C++

Introduction

The main pitfall with standard C has been identified as the lack of facilities for data abstraction. With the emergence of more abstract and modular languages like Modula-2 and Ada and Object-oriented languages like Small-Talk, BJARNE STROUSRUP at Bells Lab was motivated to develop C++ by upgrading C with appropriate mechanisms to create object like abstract data structures. The introduction of the class mechanism in C++ mainly provides the base for implementing abstract data types to suit object-oriented programming. The C++ language is thus considered as the superset of the standard C.

IDENTIFIERS AND KEYWORDS

Identifiers can be defined as the name of the variables and some other program elements using the combination of the following characters.

Alphabets : a…z, A…Z
Numerals : 0…9
Underscore : _

Eg.,

NAME, A23C, CODE, EMP_NAME

Special characters :

All characters other than listed as alphabets, numerals and underscore, are special characters.

Keywords

Keywords are also identifiers but cannot be user defined since they are reserved words.

C++

Keywords in C++

Auto break case char const continue default do double else enum extern float for goto if long register return short signed sizeof static struct switch union unsigned void volatile while

Constants

- String constants
- Numeric constants
- Character constants

String constants

A string constant is a sequence of alphanumeric characters enclosed in double quotation marks whose maximum length is 255 Characters.

Eg. “The man”
“A343”

Numeric Constants

These are positive or negative numbers.
Types of Numeric constants are :

Integer

- Integer
- Short Integer(short)
- Long Integer(long)

Float

- Single precision(float)
- Double precision(double)
- Long double

Unsigned

- Unsigned char
- Unsigned integer
- Unsigned short integer
- Unsigned long integer

Hex

- Short hexadecimal
- Long Hexadecimal

Octal

- Short octal
- Long octal

Operators

- Arithmetic operators (+, -, *, /, %)
- Assignment operators (=, +=, -=, *=, /=, %=)
- Comparison and Logical operators (<, >, <=, >=, ==,!=, &&, ||, !)
- Relational operators (<, >, <=, >=)
- Equality operators (==, !=)
- Logical operators(&&, ||, !)
- Unary operators(*, &, -, !, ++, --, type, sizeof)
- Ternary operator (?)
- Scope operator(::)
- New and delete operators

C++

Skeleton of typical C++ program

Program heading
Begin
    Type or variable declaration
    Statements of operation
    Results
End

Sample C++ program

```
#include<iostream.h>
void main()
{
    cout << “God is Great”;
}
```

iostream

The “**iostream**” supports both input/output stream of functions to read a stream of characters from the keyboard and to display a stream of objects onto the video screen.

Input and Output statements

Output Statement

The syntax…

- cout << “Message”
- cout << variable_name
- cout << “Message” << variable_name

The ‘<<’ is called as “insertion” operator.

Eg.

```
Cout << “god is great”
Cout << age
Cout << “name is …” << name
```

Input Statement

The syntax…..

Cin >> var1 >> var2 >> var3…..varn;

The ‘>>’ is called as “extraction” operator.

Eg.

```
Cout << “Enter a number”
Cin << a
```

Sample program to add, subtract, multiply and divide of the given two numbers

```
#include<iostream.h>
void main()
{
    int a,b,sum,sub,mul,div;
    cout << “Enter any two numbers “ << endln;
    cin >> a >> b;
    sum=a+b;
    sub=a-b;
    mul=a*b;
    div=a/b;
    cout << “Addition of two numbers is…” << sum;
    cout << “Mulitiplication of two numbers is…” << mul;
    cout << “Subtraction of two numbers is…” << sub;
    cout << “Division of two numbers is…” << div;
}
```

CONTROL STATEMENTS

Conditional Statements

The conditional expressions are mainly used for decision making. The following statements are used to perform the task of the conditional operations.

- if statement
- if-else statement
- switch-case statement

if Statement

The if statement is used to express conditional expressions. If the given condition is true then it will execute the statements; otherwise it will execute the optional statements.

```
If (expression)
{
Statement;
Statement;
;;
;;
}
```

if-else statement

Syntax

```
if (expression)
        statement;

else
        statement;
```

Syntax

```
if (expression)
{
        block-of-statements;
}
else
{
        block-of-statements;
}
```

Nested if

```
If (expression) {
        If (expression) {
                **********
                **********
                }
        else {
                *********
                **********
                }
else {
        if (expression) {
                *******
                ****
        }
        else {
                *********
                *********
                }
}
```

A program to read any two numbers from the keyboard and to display the largest value of them.

```
#include<iostream.h>
```

```
void main()
{
float x,y;
cout << “Enter any two numbers \n”;
cin >> x >> y;

if (x>y)
        cout << “Biggest number is…” << x << endl;
else
        cout << “Biggest number is…” << y << endl;
}
```

switch statement

The switch statement is a special multiway decision maker that tests whether an expression matches one of the number of constant values, and braces accordingly.

Syntax

```
        switch (expression) {
                case contant_1 :
                        Statements;
                case contant_2 :
                        Statements;

                        ;;
                        ;;
                        ;;
                case contant_n :
                        Statements;
                default :
                        Statement;
        }
```

C++

A program to find out whether the given character is vowel or not

```
#include<iostream.h>
void main(){
char ch;
cout << “Enter the character :”
cin >> ch;
switch(ch){
                case ‘A’ :
                case ‘a’ :
                        cout << “The given character is vowel “;
                        break;
                case ‘E’ :
                case ‘e’ :
                        cout << “The given character is vowel “;
                        break;
                case ‘I’ :
                case ‘i’ :
                        cout << “The given character is vowel “;
                        break;
                case ‘O’ :
                case ‘o’ :
                        cout << “The given character is vowel “;
                        break;
                case ‘U’ :
                case ‘u’ :
                        cout << “The given character is vowel “;
                        break;
                default :
                        cout << “The given character is not vowel”;
                        break;
                }
```

Loop Statements

A set of statements are to be executed continuously until certain condition is satisfied. There are various loop statements available in C++.

- for loop
- while loop
- do-while loop

for loop

This loop consists of three expressions. The first expression is used to initialize the index value, the second to check whether or not the loop is to be continued again and the third to change the index value for further iteration.

Syntax

```
for (initial_condition; test_condition; increment or decrement value)
{
statement_1;
statement_2;
;;
;;
}
```

A Program to find the sum and average of given numbers

```
#include<iostream.h>
void main(){
```

```
int n;
cout << “Enter the no. of terms…”;
cin >> n;
float sum = 0;
float a;
for (int i=0; i<=n-1;++i) {
        cout << “Enter a number  :\n”;
        cin >> a;
        sum = sum+a;
}
float av;
av = sum/n;
cout << “sum= “ << sum << endl’
cout << “Averave = << av << endl;
}
```

Nested For-Loops

```
for(i=0;i<=n; ++i){
        for(j=0; j<=m ; ++j)
                statements;
                statements;
                statements;
        }
```

while loop

This loop is used when we are not certain that the loop will be executed. After checking whether the initial condition is true or false and finding it to be true, only then the while loop will enter into the loop operations.

Syntax

```
While (Condition)
        Statement;
```

For a block of statements,

```
while(condition) {
        Statement_1;
        Statement_2;
        ----
        ----
}
```

Example

```
while ((character = cin.get()) != EOF )
        cout.put (character);
```

do-while loop

Whenever one is certain about a test condition, then the do-while loop can be used, as it enters into the loop at least once and then checks whether the given condition is true or false.

Syntax

```
do{
        Statement_1;
        Statement_2;
        ------
        ------
        } while (expression);
```

Example

```
Sum=0;
do
```

```
{
sum=sum+I;
i++;
}
while (i<n);
```

break statement

The "**break**" statement is used to terminate the control from the loop statements of the "case-switch" structure. This statement is normally used in the switch-case loop and in each case condition, the break statement must be used. If not, the control will be transferred to the subsequent case condition also.

Syntax

```
break;
```

Continue statement

This statement is used to repeat the same operations once again even if it checks the error.

Syntax

```
continue;
```

go to statement

This statement I used to alter the program execution sequence by transferring the control to some other part of the program.

Syntax

```
goto label;
```

There are two types of goto statements, which are **conditional** and **unconditional goto** statement.

Unconditional goto

This goto statement is used to transfer the control from one part of the program to other part without checking any condition.

Example

```
#include<iostream.h>
void main(){
        start:
        cout << “God is Great”;
        goto start;
        }
```

Conditional goto

This will transfer the control of the execution from one part of the program to the other in certain cases.

Example

```
If (a>b)
        goto big1;

big1:
```

Functions

A function definition has a name, a parentheses pair containing zero or more parameters and a body. For each parameter, there should be a corresponding declaration that occurs before the body. Any parameter not declared is taken to be an int by default.

Syntax

```
Function_type functionname(datatype arg1, datatype arg2,….)
{
```

```
        body of function;
        ------
        ------
        return value;
}
```

return keyword

This keyword is used to terminate function and return a value to its caller. This can be also used to exit a function without returning a value;

Syntax

```
return;
return(expression);
```

Types of Functions

The user defined functions may be classified in the following three ways based on the formal arguments passed and the usage of the “return” statement, and based on that, there are three types of user-defined functions.

a) A function is invoked without passing any formal argument from the calling portion of a program and also the function does not return back any value to the called function.
b) A function is invoked with formal arguments from the calling portion of a program but the function does not return back any value to the calling portion.
c) A function is invoked with formal arguments from the calling portion of a program which returns back a value to the calling environment.

Type 1

Example

```
#include<iostream.h>
```

```
void main(){
        void message() // function declaration
        message(); // function calling
}
void message(){
        ----------
        ---------- // body of the function
        ----------
}
```

Type 2

Example

```
#include<iostream.h>
void main(){
        void square(int ) // function declaration
        int a;
        -------
        -------
        square(a); // function calling
}
void square(int x){
        ----------
        ---------- // body of the function
        ----------
}
```

Type 3

Example

```
#include<iostream.h>
void main(){
        int check(int,int,char) // function declaration
        int x,y,temp;
        char ch;
        ----------
        ----------
        temp=check(x,y,ch); // function calling
```

```
}
 int check(int a, int b, char c){
        int value;
        ----------
        ---------- // body of the function
        ----------
        return(value);
}
```

A program to find the square of its number using a function declaration without using the return statement

```
#include<iostream.h>
Void main()
{
void square(int);
int max;
cout << “Enter a value for n ?\n”;
cin >> max;
for (int i=0;i<=max-1;++i)
        square(i);
}
void square(int n)
{
float v;
v=n*n;
cout << “ I = “<< n << “square = “ << value <<endl;
}
```

A program to find the factorial of a given number using function declaration with the return statement

```
#include<iostream.h>
void main(){
long int fact(int);
int x,n;
cout << “Enter the no. to find factorial “ << endl;
cin >> n;
x=fact(n);
cout << “value = “ << n << “and its factorial = “;
cout << x << endl;
}
long int fact(int n)
{
int value=1;
if (n == 1)
        return (value);
```

```
else
        {
                for (int i=1;i<=n;++i)
                        value = value * I;
                        return(value);
                }
}
```

Local and Global variables

Local variables

Identifiers declared as label, const, type variables and functions in a block are said to belong to a particular block or function and these identifiers are known as the local parameters or variables. Local variables are defined inside a function block or a compound statement.

Eg.

```
Void funct(int I, int j)
{
        int a,b; // local variables
        ::
        :;
        // body of the function
}
```

Global variables

Global variables are variables defined outside the main function block.

Eg.

```
        int a,b=10; // global variable declaration
        Void main()
        {
                void func1();
                a=20;
                ::
                :;
                func1();
        }
        func1()
        {
                int sum;
                sum=a+b;
                ::
                :;
        }
```

Storage Class Specifiers

The storage class specifier refers to how widely it is known among a set of functions in a program. In other words, how the memory reference is carried out for a variable.

- Automatic variable
- Register Variable
- Static variable
- External variable

Automatic variable

Internal or local variables are the variables which are declared inside a function.

Eg.

```
#include <iostream.h>
void main()
{
        auto int a,b,c;
        ::
        ::
}
```

Register Variable

These variables are used for fast processing. Whenever some variables are to be read or repeatedly used, they can be assigned as register variables.

Eg.

```
function ( register int n)
        {
        register char temp;
        ::
        ::
        }
```

Static Variable

Static variables are defined within a function and they have the same scope rules of the automatic variables but in the case of static variables, the contents of the variables will be retained throughout the program.

Eg.

```
Static int x,y;
```

External variable

Variables which are declared outside the main are called external variables and these variables will have the same data type throughout the program, both in main and in the functions.

Eg.

```
extern int x,y;
extern float a,b;
```

A program to display the no and its square from 0 to 10 using REGISTER variables.

```
#include<iostream.h>
void main()
{
int funt(registerint x, register int y);
register int x,y,z;
x=0;
y=0;
cout<<"x          y"<<endl;
do
{
z=funt(x,y);
cout<<x<<'\t'<<z<<endl;
++x;
++y;
}
while(x<=10);
}
int funt(register int x, register int y);
{
register int temp;
temp=x*y;
return(temp);
}
```

Program to display 1 to 10 with addition of 100 using the automatic variable and the static variable.

```
#include<iostream.h>
void main()
{
int func(int x);
int fun(int x1);
int k,s,v;
for (k=1;k<=10;k++)
{
v=func(k);
s=fun(k);
cout<<k<<'\t'<<v<<s<<endl;
}
```

```
}
func(int x)
{
int sum=100; // Automatic variable
sum+=x;
return(sum);
}
fun(int x1)
{
static int sum=100; // Static variable
sum+=x1;
return(sum);
}
```

Recursive Function

A function which calls itself directly or indirectly again and again is known as the recursive function.

Program to find the factorial of the given no using the recursive function

```
#include<iostream.h>
void main()
{
long int fact(long int);
int x,n;
cin>>n;
x=fact(n);
cout<<n<<x<<endl;
}

long int fact(long int n);
{
long int fact(long int n);
int v=1;
if(n==1) return(v);
else
{
v=n*fact(n-1);
return(v);
}
}
```

Arrays

An array is a collection of identical data objects which are stored in consecutive memory locations under a common heading or a variable name.

Syntax

Storage_class data_type array_name[expression];

Example

```
int value[10];
char line[50];
static char page[20];
```

Array initialization

```
int values[5] = {44,65,77,77,55};
char sex[] = { 'M', 'F'};
float bpay[] = {423.0,43.5,656.4};
```

A program to initialize a set of numbers in the array and to display it in a standard output device

```
#include <iostream.h>
void main()
{
int a[5]={44,67,77,88,34,55};
int I;
cout << "Contents of the array \n";
for (i=0;i<5;++i)
{
        cout << a[i] << '\t';
        }
}
```

Multidimensional Array

These are defined in the same manner as one dimensional arrays, except that a separate pair of square brackets are required for each subscript.

Syntax

Storage_class data_type arrayname[expr1][expr2].....[exprn];

Example

float coordinate x[4][4];

int value[10][5][5];

POINTERS

A pointer is a variable which holds the memory address of another variable.

Pointer operator

A pointer operator can be represented by a combination of *(asterisk) with a variable.

For eg.

int *ptr;

float *fp;

The general format of pointer declaration is…

data_type *pointer_variable;

Address operator

An address operator can be represented by a combination of & (ambersand) with a pointer variable.

For eg.

K = &ptr;

Pointer Expressions

Pointer assignment : A pointer is a variable data type and hence the general rule to assign its value to the pointer is same as that of any other variable data type.

Eg.

```
int x,y;
int *ptr1, *ptr2;
ptr1 = &x;
```

The memory address of variable x is assigned to the pointer variable ptr1.

```
y = *ptr1;
```

The contents of the pointer variable ptr1 is assigned to the variable y, not the memory address.

```
ptr1 = &x;
ptr2 = ptr1;
```

The address of the ptr1 is assigned to the pointer variable ptr2. The contents of both prt1 and ptr2 will be the same as these two pointer variables hold the same address.

A program to assign a character variable to the pointer and to display the contents of the pointer.

```
#include <iostream.h>
void main()
{
char x,y;
char *pt;
x = 'k'; // assign character
pt = &x;
y = *pt;
cout << "Value of x = " << x << endl;
cout << "Pointer value = " << y << endl;
```

}

Structure

A structure is an aggregate of several data items that can be of different types, and it is defined using the keyword **struct.** The items are called members of the structure and are logically grouped. They convey information about the same object.

Eg.

```
struct student {
        int rollno;
        char name[20];
        };
```

typedef

The typedef is used to define new data items that are equivalent to the existing data types.

Syntax

```
typedef datatype newtype;
```

Eg.

```
typedef int integer;
typedef float real;
Integer a,b;
real x,y;
```

C++

Features of C++

Class

A group of objects that share common properties and relationship. In C++, a class is a new data type that contains member variables and member functions that operates on the variables. A class is defined with the keyword **class**.

Class Object

A variable whose data type is a class.

Structure of C++ Program

A C++ program is similar to an ANSI C program except with some newly added types of declarations. Since CLASS types encapsulate all the functions, a C++ program may not have any subordinate function apart from the main function. A general structure of a program may look like

```
< include standard header files>
<include user-defined header files>
<declaration of class type>
<declaration of objects>
void main()
{
  declaration of local objects and variables
  statements
}
```

C++ class definition

Syntax: <classkey> <classname> [<:baselist>] { <member list> }

Classes are specific to C++.

<classkey> is one of the keywords "class", "struct", or "union".

<classname> can be any name unique within its scope.

<baselist> lists the base class(es) that this class derives from.

<baselist> is optional.

<member list> declares the class's data members and member functions.

Within a class,

the data are called "data members"

the functions are called "member functions"

Example:

```
class students {
  int rollno;
  int calculate(void);
};
```

Access Specifiers

public, private, and protected

Members of a class can acquire access attributes in one of two ways:
by default, or through the use of the access specifiers public, private, and protected.

Syntax:

```
public: <declarations>
private: <declarations>
protected <declarations>
```

Public

If a member is public, it can be used by any function. In C++, members of a struct or union are public by default.

Private

If a member is private, it can only be used by member functions and friends of the class in which it is declared. Members of a class are private by default.

Protected

If a member is protected, its access is the same as for private. In addition, the member can be used by member functions and friends of classes derived from the declared class, but only in objects of the derived type.

Scope Resolution Operator(::)

This operator can be used to refer to any member in the class explicitly.

Sample c++ program

```
#include<iostream.h> // include files
class student   // class declarations
{
       char name[30];
       int age;
  public:
      void getdata(void);
     void display(void);
 }
void student:: getdata(void) // member functions definitions
   {
      cout << "Enter name ";
      cin >> name;
     cout << "enter age"
      cin >> age;
 }
void student::display(void) // member function definitions
```

```
{
  cout << " name → " << name;
  cout << " age → " <<, age;
}
void main()   // main function program
 {
  student s1;  // s1 is an object for the class student
  s1.getdata();
  s1.display();
}
```

Constructor

C++ provides a special member function whose task is to initialize the objects of its class. It is special because its name is the same as the class name. The constructor is invoked whenever an object of its associated class is created. It is called constructor because it construct the values of data members of the class.

Firing of Constructor.

A Constructor fires at the time of creation of objects.

Types of Constructor

There are 4 types of constructors.

1. Default Constructor
2. Argument or Parametric Constructor
3. Copy Constructor
4. Dynamic Constructor

1. **Default Constructor :** Default Constructor fires automatically at the time of creation of the objects.
2. **Argumental Constructor :** It fires only if the argument is passed to the constructor.
3. **Copy Constructor :** If the value of one constructor is given to an another constructor, then it is called as copy constructor.

4. **Dynamic Construtor :** It is used for allocation of memory to a variable at run time.

Example :

```
# include <iostream.h>
class date
 {
   private :
        int dd;
        int mm;
        int yy;
   public :
        date () // ex. for default constructor
        {
        dd=01;
        mm=01;
        yy=2002;
        }
        date (int d,int m, int y) // ex. for argumental constructor
        {
        dd=d;
        mm=m;
        yy=y;
        }
        date(date x)
        {
           dd = x.dd;
           mm = x.mm;
           yy=x.yy;
        }
        void showdate();
        void displaydate();
};
void date ::showdate()
{
```

```
cin >> dd>>mm>>yy;
}
void date ::displaydate()
{
cout << dd<<mm<<yy;
}
void main()
{
date  d1,d2(3,3,2002);
d1.showdate();
d1.displaydate();
d2.showdate();
d2.displaydate();
date (d2);                  // ex for copy constructor
}
```

Parameterized Constructors

Parameterized constructor is used to initialize the various data elements of different objects with different values when they are created. C++ permits us to achieve this objective by passing arguments to constructor function when the objects are created. The constructors that can take arguments are called parameterized constructors.

Destructor

A destructor, as the name implies, is used to destroy the objects that have been created by a constructor. The destructor is a member function whose name is the same as the class name but is preceded by tilde. A destructor never takes any argument not does it return any value. It will be invoked implicitly by the compiler upon exit from the program to clean up the storage that is no longer accessible.

Example for Destructor :

```
class word {
```

```
        private :
        char *str_word;
        public :
        word (char *s)
        {
        str_word=new char(strlen(s));
        strcpy=(str_word,s);
        }
        int getlen()
        {
        return strlen(str_word);
        }
        char *getword()
        {
        return str_word:
        }
      ~word()
       {
        delete str_word;
        }
};
void main()
{
word *word1;
word1->word::~word;
}
```

Inheritance

Inheritance is the process of creating new classes called derived classes, from existing or base classes. The derived class inherits all the capabilities of the base classes. The derived class inherits all the capabilities of the base class but can add

embellishments and refinements of its own. The base class is unchanged by this process.

Inheritance has important advantages, most importantly it permits code reusability. Once a base class is written and debugged, if need not be touched again but can nevertheless be adapted to work in different situations. Reusing existing code saves times and money and increases program's reusability.

1. Inheritance is the concept by which the properties of one entity are available to another.
2. It allocates new class to be built from older and less specialized classes instead of being rewritten from scratch.
3. The class that inherits properties and functions is called the sub class or the derived class and the class from which they are inherited is called super class (or) the base class.
4. The derived class inherits all the properties of the base class and can add properties and refinements of its own. The base class remains unchanged.

```
# include <iostream.h>
class Base
{
public :
        void show()
        {
        cout << "\n Base";
        }};
class Der1 :public Base
{
public :
        void show()
        {
        cout <<"\nDer1";
        }
};
```

```
class Der2:public Base
{
        public :
        void show()
        {
        cout <<"\nDer2";
        }
};
void main()
{
Der1 d1;
Der2 d2;
Base *ptr;
Ptr=&d1;
Ptr->show();
Ptr=&d2;
Ptr->show();
}
```

Function overloading

This is a logical method of calling several functions with different arguments and data types that perform basically identical things by the same name.

A program to demonstrate how function overloading is carried out for swapping of two variables of the various data types.

```
#include <iostream.h>
void swap(int &x, int &y);
void swap(float a,float b);
```

```
void main()
{
int x,y;
float a,b;
cout << “Enter any two integers..” << endl;
cin >> x>>y;
cout << “Enter any two Float numbers..” << endl;
cin >> a>>b;
// swapping integer numbers
swap(x,y)
cout << “After swapping integer numbers..”<<endl;
cout <<x<<y;
// swapping float numbers
swap(a,b)
cout << “After swapping float numbers..”<<endl;
cout <<x<<y;
}
void swap(int &a, int &b)
{
int temp;
temp=a;
a=b;
b=temp;
}
void swap(float &a, float &b)
{
float temp;
temp=a;
a=b;
b=temp;
}
```

Static Data Members

Static member variables are similar to C static variable. A static member variable has special characteristics. It is initialized to zero when the first object of its class is created. No other initialization is permitted. Only one copy of that member is created for the entire class and is shared by all objects of that class, no matter how many objects are created. It is visible only within the class, but its lifetime is the entire program. Static variables are normally used to maintain values common to entire class.

Polymorphism

A property by which we can send the same message to objects of several different classes, and each respond in a different way depending on its class. We can send such message without knowing to which of the classes the objects belongs.

```
# include <iostream.h>
class date
{
    private :
        int dd;
        int mm;
        int yy;
   public :
        void showdate();
        void displaydate();
};
void date ::showdate()
{
cin >> dd>>mm>>yy;
}
void date ::displaydate()
{
cout << dd<<mm<<yy;
}
```

```
void main()
{
date  d;
d.showdate();
d.displaydate();
}
```

Virtual functions

Virtual functions let derived classes provide different versions of a base class function. You can declare a virtual function in a base class, then redefine it in any derived class, even if the number and type of arguments are the same. The redefined function overrides the base class function of the same name. Virtual functions can only be member functions.

You can also declare the functions

```
  int Base::Fun(int)
```

and

```
  int Derived::Fun(int)
```

even when they are not virtual.

The base class version is available to derived class objects via scope override. If they are virtual, only the function associated with the actual type of the object is available. With virtual functions, you can't change just the function type. It is illegal, therefore, to redefine a virtual function so that it differs only in the return type. If two functions with the same name have different arguments, C++ considers them different, and the virtual function mechanism is ignored.

A function qualified by the virtual keyword. When a virtual function is called via a pointer, the class of the objects pointed to determine which function definition will be used. Virtual functions implement polymorphism, whereby objects belonging to different classes can respond to the same message in different ways.

```
# include <iostream.h>
# include <string.h>
class Base
{
public :
virtual void show_message(void)
{
cout <<”Base class message”<<,endl:};
};
virtual void show_reverse(void)=0;
};
class Derived :public Base
{
public :
        virtual void show_message(void)
        {
        cout <<”Derived class message”<<endl;
        };
        virtual void show_reverse(void)
        {
        cout <<strrev(“Derived class message”)<<endl;
}
};
void main(void)
{
Base *poly=new Derived;
Poly->show_message();
Poly->show_reverse();
}
```

Operator Overloading

C++ enables use of system defined (or standard) operators, such as +, -, *, == etc., to act on the user-defined data structures(or objects) in a way relevant to that data

structure(or object). An operator thus may have the same LABEL OR SYMBOL as a structure operator, but associating with different parameters and resulting to different action.

We may use the plus(+) operator on different objects.

str=str1 + str2 // concatenate two strings
arr=arr1 + arr2 // add two arrays
c = c1 + c2 // add two complex numbers
r = r1 + r2 // add two rational numbers
s = s1 + s2 // union of sets s1,s2

C++ allows two variables of user-defined type with the same syntax that is applied to the basic types. C++ has the ability to provide the operators with a special meaning for a data type. The mechanism of giving such special meaning to an operator is known as Operator Overloading.

The following program segment illustrates the overloading of an assignment operator in a class.

```
#include <iostream.h>
class sample {
        private :
                int x;
                float y;
        public :
                sample(int, float);
                void operator = (sample abc);
                void display();
        };
void sample :: operator = (sample abc)
{
        x = abc.x;
```

```
        y = abc.y;
}
void main()
{
        sample obj1;
        sample obj2;
        ;;;;;
        ;;;;;
        obj1 = obj2;
        obj2.display();
}
```

new (operator) and delete (operator)

Operators that create and destroy an object

Syntax:

```
<pointer_to_name> = new <name> [ <name_initializer> ];
 delete <pointer_to_name>;
```

The "new" operator tries to create an object <name> by allocating sizeof(<name>) bytes in the heap. The "delete" operator destroys the object <name> by deallocating sizeof(<name>) bytes (pointed to by <pointer_to_name>).

The storage duration of the new object is from the point of creation until the operator "delete" deallocates its memory, or until the end of the program.

Example:

```
 name *nameptr;  // name is any non-function type
  ...
 if (!(nameptr = new name)) {
   errmsg("Insufficient memory for name");
   exit (1);
 }
```

```
// Use *nameptr to initialize new name object
 ...
delete nameptr; //destroy name; deallocate sizeof(name) bytes
```

friend (keyword)

A friend of a class X is a function or class that, although not a member of that class, has full access rights to the private and protected members of class X.

Syntax:

friend <identifier>;

In all other respects, the friend is a normal function in terms of scope, declarations, and definitions.

Example:

```
class students {
  friend department;
  int rollno;
  int calculate(void);
};
class department {
   char ugcourse[25];
   void performance(students*);
};
```

Example 2 for Friend function

```
# include <iostream.h>
class Sample
{
int a,b;
public :
```

```
friend int sum(sample object);
void set_ab(int i, int j);
};
void sample ::set_ab(int I,int j)
{
        a=i;
        b=j;
}
int sum(sample object)
// because it is friend of a samples
return object a + object b;
}
void main (void)
{
sample integer;
integer.set_ab(3,4);
cout <<sum(integer);
}
```

inline (keyword)

Declares/defines C++ inline functions

Syntax:

```
<datatype> <function>(<parameters>) { <statements>; }
inline <datatype> <class>::<function> (<parameters>) { <statements>; }
```

In C++, you can both declare and define a member function within its class. Such functions are called inline.

The first syntax example declares an inline function by default. The syntax must occur within a class definition.

The second syntax example declares an inline function explicitly. Such

definitions do not have to fall within the class definition.

Inline functions are best reserved for small, frequently used functions.

Example:

```
/* First example: Implicit inline statement */

int num;     // global num
class cat {
public:
  char* func(void) { return num; }
  char* num;
}



/* Second example: Explicit inline statement */
inline char* cat::func(void) { return num; }
```

operator (keyword)

Defines a new action

Syntax:

```
operator <operator symbol>( <parameters> )
{
   <statements>;
}
```

The keyword "operator", followed by an operator symbol, defines a new (overloaded) action of the given operator.

Example:

```
complex operator +(complex c1, complex c2)
{
   return complex(c1.real + c2.real, c1.imag + c2.imag);
```

```
}
```

this (C++ keyword)

Non-static member functions operate on the class type object with which they are called. For example, if x is an object of class X and func is a member function of X, the function call x.func() operates on x. Similarly, if xptr is a pointer to an X object, the function call xptr->func() operates on *xptr. But how does func know which x it is operating on? C++ provides func with a pointer to x called "this". "this" is passed as a hidden argument in all calls to non-static member functions.

The keyword "this" is a local variable available in the body of any nonstatic member function. The keyword "this" does not need to be declared and is rarely referred to explicitly in a function definition. However, it is used implicitly within the function for member references. For example, if x.func(y) is called, where y is a member of X, the keyword "this" is set to &x and y is set to this->y, which is equivalent to x.y.

Virtual Classes

You might want to make a class virtual if it is a base class that has been passed to more than one derived class, as might happen with multiple inheritance.

A base class can't be specified more than once in a derived class:

```
class B { ...};
class D : B, B { ... };  // ILLEGAL
```

However, a base class can be indirectly passed to the derived class more than once:

```
class X : public B { ... }
class Y : public B { ... }
class Z : public X, public Y { ... }  // OK
```

In this case, each object of class Z will have two sub-objects of class B.
If this causes problems, you can add the keyword "virtual" to a base class specifier.

For example,

```
class X : virtual public B { ... }
class Y : virtual public B { ... }
class Z : public X, public Y { ... }
```

B is now a virtual base class, and class Z has only one sub-object of class B.

Constructors for Virtual Base Classes

Constructors for virtual base classes are invoked before any non-virtual base classes. If the hierarchy contains multiple virtual base classes, the virtual base class constructors are invoked in the order in which they were declared.

Any non-virtual bases are then constructed before the derived class constructor is called. If a virtual class is derived from a non-virtual base, that non-virtual base will be first, so that the virtual base class can be properly constructed. For example, this code

```
class X : public Y, virtual public Z
   X one;
```

produces this order:

```
Z();   // virtual base class initialization
Y();   // non-virtual base class
X();   // derived class
```

Function Definition and Prototyping

The compiler must be informed of the return types and parameter types of the functions so that it will be able to check for the correct usage of the calls to those functions during compilation. To satisfy the compiler either of the following conventions may be adapted:

1. A complete function definition

```
#include< iostream.h>
int triple(int); // function prototype
void main()
{
   cout<<"\n The tripled value of f is "<<triple(25);
  }
int triple(int f)
{
    return (f * f * f);
}
```

2. A declaration of the function prototype before the function call.

```
Float square(float); or float square(float x);

Int main(void)
{
        printf("%f\n",square(5.0));
}

/* A simple C++ program to demonstrate the use of function prototype */
```

```
#include<iostream.h>
float square(float);
void print(float);
int main(void)
{
        float x;
        x=square(25.0);
        print(x);
}

/* Actual definition of the function */
        float square(float x)
{
        return(x*x);
}
void print(float x)
{
        cout<<x;
}
```

Some conventions in specifying default initializers

1. Multiple default initializers and their order:

   Multiple parameters may be given default values, but these must be assigned from the rightmost parameter to the left.

2. Default initializers and prototype definitions:

   By convention, default initializes can occur only in the prototype definitions, not in the actual function definitions.

   Eg. int power(int exponent, int base=2);// This prototype definition is correct//

```
int power(int exponent, int base=2); // Not correct//
{ /* function code */ }
```

3. No repetition of default intializer:

A default intializer can occur only once. Assume that myfunc.h contains the following prototype definition:

Int power(int exponent, int base=2);

Then,

```
#include “myfunc.h”// myfunc.h has original definition
int power(int exponent, int base=2) // Illegal, base is given with
                                        a default value//
```

4. Successive default initialization :

Applications can use this convention to customize the default initialization according to their requirement. The following cases show legal and illegal successive declarations.

Case 1:

```
int power(int exponent, base); // Original declaration
int power(int exponent, base=2); // legal(successive)
                                        redeclaration//
 int power(int exponent=0, base); // legal(successive)
                                        redeclaration//
```

Case 2:

```
int power(int exponent, base); // Original declaration
int power(int exponent=0, base); // Illegal, start from
                                        rightmost redeclaration //

 int power(int exponent=0, base=2); // Illegal, exponent already
                                        initialized //
```

Case 3:

```
int power(int exponent, base=2); // Original declaration
int power(int exponent, base); // Legal redeclaration
 int power(int exponent=0, base); // Legal redeclaration
```

TEMPLATE

A Template provides the formats of functions and type placeholder. It eliminates duplication of functions and data's. The following shows the general forms of a function template, where T is a type that the compiler will later replace.

```
Template<class T> T function_name(T param_1, T param_b)
{
// statements
}
for example
template<class T> T compare_values(T a,T b)
{
return(a+b);
}
```

The complier can replace the letter T with either the type float or int.

Function Template

The function template acts as model and it can be replaced by any type of function data types.

Simple Example:

```
#include <iostream.h>
template<class T > T add(T a, T b)
{
```

```
T c;
c=a+b;
return(c);
}
void main()
{
int a,b;
float c,d;
cin>>a>>b;
cout<<add(a,b);
cout<<add(c,d);
}
```

Generic Template:

A generic function defines a general set of operations that the function will apply to various data type. A generic function receives the type of data on which the function will operate as a parameter. Creating generic functions within our program can be useful because many algrorithms are fundamentally the same in their processing yet independent of the data type on which the algorithm operates. The word template is a key and T type is place holder of the data type.

```
template <class T type> return type function name(parameter list)
{
//statement
}
#include <iostream.h>
template<class T, class T1 > T avg(T a, T b)
{
T1 c;
c=(a+b)/2;
return(c);
}
void main()
```

```
{
int a,b;
float c;
cin>>a>>b;
c=avg(a,b);
cout<<c;
}
```

Class Template:

If a class acts a placeholder for nay data type then the class must be defined in the form of template.

```
template <class T> class someclass
{
//statement ;
}
```

Here the template not only specifies a type placeholder , it also specifies a parameter that the program can use within the template. When the program later uses the template, it can pass a parameter value to the template as shown here

```
Someclass<int 1029> this_instance
template <class T> class add
{
T a;
T b;
public:
void  sum();
void getdata();  };

template <class T>void  add<T>::sum()
{
T c;
c=a+b;
cout<<c;
}
```

```
template <class T>void  add<T>::getdata()
{
cin>>a>>b;
}
void main()
{
add a;
a.getdata();
a.sum()
}
```

LAB CYCLE

1. Write a C++ program to generate a pyramid using a set of integer numbers.
2. Write a C++ program to solve a Quadratic equation.
3. Write a C++ program to generate a Fibonacci series of 'N' numbers.
4. Write a function in C++ program to find the sum of the following series.

a) sum=1+3+5+……+n

b) sum =$x+x^2/2!+x^4/4!....+x^n/n!$

5. Write a C++ program to read a line and find out the number of vowels(a,e,I,o,u).

6. Write a C++ program to find the largest and smallest number in the given set of 'N' numbers.

7. Write a program to read any four characters and print out all the possible combinations.

8. Write a program in C++ to read a set of characters using a pointer and to print in the reverse order.

9. Write an object oriented program in C++ to read an integer number and find out the sum of all the digits until it comes to a single digit.

10. Write an oop in C++ that prints whether a given number is prime or not.

11. Write an OOP in C++ to read a number n and print it digit by digit in words using inline member function.

12. Write an OOP in C++ to read two dimensional array; find the sum of the elements row-wise and column-wise separately, and display the sums using 'new' and 'delete' operators.

13. Develop an OOP in C++ to create a data base of the following items of the derived class.

Name of the patient

Sex

Age

Ward number

Bed number

Nature of illness

Date of admission

Design a base class consisting of the data members, viz, name of the patient,sex, and age and another base class consisting of ward number,bed number and nature of the illness. The derived class consists of the data member, date of admission. Develop the program for the following facilities.

- build a master table
- list a table

- insert a new entry
- delete old entry
- search a record that is to be printed

14. Write a function overloading program to read a set of coordinates of a rectrangle.

15. Write a program in C++ to perform the following using operator overloading:

a) Area of a circle b) Area of a rectangle c) Area of triangle

16.Write an OOP in C++ using polymorphic technique that prints either the number and its square or the number and its cube from 0 to 100.

17.Write a program in C++ to perform the following using the function template concepts:

i) to read a set of integers

ii) to read a set of floating point numbers

iii) to read a set of double numbers individually.

Find out the average of the non-negative integers and also calculate the deviation of the numbers.

18.Write a program in C++ to read student's record such as name,sex,roll number, height, and weight from the specified file and to display in a sorted order.(name is the key sorting)

19. Write a program in C++ to merge two files into one file heading.

20. Write an OOP in C++ to add, subtract of any two given complex numbers using friend function.