator is only defined for the data type `integers`

It is important to emphasize that result of the division is not as we expect in general. This is because the data type of `number1` and `number2` is integer, an integer divided by an integer will give an integer response, while truncating the decimal part of the value. This makes the order of precedence of arithmetic operators very significant.

The following code will help you develop intuition of C++ Mathematics

Code 02

```
#include<iostream>
using namespace std;
int main(void)
{    // BMI Calculator
    float weight_in_kg ,height_in_meter ,bmi;
    cout<<"\t \t **Body mass index (BMI) calculator** \n";
    cout<<"\t Calculates an index that indicates"<<
        " healthy weight distribution\n";
    cout<<"Enter your weight in Kgs: ";
    cin>> weight_in_kg;
    cout <<"\nEnter your height in meters: ";
    cin>> height_in_meter;
    bmi=weight_in_kg/(height_in_meter*height_in_meter);
    cout<<"\nYour BMI value is:"<< bmi;
    cout<<"\n\n \t\t Standard BMI Values for comparison \n";
    cout<<"\n \t\t Less than 18.5 : Underweight";
    cout<<"\n \t\t Between 18.5 and 24.9 : Normal";
    cout<<"\n \t\t Between 25 and 29.9 : Overweight";
    cout<<"\n \t\t Greater than 30 : Overweight";
    return 0;
}
```

Working with +, -, * and / is very obvious. Modulus operator (%) though needs some more explanation. Modulus operator gives the value of remainder once an int is divided by other int. This is very useful operator in C++. This can be very helpful in many programming situations. The following code snippet will help develop more intuition about modulus operator.

```
int a = 1234;
cout<<a%10;
a = a/10;
cout<<endl<<a%10;
a = a/10;
cout<<endl<<a%10;
a = a/10;
cout<<endl<<a;
```

3.2 Operators Precedence and Associativity

In order to properly evaluate an expression such as $4 + 2 * 3$, we must understand both what the operators do, and the correct order to apply them. The order in which operators are evaluated in a compound expression is called operator precedence. Using normal mathematical precedence rules (which state that multiplication is resolved before addition), we know that the above expression should evaluate as $4 + (2 * 3)$ to produce the value 10.

In C++, all operators are assigned a level of precedence. Those with the highest precedence are evaluated first. You can see in the table below that multiplication and division have a higher precedence than addition and subtraction. The compiler uses these levels to determine how to evaluate expressions it encounters.

Thus, $4 + 2 * 3$ evaluates as $4 + (2 * 3)$ because multiplication has a higher level of precedence than addition.

If two operators with the same precedence level are adjacent to each other in an expression, the associativity rules tell the compiler whether to evaluate the operators from left to right or from right to left.

For example, in the expression $3 * 4 / 2$, the multiplication and division operators are both precedence level 5. Level 5 has an associativity of left to right, so the expression is resolved from left to right: $(3 * 4) / 2 = 6$.

Operator(s)	Operation(s)	Order of evaluation (precedence)
()	Parentheses	Evaluated first. If the parentheses are nested, the expression in the innermost pair is evaluated first. [Caution: If you have an expression such as $(a + b) * (c - d)$ in which two sets of parentheses are not nested, but appear “on the same level,” the C++ Standard does not specify the order in which these parenthesized sub expressions will be evaluated.]
*, /, %	Multiplication, Division, Modulus	Evaluated second. If there are several, they’re evaluated left to right.
+, -	Addition Subtraction	Evaluated last. If there are several, they’re evaluated left to right.

Following examples will help you to understand the idea of precedence and associativity

```
y = 5 / 2 * 5 + 3 * 5 + 7;  
cout<<y;
```

```
y = 5 * 5 / 2 + 3 * 5 + 7;  
cout<<y;
```

Now try the following codes

Code 03

```
1.      #include<iostream>  
2.  
3.      using namespace std;  
4.  
5.      int main()  
6.      {  
7.  
8.          int number1 = 74, number2 = 82, number3 = 88;  
9.          double average;  
10.         average = number1 + number2 + number3 / 3;  
11.         cout<<average;  
12.         return 0;  
13.     }
```

Code 04

```
1.      #include<iostream>
2.
3.      using namespace std;
4.
5.      int main()
6.      {
7.
8.          int number1 = 74, number2 = 82, number3 = 88;
9.          double average;
10.         average = (number1 + number2 + number3) / 3;
11.         cout<<average;
12.         return 0;
13.     }
```

What did you observe from the output of the above two programs? Try to explain briefly.

3.3 Special Mathematical Operators (Assignment Operators): Increment and Decrement

Incrementing (adding 1 to) and decrementing (subtracting 1 from) a variable are so common that they have their own operators in C++. There are actually two versions of each operator, a prefix version and a postfix version. Following table lists them

Operator	Symbol	Form	Operation
Prefix increment (pre-increment)	++	++x	Increment x, then evaluate x
Prefix decrement (pre-decrement)	--	--x	Decrement x, then evaluate x
Postfix increment (post-increment)	++	x++	Evaluate x, then increment x
Postfix decrement (post-decrement)	--	x--	Evaluate x, then decrement x

The prefix increment/decrement operators are very straightforward. The value of x is incremented or decremented, and then x is evaluated.

For example

```
int x = 5;
int y = ++x; // x is now equal to 6, and 6 is assigned to y
```

The postfix increment/decrement operators are a little more tricky. The compiler makes a temporary copy of x, increments or decrements the original x (not the copy), and then evaluates the temporary copy of x. The temporary copy of x is then discarded.

```
int x = 5;
int y = x++; // x is now equal to 6, and 5 is assigned to y
```

Let's examine how this last line works in more detail. First, the compiler makes a temporary copy of x that starts with the same value as x (5). Then it increments the original x from 5 to 6. Then the compiler evaluates the temporary copy, which evaluates to 5, and assigns that value to y. Then the temporary copy is discarded.

Consequently, y ends up with the value of 5, and x ends up with the value 6. Here is another example showing the difference between the prefix and postfix versions:

```
int x = 5, y = 5;
cout << x << " " << y << endl;
cout << ++x << " " << --y << endl; // prefix
cout << x << " " << y << endl;
cout << x++ << " " << y-- << endl; // postfix
cout << x << " " << y << endl;
```

This produces the output:

```
5 5
6 4
6 4
6 4
7 3
```

Special Mathematical Operators (Assignment Operators): Compound Assignments

Compound assignment operators modify the current value of a variable by performing an operation on it. They are equivalent to assigning the result of an operation to the first operand: Following table summarizes the compound assignments

Equation with Compound Assignment	Actually means
$x += 3.5$	$x = x + 3.5$
$x -= 1000$	$x = x - 1000$
$x *= 10$	$x = x * 10$
$x /= 5$	$x = x / 5$

Evaluating Expression and Equations with Mixed Data Types

You are now familiar with the idea of precedence and associativity. It is now time to clarify one very important aspects of C++ mathematics, how an expression or equation contacting mixed data types e.g. `int` and `float` is evaluated. Consider the equation for example

```
tempf=tempc*(9/5)+32;
```

One may be disguised that there is nothing wrong with the above statement, but the way C++ handle it is really important to consider. The literal 9 when divided by 5 will result in an int value whereas the user might be expecting floating result. In that case the result will be incorrect. This can be corrected by implementing the same expression with floating point literals, like

```
tempf=tempc*(9.0/5.0)+32;
```

Following will also do the job.

```
tempf=tempc*(9.0/5)+32; or tempf=tempc*(9/5.0)+32;
```

3.4 Type Casting

C++ allows to temporarily change the type of a variable for one statement, this idea is called type casting. The idea is explained in the following code.

Code 05

```
1.    #include<iostream>
2.    #include<cmath>
3.
4.    using namespace std;
5.
6.    int main ()
7.    {
8.        float num1 = -9.5;
9.        int num2 = 101;
10.       cout<<(int)num1;
11.       cout<<endl<<(float)num2/10;
12.
13.       return 0;
14.    }
```

Line number 10 will be processed by considering num1 as int and not its own type, similarly line 11 will be executed by considering num2 as floating point quantity and not int.

3.5 Advanced Mathematical Functions <cmath>

Some very useful and advanced mathematical functions are present in <cmath> library. Which can be included in a program through preprocessor directive #include<cmath>. Following are the few functions which are available in this library.

Category	Function	Description
Trigonometry	cos	Returns the cosine of an angle of x radians.
	sin	Returns the sine of an angle of x radians.
	tan	Returns the tangent of an angle of x radians.
	acos	The acos function computes the principal value of the arc cosine of x . A domain error occurs for arguments not in the range $[-1, +1]$.
	asin	The asin function computes the principal value of the arc sine of x . A domain error occurs for arguments not in the range $[-1, +1]$.
	atan	The atan function returns the arc tangent in the range $[-\pi/2, +\pi/2]$

Exponential and logarithmic function	exp	Returns the base-e exponential function of x , which is e raised to the power x : e^x .
	log	Returns the natural logarithm of x . If the argument is negative, a domain error occurs.
	log10	Returns the common (base-10) logarithm of x . If the argument is negative, a domain error occurs.
Power Functions	pow	Returns base raised to the power exponent: e.g. <code>pow(7.0, 3.0)</code> ; will find 7^3
	sqrt	Returns the square root of x . If x is negative, a domain error occurs:
	cbrt	Returns the cubic root of x .
Rounding and Remainder Functions	ceil	Rounds x upward, returning the smallest integral value that is not less than x .
	floor	Rounds x downward, returning the largest integral value that is not greater than x .
	fmod	Returns the floating-point remainder of <i>numer/denom</i>
	trunc	Rounds x toward zero, returning the nearest integral value that is not larger in magnitude than x .
	round	Returns the integral value that is nearest to x , with halfway cases rounded away from zero.
Other Functions	fabs	Returns the absolute value of x : $ x $.
	abs	Returns the absolute value of x : $ x $.

Example to use trigonometric functions

Code 06

```

1.    #include<iostream>
2.    #include<cmath>
3.
4.    using namespace std;
5.
6.    int main()
7.    {
8.
9.        const double pi = 3.141592;
10.       double angle = pi/6;
11.       cout<<endl<<"***** Calculating Trigonometric Ratios
*****"<<endl;
12.       cout<<endl<<"All calculations on Angle "<<angle<<"
Radians"<<endl;
13.       cout<<endl<<"cos("<<angle<<") "<<"= "<<cos(angle)<<endl;

```



```
14.      cout<<endl<<"sin("&<<angle<<") "<<= "<<sin(angle)<<endl;
15.      cout<<endl<<"tan("&<<angle<<") "<<= "<<tan(angle)<<endl;
16.      cout<<endl<<"***** Calculations Terminated
          *****"<<endl;
17.
18.      return 0;
19.  }
```

Example to use exponential and logarithmic functions

Code 07

```
1.      #include<iostream>
2.      #include<cmath>
3.
4.      using namespace std;
5.
6.      int main()
7.      {
8.          double num = 10.3;
9.          cout<<endl<<"exp("&<<num<<") "<<= "<<exp(num)<<endl;
10.         cout<<endl<<"log("&<<num<<") "<<= "<<log(num)<<endl;
11.         cout<<endl<<"log10("&<<num<<") "<<= "<<log10(num)<<endl;
12.
13.         return 0;
14.     }
```

Example to use power functions

Code 08

```
1.      #include<iostream>
2.      #include<cmath>
3.
4.      using namespace std;
5.
6.      int main()
7.      {
8.          double num1 = 10.3, num2 = 2.0;
9.          cout<<endl<<"pow("&<<num1<<","<<num2<<") "<<=
          "<<pow(num1,num2)<<endl;
10.         cout<<endl<<"sqrt("&<<num1<<") "<<= "<<sqrt(num1)<<endl;
11.         cout<<endl<<"cbrt("&<<num1<<") "<<= "<<cbrt(num1)<<endl;
12.
13.         return 0;
14.     }
```

Example to use power rounding functions

Code 09

```
1. #include<iostream>
2. #include<cmath>
3.
4. using namespace std;
5.
6. int main()
7. {
8.     double num1 = 2.3,num2 = 3.8,num3 = 5.5,num4 = -2.3,num5 = -3.8,num6 = -
        5.5;
9.     cout<<"value\tround\tfloor\tceil\ttrunc\n";
10.     cout<<"-----\t-----\t-----\t----\t-----\n";
11.     cout<<num1<<"\t"<<round(num1)<<"\t"<<floor(num1)<<"\t"<<ceil(num1)<<
        "\t"<<trunc(num1)<<"\n";
12.     cout<<num2<<"\t"<<round(num2)<<"\t"<<floor(num2)<<"\t"<<ceil(num2)<<
        "\t"<<trunc(num2)<<"\n";
13.     cout<<num3<<"\t"<<round(num3)<<"\t"<<floor(num3)<<"\t"<<ceil(num3)<<
        "\t"<<trunc(num3)<<"\n";
14.     cout<<num4<<"\t"<<round(num4)<<"\t"<<floor(num4)<<"\t"<<ceil(num4)<<
        "\t"<<trunc(num4)<<"\n";
15.     cout<<num5<<"\t"<<round(num5)<<"\t"<<floor(num5)<<"\t"<<ceil(num5)<<
        "\t"<<trunc(num5)<<"\n";
16.     cout<<num6<<"\t"<<round(num6)<<"\t"<<floor(num6)<<"\t"<<ceil(num6)<<
        "\t"<<trunc(num6)<<"\n";
17.
18.     return 0;
19. }
20.
```

Exercise

1. Using compound assignment operators, write a program that generates the following output:

x = 2.5	y = 10
x = 25.0	y = 15
x = 250.0	y = 20
x = 2500.0	y = 25

Initialize x as float with value of 2.5 and y as int with value 10. In each successive stage, use *= operator for x and += operator for y to achieve the desired values.

[illegible]

2. Write a program that asks the user to enter the length of base and perpendicular of a right angle triangle. Then it determines the length of hypotenuse, angle between base and hypotenuse and angle between hypotenuse and perpendicular. Also find the sine and cosine values of these angles. (For hint refer to basic trigonometry from any mathematics book)

[illegible]

3. Write a program that asks the user to enter coefficients a, b and c of the standard quadratic equation:

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

The program then should compute and display discriminant

$$|b^2-4ac|$$

And the roots of equation

$$x_1 = -b + \sqrt{b^2 - 4ac}/2a$$

$$x_2 = -b - \sqrt{b^2 - 4ac}/2a$$

Finally, give opinion on how the program could be made more general to different input conditions

[illegible]

Lab Session 4

Objective:

Decision Making in C++

In this lab session, we shall cover the following objectives

- General idea of decision making in Programming
- Operators used in decision making
- The if() and if() – else statements
- The if() – else if() structure

4.1 General idea of decision making in the Programming

The idea of decision making allows to control the flow of the program. So far, every program that we have discussed was executed from start to end. Often it is required to control the flow of a program so that a certain piece of code is only executed if a certain condition is met. The ability to control the flow of a program, letting it make decisions on what code to execute, is valuable in programming. The **if** statement allows to control if a program enters a section of code or not based on whether a given condition is true or false. One of the important functions of the if statement is that it allows the program to select an action based upon the user's input. For example, by using an if statement to check a user entered password, your program can decide whether a user is allowed access to the program.

Decision making in programming is done in terms of testing an expression (logical or relational). The result of the test is either TRUE or FALSE. A TRUE leads to the execution of a specified piece of code, whereas a FALSE leads to two possibilities; either a piece of code is executed that is different from the TRUE case or a branch takes place. An important note regarding decision making structures is that they are not loops; they are executed only once.

4.2 Operators Used in Decision Making

Arithmetic operators are incapable of generating TRUE or FALSE. For this we need operators that can result in YES and NO. In other words operators that can produce Boolean output. There are two such operators in C++

- i. Relational Operators
- ii. Logical Operators

Relational operators are described in the following table

Operator	Description	Example
>	Greater than	5 > 4 is TRUE
<	Less than	4 < 5 is TRUE
>=	Greater than or equal	4 >= 4 is TRUE
<=	Less than or equal	3 <= 4 is TRUE
==	Equal to	5 == 5 is TRUE
!=	Not equal to	5 != 4 is TRUE

The following program will describe the use of these operators

Code 01

```
#include<iostream>
using namespace std;
int main(void)
{
    int num1=105, num2=34;
    float pi=3.1412, x=123.5;
    string password="abcd1234";
    bool result=(num1>num2);
    cout<<num1<<" "<<num2<<"\t1=true, 0=false";
    cout<<"\nanswer="<<result<<endl;
    cout<<pi<<"="<<x<<"\t1=true, 0=false";
    cout<<"\nanswer="<<(pi==x)<<endl;
    cout<<"Is the password correct?\t1=yes, 0=no";
    cout<<"\nanswer="<<(password=="abcd1234");
    return 0;
}
```

What did you understand from the results of above program?

Logical operators

The logical operators apply logic functions (NOT, AND, and inclusive OR) to boolean arguments. These are helpful to take decision based on multiple conditions. Following tables summarize these operators.

Operator	C++ Symbol
AND	&&
OR	
NOT	!

AND (&&) and OR (||) Operator

A	B	A&&B	A B
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

NOT (!) Operator

A	!A
0	1
1	0

The following program will demonstrate the use of logical operators

Code 02

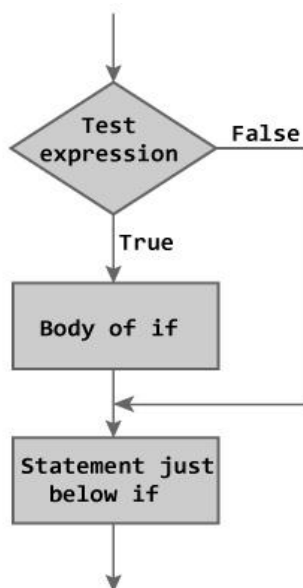
```
#include<iostream>
using namespace std;
int main(void)
{
    bool a=true, b=0, c=0, result;
    result=(c||a);
    cout<<"a AND b="<<(a&&b);
    cout<<"\nc OR a="<<result;
    cout<<"\nNOTa="<<(!a);
    cout<<"\nNOTb="<<(!b);
    cout<<"\nNOTc="<<(!c);
    return 0;
}
```

Explain in few lines what you understood from the above program

4.3 The if() and if() – else statements

```
if (testExpression)
{
    // statements
}
```

The if statement evaluates the test expression inside parenthesis. If test expression is evaluated to true, statements inside the body of **if** is executed. If test expression is evaluated to false, statements inside the body of **if** is skipped. This can be demonstrated with the following flow-chart.



The following program will help to demonstrate the idea.

Code 03

```
#include<iostream>
using namespace std;
int main(void)
{
    //Calculator Program
    double operand1, operand2, result;
    char operation;
    cout<<"\t***Calculator Program***";
    cout<<"\nEnter the desired expression"
```



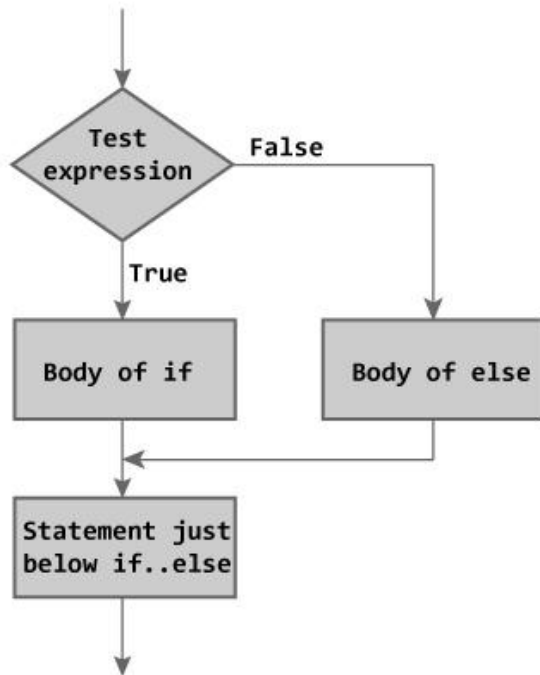
```
        <<"with spaces<eg 12.6 + 4.32>";
cin>>operand1>>operation>>operand2;
if(operation=='+')
{
    result=operand1+operand2;
}
if(operation=='-')
{
    result=operand1-operand2;
}
if(operation=='*')
{
    result=operand1*operand2;
}
if(operation=='/')
{
    result=operand1/operand2;
}
if(operation!='+' || operation!='-' ||
    operation!='*' || operation!='/' )
{
    cout<<"\nInvalid Operator";

}
cout<<"\n\nresult="<<result;

return 0;
}
```

What did you understand from the results of above program?

The **if else** executes the codes inside the body of **if** statement if the test expression is true and skips the codes inside the body of **else**. If the test expression is false, it executes the codes inside the body of **else** statement and skips the codes inside the body of **if**. The following flow chart and program will help you understand the idea.

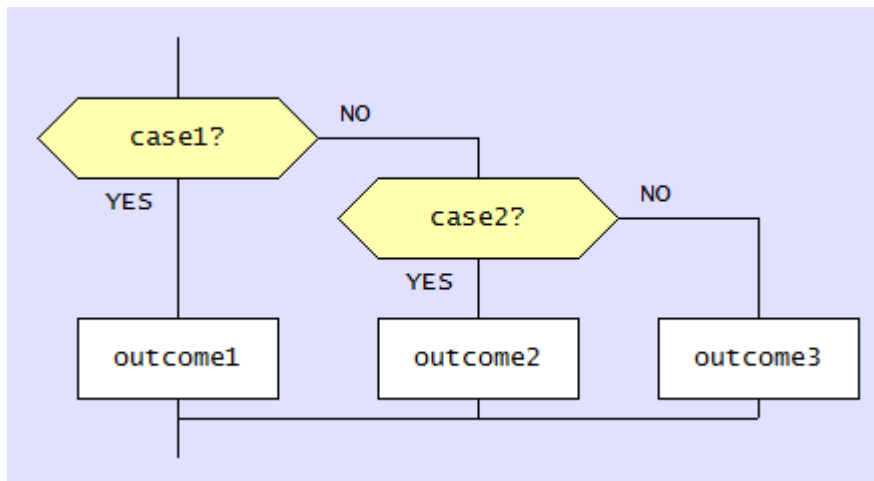


Code 04

```
#include<iostream>
using namespace std;
int main(void)
{
    string stored_password="abcd1234";
    string user_password;
    cout<<"Enter password:";
    getline(cin,user_password);
    if(user_password==stored_password)
    {
        cout<<"\nAccess granted\n";
    }
    else
    {
        cout<<"\nAccess denied\n";
    }
    return 0;
}
```

4.4 The if() – else if () statement

Occasionally, programs need to decide between a set of given conditions to perform an operation. This is called 'Multiple Selection'. In our calculator program, we performed multiple selection by using multiple if() statements. This is an improper method. Multiple selection can be performed by using if() – else if () statement. Following example and flow chart will be helpful to understand.



Code 05

```
#include<iostream>
using namespace std;
int main(void)
{ //Calculator Program
  double operand1, operand2, result;
  char operation;
  cout<<"\t***Calculator Program***";
  cout<<"\nEnter the desired expression"
    <<"with spaces<eg 12.6 + 4.32>";
  cin>>operand1>>operation>>operand2;
  if(operation=='+')
  {
    result=operand1+operand2;
  }
  else if(operation=='-')
  {
    result=operand1-operand2;
  }
  else if(operation=='*')
  {
    result=operand1*operand2;
  }
  else if(operation=='/')
  {
```

```

        result=operand1/operand2;
    }
    else
    {
        cout<<"\nInvalid Operator\n";
        return 0;
    }

    cout<<"\n\nresult="<<result;

    return 0;
}

```

Exercise

Task 1:

Write a program that asks user to enter 3 numbers and then finds the largest and smallest among them and displays both largest and smallest number. This program can be written in many ways. Provide at-least two methods.

Note: This program can be written using Multiple *if()* statements, Multiple *if()-else* statements or *if()* *else*. And of course there are other methods also. This exercise is a test of your thinking abilities.

[illegible]

This image shows a single sheet of white paper with horizontal blue ruling lines. The lines are evenly spaced and run across the width of the page. There are no margins, text, or other markings on the paper.

Task 2:

C++ provides an alternate approach for `if () - else if ()` statements, that is `switch () - case` statement. Use literature and internet resources to understand using it. Then write a calculator program written in lab session using *switch ()-case* statements. If the user entered the operator other than `+, -, *, /` then program should print “Invalid Operator” on screen.

[illegible]

Lab Session 05

Objective:

Repetition with for () loop

In this lab session, we shall cover the following objectives

- The idea of loop
- The for () loop
- Nested for() loop

5.1 The idea of loop and its need

One of the very powerful control structures is Repetition Statements in C++. Repetition statements allow to repeat a block of code until a certain condition is true. Repetition statements are commonly referred as loops and they can be implemented with the following statements

- for
- while
- do while

In this lab we shall discuss (i) whereas (ii & iii) will be discussed in the next. Loops are helpful when a certain piece of code is required to be executed in a repeated manner. This can save a lot of precious time and laborious efforts.

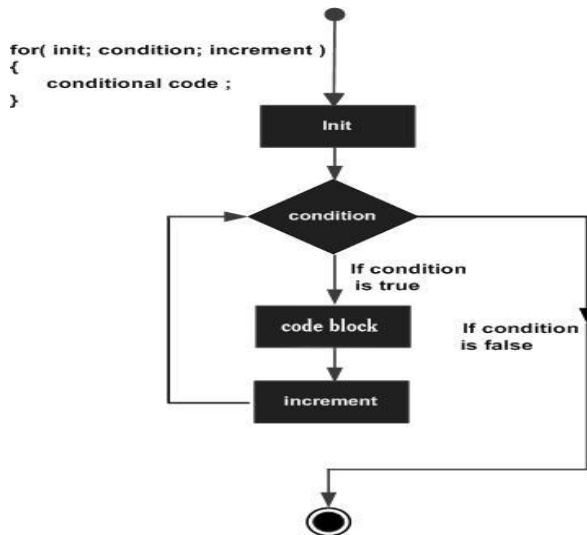
5.2 The for() loop

A for loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times. The syntax of a for loop in C++ is

```
for ( init; condition; increment ) {  
    statement(s);  
}
```

Here is the flow of control in a for loop. The init step is executed first, and only once. This step allows you to declare and initialize any loop control variables. Next, the condition is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute and flow of control jumps to the next statement just after the for loop. After the body of the for loop executes, the flow of control jumps back up to the increment statement. This statement allows you to update any loop control variables. The condition is now evaluated again. If it is true, the loop executes and the process repeats

itself (body of loop, then increment step, and then again condition). After the condition becomes false, the for loop terminates. Following diagram explains the whole process.



Code 01

```
#include<iostream>
using namespace std;
int main(void)
{
    int num;
    for(int num=0;num<=10;num++)
    {
        cout<<"\n num = "<<num;
    }
    return 0;
}
```

Observe the output of this program and describe in your words

Following code will further strengthen your understanding of for loop

Code 02

```
#include<iostream>
using namespace std;
int main(void)
{    //Calculating Power (base^exponent)
    int base,exponent,answer,counter;
    cout<<"Enter a number(integer):";
    cin>>base;
    cout<<"Enter an exponent(integer):";
    cin>>exponent;
    answer=1;//running product variable
    for(counter=exponent;counter>0;counter=counter-1)
    {
        answer=answer*base;
    }
    cout<<"\n"<<base<<" raised to power "<<exponent
        <<" = "<<answer;
    return 0;
}
```

5.3 Nested for () loop

Placing a loop inside the body of a loop is called nesting the loops. This idea is so useful to code solutions for many real life computational problems. Following is the syntax of nested for loops

```
for ( init; condition; increment ) {
    for ( init; condition; increment ) {
        statement(s);
    }
    statement(s); // you can put more statements.
}
```

C++ allows 256 levels of nesting.

Following examples will help to understand the idea

Code 03

```
#include<iostream>
using namespace std;
int main(void)
{    // 10x10 Grid of a character
    int row,col;
    char display_char;
    cout<<"Enter a character for display:";
    cin>>display_char;
    cout<<endl<<endl;
    for(row=1;row<=10;row++)
```

```
{
    for(col=1;col<=10;col++)
    {
        cout<<display_char;

    }
    cout<<endl;
}
return 0;
}
```

Draw output here

Code 04

```
#include<iostream>
using namespace std;

int main()
{
for(int row = 1; row <= 5; ++row)
{
    for(int col = 1; col <= 5; ++col)
    {
        cout<<row<<" * "<<col<<" = "
        <<row * col<<"\t";
    }
    cout<<endl;
}
}
```

Draw the output here

Task 1:

a)

```
      *
```

```
    ***
```

```
  *****
```

```
*****
```

b)

1
121
12321
1234321

[illegible]

[illegible]

Task 2:

Using for() loops, write a program that displays all possible combination of 6 bit binary number. (Hint: You shall need 6 int variables for the six digits)

Sample Run: 000000, 000001, 000010, 000011, 111111

This image shows a single sheet of white paper with horizontal ruling lines. The lines are evenly spaced and run across the width of the page. There are no margins, text, or other markings on the paper.

Lab Session 06

Objective:

Repetition with while () loop

In this lab session, we shall cover the following objectives

- The idea of sentinel controlled loop
- The while () loop
- Nested do while() loop

6.1 The idea of sentinel control loop

Loops that do not have a pre-defined ending point and terminate when the termination condition has arrived. Unlike exhaustive loops, the termination condition in these loops is provided by manipulations within the loop. Sometimes, loop control may need to be based on the value of what we are processing. In this case, we would use sentinel-controlled repetition. Sentinel-controlled repetition is sometimes called indefinite repetition because it is not known in advance how many times the loop will be executed. It is a repetition procedure for solving a problem by using a sentinel value (also called a signal value, a dummy value or a flag value) to indicate "end of data entry". The sentinel value itself is not a part of the processed data. C++ provides while and do while statements for implementing sentinel loops.

6.2 The while () loop

Generally, a while loop contains the following components

- i. Loop control variable: A variable mostly inside the relational expression.
- ii. Relational Expression
- iii. Body (Multiple statements)
- iv. A statement that makes the relational expression false.

The following code will explain these terms

Code 01

```
#include<iostream>
#include<conio2.h>
using namespace std;
int main(void)
{
    char guess;
    cout<<"Press any key from keyboard :";
    cout<<"\n This program shall end only"
        <<" when you press the secret key";
    guess=getche();
    while( guess!='x' )
    {
        cout<<"\n Wrong input, try another key:";
        guess=getche();
    }
    cout<<"\nEureka! You have discovered it.";
    getch();
    return 0;
}
```

Observe the output of this program and describe in your words

Another example to explain the idea

Code 02

```
#include<iostream>
#include<iomanip>
#include<conio2.h>
using namespace std;
int main(void)
{
    char option='y';
    double num;
    int counter;
    cout<<"\t\t****Multiplication Tables****";
    while( option!='n' )
    {
        cout<<"\n\nEnter a number:";
        cin>>num;
        for(counter=1;counter<=15;counter++)
        {
            cout<<left;
            cout<<setw(10)<<num<<"*"<<right
                <<setw(10)<<counter<<"="<<
                setw(10)<<num*counter<<endl;
        }
        cout<<"\n\nDo you like to continue?(y or n):";
        option=getche();
    }
    if(option=='n')
    {
        cout<<"Thanks for using this program";
    }
    getch();
    return 0;
}
```

6.3 The do while loop

Unlike for and while loops, which test the loop condition at the top of the loop, the do...while loop checks its condition at the bottom of the loop. A do...while loop is similar to a while loop, except that a do...while loop is guaranteed to execute at least one time. Following code will help to understand the idea.

Code 03

```
#include<iostream>
#include<iomanip>
#include<conio2.h>
using namespace std;
int main(void)
{
    char option;
    double num;
    int counter;
    cout<<"\t\t****Multiplication Tables****";
    do
    {
        cout<<"\n\nEnter a number:";
        cin>>num;
        for(counter=1;counter<=10;counter++)
        {
            //cout<<left;
            cout<<setw(10)<<num<<"*"<<right
                <<setw(10)<<counter<<"="<<
                setw(10)<<num*counter<<endl;
        }
        cout<<"\n\nDo you like to continue?(y or n):";
        option=getche();
    }
    while( option!='n' );
    if(option=='n')
    {
        cout<<"Thanks for using this program";
    }
    getch();
    return 0;
}
```

Exercise

Task 1:

Write a program that continuously asks user to enter an integer and displays the SUM of the current input with all previous input. The program continuous to run until the SUM value is less than equal to 100. Use while() loop.

Sample Run:

Enter an integer: 12 [Enter]

Running Sum = 12

Enter an integer: 10 [Enter]

Running Sum = 22

Enter an integer:70 [Enter]

Running Sum = 92

.
.
.

Sum exceeds 100. Program terminated.

Task 2:

Write a program that counts number of digits in an integer entered by the user. Use while() loop.

Sample Run:

Enter an integer: 123456 [Enter]

No. of digits = 6

This image shows a blank sheet of white paper with horizontal ruling lines. The lines are evenly spaced and run across the width of the page. There are no margins, text, or other markings on the paper.

Lab Session 07

OBJECTIVES:

- Algorithms with loops
- Example Program for Executing Algorithms using Loops
 1. Random Number Generation
 2. Fibonacci Sequence
 3. GCD using Naïve Method
 4. GCD using Euclid Slow Method
 5. GCD using Fast Euclid Method
 6. Square Root using Newton Raphson's Method
 7. Square Root using Bisection Method
 8. Trapezoidal Integration
 9. Prime and Composite Detection

THEORY & PROGRAMS:

What is Algorithm?

In Programming, Algorithm is a set of well-defined instructions in sequence to solve a problem. An algorithms should have a clear stopping point

Random Number Generation:

In C++, rand() function defined in cstdlib is used to generate random numbers in the range of 0 to RAND_MAX (a symbolic constant defined in cstdlib).

Let's have a look!

```
//Random Number Generation
#include<iostream>
#include<cstdlib>
using namespace std;
int main()
{
    for(int i=0; i<10; i++)
    {
        cout<<rand()<<endl;
    }
}
```

To generate random numbers in a specific range, modulo operator is used. Let's have a look!

Fibonacci Sequence:

Fibonacci Sequence is characterized by the fact that every number in it is a sum of two preceding ones.
1,1,2,3,5,8,....

```
#include<iostream>
using namespace std;
int main()
{
    int n,counter;
    char x;
    int prev_term = 1,curr_term = 1,sum;
    cout<<"Enter number of terms to generate: ";
    cin>>n;
    cout<<prev_term<<" ";
    for(counter=1; counter<n; counter++)
    {
        if(counter%10 == 0)cout<<endl;
        cout<<curr_term<<" ";
        sum = prev_term + curr_term;
        prev_term = curr_term;
        curr_term = sum;
    }
}
```

GCD:

In mathematics, GCD of a two or more integers, is the largest positive integer that divides the number without remainder.

There are three methods to calculate GCD:

1. Using Naïve Method
2. Euclid Slow Method
3. Euclid Fast Method

```
#include<iostream> //Using Naïve Method
using namespace std;
int main()
{
    int first_num, second_num, gcd;
    cout<<"Enter first number: ";
    cin>>first_num;
    cout<<"Enter second number: ";
    cin>>second_num;
    for(int i=1;i<=first_num&& i<=second_num; i++)
    {
        if(first_num%i==0 && second_num%i==0)
        {
            gcd = i;
        }
    }
    cout<<"Greatest Common Divisor(GCD): "<<gcd<<endl;
}
```

```
#include<iostream>                //Using Euclid Slow Method
using namespace std;
int main()
{
    int num1, num2;
    cout<<"Enter first number: ";
    cin>>num1;
    cout<<"Enter second number: ";
    cin>>num2;
    while(num1 != num2)
    {
        if(num1>num2)
        {
            num1 = num1 - num2;
        }
        else
        {
            num2 = num2 - num1;
        }
    }
    cout<<"GCD is: "<<num1;
}
```

```
#include<iostream>                //Using Euclid Fast Method
using namespace std;
int main()
{
    int num1,num2,r,a,b;
    cout<<"Enter first number: ";
    cin>>num1;
    cout<<"Enter second number: ";
    cin>>num2;
    if(num1>num2)
    {
        a = num1;
        b = num2;
    }
    else
    {
        a = num2;
        b = num1;
    }
    while(b != 0)
    {
        r = a%b;a = b;b = r;
    }
    cout<<"GCD is: "<<a;
}
```

The Euclidean Algorithm is based on the principle that the GCD of two numbers doesn't change if the larger number is replaced by its difference with the smaller number. For example: GCD of 252 and 105 is 21 and the same number 21 is also the GCD of 105 and $147=252-105$.

A more efficient version of the algorithm is to replace subtraction of larger and smaller number by division of larger and smaller and replacing the number by remainder.

Newton Raphson's Method:

Newton Raphson's method is a method of finding successively better approximation to the roots of real valued function.

$$x: f(x) = 0$$

To find the value of x following iterative formula is used.

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Let us now calculate square root of a number using Newton's Raphson's method.

```
#include<iostream>
#include<cmath>
using namespace std;
int main()
{
    double N,root,counter=0;
    cout<<"Enter a +ve number to calculate its square root: ";
    cin>>N;
    root = N/2.0;
    while( fabs(((root*root)-N))>0.0001 )
    {
        root = root - (((root*root)-N)/(2.0*root));
        counter++;
    }
    cout<<"Square root of "<<N<<" is: "<<root;
}
```

Bisection Method:

The bisection method in [mathematics](#) is a [root-finding method](#) that repeatedly [bisepts](#) an [interval](#) and then selects a subinterval in which a [root](#) must lie for further processing.

The method is applicable for numerically solving the equation $f(x) = 0$ for the [real](#) variable x , where f is a [continuous function](#) defined on an interval $[a, b]$ and where $f(a)$ and $f(b)$ have opposite signs. In this case a and b are said to bracket a root since, by the [intermediate value theorem](#), the continuous function f must have at least one root in the interval (a, b) .

```
#include<iostream>
#include<cmath>
using namespace std;
int main()
{
    double a,b,c,x,N;
    cout<<"Enter a number +ve number: ";
    cin>>N;
    a = 1.0; b = N;
    x = 1.0;
    while(fabs((x*x)-N)>0.0001)
    {
        c = 0.5*(a+b);
        x = c;
        if(x*x-N<0)
            a = c;
        else
            b = c;
    }
    cout<<endl<<"Root is: "<<x;
}
```

Trapezoidal Integration:

In [mathematics](#), and more specifically in [numerical analysis](#), the trapezoidal rule (also known as the trapezoid rule or trapezium rule) is a technique for approximating the [definite integral](#).

$$\int_a^b f(x)dx$$

The trapezoidal rule works by approximating the region under the graph of the function $f(x)$ as a [trapezoid](#) and calculating its area.

The approximation the integral becomes,

$$\int_a^b f(x)dx = \frac{h}{2} \sum_{k=1}^N (f(x_{k+1}) + f(x_k))$$

```
#include<iostream>
#include<cmath>
using namespace std;
int main()
{
    double x,a,b,delta=0.0001,sum=0.0,fx1,fx2,deg_a,deg_b;
    double pi = 3.14159265359;
    int counter,counter_max;
    cout<<"Enter x_min in degrees: ";
    cin>>deg_a;
    cout<<"Enter x_max in degrees: ";
    cin>>deg_b;
    a = deg_a*pi/180.0;
    b = deg_b*pi/180.0;
    counter_max = ((b-a)/delta);
    x = a;
    for(counter=1; counter<=counter_max; counter++)
    {
        fx1 = sin(x);
        fx2 = sin(x+delta);
        sum = sum + (fx1+fx2);
        x = x + delta;
    }
    sum = 0.5*sum*delta;
    cout<<"integral = "<<sum;
    return 0;
}
```

Exercise:

Q1: Halley's Method for Determination of roots of polynomial is

$$x_{n+1} = x_n - \frac{2f(x_n)f'(x_n)}{2f'(x_n)^2 - f(x_n)f''(x_n)}$$

Write a program to find roots of the following polynomial using Halley's method, precision is 0.0001

$$f(x) = ax^3 + bx^2 + cx + d$$

Q2: Write an iterative algorithm to implement the following expansion (precision upto 0.0001)

$$\sin x = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!} x^{2n+1} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \dots \quad \text{for all } x$$

Lab Session 08

OBJECTIVES:

Arrays in C/C++.

- Understanding array as a sequential data structure
- Declaring and initializing arrays
- Using loops to manipulate/process arrays
- Working with 2D arrays
- Using functions to manipulate/process arrays

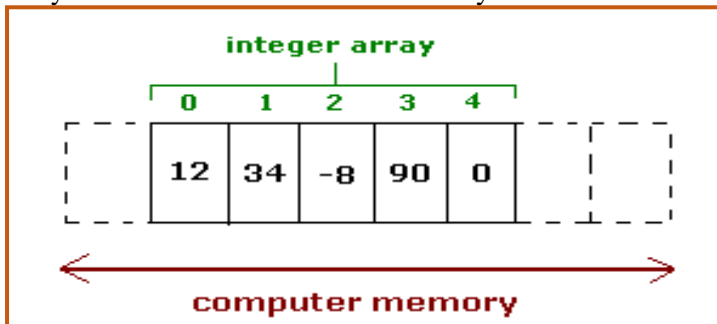
THEORY & PROGRAMS:

Array:

An array is the list of variables of a certain data type having a single name, define contiguously in the memory.

In C/C++ an array is called a “subscripted variable” for obvious reasons. It is the very first step towards the data structures.

An integer array of 5 elements would look like this in the memory. The memory locations occupied by the array are numbered 0 to 4 while the array itself is the content of these locations.



Declaring an Array:

C++ has 5 basic variable data types. Since an array is a list of variables it can also be defined as any one of them namely; *integer*, *float*, *character*, *long* and *double*.

Try the following program to understand array definition.

- `int num[3];`
 - `float temperature[7];`
 - `char myname[25];`
 - `unsigned int roll_num[270];`
 - `double cgpa[100];`
- And this is allowed:
- `const int size = 30;`
 - `int arr[size];`

The general method for defining an array is:

datatype arrayname [no. of elements]

Points to note here:

- 1) *no. of elements* is always constant and can't be taken from user
- 2) Usually *no. of elements* are given as a `#define` directive or `const int` variable
Example: `#define LIM 3,`
`const int N = 12;`
- 3) C/C++ never stops the programmer from exceeding the array limit (e.g. `cin >> num[5]`, so you need to be cautious regarding that).

Initializing an Array:

Initializing an array means declaring it and assigning some initial values to it. This can be done easily by the following syntax shown in the program.

```
#define LIM 3
int main(void)
{
    float num[LIM]={12.9, 9.0, 986.89}; // Declaration of an array
    char ch[]={'a','b','c','d'};        // Declaration of char array, no. of
                                        // elements not assigned
    cout<<"float array:"<<num[0]<<" "<<num[1]<<" "<<num[2];
    cout<<"char array:"<<ch[0] <<" "<<ch[1] <<" "<<ch[2] <<" "<<ch[3];
    return 0;
}
```

- The syntax for initializing values is simply the use of parenthesis { } and putting the values in it.
- Even when the no. of elements is not defined, the compiler fixes it to a constant value that is equal to the no. of elements actually present.
- Declaration does not require the no. of elements to be explicitly given since the compiler calculates it automatically.

Practical Programming with Arrays:

Practically speaking we exploit one special property of arrays to work with them. As the *array index* – the number written inside the square bracket – is always going to be an integer value, we define a separate integer variable and write it within the square bracket (example: *num[index]*). This empowers us to manipulate the array by changing the index variable.

Once the index variable is defined, the best way to manipulate it through a loop. The following example uses *for()* loops to perform operation on an array.

Use of *for()* and *while()* loops in Programming Arrays:

Arrays can be manipulated with *for()* loops. This is ideal in the case of arrays that need to be processed completely – all elements, first to last. But when we need part of array which has some unused locations, we need *while()* loops. In case of using *while()* loops for controlling array operations where all array locations are not utilized there are two special considerations.

In the next example we will write a program that has a character array of 25 elements. Some of these are used by the user to enter his/her name while others remain unused.

```
// Proper method of processing arrays with
// for() loops
#include<iostream>
#include<conio2.h>
using namespace std;

#define LIM 10

int main(void)
{
    // variable definition
    // 10 element float array, numbered 0 to 9
    float num[LIM];
    int index_count;// an integer variable to access different array
    locations

    //Automatic initialization with even numbers
    for(index_count=0;index_count<LIM;index_count++)
    {
        num[index_count]=(index_count+1)*2;
    }

    cout<<"\nPress any key to continue";
    getch();

    // loop for printing output
    for(index_count=0;index_count<LIM;index_count++)
    {
        cout<<"\nElement no."<<index_count<<" = "<<num[index_count];
    }

    getch();
    return 0;
}

// Note:
// 1) A for() loop is feasible when we have to process all elements of
// anarray.
// 2) We have used three loops for three tasks; input, processing, and
// output.
// We could have done this through a single loop but we shall make our
// programs modular in this way.
```

Array Processing with *while()* loop:

```
//processing arrays with while() loops
#include<iostream>
#include<conio2.h>
using namespace std;

int main(void)
{
    const int MAX=100;
    // Array/variable definition
    char name[MAX];
    int index;
    int maxindex;
    cout<<"\nEnter your name (press ESC to stop)\n";
    // Taking input in array locations:
    // index goes from 0 to unknown value
    index=0;
    name[index]=getche();
    while( name[index]!=27 )
    {
        index++;
        if(index==100)
        {
            cout<<"\nArray Overflow\n";
            break;
        }
        name[index]=getche();
    }
    maxindex=index;
    // Simple processing of name[] array, converting small into capital
case
    // index goes upto maxindex less 1
    for(index=0; index<maxindex; index++)
    {
        if((name[index]>=97) && (name[index]<=122))
        {
            name[index]=name[index]-32;
        }
    }
    cout<<"\nPrinting the processed array.\n";
    for(index=0;index<maxindex; index++)
    {
        cout<<name[index];
    }
    return 0;
}
```

Two Dimensional Arrays:

C++ allows the programmer to define and work with multi-dimensional arrays. Multidimensional arrays find extensive usage in programming specially when handling large data-set. In this second part of the lab we will see how to use 2-dimensional arrays.

Representation of 2D Array:

A 2D array, also called an array of arrays, is practically nothing but a two dimensional grid of numbers or characters. Just like 1D array which has a max length, a 2D array has both max length and max width.

	0	1	2	3
0	12.9	-78.0	90.8	0.0
1	0.0	0.0	569.7	100.0
2	55.0	-10.0	988.1	12.0

A 3x4 (2D) float array

Declaring and Initializing a 2D Array:

A 2D array requires both ROWS and COLUMNS to be defined in separate square brackets.

Example: `int num[5][10];` //a 5 row and 10 column integer array
`char names[5][20]` //a character array in which 5 names can be stored each with 20
//characters length

Declaration is also similar for a 2D array that is why using curly brackets {}.

Example: `float price[2][3] = {{12.0,3.4,56.8},{0.0,23.8,65.8}};`
`char cars[3][7] = {{'t','o','y','o','t','a','a'},{'k','i','a'},{'h','o','n','d','a'}};`
`char cars[3][7] = {"toyota","kia","honda"};`

Practical Programming with 2D Arrays:

Considerations for practical programming with 2D arrays are same as those for 1D arrays. However there is one understandable difference to access all location of a 2D array, you need a nested loop.



Using Nested *for()* loops to process arrays:

```
#include<iostream>
#include<conio2.h>
using namespace std;
#define row 5
#define col 3
int main(void)
{
    int i,j;
    float num[row][col]; // a global array
    cout<<"Enter elements of array:";
    for(i=0;i<row;i++) // loop for scanning
    {
        for(j=0;j<col;j++)
        {
            cout<<"\nEnter location "<<i<<" "<<j<<":";
            cin>>num[i][j];
        }
    }
    cout<<"\n \n";
    for(i=0;i<row;i++) // loop for printing
    {
        for(j=0;j<col;j++)
        {
            cout<<"Location "<<i<<" "<<j<<""
            <<num[i][j]<<"\t";
        }
        cout<<"\n";
    }
    getch();
    return 0;
}
```

In the previous example, the array was manipulated with nested *for()* loops. This is ideal in the case of arrays that need to be processed completely – all elements, first to last. But when we need only a part of array which has some unused locations, we need nested *while()* loops.

In the next example we will write a program that has a 2D character array of 10x25 elements. Some of these are used by the user to enter his/her name while others remain unused.

```
#include<iostream>
#include<conio2.h>
using namespace std;
#define row 10
#define col 25
int main(void)
{
    static char names[row][col]; // static class is empty
    int i,j,maxrow,maxcol;
    char ch;
    cout<<"Enter name of 10 students,press ESC to stop\n";
    i=0;
    ch=1;
    while(ch!=27)
    {
        // 1st while start
        j=0;
        cout<<"\n Enter name "<< i+1<<":";
        ch=getche();
        while((ch!=13)&&(ch!=27))
        {
            // 2nd while starts
            names[i][j]=ch;
            j=j+1;
            if(j==25)
            {
                cout<<"\n Too long name.";
                i=i-1;
                break;
            }
            ch=getche();
        }
        // 2nd while ends
        i=i+1;
        if(i==10)
        {
            cout<<"\n No. of names exceeded";
            break;
        }
        cout<<"\nPress ESC to stop, any other key to continue.";
    }
    // 1st while ends
    maxrow=i;
    cout<<endl<<endl;
    for(i=0;i<maxrow;i++) // loop for printing
    {
        for(j=0;j<col;j++)
        {
            cout<<names[i][j];
        }
        cout<<endl;
    }

    getch();
    return 0;
}
```

Conclusions for practical 2D Array Programming:

- For the programs where you need to utilize the array completely, use *for()* loops – nothing special.
- For the programs in which a part of array is used, we use *while()* loops.
 - ✓ The total no. of rows and columns actually utilized by the array needs to be saved for the future use in the program.
 - ✓ *while()* loop needs a separate termination condition to avoid exceeding array limit.

Calling functions to process arrays:

Arrays can also be passed to functions for input, processing and output. One vital difference between passing variable and passing array to function is that ‘Calling’ a function with array as argument is a ‘Call by reference’ – when array is passed to function, its address is passed to it.

Method for Writing function that passes arrays:

- Write prototype with name of array to be used inside function with proper data-type.
int bubblesort(float array[], int N, char order);
- Write the function inside which the array is processed. Note that this function shall modify the original array in *main()* function, thus no need to return the array.

```
#include<iostream>
#include<conio2.h>
using namespace std;
// Function Prototype
int bubblesort(float array[], int N, char order);
// array is the array to sort, N is the array size
// and order dictates ascending/descending sorting
int main(void)
{
    float num[10]={10.1,2.0,34.5,4.6,-5.7,
                  6.2,77.0,18.8,9.4,0.0};

    char option;
    int index;
    cout<<"\nPrinting the given array.\n\n";
    for(index=0;index<10; index++)
        cout<<num[index]<<endl;
    cout<<"\nHow would you like to sort it?"<<
        " press a for ascending d for";
    option=getche();
    bubblesort(num, 10, option); // Function Call
    cout<<"\n\nPrinting the sorted array.\n\n";
    for(index=0;index<10; index++)
        cout<<num[index]<<endl;
    getch();
    return 0;
} // End of main()

int bubblesort(float array[], int N, char order)
{
    float buffer;
    int pivot, index;
    // Defensive Condition: in case of error in N and order values
    if((N<=0)||((order!='a')&&(order!='d'))){
        return 0;
    }
    for(pivot=0;pivot<N-1;pivot++)
    {
        for(index=pivot;index<N;index++)
        {
            if(order=='a')
            {
                if(array[pivot]>array[index])
                {
                    buffer=array[pivot];
                    array[pivot]=array[index];
                    array[index]=buffer;
                }
            }
            if(order=='d')
            {
                if(array[pivot]<array[index])
                {
                    buffer=array[pivot];
                    array[pivot]=array[index];
                    array[index]=buffer;
                }
            }
        }
    }
    return 1;
}
```

Important Considerations for passing Arrays to Function:

- An array is passed to function by reference; the actual array address is given to the function hence making return un-necessary.
 - In function prototype and definition, array name is followed by empty square bracket.
int bubblesort(float array[], int N, char order);
- For 2D arrays, function prototype and definition has array name followed by 2 pairs of square bracket – for rows and columns. But in 2D arrays the second pair contains the no. of columns while the first pair remains empty.
void printarray(array[][5], int size);
- For any array function, the function call only requires the array name to be passed, not the brackets.
bubblesort(num, 10, 'a');

Exercise:

Q1: Read in 20 numbers in an array, each of which is in between 10 and 100 – if the number is not in this range, ask user to re-enter. As each number is read by the program, print it only if it is not a duplicate of a number already read.

Q2: Write a simple database program that stores name, roll no., and cgpa in FE , all in separate arrays, for 25 students. The program should be able to let the user enter records, display records and replace any one of the records (switch()-case can be used to give these options to user). The program must continue until ESC is pressed. [Note: A single 'record' means name, roll no., and cgpa of one student]..

Lab Session 09

OBJECTIVES:

To become familiar with User Defined Functions in C++.

- Procedure to write simple C/C++ functions
- Understanding the process of function call and their handling by computer.
- Writing functions that accepts values and reference.
- Making your header file to keep your function in one place

THEORY & PROGRAMS:

Functions:

Function is a self-contained one piece of code with some inputs upon which it does processing and returns the output.

Number of inputs in a function can be any value and any data-type. However, in a C/C++, function can only return one output. This is handicap when writing functions required to generate multiple outputs.

Procedure for writing Functions:

There are three steps for writing Functions,

Step 1: Specify input(s) / output and write “prototype” before *int main (void)*

Step 2: Write function “definition” (the actual piece of code) after *int main (void)*, make sure you have returned the correct output at the completion of definition.

Step 3: “Call” the function where ever it is needed in the *main()* function.

Why Use Functions:

Functions provide a number of benefits that make them extremely useful in non-trivial programs.

- Organization: As programs grow in complexity, having all the code live inside the *main()* function becomes increasingly complicated. A function is almost like a mini-program that we can write separately from the main program, without having to think about the rest of the program while we write it. This allows us to divide complicated tasks into smaller, simpler ones, and drastically reduces the overall complexity of our program.
- Reusability: Once a function is written, it can be called multiple times from within the program. This avoids duplicated code and minimizes the probability of copy/paste errors. Functions can also be shared with other programs, reducing the amount of code that has to be written from scratch (and retested) each time.
- Testing: Because functions reduce code redundancy, there’s less code to test in the first place. Also because functions are self-contained, once we’ve tested a function to ensure it works, we don’t need to test it again unless we change it. This reduces the amount of code we have to test at one time, making it much easier to find bugs (or avoid them in the first place).

- **Extensibility:** When we need to extend our program to handle a case it didn't handle before, functions allow us to make the change in one place and have that change take effect every time the function is called.
- **Abstraction:** In order to use a function, you only need to know its name, inputs, outputs, and where it lives. You don't need to know how it works, or what other code it's dependent upon to use it. This is super-useful for making other people's code accessible (such as everything in the standard library).

To Calculate Factorial of a Number:

```
//Program that calculates factorial of a number
#include<iostream>
#include<iomanip>
using namespace std;

//Function Prototype
int factorial(int);

int main(void)
{
    int number, fact;
    cout<<"Enter a positive number:";
    cin>>number;
    cout<<"\nFactorial of "<<number
        <<" is "<<factorial(number);
    return 0;
}

//Function Definition
int factorial(int num)
{
    int counter, answer=1;
    if(num<0 || num>15)
    {
        return 0; //indicates error
    }
    if(num==0)
    {
        return 1; //special case
    }
    for(counter=1;counter<=num;counter++)
    {
        answer=answer*counter;
    }
    return answer;
}
```

Functions that returns nothing – Void Functions:

```
#include<iostream>
#include<conio2.h>
using namespace std;

void delay(int num);

int main(void)
{
    getch();
    cout<<"Testing void function delay()\n";
    delay(1);
    cout<<"This line is printed after delay value 1\n";
    delay(2);
    cout<<"This line is printed after delay value 2\n";
    delay(5);
    cout<<"This line is printed after delay value 5\n";
    return 0;
}

void delay(int num)
{
    for(int counter=1;counter<=num*100000000;counter++)
    {

    }
}
```


Functions that returns Boolean – Predicate Functions:

```
#include<iostream>
#include<conio2.h>
#include<cmath>
using namespace std;

bool iseven(int num);

int main (void)
{
    int N;
    cout<<"Enter a number:";
    cin>>N;
    if(iseven(N)==1)
    {
        cout<<"\nThe number is even.";
    }
    else
    {
        cout<<"\nThe number is odd.";
    }
    return 0;
}

bool iseven(int num)
{
    bool result;
    if(num%2==0)
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

Functions with two or more Outputs – using Global Variables:

```
// Function to calculate two roots of
// quadratic equation and give back to
// main()
#include<iostream>
#include<conio2.h>
#include<cmath>
using namespace std;
float quadroots(float,float,float);
// Function Prototype:quadroots() returns the
// 1st root of equation, second root is saved
// in global variable root2

float root2;
// Global Variable: Can be seen and
// manipulated by all functions

int main(void)
{
    float num1,num2,num3,root1;
    cout<<"\t\tProgram for calculating roots"
        <<" of Quadratic Equation\n\t\t"
        <<" of the form ax^2+bx+c=0";
    cout<<"\nEnter the 1st co-efficient a:";
    cin>>num1;
    cout<<"\nEnter the 2nd co-efficient b:";
    cin>>num2;
    cout<<"\nEnter the constant c:";
    cin>>num3;
    root1=quadroots(num1,num2,num3);
    //Function Call: root1 is returned by quadroots()
    cout<<"The roots are "<<root1<<" "<<root2;
    getch();
    return 0;
}

float quadroots(float a,float b,float c) // Function Definition
{
    float root1;
    // Discriminant is -ve, complex roots
    if(((b*b)-(4*a*c))<0.0)
    {
        cout<<"\nRoots are imaginary,"
            <<" can't be calculated\n";
        root1=0.0;
        root2=0.0;
    }
    else
    {
        root1=(-b+sqrt((b*b)-(4*a*c)))/(2*a);
        root2=(-b-sqrt((b*b)-(4*a*c)))/(2*a);
    }
    return root1;
}
```

Understanding Global Variables:

- Global variables are seen by and accessible to all functions.
- They are declared outside the scope of any function
- Care should be taken in using Global variables as they can be changed by any function in program.
- Declaring many Global variables is a bad programming practice.

Understanding “Call by Reference”:

- Function call by reference means sending the variable address instead of its value.
- Variable can be sent to a function by reference, by writing address operator (&) before its name in function prototype and definition

Functions with two or more Outputs – using call by reference:

```
// Function to calculate two roots of
// quadratic equation and give back to
// main()
#include<iostream>
#include<conio2.h>
#include<cmath>
using namespace std;
void quadroots(float a,float b,float c, float &root1,float &root2);
// Function Prototype:quadroots() returns the
// root of equation,through arguments passed
// as input

int main(void)
{
    float num1,num2,num3,r1,r2;;
    cout<<"\t\tProgram for calculating roots of Quadratic Equation\n\t\t"
        <<" of the form ax^2+bx+c=0";
    cout<<"\nEnter the 1st co-efficient a:";
    cin>>num1;
    cout<<"\nEnter the 2nd co-efficient b:";
    cin>>num2;
    cout<<"\nEnter the constant c:";
    cin>>num3;
    quadroots (num1,num2,num3,r1,r2);
    //Function Call: nothing is returned,
    //root values come back via input arguments
    //r1 and r2
    cout<<"The roots are "<<r1<<" "<<r2;
    getch();
    return 0;
}

void quadroots(float a,float b,float c,float &root1,float &root2)
{
    // Discriminant is -ve, complex roots
    if(((b*b)-(4*a*c))<0.0)
    {
        cout<<"\nRoots are imaginary,"
            <<" can't be calculated\n";
        root1=0.0;
        root2=0.0;
    }
    else
    {
        root1=(-b+sqrt((b*b)-(4*a*c)))/(2*a);
        root2=(-b-sqrt((b*b)-(4*a*c)))/(2*a);
    }
    //return is not needed
}
```

Exercise:

Q1: Write program with a function that accepts 3 int type numbers and returns the smallest among them. The function is called minimum().

Q2: Write a void function that generates a precise delay of 2 seconds whenever it is called. The function should contain clock() function or time() function from ctime, for precise timing.

Lab Session 10

OBJECTIVES:

Recursive Functions in C/C++

THEORY & PROGRAMS:

Recursion:

Recursion in Computer science is a method where the solution to a problem depends on a solution to smaller instances of the same problem. The power of recursion evidently lies in the possibility of defining infinite set of objects by a finite statement.

Recursive Functions:

Recursive function is the one that calls itself to repeat the code. Recursive function calls generally work just like normal function calls. The most important difference with recursive function is you must include a recursive termination condition, or they will run forever (actually, until the call stack runs out of memory). A recursive *termination condition* is a condition that, when met, will cause the recursive function to stop calling itself.

Recursive termination generally involves *if* statement.

Have a look at an examples on how recursion process can be used to convert decimal into binary and to calculate factorial of a number.

```
//program to convert to decimal number into binary
#include<iostream>
using namespace std;
void convertToBin(unsigned int n);
int main()
{
    unsigned int num;
    cout<<"Enter a decimal number: ";
    cin>>num;
    convertToBin(num);
}
void convertToBin(unsigned int n)
{
    if((n/2) != 0)
    {
        convertToBin(n/2);
    }
    cout<<n%2;
}
```

```
//program to calculate the factorial of a number
#include<iostream>
using namespace std;
unsigned long factorial(unsigned long val);
int main()
{
    unsigned long num;
    cout<<"Enter a number to find its factorial: ";
    cin>>num;
    cout<<"Factorial of a "<<num<<" is: "<<factorial(num);
}
unsigned long factorial(unsigned long val)
{
    if(val == 1 || val == 0)
    {
        return 1;
    }
    if (val>1)
    {
        return val*factorial(val-1);
    }
}
```

Fabonacci sequence for nth number can be calculated using recursive functions.

```
//program to display nth fabonacci numbers
#include <iostream>
int fibonacci(int number)
{
    if (number == 0)
        return 0; // base case (termination condition)
    if (number == 1)
        return 1; // base case (termination condition)
    return fibonacci(number-1) + fibonacci(number-2);
}
// And a main program to display the first 13 Fibonacci numbers
int main()
{
    for (int count=0; count < 13; ++count)
        std::cout << fibonacci(count) << " ";

    return 0;
}
```

Recursion vs Iteration:

One question that is often asked about recursive functions, “Why use a recursive function if the same can be done through iteration (using *for* or *while* loops)?” It turns out that you can always solve the recursive problem iteratively. However, for non-trivial problems, the recursive version is often much simpler to write (and read).

Iterative functions (those using *for* or *while* loop) are always more efficient than recursive counterparts. This is because every time you call a function there is some amount of overhead that takes place in pushing and popping stack frames. Iterative functions avoid this overhead.

That’s not to say iterative functions are always a better choice. Sometimes the recursive implementation of a function is so much cleaner and easier to follow that incurring a little extra overhead is more than worth it for the benefit in maintainability, particularly if the algorithm doesn't need to recurse too many times to find a solution.

In general, recursion is a good choice when most of the following are true:

- The recursive code is much simpler to implement.
- The recursion depth can be limited (e.g. there’s no way to provide an input that will cause it to recurse down 100,000 levels).
- The iterative version of the algorithm requires managing a stack of data.
- This isn’t a performance-critical section of code.

However, if the recursive algorithm is simpler to implement, it may make sense to start recursively and then optimize to an iterative algorithm later.

Exercise:

Q1: Write a recursive function to implement Newton Raphson Method algorithm to determine square root of a number.

Q2: Write a recursive function to find Greatest Common Divisor of two numbers using Euclid Remainder Algorithm.

Q3: Test for a number if it is prime or composite using recursion.

Q4: Write a recursive function to implement the following expansion (precision upto 0.0001)

$$\sin x = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!} x^{2n+1} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \dots \quad \text{for all } x$$

Lab Session 11

OBJECTIVE:

Introduction to Pointers in C/C++

THEORY & PORGRAMS:

Pointers:

A pointer is a variable that holds a *memory address* as its value. Pointers are typically seen as one of the most confusing part of the C++ language, but are surprisingly simple when explained properly.

Before moving deeper into Pointers we need to understand two basic concept related to it.

The *address-of* Operator (&):

The *address-of* operator (&) allows us to see what memory address is assigned to a variable. This is

```
#include <iostream>

int main()
{
    int x = 5;
    std::cout << x << '\n'; // print the value of variable x
    std::cout << &x << '\n'; // print the memory address of variable x

    return 0;
}
```

pretty straightforward.

Note: Although the *address-of* operator looks just like the bitwise-and operator, you can distinguish them because the address-of operator is unary, whereas the bitwise-and operator is binary.

The *Dereference* Operator (*):

The dereference operator (*) allows us to get the value at a particular address:

```
#include <iostream>

int main()
{
    int x = 5;
    std::cout << x << '\n'; // print the value of variable x
    std::cout << &x << '\n'; // print the memory address of variable x
    std::cout << *x << '\n'; // print the value at the memory address of variable x

    return 0;
}
```

Note: Although the dereference operator looks just like the multiplication operator, you can distinguish them because the dereference operator is unary, whereas the multiplication operator is binary.

Declaring a Pointer:

Pointer variables are declared just like normal variables, only with an asterisk between the data type and the variable name.

For e.g:

```
int *iPtr;           //a pointer to an integer value
```

```
double *dPtr;        //a pointer to a double value
```

Note that the asterisk here is not a dereference. It is part of the pointer declaration syntax.

Syntactically, C++ will accept the asterisk next to the data type, next to the variable name, or even in the middle.

However, when declaring multiple pointer variables, the asterisk has to be included with each variable.

If we get used to declare pointers with asterisk next to data type and we are declaring multiple variables, then the first declared variable will be the pointer but the other will just be a plain variable.!

Have a look at this.

```
int* iPtr1, iPtr2;    //iPtr1 is a pointer but iPtr2 is not
```

For this reason, when declaring pointers, it is recommended to put asterisk next to variable name.

Assigning a value to a Pointer:

Since pointers only hold addresses, when we assign a value to a pointer, that value has to be an address. One of the most common things to do with pointers is have them hold the address of a different variable.

To get the address of a variable, we use the address-of operator:

```
int value = 5;
```

```
int *ptr = &value;    //initialize ptr with address of a variable value
```

Conceptually you think of the above snippet like this:



The type of the pointer has to match the type of the variable being pointed to:

```
int iValue = 5;
```

```
double dValue = 7.0;
```

```
int *iPtr = &iValue;    //ok
```

```
double *dPtr = &dValue; //ok
```

```
iPtr = &dValue;         //wrong
```

```
dPtr = &iValue;         //wrong
```

Note that the following is also not legal:

```
int *ptr = 5;
```

This is because pointers can only hold addresses, and integer literal 5 doesn't have a memory address. If you try this, the compiler will tell you it cannot convert an integer to an integer pointer.

Dereferencing Pointers:

Once we have a pointer variable pointing at something, the other common thing to do with it is dereference the pointer to get the value of what it's pointing at. A dereferenced pointer evaluates to the *contents* of the address it is pointing to.

```
int value = 5;
cout << &value;    //prints address of value
cout << value;      //prints contents of value
int *ptr = &value  //ptr points to a value
cout << ptr;        //prints address held in ptr, which is &value
cout << *ptr;       //dereference ptr (get the value that ptr is pointing to)
```

This is why pointers must have a type. Without a type, a pointer wouldn't know how to interpret the contents it was pointing to when it was dereferenced.

What good are Pointers:

At this point, pointers may seem a little silly, academic, or obtuse. Why use a pointer if we can just use the original variable?

It turns out that pointers are useful in many different cases:

- Arrays are implemented using pointers. Pointers can be used to iterate through an array (as an alternative to array indices).
- They are the only way you can dynamically allocate memory in C++.
- They can be used to pass a large amount of data to a function in a way that doesn't involve copying the data, which is inefficient.
- They can be used to pass a function as a parameter to another function.

Passing Pointers to functions in C++:

C++ allows us to pass a pointer to a function. To do so, simply declare the function parameter as a pointer type. Following a simple example where we pass an unsigned long pointer to a function and change the value inside the function which reflects back in the calling function.

Following example calculates the average value of numbers stored in an array using pointer passed an argument

```
#include <iostream>
using namespace std;

// function declaration:
double getAverage(int *arr, int size);

int main () {
    // an int array with 5 elements.
    int balance[5] = {1000, 2, 3, 17, 50};
    double avg;

    // pass pointer to the array as an argument.
    avg = getAverage( balance, 5 ) ;

    // output the returned value
    cout << "Average value is: " << avg << endl;
```

Following example calculates the average value of numbers stored in an array by using pointer passed as an argument.

```
#include <iostream>
#include <ctime>

using namespace std;
void getSeconds(unsigned long *par);

int main () {
    unsigned long sec;

    getSeconds( &sec );

    // print the actual value
    cout << "Number of seconds :" << sec << endl;

    return 0;
}

void getSeconds(unsigned long *par) {
    // get the current number of seconds
    *par = time( NULL );
    return;
}
```

Exercise:

Q1: Selection sort algorithm can be used to sort an array in ascending order. The first iteration of the algorithm selects the smallest element in the array and swaps it with the first element. The second iteration selects the second-smallest element (which is the smallest element of the remaining elements) and swaps it with the second element. The algorithm continues until the last iteration selects the second-largest element and swaps it with the second-to-last index, leaving the largest element in the last index.

As an example, consider the array

34 56 4 10 77 51 93 30 5 52

A program that implements the selection sort first determines the smallest value (4) in the array, which is contained in element 2. The program swaps the 4 with the value in element 0 (34), resulting in

4 56 34 10 77 51 93 30 5 52

The program then determines the smallest value of the remaining elements (all elements except 4), which is 5, contained in element 8. The program swaps the 5 with the 56 in element 1, resulting in

4 5 34 10 77 51 93 30 56 52

On the third iteration, the program determines the next smallest value, 10, and swaps it with the value in element 2 (34).

4 5 10 34 77 51 93 30 56 52

The process continues until the array is fully sorted.

4 5 10 30 34 51 52 56 77 93

Using pass by reference feature of pointers, implement the selection sort algorithm.

Q.2 Explore all methods of viewing addresses of variables. Also explore address storing mechanisms (pointer variables). Finally use this knowledge to access and manipulate arrays and call multiple variables from functions using pointers.

Lab Session 12

OBJECTIVES:

An Introduction to Filing in C/C++

THEORY & PROGRAMS:

File:

The information / data stored under a specific name on a storage device, is called a file.

Stream:

It refers to a sequence of bytes.

Text file:

It is a file that stores information in ASCII characters. In text files, each line of text is terminated with a special character known as EOL (End of Line) character or delimiter character. When this EOL character is read or written, certain internal translations take place.

Classes for Stream Operation:

ofstream: Stream class to write on files

ifstream: Stream class to read from files

fstream: Stream class to both read and write from/to files

Opening a File:

Opening File using constructor

```
ofstream outFile("sample.txt");    //output only
```

```
ifstream inFile("sample.txt"); //input only
```

Opening File using open()

```
ofstream outFile;
```

```
outFile.open("sample.txt");
```

```
ifstream inFile;
```

```
inFile.open("sample.txt");
```

File mode Parameter	Meaning
ios::app	Append to end of file
ios::in	Open file for reading only
ios::out	Open file for writing only

Each of the open member functions of classes ofstream, ifstream and fstream has a default mode that is used if the file is opened without a second argument:

Class	Default Mode Parameter
ofstream	ios::out
ifstream	ios::in
fstream	ios::in ios::out

For fstream, the default value is only applied if the function is called without specifying any value for the mode parameter. If the function is called with any value in that parameter the default mode is overridden, not combined.

Closing a File:

```
outFile.close();  
inFile.close();
```

Basic Operation in Text File in C++:

File I/O is a five-step process:

1. Include the header file fstream in the program.
2. Declare file stream object.
3. Open the file with the file stream object.
4. Use the file stream object with >>, <<, or other input/output functions.
5. Close the files.

Following are the example programs to illustrate the concept.

Program to write in a text file.

```
#include <fstream>  
using namespace std;  
  
int main()  
{  
    ofstream fout;  
    fout.open("out.txt");  
  
    char str[300] = "Time is a great teacher but  
                    unfortunately it kills all its pupils. Berlioz";  
  
    //Write string to the file.  
    fout << str;  
  
    fout.close();  
    return 0;  
}
```

Program to read from text file and display it.

```
#include<fstream>
#include<iostream>
using namespace std;

int main()
{
    ifstream fin;
    fin.open("out.txt");

    char ch;

    while(!fin.eof())
    {
        fin.get(ch);
        cout << ch;
    }

    fin.close();
    return 0;
}
```

Where, *eof()* function returns a true (non-zero) if end of the file is encountered while reading; otherwise return false (zero). This while loop will continue to run as long as we reached the end of the file.

get() function is used take a single character from text file and print it on console.

Program to count number of character

```
#include<fstream>
#include<iostream>
using namespace std;

int main()
{
    ifstream fin;
    fin.open("out.txt");

    int count = 0;
    char ch;

    while(!fin.eof())
    {
        fin.get(ch);
        count++;
    }

    cout << "Number of characters in file are " << count;

    fin.close();
    return 0;
}
```


Program to copy content of file to another

```
#include<fstream>
using namespace std;

int main()
{
    ifstream fin;
    fin.open("out.txt");

    ofstream fout;
    fout.open("sample.txt");

    char ch;

    while(!fin.eof())
    {
        fin.get(ch);
        fout << ch;
    }

    fin.close();
    fout.close();
    return 0;
}
```

Q.1 Develop a simple text editor application with File, Edit and Fonts Menus. It should be able to create new files, display previously stored text files, edit files and save any changes.