Ahsanullah University of Science and Technology

Department of Electrical and Electronic Engineering

LABORATORY MANUAL

FOR

ELECTRICAL AND ELECTRONIC SESSIONAL COURSE

Student Name :

Student ID :

Course no : EEE 2110

Course Title : Course Title: Programming Language Lab

For the students of

Department of Electrical and Electronic Engineering

2nd Year, 1st Semester

LAB NO : 01
LAB NAME: Introduction to the Local Computer System and the Execution of First C++ Program.

Objective:

The purpose of this laboratory session is to introduce you to the computer system that you will use in the remaining sessions. The materials supplied during this period will teach you to:

1. Gain access to the machine.
2. First time handling with Visual C++ 8.0
3. Enter a C++ program and execute it.

Gaining Access to the Machine

Those of you using this manual may be working with a wide variety of computer equipment. Some may use individual autonomous machines, others may use individual machines that share resources over a network, while others may use remote terminals connected to a central computing facility that serves many users at once. No matter what the machine, however, your first contact with the computer will be through its operating system—a program that coordinates the activities of the machine and performs tasks as directed by the machine's user (or users). In the all lab sessions, your operating system will be Windows.

The Program Preparation Process

The steps required to develop programs using the C++ language will depend on the computer system being used. However, some features of the process are common to all systems. As a first step, the programmer uses a program called an editor to type a C++ language version of the program being developed. This editor may be a stand-alone utility program or a part of an integrated software development package. Once the program has been typed, it is usually saved as a file in mass storage. This version of the program is known as the source program because it is the initial, or source, version of the program. It is this version to which you will return when alterations to the program are required.

A program in its source form cannot be executed by the computer; it must first be translated into the machine's own low-level language. This translation process is performed by a program known as a translator (or compiler).

The compiler`s working procedure is shown in flow chart.

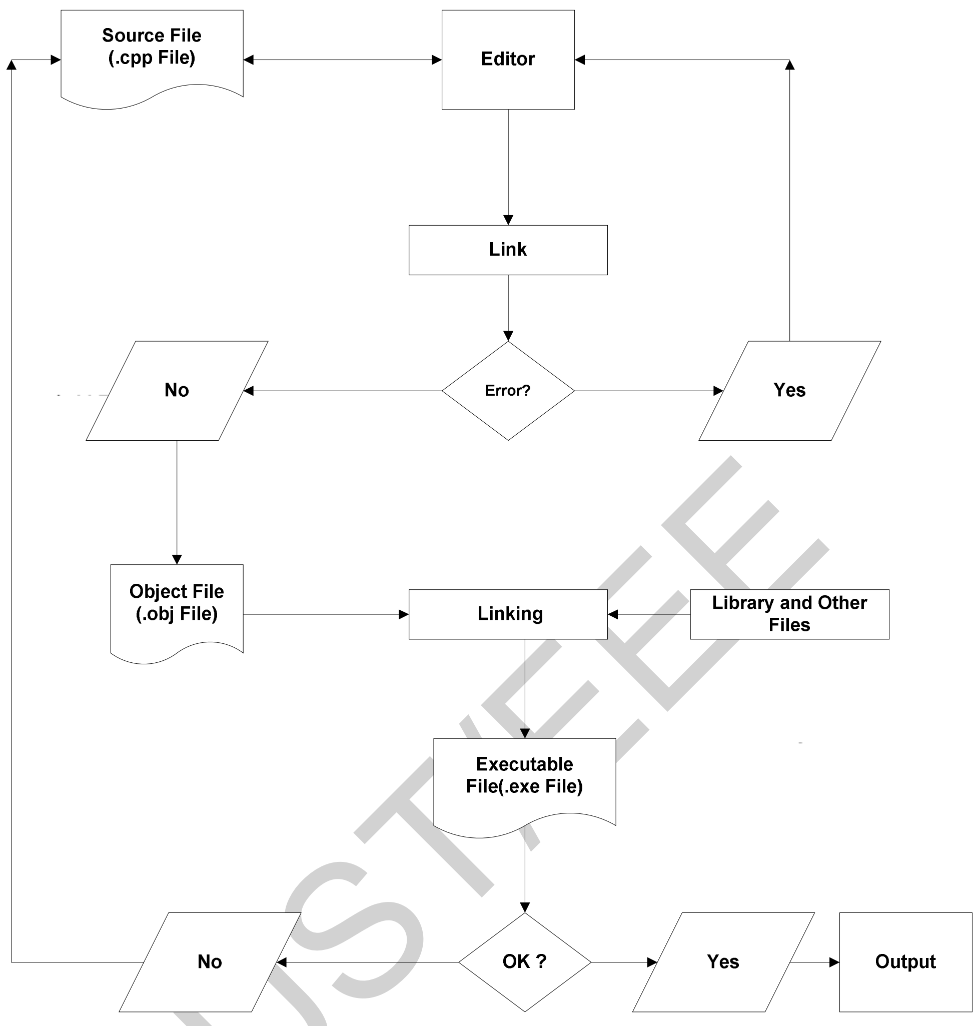


Fig: Compiler`s Working Principle

Mirosoft Visual Studio 2008

Microsoft Visual Studio 2008 is a software development environment that allows a programmer to write, compile, link, execute, test, and debug C++ programs.

1. Creating a Folder:

a.Make a new folder on the desktop.

b.Name it as “EEE2110_[*yourgroup number]*”

2. Launching Microsoft Visual Studio 2008:
a.First make a folder on the desktop. Name it as *EEE2110*.

b.Click on **Start →Programs →Microsoft Visual Studio 2008 .**

c. Visual C++ window will be appeared.

3.Creating a New Project:

a.Click on **File** →click on **New** →click on **Project**.

b.A new popup window will be appeared.

c. i.Select **Win32** under the headline of **Visual C++** of **project types**.

ii. Select **Win32 Console Application** under the headline of **Visual Studio installed templates** of **templates** types.

iii. Give a project **Name** for an example helloworld **and** give a **location** in that window and click **OK**.

d. A **win32 Application Wizard** will be appeared.

Click **Next** →select **Empty Project** in **Additional options** →click **Finish**.

4.Creating a file:

a. Right click on the **Source Files** of your project (for example Helloworld) and click **Add**→click **New Item**.

b. A new window will be appeared.

i.In that window select **Visual C++** listed in **Categories** and select **C++ File(.cpp)** listed in **Templates**.
ii. Give a **Name** of the file for an example lab01 and click **Add**.

c. A new window will be appeared named your file name.cpp for example lab01.cpp. This file is **text editor** where you can write and edit your codes.

5.Building and running a Project:

a. After finishing writing your program on the text editor to build the program Click on **Build** in the toolbar→click **Build solution**.

b. If it compiles successfully, the window at the bottom of your screen will display-

```
1>Helloworld- 0 errors(s) , 0 warnings(s)
 -----------------Build: 1 succeeded, 0 failed
```

c. To execute your code click on **Tools** →click **Customize**→click **Commands**→select **Debug** listed in commands→select and drag **Start Without Debugging** to the toolbar and close the Customize window.

d. By clicking **Start Without Debugging** in the toolbar a window should popup with the output from your program.

6.Debugging C++ Source Code:

a.If you have compiler errors, click the mouse in the window at the bottom of your screen. To see the error message first, scroll to the top of that window. If you doubleclick on an error message, a blue arrow in the coding window will identify the statement corresponding to the error message.

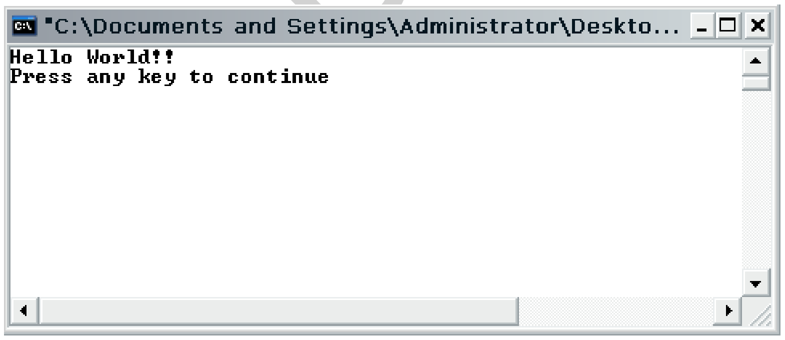
b.Return to the coding window and correct the error.

c. Recompile the program.

Example: 01

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

Output of the program:



```
Hello World!!
Press any key to continue
```

Post-Laboratory Problems

1. Write a program that displays five of your favorite colors name in (i) 1 line (ii) 5 lines.

2. What is the error in the following program? Write down the exact output of the following program after correcting the error.

```
include <iostream.h>
int main()
{
cout << "Patriot " << "G" << "a" << "m" << "e" << "\n"; cout
<< "\t Written" << " " << "B" << "y" << "\n";
cout << "\t\t Tom" << " " << "C" << "lancy\n\n\n";
return 0;
 }
```

3. What is the error in the following program? Write down the exact output of the following program after correcting the error.

```
#include <iostream>
int main()
   // prints "Hello, World!":
cout << "Hello, World!\n"
}
```

4. Write and run a program that prints the sum, difference, product, quotient, and remainder of two integers that are input interactively.

5.Write and run a program that prints the first letter of your last name as a block letter in a7 ×7 grids of stars.

LAB NO : 02

LAB NAME: **Introduction of variables, cin function and simple Arithmetic Operators, Composite Assignment Operator, Increment & Decrement.**

Objective:

The purpose of this laboratory session is to introduce you to the variables that are used in C++ with some common applications as well as cin function. Also some applications simple arithmetic operations are introduced.

Variables:

Variables are the lifeblood of any programming language. Without them, the results of a program would always be the same. This is because variables are the "mailboxes" in which data necessary for program execution to run. You can store many kinds of data in these "mailboxes." Integers, Floating-point numbers, and Characters are a few. The following program is simply going to set a variable and print it out.

```
// prints "n = 44":
#include <iostream.h>
int main()
{
int n;
n = 44;
cout << "n = " <<n<<endl;
return 0;
}
```

After that you see a new statement that looks like **int n;** This declares a variable named **n** of type **int**. **int** means integer. An integer is a type which can only contain non-floating point values. In other words, only whole numbers. Other typesof variables and their keywords are:

Integer	`int`
Long integer	`long int`
Short integer	`short int`
Character	`char`
Floating Point	`float`

Then the line **n = 44;** is commonly known as initialization of variable. This line simply assigns the value of 44 to n. The "mailbox" of the variable now has a letter that says "44". The = operator is called the assignment operator. It assigns the value of whatever is on the right to whatever is on the left. This data can later be used in our program for output, as the next line indicates: **cout << "n = " << n << endl;** This line tells the computer to output the value stored in n (44) with a text “n= ”and then an end-of-line character. It is done the same way with most variables; simply append an extra << operator for each value you wish to attach.

The Input Operator:
In C++, input is almost as simple as output. The input operator “>>” [*also known as get operator /extraction operator*] works like the output operator “<<”. The function for input is “cin”. Following program will clarify the operation of cin:

Example 01:

```
#include <iostream.h>
int main()
{
int m, n;
cout << "Enter two integers (use SPACE to separate): ";
cin >> m >> n;
cout << "m = " << m << ", n = " << n <<endl;
double x, y, z;
cout<< "Enter three decimal numbers (use SPACE to separate): ";
cin >> x >> y >> z;
cout << "x = " << x << ", y = " << y << ", z = " << z <<endl;
char c1, c2, c3, c4;
cout << "Enter four characters (use SPACE to separate): ";
cin >> c1 >> c2 >> c3 >> c4;
cout << "c1 = " << c1 << ", c2 = " << c2 << ", c3 = " << c3;
cout << ", c4 = " << c4 << endl;
return 0;
}
```

Arithmetic Operators:

The following two examples will show the exact use of different types of arithmetic operators:

Example 2 :

```
/* Integer Arithmetic
Testing the operators +, -, *, /, and % */
#include <iostream.h>
int main()
{
int
m=54;
int
n=20;
cout << "m = " << m << " and n = " << n << endl;
cout << "m+n = " << m+n <<endl;
cout << "m-n = " << m-n <<endl;
cout << "m*n = " << m*n <<endl;
cout << "m/n = " << m/n <<endl;
cout << "m%n = " << m%n <<endl;
return 0;
}
```

Example 3:

```
/* Floating-Point Arithmetic
Testing the floating-point operators +, -, *, and / */
#include <iostream.h>
int main()
{
double
x=54.0;
double
y=20.0;
cout << "x = " << x << " and y = " << y <<endl;
cout << "x+y = " << x+y <<endl;
cout << "x-y = " << x-y <<endl;
cout << "x*y = " << x*y <<endl;
cout << "x/y = " << x/y <<endl;
cout<< ”x%y  = ‟ << x%y <<endl;
return 0;
}
```

Example 4:

To observe the storage sizes of fundamental data types compile, build and run the following program. Write the observed output.

```
#include <iostream.h>
int main()
{
cout << "Number of bytes used:\n";
cout << "\t char: " << sizeof(char) <<endl;
cout << "\t short: " << sizeof(short) <<endl;
cout << "\t int: " << sizeof(int) <<endl;
cout << "\t long: " << sizeof(long) << endl;
cout << "\t unsigned char: " << sizeof(unsigned char) <<endl;
cout << "\tunsigned short: " << sizeof(unsigned short) <<endl;
cout << "\t unsigned int: " << sizeof(unsigned int) <<endl;
cout << "\t unsigned long: " << sizeof(unsigned long) <<endl;
cout << "\t signed char: " << sizeof(signed char) <<endl;
cout << "\t float: " << sizeof(float) <<endl;
cout << "\t double: " << sizeof(double) <<endl;
cout << "\t long double: " << sizeof(long double) <<endl;
return 0;

}
```

Example 5:

At its closest point to Earth, Mars is approximately 34,000,000 miles away. Assuming there is someone on Mars that you want to talk with, what is the delay between the time a radio signal leaves Earth and the time it arrives on Mars? This project creates a program that answers this question. Recall that radio signals travel at the speed of light, approximately 186,000 miles per second. Thus, to compute the delay,you will need to divide the distance by the speed of light. Display the delay in terms of seconds and also in minutes.

```
 #include<iostream>
using namespace std;
int main()
{
double distance;
double lightspeed;
double delay;
double delay_in_min;
distance = 34000000.0; // 34,000,000 miles
lightspeed = 186000.0; // 186,000 per second
delay = distance / lightspeed;
cout << "Time delay when talking to Mars: " << delay << " seconds.\n";
delay_in_min = delay / 60.0;
cout << "This is " << delay_in_min << " minutes.";
return 0;

}
```

Compile and run the program. The following result is displayed:

```
Time delay when talking to Mars: 182.796 seconds.

This is 3.04659 minutes.
```

Increment and Decrement:

The values of integral objects can be incremented and decremented with the ++ and -- operators, respectively. Each of these operators has two versions: a "pre" version and a "post" version. The "pre" version performs the operation (either adding 1 or subtracting 1) on the object before the resulting value is used in its surrounding context. The "post" version performs the operation after theobject's current value has been used.

Both the increment and decrement operators can either precede (prefix) or follow (postfix) the operand.
For example,
x = x + 1;can be written as
++x; // prefix form
or as
x++; // postfix form
In this example, there is no difference whether the increment is applied as a prefix or a postfix.However, when an increment or decrement is used as part of a larger expression, there is an importantdifference. When an increment or decrement operator precedes its operand, C++ will perform theoperation prior to obtaining the operand’s value for use by the rest of the expression. If the operatorfollows its operand, then C++ will obtain the operand’s value before incrementing or decrementing it.
Consider the following:
x = 10; y = ++x;
In this case, y will be set to 11. However, if the code is written as
x = 10; y = x++;
then y will be set to 10. In both cases, x is still set to 11; the difference is when it happens. There aresignificant advantages in being able to control when the increment or decrement operation takes place.

Example 6:

```
#include<iostream.h>
int main()
{
int m,n;
m=44;
n=m++;
cout<<"the value of m\t"<<m<<"\t"<<"the value of n\t"<<n<<endl;
m=44;
n=++m;
cout<<"the value of m\t"<<m<<"\t"<<"the value of n\t"<<n<<endl;
m=44;
n=m--;
cout<<"the value of m\t"<<m<<"\t"<<"the value of n\t"<<n<<endl;
m=44;
n=--m;
cout<<"the value of m\t"<<m<<"\t"<<"the value of n\t"<<n<<endl;
return 0;
}
```

Composite assignment operators:

The standard assignment operator in C++ is the equals sign =. In addition to this operator, **C++** alsoincludes the following composite assignment operators *:* +=, -= , *=, /=, and %=. When appliedto a variable on the left, each applies the indicated arithmetic operation to it using the value of theexpression on the right .

Example 7:

Applying Composite Arithmetic Assignment Operators

```
# include<iostream.h>
int main()
{int n=22;
cout << "n = " <<n<<endl;
n +=9; // adds 9 to n
cout << "After n += 9, n = " <<n<<endl;
n -= 5; // subtracts 5 from n
cout << "After n -= 5, n = " <<n<<endl;
n *= 2; // multiplies n by 3
cout « "After n *= 2, n = " <<n<<endl;
n /= 3; // divides n by 9
cout << "After n /= 3, n = " <<n<<endl;
n %= 7; // reduces n to the remainder from dividing by 4
cout << "After n %= 7, n = " <<n<<endl;
return 0;
}
```

The output would be:

```
n = 22
After n + = 9, n = 31
After n -= 5, n = 26
After n *= 2, n = 52
After n /= 3, n = 17
After n %= 7, n = 3
```

Type conversions:

C++ also converts integral types into floating point types when they are expected. For example,

```
int n = 22;
float x = 3.14159;
x += n; // the value 22 is automatically converted to 22.0
cout << x - 2 << endl; // value 2 isautomatically converted to 2.0
```

Converting from **integer to float** like this is what one would expect and is usually taken for granted.But converting from a **floating point type to an integral type** is not automatic.
In general, if **T** isone type and **v** is a value of another type, then the expression **T(v)** converts v to type T. This is called type **casting**. For example, if *expr* is a floating-point expression and n is a variable of type `int`, then `n = int{expr);`converts the value of *expr* to type `int` and assigns it to n. The effect is to remove the real number's fractional part, leaving only its whole number part to be assigned to n. For example, 2.71828 would be converted to 2. Note that this is ***truncating,*** not ***rounding.***

Example 8:

This program casts a double value into int value:

```
void main( )
{ double v =1234.56789;
int n = int(v);
cout << "V = " <<v<< ",n=" <<n<<endl;}

v = 1234.57, n = 1234
```

The double value 1234.56789 is converted to the int value 1234.
When one type is to be converted to a "higher" type, the type case operator is not needed. This iscalled *type promotion*. Here's a list of *promotion* from char all the way up to double:

Char< short <int < long< float <double

Note that type casting and promotion convert the type of the **value** of a variable or expression, but it does not change the type of the variable itself.

Example 9:

This program promotes a char to a short to an int to a float to a double:

```
 void main( )

{char c= ‘A’ ;
cout<< “char c=”<<c<<endl;
 short k= c ;
cout<< “short k=”<<k<<endl;
 int m= k ;
cout<< “int m=”<<m<<endl;
long n= m ;
cout<< “long n=”<<n<<endl;
float x= n ;
cout<< “float x=”<<x<<endl;
double y= x;
cout<< “double y=”<<y<<endl;
}
```

Numeric overflow:

On most computers the long int type allows 4,294,967,296 different values. That'sa lot of values, butit's still finite. Computers are finite, so the range of any type must also be finite. But in mathematicsthere are infinitely many integers. Consequently, computers are manifestly prone to error when theirnumeric values become too large. That kind of error is
called numeric overflow.

Example 10:

```
#include<iostream.h>
int main()
 // prints n until it overflows:
int n=1000;
cout << "n = " <<n<<endl;
n *= 1000; // multiplies n by 1000
cout << "n = " <<n<<endl;
n *= 1000; // multiplies n by 1000
cout << "n = " <<n<<endl;
n *= 1000; // multiplies n by 1000
cout << "n = " <<n<<endl;
return 0;
}
```

n = 1000 n = 1000000 n = 1000000000 n = -727379968 ... This shows that the computer that ran this program cannot multiply 1,000,000,000 by 1000 correctly. Integer overflow "wrapsaround" to negative integers. Floating-point Overflow “sinks' into the abstract notion of infinity.

Laboratory Problems:

1. Write a program for a calculator that will take two decimal numbers from a user. It’ll perform the addition, subtraction and multiplication; and then the calculation will be shown on the output screen.

2.Write a program that asks the user to type the width and the length of a rectangle and then outputs to the screen the area and the perimeter of that rectangle.

3.Write a program that converts centimeters to inches. [Hints: 1 cm = .393 inch.]

4.Write a program that convert any temperature from Celsius scale to Fahrenheit scale. [Hints: relation between Celsius and Fahrenheit isC=5/9×(F-32); where F is Fahrenheit and C is Celsius.]

5.Write a program for Floating-point Overflow.

6.Write a program that implements the *quadratic formula* to solve quadratic equations.
[HINTS] #include <cmath> // defines the sqrt() function
cout « "The equation is: " << a << "*x*x + " << b « "*x + "<< c << " = 0" << endl;

7.You will create a program that computes the regular payments on a loan, such as a car loan. Given the principal, the length of time, number of payments per year, and the interest rate, the program will compute the payment.

LAB NO : 03

LAB NAME : **Conditional statements.**

Objective:

The purpose of this laboratory is to know the conditional statements i.e. if, if-else, switch, conditional operator, compound conditions.

If statement:

The if statement allows the programmer to change the logical order of a program; i.e., it makes theorder inwhich the statements are executed differ from the order in which they are listed in the program.

The if-then statement uses a Boolean expression to determine whether to execute a statement or to skip it. Hereis the syntax template: *if (Expression) Statement*
The expression in parentheses can be of any simple data type. Almost without exception, this will be a logical(Boolean) expression; if not, its value is implicitly coerced to type bool (nonzero value means true, zero valuemeans false). Now, let's look at the following statement:

```
if (number < 0)
number = 0;
sum = sum + number ;
```

The expression (number < 0) is evaluated. If the result is true, the statement number = 0; is executed. If the
result is false, the statement is skipped. In either case,the next statement to be executed is

```
sum = sum + number ;
```

Example 01:

// this program finds the minimum of three integers.

```
int main()
{ int n1, n2, n3;
  cout << "Enter three integers: ";
  cin >> n1 >> n2 >> n3;
  int min=n1;              // now min <= n1
  if (n2 < min) min = n2;  // now min <= n1 and min <= n2
  if (n3 < min) min = n3;  // now min <= n1, min <= n2, and min <= n3
  cout << "Their minimum is " << min << endl;
}
```

if-else Statement:

if-else statement uses a Boolean expression to determine which one of the two statements to execute.
Here is the syntax template:

```
if (Expression)
Statement1A
else
Statement1B
```

The expression in parentheses will be evaluated with the result of true or false.

Here is an example

Example 02:

```
#include<iostream>
#include<cstdlib>
using namespace std;
void main()
{
 int magic;//magic number
 int guess;//user's guess
 magic=rand();//get a random number
 cout<<"Enter your guess: ";
 cin>>guess;
 if(guess==magic) cout<<"**Right Guess**";
//If the guess matches the right number then the messege is displayed.
}
```

This program uses the ‘if’ statement to determine whether the user’s guess matches the magic number. If it does, the message is printed on the screen. Taking the Magic Number program further, the next version uses the else to print a message when the wrong number is picked:

Example 03:

```
#include<iostream>
#include<cstdlib>
using namespace std;
void main()
{
 int magic;//magic number
 int guess;//user's guess
 magic=rand();//get a random number
 cout<<"Enter your guess: ";
 cin>>guess;
 if(guess==magic) cout<<"**Right Guess**";
//If the guess matches the right number then the messege is
displayed.
 else cout<<"Sorry You are wrong";
}
```

Nested if Statement:

An if-then statement uses Boolean expression to determine whether to execute or skip a statement. An if-thenelse statement uses a Boolean expression to determine which one of the two statements to execute. The statements to be executed or skipped could be simple statements or compound statements (blocks). They also can be an if Statement. An if statement within another if statement is called ***a nested if statement***.The following example is a nested if statement.

Example 04:

```
#include<iostream>
 #include<cstdlib>
using namespace std;
void main()
{
int magic;//magic number
 int guess;//user's guess
 magic=rand();//get a random number
cout<<"Enter your guess: ";
cin>>guess;
if(guess==magic)
 { cout<<"**Right Guess**\n";
  cout<<magic<< is the number\n";
   }
 else
 {cout<<"Sorry You are wrong";
 if( guess>magic)
  cout<<"Your guess is too high";
 else cout<<"Your guess is too low.\n”;
 }
}
```

Nested if…else statement:

Example 05:

// to find the minimum of three integers.

```
#include<iostream>
using namespace std;
 int main()
 { int n1, n2, n3;
   cout << "Enter three integers: ";
   cin >> n1 >> n2 >> n3;
   if (n1 < n2)
     if (n1 < n3) cout << "Their minimum is " << n1 << endl;
     else cout << "Their minimum is " << n3 << endl;
   else  // n1 >= n2
     if (n2 < n3) cout << "Their minimum is " << n2 << endl;
  else cout << "Their minimum is " << n3 << endl;
}
```

The switch statement:

The switch statement is a selection statement that can be used instead of a series of if-then- else statements. Alternative statements are listed with a switch label in front of each. A switch label is either a case label or the word default. A case label is the word case followed by a constant expression. An integral expression is called a switch expression and is used to match one of the values on the case labels. The statement associated with the value that is matched is the statement that is executed. Execution then continues sequentially from the matched label until the end of the switch statement is encountered or a break statement is encountered. Here is the syntax template for the Switch statement,

integral or `enum expression` is an expression of integral type —char, short, int, long, `bool` or of `enum` type which will be discussed later. In a case label, `constant expression`is an integral or `enum` expression whose operands must be literal or named constants.
Now let's look at the following C++ code:

Example 06:

```
#include<iostream>
using namespace std;
void main()
{
cout<<"\t****MATH OPERATION****"<<endl;
cout<<"\t. . . . . . . . . . . . . . . . . . . . "<<endl;
cout<<"\t[1] Addition "<<endl;
cout<<"\t[2] Subtraction"<<endl;
cout<<"\t[3] Multiplication"<<endl;
cout<<"\t[4]Division"<<endl;
float num1,num2;
cout<<"Enter frist number";
cin>>num1;
cout<<"Enter second number"; cin>>num2;
int choice;
cout<<"Enter your choice[1/2/3/4]:";
cin>>choice;
float result;
switch(choice)
  {
      case 1: result=num1+num2;cout<<"Sum="<<result;break;
      case 2: result=num1-num2;cout<<"Sub="<<result;break;
      case 3: result=num1*num2;cout<<"Mul="<<result;break;
      case 4: result=num1/num2;cout<<"Div="<<result; break;
      default : cout<<"Invalid choice"<<endl;
      }
}
```

THE CONDITIONAL EXPRESSION OPERATOR

C++ provides a special operator that often can be used in place of the **if...else** statement.
It is called the *conditional expression operator*. It uses the **?** and the **:** symbols in this syntax:

condition **?** *expression1* **:** *expression2*

It is a *ternary operator*; *i.e.*, it combines three operands to produce a value. That resulting value is either the value of *expression1* or the value of *expression2* , depending upon the Boolean value of the *condition*. For example, the assignment

min = **(x<y ? x : y)** ; // finds the minimum of two integers.

Example 07:

```
#include<iostream.h>
int main()
{int m,n;
cout << "Enter two integers: ";
cin >> m >> n;
cout << ( m<n ? m : n ) << " is the minimum." << endl;
return 0;
}
```

Example 08:

// Finds the minimum of three integers.

```
#include<iostream>
using namespace std;
int main()
{int a,b,c;
cout << "Enter two integers: ";
cin>>a>>b>>c;
int min=(a<b?(a<c?a:c):(b<c?b:c));
cout<< “minimum number=”<<min<<endl;
}
```

Compound conditions:

Conditions such as `n % d` and `x >= y` can be combined to form compound conditions. This is done using the *logical operators* `&&` (and), `||` (or), and `!` (not). They are defined by

`p && q`	evaluates to true if and only if both `p` and `q` evaluate to true
`p || q`	evaluates to false if and only if both `p` and `q` evaluate to false
`!p`	evaluates to true if and only if `p` evaluates to false

For example, `(n % d || x >= y)` will be false if and only if `n % d` is zero and `x` is less than `y`.

The definitions of the three logical operators are usually given by the *truth tables* below.

p	q	p && q
T	T	T
T	F	F
F	T	F
F	F	F

p	q	p || q
T	T	T
T	F	T
F	T	T
F	F	F

p	!p
T	F
F	T

These show, for example, that if `p` is true and `q` is false, then the expression `p && q` will be false and the expression `p || q` will be true.

Example 09:

```
//to find the minimum of three integers.

#include<iostream>
using namespace std;
void main( )
{ int n1, n2, n3;
  cout << "Enter three integers: ";
  cin >> n1 >> n2 >> n3;
  if (n1 <= n2 && n1 <= n3) cout << "Their minimum is " << n1 <<endl;
  if (n2 <= n1 && n2 <= n3) cout << "Their minimum is " << n2 <<endl;
  if (n3 <= n1 && n3 <= n2) cout << "Their minimum is " << n3 <<endl;
}
```

Laboratory Problems:

1. Write a program to find the roots of a quadratic equation including complex solution

2. An AC power supply of V volts is applied to a circuit load with impedance of Z (Ø) with currentI. Wrote a program to Display the real power P, the reactive power R, the apparent power Sand the power factor PFof the load. Test the programwith a voltage of 120 volts and an impedance of 8 ohms at 30degrees{Hints : for cos and sin function use `<cmath>]` .

3.Write and run a program that reads the user's age and then prints "You are a child." if the age < 18, "You are an adult." if 18 <age < 65, and "You are a senior citizen." if age >65.

4.Modify the example of the switch statement.

LAB NO : 04

LAB NAME : Iterations

Objective:

Iteration is the repetition of a statement or block of statements in a program. C++ has three iteration statements: the `while` statement, the `do..while` statement, and the `for` statement. Iteration statements are also called *loops* because of their cyclic nature.

THE while STATEMENT

The syntax for the `while` statement is
while (*condition***)** *statement***;**

where *condition* is an integral expression and *statement* is any executable statement. If the value of the expression is zero (meaning "false") then the *statement* is ignored and program execution immediately jumps to the next statement that follows the **while** statement. If the value of the *expression* is nonzero (meaning "true") then the *statement* is executed repeatedly until the *expression* evaluates to zero. Note that the *condition* must be enclosed by parentheses.

Example :

This program computes the sum of reciprocals $s = 1 + 1/2 + 1/3 + \ldots\ldots\ldots + 1/n$, where n is the smallest integer for which $n \geq s$:

```
#include<iostream.h>
int main()
{ int bound;
cout << "Enter a positive integer: ";
cin >> bound;
double sum=0.0;
int i=0;
while (sum < bound)
sum += 1.0/++i;
cout << "The sum of the first " << i << " reciprocals is " << sum << endl;
return 0;
}
```

The Fibonacci Numbers:

The *Fibonacci numbers* $F0$, $F1$, $F2$, $F3$, ... are defined recursively by the equations
For example, letting $n = 2$ in the third equation yields
$F2 = F2–1 + F2–2 = F1 + F0 = 0 + 1 = 1$ Similarly, with $n = 3$,
$F3 = F3–1 + F3–2 = F2 + F1 = 1 + 1 = 2$ and with $n = 4$,
$F4 = F4–1 + F4–2 = F3 + F2 = 2 + 1 = 3$
The first ten Fibonacci numbers are shown

Example:

```
#include<iostream.h>
int main()
{ long bound;
cout << "Enter a positive integer: ";
cin >> bound;
cout << "Fibonacci numbers < " << bound << ":\n0,1";
long f0=0,f1=1;
while (true)
 { long f2 = f0 + f1;
   if (f2 > bound) break; // terminates the loop immediately
 cout << "," << f2;
  f0 = f1;
  f1 = f2;
  }
}
```

Do yourself:

1. Write a program to calculate sum of numbers until a negative number is keyed.
2. Write a program to test a number if it is prime number or not.(using while loop)
3. write a program to find the sum=1+3+5+…+n

THE do..while STATEMENT

The syntax for the `do..while` statement is

do
  {*statement*
  } **while (***condition***);**

where *condition* is an integral expression and *statement* is any executable statement. It repeatedly executes the *statement* and then evaluates the *condition* until that condition evaluates to `false`. The `do..while` statement works the same as the `while` statement except that its condition is evaluated at the end of the loop instead of at the beginning. This means that any control variables can be defined within the loop instead of before it. It also means that a `do...while` loop will always iterate at least once, regardless of the value of its control condition

Example1

```
#include<iostream>
using namespace std;
void main()
{
 int num,digit;
cout<<"\nEnter your number";
cin>>num;
cout<<"\nThe number in reverse
order is :";
do
 {
  digit=num%10;
  cout<<digit;
  num/=10;
 }while(num!=0);
}
```

Output:

```
Enter your number 4567
The number in reverse order is 7654…
```

Example 2

```
#include<iostream>
using namespace std;
void main()
{
 int num,digit,sum=0;
 int actnum;
cout<<"\nEnter your number";
cin>>num;
actnum=num;
do
 {
  digit=num%10;

  num/=10;
  sum+=digit;
 }while(num!=0);
 cout<<"\nSum of Digits of the
number "<<actnum<<"is"<<sum;
```

Output:

```
Enter your number 4567
Sum of Digits of the number 4567 is 22
```

Do yourself:

1. Find the factorial of a given number
2. Find the sum and average of the given numbers using the `do..while` loop.
3. Find the sum of the even numbers using `do..while` loop

THE for STATEMENT

The syntax for the for statement is

for (*initialization***;** *condition***;** *update***)** *statement***;**

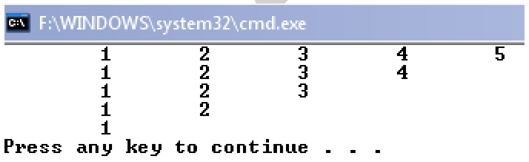
where *initialization*, *condition*, and *update* are optional expressions, and *statement* is any executable statement. The three-part (*initialization***;** *condition***;** *update***)** controls the loop. The *initialization* expression is used to declare and/or initialize control variable(s) for the loop; it is evaluated first, before any iteration occurs. The *condition* expression is used to determine whether the loop should continue iterating; it is evaluated immediately after the initialization; if it is true, the statement is executed. The *update* expression is used to update the control variable(s); it is evaluated after the statement is executed. So the sequence of events that generate the iteration are:

1. evaluate the *initialization* expression;
2. if the value of the *condition* expression is false, terminate the loop;
3. execute t he *statement*;
4. evaluate the *update* expression;
5. repeat steps 2–4.

Example1

```
#include<iostream>
using namespace std;
void main()
{
 int i=0,j=0;
 for(i=5;i>=1;i--)
{
       for(j=1;j<=i;j++)
       cout<<"\t"<<j;
        cout<<'\n';
 }
}
```

OUTPUT:

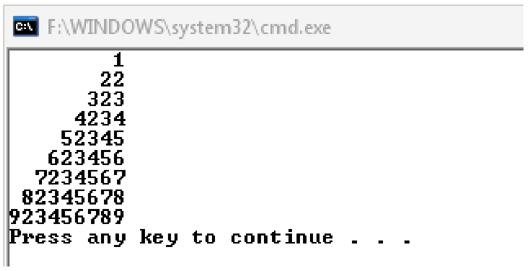


```
        1       2       3       4       5
        1       2       3       4
        1       2       3
        1       2
        1
Press any key to continue . . .
```

Example 2

```
#include<iostream>
using namespace std;
void main()
{
 int i,j,k,l;
 for(i=1;i<=9;i++)
 {
    {
      for(j=1;j<=9-i;j++)
           cout<<" ";
      for( k=i;k>=i;k--)
           cout<<k;
      for(l=2;l<=i;l++)
           cout<<l;
      cout<<endl;
 }
 }
```

OUTPUT



```
        1
       22
      323
     4234
    52345
   623456
  7234567
 82345678
923456789
Press any key to continue . . .
```

Do yourself:

1. Write a program to find the following output

```
1 2 3 4 5                          1              5  4  3  2  1
1 2 3 4                        1   2              5  4  3  2
1 2 3                      1   2   3              5  4  3
1 2                    1   2   3   4              5 4
1                                                 5
```

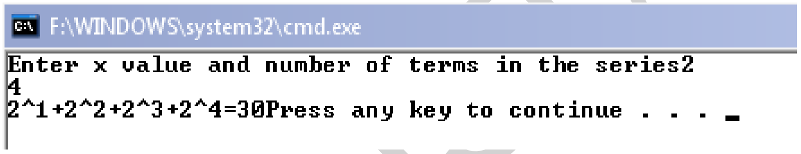
Summation:

Example01

Write a program to sum the series

$SUM=X+X^2+X^3+\ldots..+X$

```
#include<iostream>
using namespace std;
void main()
{int i,n,x,t,sum=0;
cout<<"Enter x value and number of terms in the series";
cin>>x>>n;
t=x;
for(i=1;i<=n;i++)
{
	sum+=t;
	t*=x;
}
for(i=1;i<n;i++)
cout<<x<<"^"<<i<<"+";
cout<<x<<"^"<<n<<"="<<sum;
}
```



```
Enter x value and number of terms in the series2
4
2^1+2^2+2^3+2^4=30Press any key to continue . . .
```

Example02

Write a program to find the sum of the following series:

$1^3-2^3+3^3-4^3+\ldots\ldots\ldots-n^3$

```
#include<iostream>
using namespace std;
void main()
{
int i=0,sign=1,sum=0;int n;
cin>>n;
while(i++<n)
{int j=i*i*i;int temp=sign*j;sum+=temp;sign=(-1)*sign;}cout<<"Sumis:"<<sum;
}
```

Output

3

Sum is 20

Laboratory Problems:

01.Write programs to find the sun of the following series.

$sum=1+2^2+4^2+\ldots..+n^2$

$sum=1-3^2+5^2-\ldots\ldots+n^2$

02. Submit all exercise problems.

LAB NO: 05
LAB NAME: Function

Objective:
The purpose of this laboratory session is to introduce you to the function (in-built & user defined).

FUNCTION:

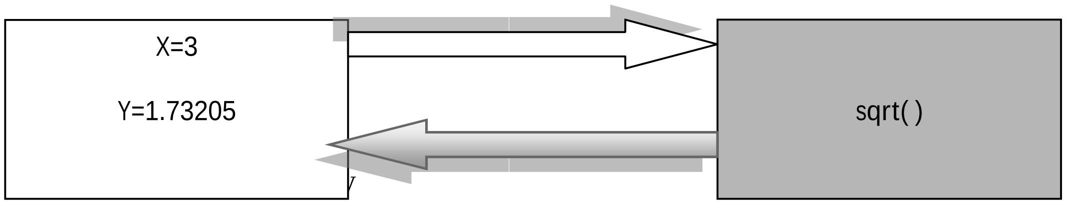
- A C program consists of one or more functions
- All C programs MUST have a main() function
- A function in C performs a particular task e.g. print info on the screen, compute, etc
- Execution of the program begins at the first statement of main ( ) function
- main ( ) function usually invokes (call) other functions to perform its job. Some functions are defined in the same program, others are provided by libraries e.g. sin(angle_ param) function is provided by math.h library.

STANDARD C++ LIBRARY FUNCTIONS:

Here is a simple program that uses the predefined square root function:

```
#include <cmath> // defines the sqrt() function
#include <iostream> // defines the cout object
using namespace std;
int main()
{ // tests the sqrt() function:
for (int x=0; x < 6; x++)
cout << "\t" << x << "\t" << sqrt(x) << endl;
}
```

A function like `sqrt()` is executed by using its name as a variable in a statement, like this:
`y = sqrt(x);`
This is called *invoking* or *calling* the function. Thus in the above Example, the code `sqrt(x)` *calls* the `sqrt()` function. The expression `x` in the parentheses is called the *argument* or *actual parameter* of the function call, and we say that it is *passed by value* to the function. So when `x` is 3, the value 3 is passed to the `sqrt()` function by the call `sqrt(x)`.



USER-DEFINED FUNCTIONS:

More common arrangement is to list only the function's header above the main program, and then list the function's complete definition (head and body) below the main program.
.In this arrangement, the function's declaration is separated from its definition. A function *declaration* is simply the function's head, followed by a semicolon. A function *definition* is the complete function: header and body. A function declaration is also called a function *prototype*.
A function declaration is like a variable declaration; its purpose is simply to provide the compiler with all the information it needs to compile the rest of the file. The compiler does not need to know how the function works (its body). It only needs to know the function's name, the number and types of its parameters, and its return type. This is precisely the information contained in the function's head.

Example 01.

```
#include<iostream.h>
int max(int,int);
// returns larger of the two given integers:
int main()
{ // tests the max() function:
int m,n;
do
{ cin >> m >> n;
cout << "\tmax(" << m << "," << n << ") = " << max(m,n) << endl;
}
while (m != 0);
}
int max(int x,int y)
{ if (x < y) return y;
else return x;
}
```

Example 02

```
#include<iostream.h>
long fact(int);
// returns n! = n*(n-1)*(n-2)*...*(2)(1)
int main()
{ // tests the factorial() function:
for (int i=-1; i < 6; i++)
cout << " " << fact(i);
cout << endl;
}
long fact(int n)
{ // returns n! = n*(n-1)*(n-2)*...*(2)(1)
if (n < 0) return 0;
int f = 1;
while (n > 1)
f *= n--;
return f;
}
```

Exercise:

1. Write and test the following `average()` function that returns the average of four numbers: `float average(float x1,float x2,float x3,float x4)`

2. Write and test the following `min( )` function that returns the smallest of four given integers: `int min(int,int,int,int);`

void FUNCTIONS :

A function need not return a value. In other programming languages, such a function is called a *procedure* or a *subroutine*. In C++, such a function is identified simply by placing the keyword `void` where the function's return type would be.
A type specifies a set of values. For example, the type `short` specifies the set of integers from –32,768 to 32,767. The `void` type specifies the empty set. Consequently, no variable can be declared with `void` type. A `void` function is simply one that returns no value.

Example 04

```
#include<iostream.h>
void printDate(int,int,int);
// // prints the given date in literal form;
int main()
{ // tests the printDate() function:
int month,day,year ;
do
{ cin >> month >> day >> year;
printDate(month,day,year);
}
while (month > 0);
}
void printDate(int m,int d, int y)
{ // prints the given date in literal form:
if (m < 1 || m > 12 || d < 1 || d > 31 || y < 0)
{ cerr << "Error: parameter out of range.\n";
return;
}
switch (m)
{ case 1: cout << "January "; break;
case 2: cout << "February "; break;
case 3: cout << "March "; break;
case 4: cout << "April "; break;
case 5: cout << "May "; break;
case 6: cout << "June "; break;
case 7: cout << "July "; break;
case 8: cout << "August "; break;
case 9: cout << "September "; break;
case 10: cout << "October "; break;
case 11: cout << "November "; break;
case 12: cout << "December "; break;
}
cout << d << "," << y << endl;
}
```

BOOLEAN FUNCTIONS:

In some situations it is helpful to use a function to evaluate a condition, typically within an `if` statement or a `while` statement. Such functions are called *boolean functions* after the British logician George Boole (1815-1864) who developed boolean algebra.

Example 05.

```
#include <cmath> // defines the sqrt() function
#include <iostream> // defines the cout object
using namespace std;
bool isPrime(int);
// returns true if n is prime,false otherwise;
int main()
{ for (int n=0; n < 80; n++)
if (isPrime(n)) cout << n << " ";
cout << endl;
}
bool isPrime(int n)
{ // returns true if n is prime,false otherwise:
float sqrtn = sqrt(n);
if (n < 2) return false; // 0 and 1 are not primes
if (n < 4) return true; // 2 and 3 are the first primes
if (n%2 == 0) return false; // 2 is the only even prime
for (int d=3; d <= sqrtn; d += 2)
if (n%d == 0) return false; // n has a nontrivial divisor
return true; // n has no nontrivial divisors
}
```

Output is:

```
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79
Press any key to continue...
```

Example 06

```
#include<iostream.h>
bool isLeapYear(int);
// returns true iff y is a leap year;
int main()
{ // tests the isLeapYear() function:
int n;
do
{ cin >> n;
if (isLeapYear(n)) cout << n << " is a leap year.\n";
else cout << n << " is not a leap year.\n";
}
while (n > 1);
}
bool isLeapYear(int y)
{ // returns true iff y is a leap year:
return y % 4 == 0 && y % 100 != 0 || y % 400 == 0;
}
```

Exercise:

3 Write and test the following `isSquare()` function that determines whether the given integer is a square number: `int isSquare(int n)`
The first ten square numbers are 0, 1, 4, 9, 16, 25, 36, 49, 64, and 81.

4 Improve the program using compound condition that is more efficient due to short circuiting.

PASSING BY REFERENCE:

The read-only, pass-by-value method of communication is usually what we usually want for functions. It makes the functions more self-contained, protecting them against accidental side effects. However, there are some situations where a function needs to change the value of the parameter passed to it. That can be done by passing it *by reference*.
To pass a parameter by reference instead of by value, simply append an ampersand, &, to the type specifier in the functions parameter list. This makes the local variable a reference to the argument passed to it. So the argument is *read-write* instead of read-only. Then any change to the local variable inside the function will cause the same change to the argument that was passed to it.

This example shows the difference between passing by value and passing by reference:

Example 06:

```
#include<iostream.h>
void f(int,int&);
// changes reference argument to 99:
int main()
{ // tests the f() function:
int a = 22,b = 44;
cout << "a = " << a << ", b = " << b << endl;
f(a,b);
cout << "a = " << a << ", b = " << b << endl;
f(2*a-3,b);
cout << "a = " << a << ", b = " << b << endl;
}
void f(int x,int& y)
{ // changes reference argument to 99:
x = 88;
y = 99;
}
```

The output is:

```
a = 22, b = 44
a = 22, b = 99
a = 22, b = 99
```

The call `f(a,b)` passes `a` by value to `x` and it passes `b` by reference to `y`. So `x` is a local variable that is assigned `a`'s value of 22, while `y` is an alias for the variable `b` whose value is 33. The function assigns 88 to `x`, but that has no effect on `a`. But when it assigns 99 to `y`, it is really assigning 99 to `b`, because `y` is an alias for `b`. So when the function terminates, `a` still has its original value 22, while `b` has the new value 99. The argument `a` is read-only, while the argument `b` is read-write.

OVERLOADING:

C++ allows you to use the same name for different functions. As long as they have different parameter type lists, the compiler will regard them as different functions. To be distinguished,

the parameter lists must either contain a different number of parameters, or there must be at least
one position in their parameter lists where the types are different.

Example 08:

```
#include<iostream.h>
int max(int,int);
int max(int,int,int) ;
int main()
{ cout << max(99,77) << " " << max(55,66,33);
}
int max(int x,int y)
{ // returns the maximum of the two given integers:
return (x > y ? x : y);
}
int max(int x,int y, int z)
{ // returns the maximum of the three given integers:
int m = (x > y ? x : y); // m = max(x,y)
return (z > m ? z : m);
}
```

Using the return Statement to Terminate a Program:

Example 09:

```
int main()
{ // prints the quotient of two input integers:
int n,d;
cout << "Enter two integers: ";
cin >> n >> d;
if (d == 0) return 0;
cout << n << "/" << d << " = " << n/d << endl;
}
```

Using the `exit()` **Function to Terminate a Program:**

Example 10:

```
#include <cstdlib> // defines the exit() function
#include <iostream> // defines the cin and cout objects
using namespace std;
double reciprocal(double x);
int main()
{ double x;
cin >> x;
cout << reciprocal(x);
}
double reciprocal(double x)
{ // returns the reciprocal of x:
if (x == 0) exit(1); // terminate the program
return 1.0/x;
}
```

DEFAULT ARGUMENTS :

This function evaluates the third degree polynomial $\mathbf{a_0+a_1x+a_2x^2+a_3x^3}$. The actual evaluation is

done using Horner's Algorithm, grouping the calculations as $\boldsymbol{a_0 + (a_1 + (a_2 + a_3x)x)x}$ for greater efficiency:

Example 11.

```
#include<iostream.h>
double p(double, double, double=0,double=0, double=0);
int main()
{ // tests the p() function:
double x = 2.0003;
cout << "p(x,7) = " << p(x,7) << endl;
cout << "p(x,7,6) = " << p(x,7,6) << endl;
cout << "p(x,7,6,5) = " << p(x,7,6,5) << endl;
cout << "p(x,7,6,5,4) = " << p(x,7,6,5,4) << endl;
}
double p(double x,double a0,double a1,double a2,double a3)
{ // returns a0 + a1*x + a2*x^2 + a3*x^3:
return a0 + (a1 + (a2 + a3*x)*x)*x;
}
```

Exercise:

1. Write and test the following `computeSphere()` function that returns the volume `v` and the surface area `s` of a sphere with given radius `r`:
   `void computeSphere(float& v,float& s,float r).`

2. Write and test a function that uses the greatest common divisor function

3. Write a program to find a root (for polynomial of a given maximum degree 4, i.e. a function of the form $\mathbf{c_4\,x^4 + c_3\,x^3 + c_2\,x^2 + c_1\,x + c_0}$ for arbitrary values of the coefficients c_4, c_3, c_2, c_1, and c_0.
   The formula for evaluating the polynomial at a given value of x is
   $\mathbf{val = (((c_4 * x + c_3)*x + c_2)*x + c_1)*x + c_0}$
   The program is to prompt for and read the values of the coefficients.

4. Write and test the following `power()` function that returns `x` raised to the power `n`, where `n` can be any integer:
   `double power(double x,int p);`
   Use the algorithm that would compute x_{20} by multiplying 1 by x 20 times.

5. Write a program to find the following series:

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + ... + \frac{x^n}{n!} \qquad \text{sum} = x + \frac{x^2}{2!} + \frac{x^4}{4!} + \frac{x^6}{6!} + ... + \frac{x^n}{2n!}$$

LAB NO: 06

LAB NAME : Array

Arrays:

An array allows you to store and work with multiple values of the same data type. An *array* is a sequence of objects all of which have the same type. The objects are called the *elements* of the array and are numbered consecutively 0, 1, 2, 3, ... These numbers are called *index values* or *subscripts* of the array. The term "subscript" is used because as a mathematical sequence, an array would be written with subscripts: a_0, a_1, a_2, …. The subscripts locate the element's position within the array, thereby giving *direct access* into the array.
If the name of the array is a, then a[0] is the name of the element that is in position 0, a[1] is the name of the element that is in position 1, *etc*. In general, the *i*th element is in position *i*–1.
So if the array has *n* elements, their names are a [0], a[1], a[2], …, a[n-1].
We usually visualize an array as a series of adjacent storage compartments that are numbered by their index values. For example, the diagram here shows an array named a with 5 elements: a[0] contains 11.11, a[1] contains 33.33, a[2] contains 55.55, a[3] contains 77.77, and a[4] contains 99.99. The diagram actually represents a region of the computer's memory because an array is always stored this way with its elements in a contiguous sequence.

a[0]	11.11
a[1]	33.33
a[2]	55.55
a[3]	77.77
a[4]	99.99

The method of numbering the *i*th element with index *i*–1 is called *zero-based indexing*. It guarantees that the index of each array element is equal to the number of "steps" from the initial element a[0] to that element. For example, element a[3] is 3 steps from element a[0].
Virtually all useful programs use arrays. If several objects of the same type are to be used in the same way, it is usually simpler to encapsulate them into an array.

INITIALIZING AN ARRAY:

a[0]	55.5
a[1]	66.6
a[2]	77.7
a[3]	0.0
a[4]	0.0

In C++, an array can be initialized with an optional *initializer list*, like this:

```
float a[] = {22.2,44. 4,66.6 };
```

The values in the list are assigned to the elements of the array in the order that

they are listed. The size of the array is set to be equal to the number of values in the initializer list. So this single line of code declares `a` to be an array of 3 floats and then initializes those for elements with the four values given in the list.

Example 01:

```
#include<iostream.h>
int main()
{ float a[] = { 22.2,44.4, 66.6 };
int size = sizeof(a)/sizeof(float);
for (int i=0; i<size; i++)
cout  <<  "\ta["  <<  i  <<  "]  =  "  <<  a[i]  <<  endl;
}
```

The output is :

```
a[0] = 22.2
a[1] = 44.4
a[2] = 66.6
```

PASSING AN ARRAY TO A FUNCTION:

Example 02:

```
// Passing an Array to a Function that Returns its
Sum// #include<iostream.h>
int sum(int[],int);
int main()
{ int a[] = { 11,33, 55,77 }; int
size = sizeof(a)/sizeof(int);
cout  <<  "sum(a,size)  =  "  <<  sum(a,size)  <<  endl;
}
int sum(int a[],int
n) { int sum=0;
for (int i=0; i<n;
i++) sum += a[i];
return  sum;
}
```

Example 03 :

```
//  that  inputs  are  stored  in  reverse
#include<iostream.h>
int main()
{ const int SIZE=5;
double a[SIZE];
cout << "Enter " << SIZE << "
numbers:\n";  for (int i=SIZE-1; i>=0; i--)
{ cout << "\ta[" << i << "]:
"; cin >> a[i];
}
cout << "In reverse order,they
are:\n";  for (int i=0; i<SIZE; i++)
cout  <<  "\ta["  <<  i  <<  "]  =  "  <<  a[i]  <<  endl;
```

This program uses a `read()` function to input values into the array `a` interactively. Then it uses a `print()` function to print the array:

Example 04:

```
#include<iostream.h>
void read(int[],int&);
void print(int[],int);
int main()
{ const int MAXSIZE=100;
int  a[MAXSIZE]={0},size;
read(a,size);
cout << "The array has " << size << " elements:
"; print(a,size);
}
void  read(int  a[],in  t&  n)
{  cout  <<  "Enter  integers.  Terminate  with
0:\n"; n = 0;
do
{  cout  <<  "a["  <<  n  <<  "]:  ";
cin  >>  a[n];
} while (a[n++] != 0 && n <
MAXSIZE); --n; // don't count the 0
}
void print(int a[],i nt
n) { for (int i=0; i<n;
i++) cout << a[i] << " ";
}
```

Exercise:

1. Write a program that will calculate the average of the data that has been given by the user. Use array to do this.

2. Write and test the following function that returns the minimum value among the first *n* elements:

```
float  min(  float  a[  ],  int  n)
```

MULTIDIMENSIONAL ARRAYS:

The arrays we have used previously have all been *one-dimensional*. This means that they are *linear*; *i.e.*, sequential. But the element type of an array can be almost any type, including an array type. An array of arrays is called a *multidimensional array*. A one-dimensional array of one-dimensional arrays is called a two-dimensional array; a one-dimensional array of two-dimensional arrays is called a three-dimensional array; *etc.* The simplest way to declare a multidimensional array is like this: double a [32][10][4];
This is a three-dimensional array with dimensions 32, 10, and 4. The statement
a [25][8][3] = 99.99 would assign the value 99.99 to the element identified by the multi-index (25,8,3).

Example 05 :

```
#include<iostream.h>
//This program shows how a two-dimensional array can be processed//
void read(int a[][5]);
void print(cont int
a[][5]); int main()
{ int a[3][5];
read(a);
print(a);
}
void  read(int  a[][5])
{   cout   <<   "Enter   15   integers,5   per
row:\n"; for (int i=0; i<3; i++)
{  cout  <<  "Row  "  <<  i  <<  ":  ";
for (int j=0; j<5;
j++) cin >> a[i][j];
}
}
void  print(const  int  a[][5])
{ for (int i=0; i<3;
i++) { for (int j=0;
j<5; j++) cout << " " <<
a[i][j]; cout << endl;
}
}
```

The output is :

```
Enter 15 integers,5 per
row: Row 0: 44 77 33 11 44
Row  1:   60 50 30 90 70
Row  2:   85 25 45 45 55
44  77  33  11  44
60  50  30  90  70
85  25  45  45  55
```

Example 06:

```
#include<iostream.h>
int numZeros(int a[][4][3],int n1,int n2,int
n3); int main()
{ int a[2][4][3] = { { {5, 0, 2}, {0, 0, 9}, {4, 1, 0}, {7, 7, 7}
}, { {3,0,0}, {8,5,0}, {0,0,0}, {2,0,9} }
};
cout  <<  "This  array  has  "  <<  numZeros(a,2,4,3)  <<  "  zeros:\n";
}
int numZeros(int a[][4][3],int n1,int n2,int
n3) { int count = 0;
for (int i = 0; i < n1; i++)
for (int j = 0; j < n2; j++)
for (int k = 0; k < n3; k++)
if (a[i][j][k] == 0)
++count; return count;
}
```

Notice how the array is initialized: it is a 2-element array of 4-element arrays of 3 elements each. That makes a total of 24 elements. It could have been initialized like this:

int
a[2][4][3]={5,0,2,0,0,9,4,1,0,7,7,7,3,0,0,8,5,0,0,0,0,2,0,9};
or like this:
int a[2][4][3]={{5,0,2,0,0,9,4,1,0,7,7,7},{3,0,0,8,5,0,0,0,0,2,0,9}};
But these are more difficult to read and understand than the three-dimensional initializer list. Also notice the three nested **for** loops. In general, processing a *d*-dimensional array is done with *d* **for** loops, one for each dimension.

Exercise:

3. Write a program that will multiply two matrixes of any columns and row if it passes the test for the conditions of matrix multiplication.

4. Write and test the following function:

```
double  stdev(double  x[],int  n);
```

The function returns the *standard deviation* of a data set of *n* numbers $x_0,\ldots,x_{n-1}$ defined by the formula

$$sum = \sqrt{\frac{\sum_{i=0}^{n-1}\left(x_i - \bar{x}\right)^2}{n-1}}$$

where $\mathbf{x_i}$ is the mean of the data. This formula says: square each deviation `(x[i] - mean)` sum those squares; divide that square root by `n-1`; take the square root of that sum.

LAB NO: 07

LAB NAME: Pointer

POINTER:

A pointer is an object that contains a memory address. Very often this address is the location of another object, such as a variable. For example, if x contains the address of y, then x is said to “point to” y. Pointer variables must be declared as such. The general form of a pointer variable declaration is

`Type* pointername;`

Here, pointer variables have the derived type “pointer to T”,where T is the type of the object to which the pointer points. The derived type is denoted by T*. pointername is the name of the pointer variable. For example, to declare ip to be a pointer to an int, use this declaration:

`int* ip;`

Since the derived type of ip is int, it can be used to point to int values. Here, a float pointer is declared:

`float* fp;`

In this case, the derived type of fp is float, which means that it can be used to point to a float value.

Example 01:

```
#include <iostream>
using namespace std;
int main()
{
int total;
int* ptr;
total = 3200; // assign 3,200 to total
ptr = &total; // get address of total
cout<<”ptr=”<<ptr<<endl;
cout <<”&ptr=”<<&ptr<<endl;
return 0;
}
```

Reference operator (&):

The address that locates a variable within memory is what we call a reference to that variable. This reference to a variable can be obtained by preceding the identifier of a variable with an ampersand sign (&), known as reference operator, and which can be literally translated as "address of". For example:

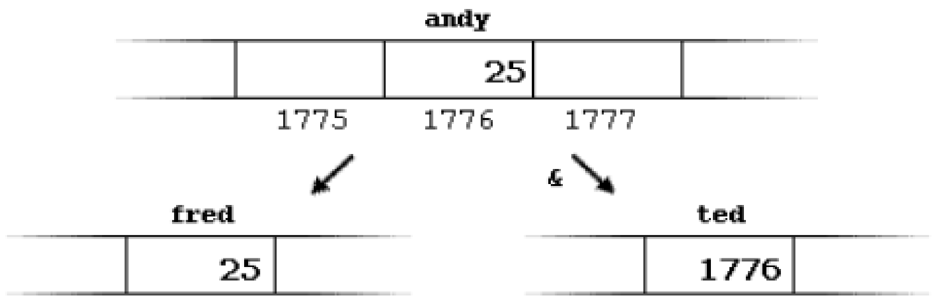
ted = &andy;

This would assign to ted the address of variable andy, since when preceding the name of the variable andy with the reference operator (&) we are no longer talking about the content of the variable itself, but about its reference (i.e., its address in memory).

Consider the following code fragment:

andy = 25;
fred = andy;
ted = &andy;

The values contained in each variable after the execution of this, are shown in the following diagram:



Dereference operator (*):

We have just seen that a variable which stores a reference to another variable is called a pointer. Pointers are said to "point to" the variable whose reference they store.

Notice the difference between the reference and dereference operators:

1. & is the reference operator and can be read as "address of"
2. * is the dereference operator and can be read as "value pointed by"

Thus, they have complementary (or opposite) meanings. A variable referenced with & can be dereferenced with *

Example 02:

```
// my first pointer
#include<iostream>
using namespace std;
int main ()
{
int firstvalue;
cin>>firstvalue;
int* mypointer;
mypointer = &firstvalue;
cout << "firstvalue is " << firstvalue <<endl;
cout << " mypointer " << mypointer <<endl;
cout << " *mypointer" << *mypointer <<endl;
return 0;
}
```

Example 03:

```
// more pointers
#include <iostream>
using namespace std;
int main ()
{
int firstvalue = 5, secondvalue = 15;
int* p1, int* p2;
p1 = &firstvalue; // p1 = address of firstvalue
p2 = &secondvalue; // p2 = address of secondvalue
*p1 = 10; // value pointed by p1 = 10
*p2 = *p1; // value pointed by p2 = value pointed by p1
p1 = p2; // p1 = p2 (value of pointer is copied)
*p1 = 20; // value pointed by p1 = 20
cout << "firstvalue is " << firstvalue << endl;
cout << "secondvalue is " << secondvalue << endl;
return 0;
}
```

Pointers and arrays:

The concept of array is very much bound to the one of pointer. In fact, the identifier of an array is equivalent to the address of its first element, as a pointer is equivalent to the address of the first
element that it points to, so in fact they are the same concept. For example, supposing these two declarations

Example 04:

```
// more pointers
#include <iostream>
using namespace std;
int main ()
{
int numbers[5];
int* p;
p = numbers; *p = 10;
p++; *p = 20;
p=&numbers[2];*p=30;
p=numbers+3;*p=40;
p=numbers;*(p+4)=50;
for (int n=0; n<5; n++)
cout << numbers[n] << ", ";
return 0;
}
```

Pointer arithmetic's:

To conduct arithmetical operations on pointers is a little different than to conduct them on regular integer data types. To begin with, only addition and subtraction operations are allowed to be conducted with them, the others make no sense in the world of pointers. But both addition and subtraction have a different behavior with pointers according to the size of the data type to which they point.
When we saw the different fundamental data types, we saw that some occupy more or

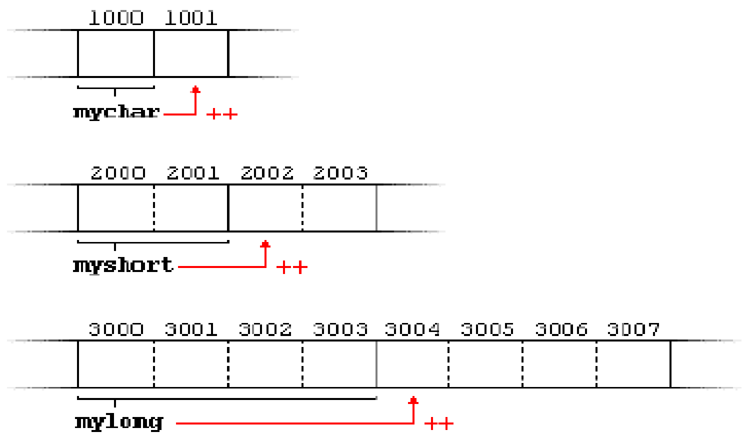
less space than others in the memory. For example, let's assume that in a given compiler for a specific machine, char takes 1 byte, short takes 2 bytes and long takes 4. Suppose that we define three pointers in this compiler:

```
Char* mychar;
Short* myshort;
```

and that we know that they point to memory locations 1000, 2000 and 3000 respectively.So if
we write:

```
mychar++;
myshort++;
mylong++;
```

mychar, as you may expect, would contain the value 1001. But not so obviously, myshort would contain the value 2002, and mylong would contain 3004, even though they have each been increased only once. The reason is that when adding one to a pointer we are making it to point to the following element of the same type with which it has been defined, and therefore the size in bytes of the type pointed is added to the pointer



Pointers to functions:

C++ allows operations with pointers to functions. The typical use of this is for passing a function as an argument to another function, since these cannot be passed dereferenced. In order to declare a pointer to a function we have to declare it like the prototype of the function except that the name of the function is enclosed between parentheses () and an asterisk (*) is inserted before
the name:

Example 05:

```
// pointer to functions
#include <iostream>
using namespace std;
int addition (int a, int b)
{return (a+b); }
int subtraction (int a, int b)
{return (a-b); }
int operation (int x, int y, int (*functocall)(int,int))
{ int g;
g = (*functocall)(x,y);
return (g);
}

int main()
{ int m,n;
int (*minus)(int,int) = subtraction;
m = operation (7, 5, addition);
n = operation (20, m, minus);
cout <<n;
return 0; }
```

In the example, minus is apointer to a function that has two parameters of type int. It is immediately assigned to point to the function subtraction, all in a single line:

```
int (* minus)(int,int) = subtraction;
```

Exercise:

1.Write a programto demonstrate a pointer to a function is declared to perform simple arithmetic operations such as addition, subtraction, multiplication and division of two numbers.

LAB NO: 08
LAB NAME: strings

Cstrings:

A Cstring is a sequence of contiguous characters in memory terminated by NUL character '\0'.C-strings are accessed by variables of type char* (pointer to char). For an example, if s has type char*,then

cout<<s<<endl**;**

will print the characters stored in the memory beginning at the address s and ending with the first occurrence of NUL charater.

In C++, a C-string is an array of characters with the following important features:

i) An extra component is appended to the end of the array, and its value is set to the NUL character '\0'. This means the total number to characters is the array is always one more than the string length.

ii) The C-string may be initialized with a string literal like this:
char s[]=" Abcd"
Note that this string has 5 elements : 'A' , ' b' , ' c' , ' d' and '\0' .

iii) The entire C-string may be output as a single object, like this:
cout<<s;
The system will copy characters from s to cout until the NUL character '\0' is encountered.

iv) The entire C-string may be input as a single object, like this:
cin>>s;
The system will copy characters from cin into s until a white space character is encountered. The user must ensure that s is defined to be a character string long enough to hold the input.

v) The C header file <cstring> provides a wealth of special functions such as strlen( ), strcpy( ), strncpy( ), strcat( ), strcmp( ), strncmp( ) etc. for manipulating C-strings.

Example 01:

```
#include<iostream>
#include<cstring>
using namespace std;
void main( )
{char a[ ]= "ABCdef " ;
cout<<a<<endl;
cin>>a;
cout<<<a<endl;
char b[50];
cin.getline(b,50);
cout<<b<<endl;
}
```

Example 02:

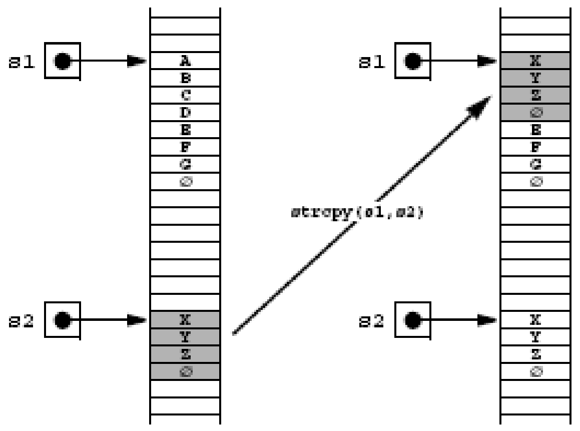
The strlen( ) function returns the number of characters in the string that precede the first occurrence of the NUL character **'\0'.**

```
#include<iostream>
#include<cstring>
using namespace std;
int main()
{ char s[] = "ABCDEFG";
  cout << "strlen(" << s << ") = " << strlen(s) << endl;
  cout << "strlen(\"\") = " << strlen("") << endl;
  char buffer[80];
  cout << "Enter string: ";  cin >> buffer;
  cout << "strlen(" << buffer << ") = " << strlen(buffer) << endl;
}
```

Example 03:

If s1 and s2 are two strings then the function strcpy(s1,s2) copies s2 into s1.

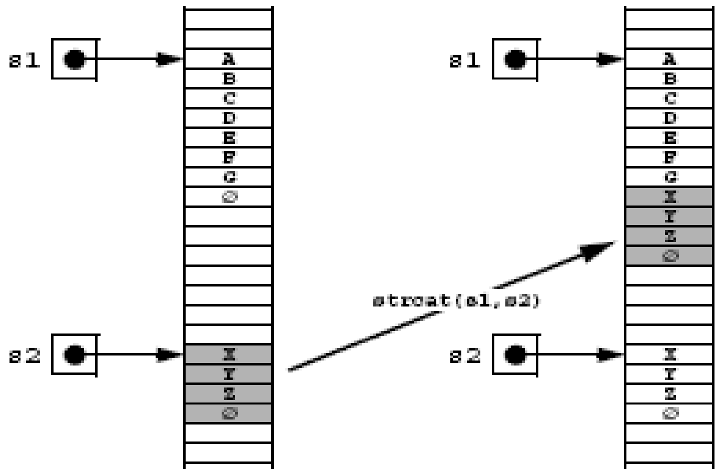
```
#include<iostream>
#include<cstring>
using namespace std;

int main()
{ char s1[] = "ABCDEFG";
  char s2[] = "XYZ";
  cout << "Before strcpy(s1,s2):\n";
  cout << "\ts1 = [" << s1 << "], length = " << strlen(s1) << endl;
  cout << "\ts2 = [" << s2 << "], length = " << strlen(s2) << endl;
  strcpy(s1,s2);
  cout << "After strcpy(s1,s2):\n";
  cout << "\ts1 = [" << s1 << "], length = " << strlen(s1) << endl;
  cout << "\ts2 = [" << s2 << "], length = " << strlen(s2) << endl;
  cout<< " s1=" << s1<<endl;
 }
```



Example 04:

If s1 and s2 are two strings then the function strcat(s1,s2) appends s2 onto the end of s1.

```
#include<iostream>
#include<cstring>
using namespace std;
 int main()
 { char s1[] = "ABCDEFG";
   char s2[] = "XYZ";
   cout << "Before strcat(s1,s2):\n";
   cout << "\ts1 = [" << s1 << "], length = " << strlen(s1) << endl;
   cout << "\ts2 = [" << s2 << "], length = " << strlen(s2) << endl;
   strcat(s1,s2);
   cout << "After strcat(s1,s2):\n";
   cout << "\ts1 = [" << s1 << "], length = " << strlen(s1) << endl;
   cout << "\ts2 = [" << s2 << "], length = " << strlen(s2) << endl;
    cout<< " s1=" << s1<<endl;
    }
```

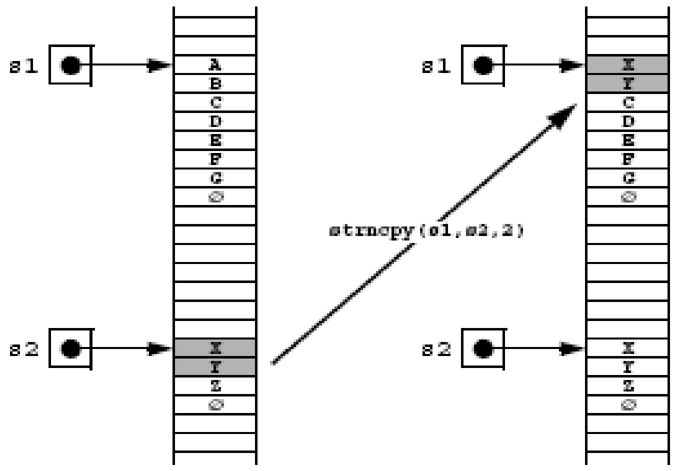


Example 05:

If s1 and s2 are two strings then the call strncpy(s1,s2,n) replaces the first n characters of s1 by the first n characters of s2 leaving the rest of s1 unchanged.

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

```
int main()
{ char s1[] = "ABCDEFG";
  char s2[] = "XYZ";
  cout << "Before strncpy(s1,s2,2):\n";
  cout << "\ts1 = [" << s1 << "], length = " << strlen(s1) << endl;
  cout << "\ts2 = [" << s2 << "], length = " << strlen(s2) << endl;
  strncpy(s1,s2,2);
  cout << "After strncpy(s1,s2,2):\n";
  cout << "\ts1 = [" << s1 << "], length = " << strlen(s1) << endl;
  cout << "\ts2 = [" << s2 << "], length = " << strlen(s2) << endl;
  cout<< " s1=" << s1<<endl;
}
```



Example 06:

```
// this function copies s2 into s1 without using predefined function strcpy( ).
#include<iostream>
#include<cstring>
using namespace std;
char* stringcopy(char* s1, char* s2)
{char* p=s1;
 for( ; *s2; p++, s2++)
    *p=*s2;
 *p= '\0' ;
 return s1 ;
 }

void main( )
{char a1[ ]= " ABCDEFX " ;
 char a2[ ]= " XYZ " ;
stringcopy(a1,a2);
cout<<a1<<endl;
}
```

Example 07:

```
// To capitalize the given string.
 void main( )
{char s[ ]= " Today is Monday." ;
for(int i=0; s[i]!= '\0'; i++)
  if(s[i]>= 'a' && s[i]<= 'z')
      s[i]=s[i]-('a'-'A');
cout<< "s="<<s<<endl;
}
```

Example 08:

```
#include<iostream>
#include<string>
using namespace std;
int main( )
{string str1= "Hello" ;
  string str2= " World" ;
string str3; int len;
//copies str1 into str3.
str3=str1;
cout<< "str3="<<str3<<endl;
//concatenates str1 and str2.
str3=str1+str2;
cout<< "str3="<<str3<<endl;
//total length of str3 after concatenation.
len=str3.size( );
cout<< "str3.len="<<len<<endl;
return 0;}
```

Exercise:

1. Write a program that counts the VOWELS of a string.
2. Write a program that reads one line of text and then prints it with all its blanks removed.
3. Write a program that reads one line of text and then prints the line in reverse order.
   For an example:
   Input: Today is Monday.
   Output: .yadnoM si yadoT
4. Catenate two strings without using strcat( ).
5. Catenate two strings such that the function appends the first n characters of 2nd string to s1 string without using strncat( ).

LAB NO : 09
LAB NAME: Classes

Introduction:

A *class* is like an array: it is a derived type whose elements have other types. But unlike an array, the elements of a class may have different types. Furthermore, some elements of a class may be functions, including operators.

Although any region of storage may generally be regarded as an "object", the word is usually used to describe variables whose type is a class. Thus "object-oriented programming" involves programs that use classes. We think of an object as a self-contained entity that stores its own data and owns its own functions. The functionality of an object gives it life in the sense that it "knows" how to do things on its own.

Class Declaration:

Here is a declaration for a class whose objects represent rational numbers (*i.e.*, fractions):

```
class Ratio
{ public:
    void assign(int, int);
    double convert();
    void invert();
    void print();
  private:
    int num, den;
};
```

The declaration begins with the keyword `class` followed by the name of the class and ends with the required semicolon. The name of this class is `Ratio`.

The functions `assign()`, `convert()`, `invert()`, and `print()` are called *member functions* because they are members of the class. Similarly, the variables `num` and `den` are called *member data*. Member functions are also called *methods* and *services*.

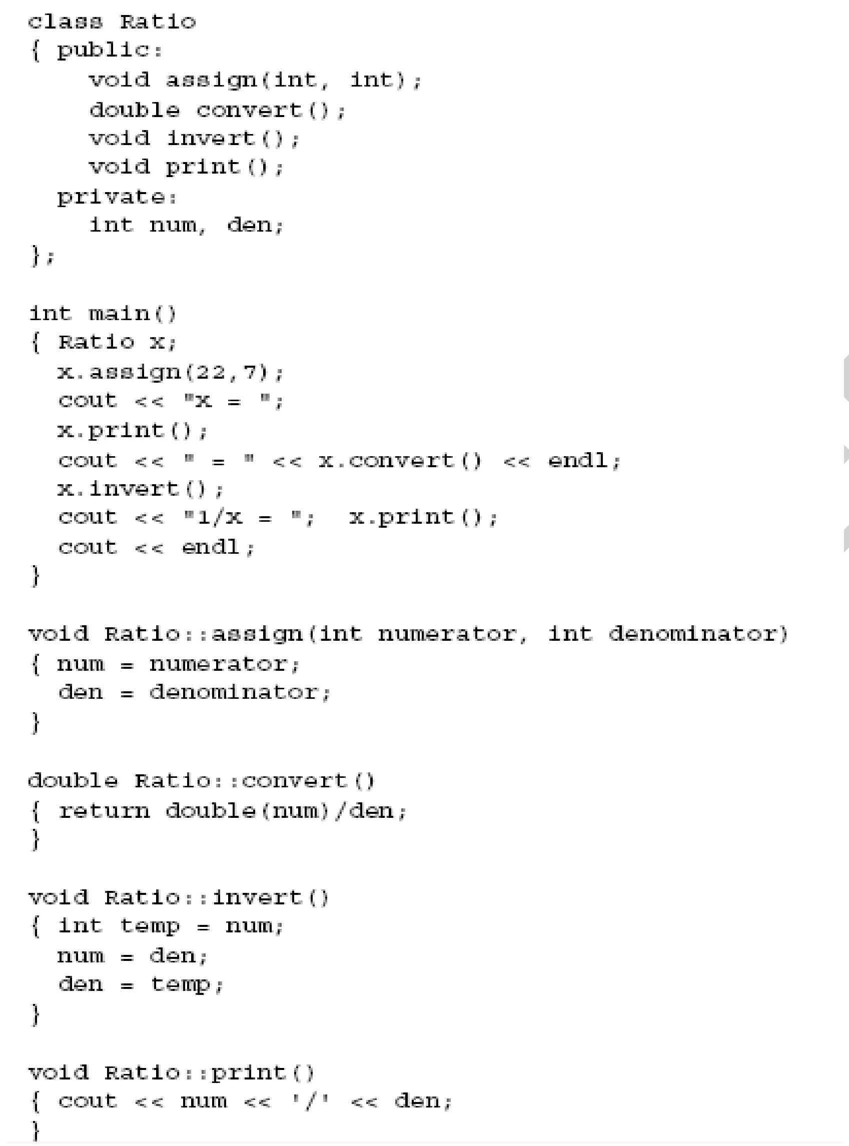
In this class, all the member functions are designated as `public`, and all the member data are designated as `private`. The difference is that `public` members are accessible from outside the class, while `private` members are accessible only from within the class. Preventing access from outside the class is called "information hiding." It allows the programmer to compartmentalize the software which makes it easier to understand, to debug, and to maintain.

The following example shows how this class could be implemented and used.

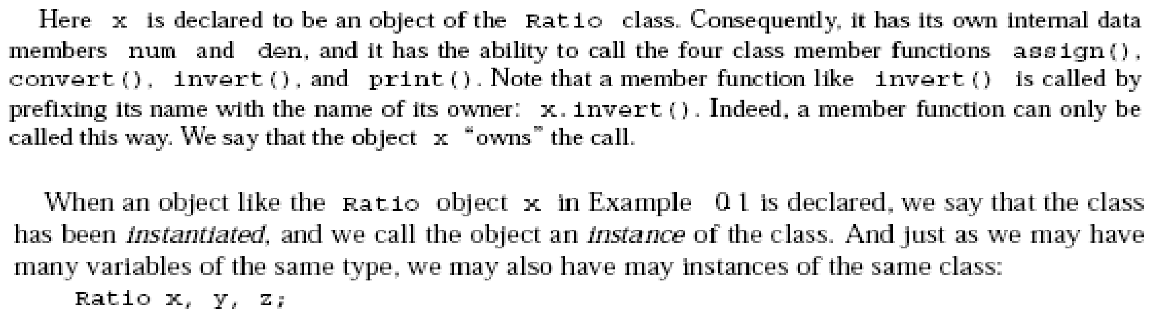
Example: 01

//Implementing the Ratio class.

#include<iostream.h>



```
class Ratio
{ public:
     void assign(int, int);
     double convert();
     void invert();
     void print();
  private:
     int num, den;
};

int main()
{ Ratio x;
  x.assign(22,7);
  cout << "x = ";
  x.print();
  cout << " = " << x.convert() << endl;
  x.invert();
  cout << "1/x = ";  x.print();
  cout << endl;
}

void Ratio::assign(int numerator, int denominator)
{ num = numerator;
  den = denominator;
}

double Ratio::convert()
{ return double(num)/den;
}

void Ratio::invert()
{ int temp = num;
  num = den;
  den = temp;
}

void Ratio::print()
{ cout << num << '/' << den;
}
```



Here `x` is declared to be an object of the `Ratio` class. Consequently, it has its own internal data members `num` and `den`, and it has the ability to call the four class member functions `assign()`, `convert()`, `invert()`, and `print()`. Note that a member function like `invert()` is called by prefixing its name with the name of its owner: `x.invert()`. Indeed, a member function can only be called this way. We say that the object `x` "owns" the call.

When an object like the `Ratio` object `x` in Example 01 is declared, we say that the class has been *instantiated*, and we call the object an *instance* of the class. And just as we may have many variables of the same type, we may also have may instances of the same class:

```
     Ratio x, y, z;
```

Example : 02

//A self contained implementation of the Ratio class.

```
class Ratio
{ public:
    void assign(int n, int d) { num = n; den = d; }
    double convert() { return double(num)/den; }
    void invert() { int temp = num; num = den; den = temp; }
    void print() { cout << num << '/' << den; }
  private:
    int num, den;
};
```

Example: 03

```
#include <iostream>
using namespace std;
class Rectangle
{
private:
        double length;
        double width;
public:
        void setLength(double);
        void setWidth(double);
        void getArea( );
};
void Rectangle::setLength(double len)
{       if (len >= 0)
          length = len;
        else
        {       length = 1.0;
                cout << "Invalid length. Using a default value of 1.\n";
        }
}
void Rectangle::setWidth(double w)
{       if (w >= 0)
          width = w;
        else
        {       width = 1.0;
                cout << "Invalid width. Using a default value of 1.\n";
        }
}
```

```
void  Rectangle::getArea( )
{
        double ar;
        ar=length*  width;
        cout << "\nHere is the rectangle's data:\n";
        cout << "Length: " << length << endl;
        cout << "Width : " << width << endl;
        cout << "Area : " << ar<< endl;
}
int main( )
{
        Rectangle box;
        double boxLength, boxWidth;

cout << "This program will calculate the area of a rectangle.\n";
cout << "Enter the length : ";
cin >> boxLength;
cout << "Enter the width : ";

cin >> boxWidth;
box.setLength(boxLength);
box.setWidth(boxWidth);
box.getArea( );
return 0;
}
```

Exercise:

1. Using Class show Fibonacci numbers in the output.