SATHYABAMA UNIVERSITY

(Established under Section 3, UGC Act 1956)

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

SCSX1014 OBJECT ORIENTED PROGRAMMING

SCSX1014 OBJECT ORIENTED PROGRAMMING

CHARACTERISTICS OF PROCEDURE ORIENTED PROGRAMMING (C)

Emphasis is on doing things (algorithms)
Larger programs are divided into smaller programs known functions.
Most of the functions share global data.
Data move openly around the system from function to functions
Functions transforms data from one form to another
Employs TOP-DOWN approach in program design.

CHARACTERISTICS OF OBJECT ORIENTED PROGRAMMING (C++)

Emphasis is on data rather than procedure.
Programs are divided into what known as OBJECTS
Data structures are designed such that they characterize the objects
Functions that operate on the data of an object are tied together in the data structure.
Data is hidden and cannot be accessed by external functions.
Objects may communicate with each other through functions
New data and functions can be easily added whenever necessary
Follows BOTTOM-UP approach in program design.

DIFFERENCE BETWEEN C & C++:

C is an procedure oriented programming language, whereas C++ is an object oriented programming language .The need for an object oriented environment is because procedure oriented programming language such as C fails to show the desired results for larger programs inters of bug free, complexity & reusable programs. C++ eliminates the pitfalls faced by C language.

There are many differences which makes C++ as an apt alternative to C. There are many concepts in C++ which C language doesn't support. They are data hiding, data encapsulation, inheritance, polymorphism, classes, objects, operator overloading & message passing.

As mentioned earlier using these concepts can eliminate the complexity of C for larger programs.

The main difference between C & C++ is where we use the class (user defined data type), Through class we derive the other concepts. Class is similar to structure in C. In structure only mixed data type can be declared, whereas C++ allows us to declare & define functions in addition to the data's. This makes the main function more simple & grouping all functions into a single class.

Operator overloading concept in C++ makes us to create a new language of or own. Through this we can perform arithmetic operations on variables of class i.e. objects which makes the program easier to understand. In C we can't perform arithmetic operation on structure variable as we do in C++.

In c for larger programs the presence of redundant codes is unavoidable. This can be eliminated in C++ by following inheritance.

In C there may be accidental invading of codes of one program by another program. This can be eliminated in C++ which supports data hiding which helps the programmer to build secure programs that cannot be invaded by code in other parts of the program.

In C, if we want to add additional features to the program without modifying the function is very difficult. This can be eliminated in C++ which supports inheritance where we can add additional features to an existing class without modifying it. This concept provides the idea of reusability of the programs.

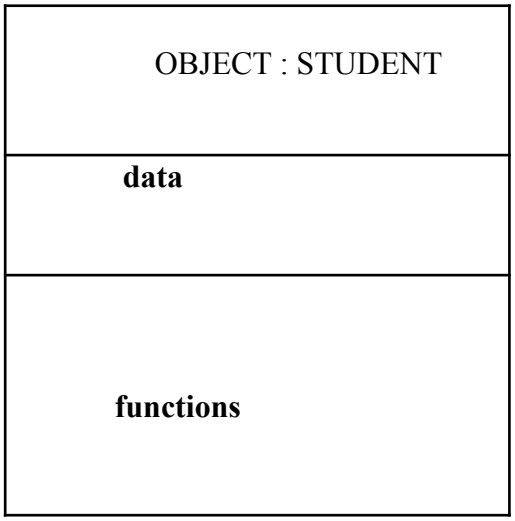
CONCEPTS OF OOP.

General concepts of oops comprises the following

1 Objects
1. Classes
2. Data abstraction
3. Data encapsulation
4. Inheritance
5. Polymorphism
6. Dynamic binding
7. Message passing

1. **object**

Object is an entity that can store data and, send and receive messages. They are runtime entities, they may represent a person, a place a bank account, a table of data or any item that the program must handle. It is an instance of a class.
They may also represent user-defined data such as vectors, time and lists. When a program is executed, **the object interact by sending messages to one another**. Each object contain data and code to manipulate the data objects can interact without having to know details of each other's data or code. It is sufficient to know the type of message accepted and the type of response returned by the objects.



2. **classes**

A class is a collection of objects of similar type. Classes are user defined data types and behave like the built in types of a programming language. For example mango, apple and orange are members of the class fruit. Then the statement FRUIT MANGO; will create an object mango

belonging to the class fruit. The syntax used to create an object is no different than the syntax used to create an integer object in C. if **fruit** has been defined as a class, then the statement

fruit mango;

will create an object **mango** belonging to the class **fruit**.

3. **Data abstraction and encapsulation**

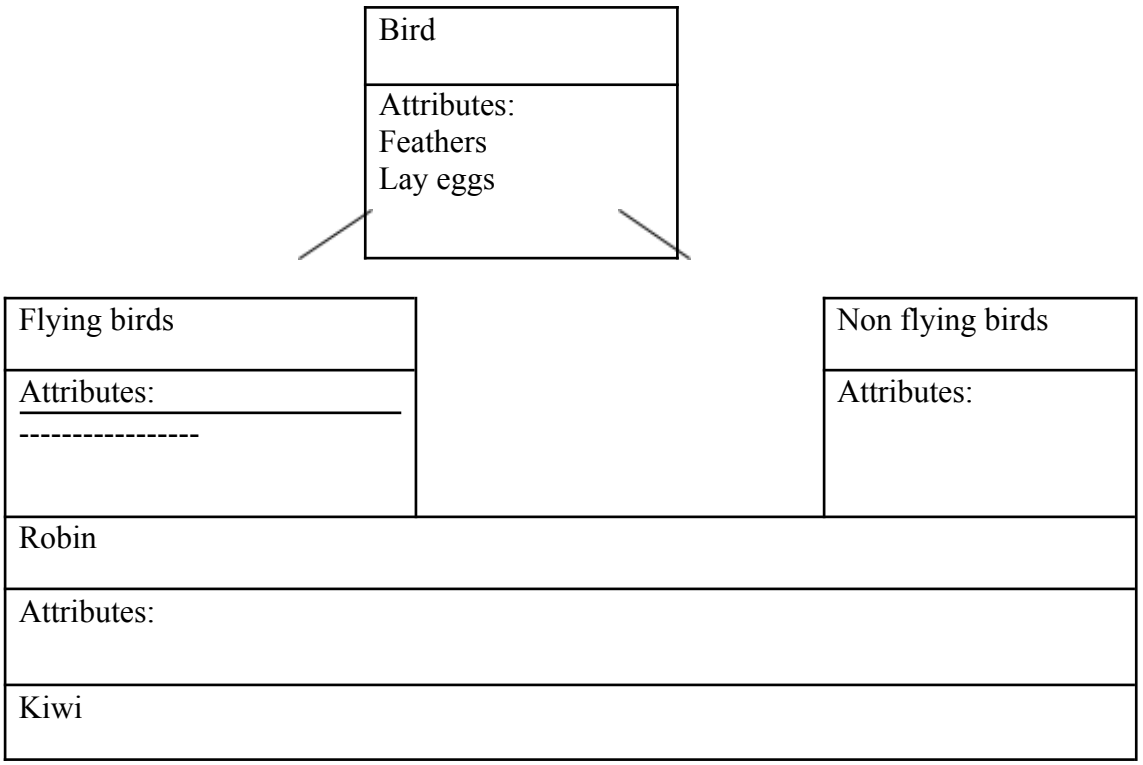
The wrapping up of data and its functions into a single unit (class) is known as encapsulation. The data is not accessible to the outside world and only those functions which are wrapped in the class can assess it> these functions provide the interface between the objects data and the program> this insulation of data from direct access by the program is called **DATA HIDING (or data abstraction)**
Since the classes use the concept of data abstraction they are known as ABSTRACT DATA TYPES

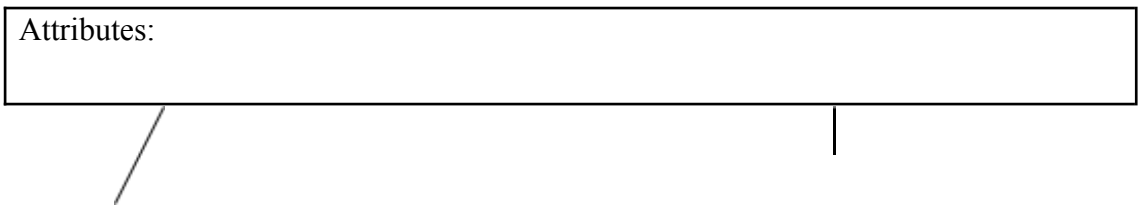
(ADT)

4. **inheritance**

In heritance is the process by which objects of one class acquire the properties of objects of another class. It supports the concept of hierarchical classification. For example the bird **robin** is a part of the class **flying birds** which again a part of **bird**. As given in the diagram below each derived class shares common characteristics with the class from which it is derived.

In **OOP**, the concept of inheritance provides the idea of reusability. This means that we can add additional features to an existing class without modifying it. This is possible by deriving a new class from the existing one. The new class will have the combined features of both the classes. The real appeal and power of the inheritance mechanism is that it allows the programer to reuse a class that is almost, but not exactly, what he wants.

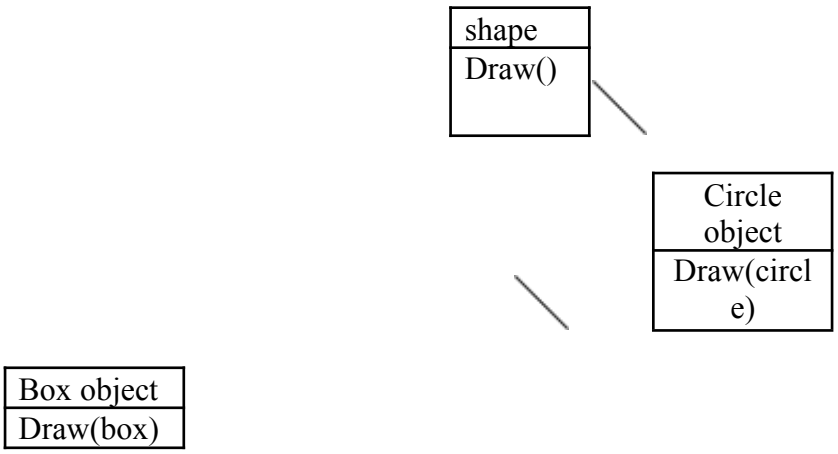




5. **Polymorphism**

Polymorphism means the ability to take more than one form. For example an operation may exhibit different behavior in different instances. The behavior depends upon the types of data used in the operation. For example consider the operation addition. For two numbers, the operation will generate a sum . if the operands are strings, then the operation would produce a third string by concatenation.

Here in the below given diagram a single function draw () does different operation according to the behavior of the type derived. I.e. Draw () function works in different form.



Polymorphism plays an important role in allowing objects having internal structures to share the same external interface. This means that a general class of operations may be accessed in the same manner even though specific actions associated with each operation may differ. Polymorphism is extensivel used in inheritance.

6. Dynamic binding

Dynamic binding means that the code associated with a given procedure call is not known until the time of the call at run-time. It is associated with polymorphism and inheritance. A function call is associated with a polymorphic reference depends on the dynamic type of that reference.

7. **message communication**

An object oriented program consists of a set of objects that communicate with each other. Objects communicate with one another by sending and receiving information much the same way as people pass messages to one another. Objects have a life cycle. They can be created and destroyed. Communication with an object is feasible as long as it is alive.

Benefits/advantages of oops

- Through inheritance we can eliminate redundant code and extend the uese of existing classes.
- We can build programs from the standard working modules that communicate with one another, rather than having to start wrioting the code from scratch. This leads to saving of development time and higher productivity.
- The principle of data hiding helps the programmer to build secure programs that cannot be invalid by code in other parts of the program.
- It is possible to have multiple instances of an object to coedist without any interference.
- It possible to map objects in the problem domain to those objects in the program
- It is easy to partition the work in a project based on objects.
- The data centered design approact enables us top captyre more details of a model in implementable form.
- Software conmplexityc can be easily managed.
- Message passing is allowed in oops thru functions.
- OOP system can be easily upgraded.

Applications of OOP

Some of the applications of oop's are

- Real_time systems
- Simulation and modeling
- Object _ oriented databases
- Hypertext, hypermedia and expertex
- AI and expert systems
- Neural networks and parallel programming
- Decision support and office automation systems
- CIM/CAD/CAM systems

Input statement

The basic input statement in C++ is **cin,** if we are using the input and output statement we must use the header file **iostream.h** . the general format of the input statement is given below.(this **cin** is similar to the **scanf** statement in **C**)

```
Cin>>list of variables
```

E.g.

```
Int. a,b,a
Cin >>a>>b>>c        /* here we are reading three variables*/
```

Output statement

The basic output statement in C++ is **cout,** if we are using the input and output statement we must use the header file **iostream.h** . the general format of the output statement is given below.(this **cout** is similar to the **printf** statement in **C**)

```
Cout>>list of
variables
```

Eg.

```
Int a,b,c
Cin >>a>>b>>c        /* here we are reading three variables, it is like Scanf in c*/
Cout<<a<<b<<c      /* here we are printing three variables, it is like printf in c*/
```

TOKENS

The smallest individual units in a program are known as tokens. C++ has the following tokens

- Keywords.
- Identifiers
- Constants
- Strings
- Operators

Keywords

Keywords are explicitly reserve words and they cannot be used as program variables or other userdefined program elements.

Eg.

Break, new, private, protected, cin ,cout, void, inline, delete, int char, etc.

Identifiers

Identifiers refer to the names of variables, functions, arrays, classes, etc. created by the programmer. They are the fundamental requirement of any language. The rules for naming an identifier are

Only alphabetic characters, digits and underscore are permitted.

The name cannot start with the digit.

Upper case and lowercase4 letters are distinct

A declared keyword cannot be used as a variable name.

Basic data types

The different basic data types are classified as follows

1.user defined data types (eg. Structure, union, class, enumerated)

2.built in type (eg. Int , char , float, double)
3.derived data type (eg. Array, function, pointer.)

Operators

Apart from the operators present in c additionally we have

- Insertion operator <<
- Extraction operator >>
- Scope resolution operator ::
- Pointer to member declarator ::*
- Pointer to member operator ->*
- Pointer to member operator .*
- Memory release operator **delete**
- Line feed operator **endl**
- Memory allocation operator **new**
- Field width operator **setw**

Control structures

Note : refer C notes in the first unit.

Functions`

We know that the functions play an important role in C, dividing a program into many functions is one of the major principles of top-down ,structured programming. Another use of the functions is that it is possible to reduce the size of a program by calling and using them in different places in the program. In fact C++ has added many new features to functions to make hem more reliable and flexible.

Function prototyping

Function prototyping is one of the major improvements added to c++ functions. The prototype describes the function interface to the compiler by giving details such as the number and type of arguments and the type of return values. C ++ makes prototyping essential, where in ANSI C prototyping is an optional one.

Function prototyping is a declaration statement in the calling program and is of the following form:

```
Type function_name(argument list);
```

The argument list contains the types and the names of arguments that must be passed to the function.

Eg: **float volume(int x, float y, float z)**

Note that the each argument variable must be declared independently inside the parenthesis that is the combined declaration is not allowed as in the below statement.

Float volume(int x, float x,y);

In the function declaration the names of the arguments are dummy variables and therefore, they are optional. That is this form.

Float volume(int,float,float)

Is acceptable at the place of declaration. At this place the compiler checks for the type of arguments , when the function is called.

The general synatax for a function is given below.

```
return_type function_name(argument list)
{
body of function
return;
}
```

Call by reference

In C a function call passes arguments by values. The called function creates a new set of variables and copies the values of arguments into them. That type of calling the function by passing the values is called as *call by value*. The provision of the reference variable in C++ permits us to pass parameters to the functions by reference. When we pass arguments by reference, the 'formal' arguments in the called functions become aliases to the 'actual' arguments in the calling function. This means that when the function is working with its own arguments, it isactually working on the original data. Consider the following function"

```
Void swap(int &a, int &b)
{
int t=a;
a=b;
b=t;
}
```

the above written function can be called using the function in the main program in such a way.

```
Swap(m,n)
```

Where the m and n values are interchanged (*call by reference*).
In c++ the *call by reference* can be done using the address varible and the pointer variable. The example is given below. The function can be written for the above example using call by reference as.

```
Void swap(int *a, int *b)
{
int t=*a;
*a = *b;
*b = t;
}
```

here in the above function poi nter variable is used since we are going to call this function using the address variable **&**. The calling function which is written in the main is,

```
void swap(&x,&Y)
```

where we are calling the function using the address rather the using the value , this type of calling the function using the address is called as *call by reference.*

Return by reference

A function can also return a reference. That is return the value by its address. Consider the following the function

```
Int & max(int &x,int &y)
                                        {
if (x>y)
return x;
else
return y;
}
```

since the return type of the function **max()** is **int &** , the function returns reference to x or y. (not the values). Then a function call **max(a,b)** will yield a reference to **a or b** depending on their values. This means the function call can appear on the left hand side of an assignment statement.

Inline function

One of the objective of usin g functions in a program is to save some memory space , which becomes appreciable when a function is likely to be called many times. However, every time a function is called, it takes a lot of extra time in executing a series of instructions for tasks such as jumping to the function, saving registers, pushing arguments into the stack and returning to the calling function. When a function is small a substatial percentage of execution time may be spent in such overheads. C++ has a solution to this problem. To eliminate the cost of calls to small function C++ proposes a new feature called *inline function.* **An inline function is a function that is expanded in a line, this function keeps a request to compiler to give preference to execute it as a request, but it does not makes a commands to give that preference to execute.** An inline function can be defined as below

```
Inline
function_header
{
function body
}
```

Example;

```
Inline int cube(int a)
{
return (a*a*a);
}
```

the above inline function can be invoked by the statement like;

```
c = cube(3);
d = cube(2+2);
```

the output of the above statements will be 27 and 64 for c and d respectively.
Remember that the **inline** keyword mearly sends a request , not a command to the compiler. The compiler may ignore this request if the the function definition is too long or too complicated and compile the function as a normal function.
Some of the situation where the **inline function may not work are:**

- For the function returning values, if a loop, a **switch**, or a **goto** exists.
- For function not returning values, if a return statement exists.
- If a function contains static variables.

- If **inline** function are recrusive.

Default arguments

C++ allows us to call a functio without specifying all its argumens. In such cases, the functions assigns a default value to the parameter which does not have a matching argument in the function call. Default values are specified when the function is declared . the compiler looks at the prototype to see how many arguments a function uses and the alers the program for possible default values. Here in the example of a prototype with default values.

float amount **(**float principal, int period, **float rate = 0.15)**

Here in the above declaration the **rate** is given some default value . the function can be called in the main program in such a way as written below.

A= amount(5000,29); // one argument missing

This function call passes the value 5000 to **principal** and 7 to **period** and then lets the function use default value of 0.15 for **rate** . the function call

Value = amount(5000 , 5 , 0.12); // no missing argument

Passes an explicit value of 0.12 to **rate.**

A default argument is chedcked for type at the time of declaration and evaluated at the time of call .we can have defaults value from right to left. We cannot provide a default value to a particular argument in the middle of the argument list.

Constant argument

In c++, an argument to a function can be declared as **const** as shown below.

```
int strlen(const char *p);
int length(const string &s);
```

the qualifier **const** tells the compiler that the function should not modify the argument. The compiler will generate an error when this condition is violated . this type of declaration is significant only when we pass arguments by reference or pointers.

Function overloading

Overloading refers to the use of the same thing for different purposes. C++ allows overloading of functions. This means that we can use the same function name to create functions that perform a variety of different tasks. This is known as ***function polymorphism*** in OOP.
The function would perform different operations depending on the argument list in the function call. The correct function to be invoked is determined by checking the number and type of the arguments but not on the function type. For example, an overloaded **add()** function handles different types of data as shown below.

```
//declarations

int add (int a, int b);
int add (int a, int b, int c);
double add ( double x,double y);
double add ( double x, int q);

// function calls

cout<<add(5,10);
```

```
cout<<add(15,10.0);
cout<<add(5.10,15);
cout<<add(0.75,5);
```

a function call first matches the prototype having the same number and type of arguments and then calls the appropriate function for execution. A best match must be unique. The function selection involves the following steps:
1. the compiler first tries to find an exact match in which the types of actual arguments are the same and use that function.

8. if an exact match is not found , the compiler uses the integral promotions to the actual arguments, suc as ,
   char to **int**
   float to **double**

to find a match.

9. when either of them fails the compiler tries to use the built_in conversions(the implicit assignment conversions) to the actual arguments and then uses the function whose match is unique. If the conversion is possible to have multiple matches, then the compiler will generate an error message. sUppose we use the following two functions:

   ```
   long square(long n)
   double square(double x)
   ```

a function call such as

```
square(10)
```

will cause an error because **int** argument can be converted to eithe long or double thereby creating an ambigous situations as to which version square() should be used.

10. if all of the above steps fail, then the compiler will try the user_defined conversions in combination with integral promotions and built_in conversions to find a unique match. User defined conversions are often used in handling class functions.

Classes and Objects

The most important feature of C++ is the “class”. A class is an extension of the idea of structure used in C. it is a new way of creating and implementing a user_defined data type.

A class is a way to bind the data and its associated functions together. It allows the data (and functions) to be hidden, if necessary, from external use. A class specification has two parts:

1. class declaration
2. class function definitions

the describes the type and the scope of its members. The class function definitions describe how the class functions are implemented.
The general form of a **class** declaration is:

```
Class class_name
{
  private:
       variable declarations;
        function declarations:
 public:
        variable declarations;
         function declarations:
};
```

The keyword **class** specifies that what follows is an abstract data of type *class_name*. The body of the class is enclosed within braces and terminated by a semicolon. The class body contains the declarations of variables and functions. These variables and functions are collectively known *members.* They are usually grouped under two sections namely *private* and *public* to denote which of them are *members* of *private* and *public.* The keywords private and **public** and **private** are known as visibility labels. Note that these keywords are followed by a colon (:).

The members that have been declared as **private** can be accessed only from within the class. From the other hand the **public** members can be accessed from outside the class also. The *data hiding* is the key future of object oriented programming. The use of the keyword **private** is optional, by default the members of the class are **private** . if both the labels are missing, then by default , all the members are **private**. Such a class is completely hidden from the outside world.

The variables which are declared inside the class are known as *data members* and the functions are known as *member functions.* Only the member functions can have access to the private data members and private functions. The binding of data and functions together into a single class_type variable is referred to as ***encapsulation.***

A simple example for **class**

```
Class item
{
        int number;     //variables declared in the
        float cost;     //private section
public:
        void getdata(); //function declared in public section
        void putdata();//function declared in public section
}
```

here above the **class** is given a name as **item** . the class contains two data members and two function members. The data members are private by default. The function **getdata()** can be used to read values for *number* and *cost* and **putdata()** to print the values. Note that the function is ***declared*** and ***not defined***. Actually the function definition appears after the program. The data members are usually declared as **private** and the member functions as **public.**

<u>Creating objects</u>

The above example class **item** shows what that class will contain, if we want to create an object of that class, we must use the class name in such a way.

```
Item x;
```

Which creates a variable **x** of type **item.** In C++ , the class variables are known as ***objects***. Therefore **x** is called as an object of type **item.** We may also declare more than one object in one statement.
Eg. **Item x,y,z;**

Objects can also be created by placing the objects name immediately after the closing brace, as we do in the case of structures. That is, the definition

```
Class item
{
…….
…….
…….
} x , y , z ;
```

would create the objects **x,y,z** of type **item**.

Accessing class members

As pointed out earlier, the private data of a class can be accessed only through the member functions of that class. The **main()** cannot contain statements that access **number** and **cost** directly. The following is the format for calling a member functions.

```
Object_name . function_name
(actual-arguments);
```

For example the function call statement

```
x.getdata();
```

is an valid statement to read the values of **number** and **cost** of the object **x** by implementing the **getdata()** function. The assignments occur in the actual functions.

The variables declared as **public** can be accessed by the objects directly. Eg.

```
Class xyz
                                        {
        int x;
        int y;
public:
        int z;
        ……..
        ……..
}


main()
{
……..
……..
xyz     p;  //creating an object of class xyz
p.x = 0;  //error, x is private
p.z = 10          //ok z is public
```

```
……..
……..
}
```

in the above example it makes an error when the private variables are assigned some values, it does not create any error if the public variables are assigned directly.

Defining member function

Member functions can be defined in two places:

- Outside the class definition
- Inside the class definition

Where ever the class is defined , it does the same job

Outside the class definition

Member functions that are declared inside the class have to be defined outside the class separately. Their definitions are very much like the normal functions. They should have a function header and a function body. An important difference between a member function and normal function is that a member function incorporates a membership 'identity label' in the header. This 'label' in the header tells which **class** the function belongs to. The general form of member function is :

```
Return_type class_name : : function –name(argument
declaration)
{
function body
}
```

The membership label *class-name* :: tells the compiler that the *function-name* belongs to the class *class-name*. That is the scope of the function is restricted to the *class-name* specified in the header line. The symbol **::** is called *scope resolution operator.*

Consider the example below that we have declared the function inside the class (**getdata()** and **putdata()**), and defining the function outside the class using the *scope resolution operator* **(: :)**

```
#include<iostream.h>
#include<conio.h>
class item
{
        int number;
        float cost;
public:
        void getdata(); //function declared//
        void putdata();   //function declared//
};


void item : : getdata()  //defining the function getdata()
        {
        cout<<"enter the value of number and cost";
        cin>>number>>cost;
        }
```

```
void item : : putdata()  //defining the function putdata()
        {
        cout<<number<<cost;
        }


main()
{
item x;  //creating an object x of class item
x.getdata(); //calls the function getdata()
x.putdata();//calls the function putdata()
}
```

inside the class definition

here in the above program the function is defined outside the class using the *scope resolution operator* .

those functions **getdata()** and **putdata()** can also be written inside the class as shown belos:

```
#include<iostream.h>
#include<conio.h>
class item
{
        int number;
        float cost;
public :
        void getdata()  //declaring and defining the function inside the class
        {
        cout<<"enter the value of number and cost";
        cin>>number>>cost;
        }

        void  putdata()  //declaring and defining the function putdata()  inside the class
        {
        cout<<number<<cost;
        }
};

main()
{
item x;
x.getdata();
x.putdata();
}
```

both the functions does the same job without any changes

Friend functions

we have been emphasizing throughout this chapter that the private members cannot be accessed from outside the class. That is non_member functions cannot have an access to the private data of the class. However there could be a particular situation where we would like two classes to share a particular function. For example consider a case where two classes, **manager** and **scientist,** have been defined. If we would like to operate on the function **incometax()** to operate on the objects of the both the classes. In that situation , C++ allows the common function to be made *friendly* with both the classes, to make an outside function ' friendly' to a class, we have to simply declare this function as a **friend** of the class as shown below:

```
Class ex
{
……..
……
public:
……..
……
friend void xyz( );
//declaration
};
```

The function declaration should be preceeded by the keyword **friend**. The function is declared elsewhere in the program like a normal C++ program. The function definition does not use the keyword **friend** of the scope resolution operator (: :) . the functions that are declared with te keyword **friend** are called as friend functions.
A friend function although not a member function. Has a full access rights to the private members of the class.
A friend function posses certion spl characteristics:

- It is not in the scope of the class to which it has been declared as **friend.**
- Since it is not in the scope of the class, it cannot be called usig the object of that class. It can be invoked like a normal function without the help of any object.
- A dot operator is not needed to execute this funciton
- It can be declared either in the private or public part of the class without affecting its meaning
- Usually it has the objects as arguments.

```
///program involving friend function//

#include<iostream.h>

class ABC
{
        int a;
        public:
        void getval()
           {
           cin>>a;
           }
friend void max(XYZ , ABC)
```

```
};

class XYZ
{
        int x;
        public:
        void getval()
        {
          cin>>x;
        }
friend void max(XYZ , ABC);
};


void max(XYZ   m , ABC   n)
{
        if(m.x>=n.a)
        cout <<m.x;
        else
        cout <<n.a;
}

main()
{
        ABC   ob1;
        ob1. getval();
        XYZ    ob2;
        ob2. getval();
        max(ob1 ,ob2);
}
```

here in the above program a common class **max()** is used by the two classes XYZ and ABC by sharing the private data's.

constructors and destructors

If we want to initialize an ordinary variable, we will be initializing as below,

```
int m = 20;
float x = 5.37;
```

are valid initialization statements for basic data types.

C++ provides a special member function called the *constructor* which enables an object to initialize itself when it is created. This is known as *automatic initialization* of objects. It also provides another member function called the *destructor* that destroys the objects when they are no longer required.

Constructors

A constructor is 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.

1. No argument constructor

Constructor having no argument is called as no argument constructor (default constructor)

A constructor is defined as below:

```
//class with constructor


class integer
{
        int m , n;
public :
        integer (void);          //constructor declared
        ……….
        ……..
};

integer : : integer (void)       //constructor defined
{
m=0;
n =0;
}

main()
{
integer  eg1;
……..
……
……..
}
```

when a class contains a constructor like the one defined above, it is guranteed that an object created by the class will be initialized automatically. For example the declaration

integer eg1;

which is in the main creates an oject **eg1** of type **integer** but also initializes **m** and **n** to zero. *There is no need to write any statement to invoke the constructor function* (as we do with the normal function). **A constructor that accepts no parameters is called *default constructor.***

The constructor functions have some special characters:

They should be declared in the public section.

They are invoked automatically when the objects are created.

They do not have return types.

They cannot be inherited by any class.

Like other C++ functions , they can have default arguments.

Constructors cannot be **virtual**.

We cannot refer their address.

An object with a constructor (or destructor) cannot be used as a member of a union.

They make implicit calls to the operators **new** and **delete** when memory allocation is required.

2. Parameterized constructors

A constructor **integer()** , defined above, initializes the data members of all the objects to zero. However in practice it may be necessary 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 the constructor function when the objects are created. **The constructors that can take arguments are called *parameterized costructors.***

From the above example the constructor **integer()** may be modified to take arguments as shown below:

```
Class integer
{
        int m , n:
public:
        integer (int x , int y)
        ……
        ……
        ……
};

integer : : integer (int x , int y)
{
m = x;
n = y;
}
```

in the above example parameters x and y are passed to the variables m and n. In the main program we can pass the values as arguments in such a way shown below,

```
        main()
        {

        integer  eg1(10,20);
        …….
        ……
        ……..
        }
```

here above an object **eg1** is created of the class **integer** and the values 10 and 20 initialized to **m** and **n** by passing those values as arguments.

3. Multiple constructors

A class having more than one constructor is called as multiple constructor
so far we have used two kinds of constructors. They are

```
integer();        // no argument constructor (null constructor)
integer( int , int)        // argument constructor (parameterized constructor)
```

in the first case, the constructor itself supplies the data values and no values are passed by the calling programming. In the second case, the function call passes the appropriate values from **main()**. C++ permits us to use both these constructors in the same class. For example.

```
Class integer
{
int m, n;
public:
integer()                    // constructor 1
{m = 0; n = 0;}
integer( int a ,int b) //constructor 2
{m = a; n = b;}
integer ( integer & i) //constructor 3
{m = i. m; n = i.n;}
};
```

this declares three constructors for an **integer** object. The first constructor receives no arguments, the second one receives two integer arguments and third receives one **integer** object as an argument . for example, the declaration

integer a1;

would automatically invoke the first constructor and set the both **m** and **n** of **a1** to zero. The statement

integer a2 (20,40)
would call the second constructor and set both **m** and **n** of **a2** to 20 and 40 respectively, finally the statement

integer a3 (a2);

would invoke the third constructor which copies the values of **a2** and **a3** . that is , it sets the value of every data element of **a3** to the value of the corresponding data element of **a2.** as mentioned earlier, such a constructor is called the ***copy constructor.***
Sharing the same name by two or more functions is referred to as *function overloading*.
When more than one constructor is defined inside the class it is called as *constructor overloading.*

4. Constructors with default arguments(default constructors)

It is possible to define constructors with default arguments. For example from the above example **class integer**, a constructor of that class can be declared as follows:

Integer (int x, int y = 0)

The default value of the argument **y = zero** , then the statement

Integer a4 (10,40);

Assigns the value 10 to x and 40 to y. the actual parameter overrides the default value.

5. copy constructor

we have already studied something about the copy constructor, ***it is nothing but it holds the copy of another constructor.*** A copy constructor may be written as

integer (integer & i);

this example has be defined already under multiple constructor. ***As stated earlier a copy constructor is used to declare and initialize an object from another object,*** for example the statement:

integer a5 (a2);

would define the object **a5** and at the same time initialize it to the values of **a2** . another form of this statement is

integer a5 = a2 ;

the process of initializing through a copy constructor is known as *copy initialization*.

6. Dynamic constructor

Allocation of memory to objects at the time of their construction is known as *dynamic construction* ***of objects.*** The memory is allocated with the help of the **new** operator. Program shows the use of the **new** in constructors that are used to construct strings.

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

class string
{
        char    *name;
        int     length;
public :
        string ()                   //constructor 1
        {
        length = 0;
        name =  new char [length + 1];
        }

        string ( char   * s)        // constructor 2
        {
        length = strlen (s);
```

```
name = new char [length +1];
strcpy( name , s);
}

void display()
{
cout << name;
}



main()
{
char   *first  = "joseph" ;
string name1(first) , name ("louis ");
name1 . display();
name2.display();
}
```

Destructors

A destructor is used to destroy the objects that has been created by the constructor. Like constructor the destructor is a member function whose name is same as the class name but is preceeded by a *tilde*. For example the destructor of the class **integer** can be defined as below.

```
~integer ()
{ }
```

a destructor never takes argument nor does it return any value. It will be invoked implicitly by the compiler upon exit from the program to clean up storage that is no longer accessible. It is good practice to declare destructor in a program since it releases memory space for future use.

Example: **program using constructor and destructor**

```
Class integer
{
int m, n;
public:
integer()                    // constructor 1
{m = 0; n = 0;}
integer( int a ,int b) //constructor 2
{m = a; n = b;}
integer ( integer & i) //constructor 3
{m = i. m; n = i.n;}

~integer()       //destructor
{  }
};
```

```
main()
{
integer    a2;
a2.integer(10,20);  // value passing through constructor
~a2;                // destroying the object using destructor
}
```

operator overloading

operator overloading is one of the many exciting features in C++ language. It is an important technique that has the power of *extensibility* of C++. C++ tries to make the user-defined types with the same way as the built-in types. C++ has the ability to provide the operators with a special meaning for a *data type.* The mechanism of giving such meaning is known as *operator overloading.*
We can overload all the operators except the following operators.

Class member access operator (**.** , **.***)

Scope resolution operator(**: :**)

Size operator (**sizeof**)

Conditional operator (**?:**)

Although the *semantics* of an operator can be extended , we cannot change its syntax, the grammatical rules that govern its use such as the number of operands, precedence and associativity. For example the multiplication operator will enjoy higher precedence than the addition operator.

Defining operator overloading
To define an additional task to the operator, we must specify what it means in relation to the class to which the operator is applied. This is done with the special function known as *operator function*, which describes the task. The general type of the operator function is:

```
Return_type   classname : : operator op
(argument list)
{
function body
……
……..
}
```

Where *return type* is the type of value returned by the specified operation and *op* is the operator being overloaded. The *op* is preceded by the keyword **operator**. **Operator** *op* is the function name. The process of overloading involves the following steps.

First create a class that defines the data type that is to be used in the overloading operation.

Declare the operator function **operator** *op* ( ) in the public part of the class. It may either be a member function of **friend** function.

Define the operator function to implement the required operations.

Overloaded operator functions can be invoked by expressions such as

OP X or **X OP** for **unary** operator.
For binary operator **X OP Y .**

Overloading unary operator

let us consider the unary minus operator. A minus operator , when used as a unary, takes just one operand. We know that this operator changes the sign of an operand when applied to a basic data item. We will see how can we overload this operator so that it can be applied to an object in much the same way as is applied to an **int** or **float** variable. The unary minus when applied to an object should change the sign of each data items.

```
////////////////////////////over loading unary minus//////////////////////////////////

#include<iostream.h>
class space
{
        int x,y,z;
public:
        void getdata();
        void display();
        void operator  - ();
};

void space : : getdata()
{
        cin>>x>>y>>z;
}

void space : : display()
{
        cout<<x<<y<<z;
}

void space : : operator – ()
{
        x = -x; y = -y; z = -z;
}

main()
{
space   obj;
obj.getdata();  //calling the function getdata()
obj.display();  //calling the function display()
 - obj;            // calling the operator overloading
obj.display();  //calling the function display()
}
```

the program reads the input from the user, if the user is giving

x = 10, y = 20 and z = -30 as input, before calling the obverloading the function the out put will be same as the input. After calling the overloading function that is **- obj** and now if we display the out put will be x = -10 , y = -20 and
z = 30.

We can also use a friend function to overload unary minus as follows.

Friend void operator - (space &s); // declaration

```
Void operator – (space & s)              //definition
{
s . x = - s. x;
s . y = - s. y;
s. z =  - s. z;
}
```

note that the argument is passed by reference. It will not work if we pass the argument by value because only a copy of the object that activated the call is passed to **–operator** . therefore the changes made inside the operator function will not reflect in the called object.

Overloading binary operators

The above same mechanism can be used to overload a binary operator. The arithmetic expression for + can be used as follows C = A + B;
By overloading the + operator using an **operator + ( )** function. the program illustrates how this is accomplished.

```
//////////////////////////////////////////////program to overload + operator////////////////////////////////////
Class complex
{
        float    real,  imag;
public :
        complex()       // constructor
        {real = 0.0 ; imag = 0.0 ; }
        void get();                 //member function for reading the value
        complex operator + ( );        // operator overloading for + declared
        void display  ()                //member function for display
};

void get()
{
cin >>real>>imag;
}

void display()
{
cout<< real << " + j " <<imag;
}

complex complex : : operator  + (complex  c)         //define the over loading operator +
{
complex   temp;
```

```
temp.real = real + c.real;
temp.imag = imag + c . imag;
return (temp);
}

main()
{
complex c1, c2, c3;     //creating three objects
c1.get();                       //calling the function get() eg. Input  real = 45.0,  imag = 56.90
c2.get();                       //calling the function get() eg. Input  real = 12.0,  imag = 11.90
c3 = c1 + c2 ;          // calling the over loading operator
cout << c1.display();   //displaying the o/p eg. 45.0 + j 56.90
cout<< c2. display();   //displaying the o/p eg. 12.0 + j 11.90

cout<< c3.display();    //displaying the resultant o/p eg. 57.0 + j 68.80
                                i.e calculating the imaginary and real parts of the complex
number
                                input   c1 = 45.0 + j 56.90
                                input   c2 = 12.0 + j 11.90
                                ---------------------
                                output          c3 = 57.0 + j 68.80
                                ----------------------
}
```

the function is expected to add two complex values and return a complex value as the result and receives only one value as arguments. Where does the other value come from ? now let us look at the statement

c3 = c1 + c2; //invokes **operator + ( )** function

we know that the member function can be invoked by an object of the same class. Here the object **c1** takes the responsibility of invoking the function and **c2** playes the role of an argument that is passed to the function. the above **invocation** statement is equivalent to

c3 = c1 . operator + (c2) ; //usual function call syntax

therefore in the **operator +()** function , the data members of **c1** are accessed directly and the data members of **c2**(that is passed as the argument) are accessed using the dot operator. Thus both the objects are available for the function. For example in the statement.

Temp.real = real + c.real;

c.real refers to the object **c2** and **real** refers to the object **c1.temp.real** is the real part of **temp** that has been created specially to hold the results of addition of **c1** and **c2.** The function returns the complex temp to be assigned to **c3.** *As per the rule in the overloading operator of binary operators, the left hand operand is used to invoke the operator functin and the right hand operand is passed as arguments.*

Note: similar way we can write program for other unary and binary operators for overloading/////

```
///program to over load binary operator  minus//////////
Class complex
{
```

```
        float    real,  imag;
public :
        complex()       // constructor
        {real = 0.0 ; imag = 0.0 ; }
        void get();                 //member function for reading the value
        complex operator - ( );         // operator overloading for + declared
        void display  ()                    //member function for display
};

void get()
{
cin >>real>>imag;
}

void display()
{
cout<< real << " + j " <<imag;
}

complex complex : : operator  - (complex  c)          //define the over loading operator +
{
complex   temp;
temp.real = real - c.real;
temp.imag = imag - c . imag;
return (temp);
}

main()
{
complex c1, c2, c3;     //creating three objects
c1.get();                       //calling the function get() eg. Input  real = 45.0,  imag = 56.90
c2.get();                       //calling the function get() eg. Input  real = 12.0,  imag = 11.90
c3 = c1 - c2 ;              // calling the over loading operator
cout << c1.display();   //displaying the o/p eg. 45.0 + j 56.90
cout<< c2. display();   //displaying the o/p eg. 12.0 + j 11.90

cout<< c3.display();    //displaying the resultant o/p eg. 33.0 + j 45.00
                                i.e calculating the imaginary and real parts of the complex
number
                        input   c1 = 45.0 + j 56.90
                        input   c2 = 12.0 + j 11.90
                                ---------------------
                        output          c3 = 33.0 + j 45.00
                                ----------------------
}
```

rules for over loading operators

- only existing operator can be overloaded. New operators cannot be created.
- The overloaded operator must have atleast one operand that is of user defined type.
- We cannot change the basic meaning of an operator . that is we cannot redefine the plus(+)operator to substract one value from the other.

- Overloaded operators follow the syntax rules of the original operators. They cannot be overridden.
- There are some operators that cannot be overloaded.
- We cannot use friend functions to overload certain operators.
- Unary operators, overloaded through the member function take one explicit argument and return no explicit values .
- Binary operators overloaded through a friend function take two explicit arguments.
- When using binary operators overloaded through a member function, the left-hand operand must be an object of relevant class.
- Binary arithmetic operators such as + , -, *, and / must explicitly return a value . they must not attem,pt to change their own arguments.

Inheritance

The mechanism of deriving a new class from an old one is called inheritance . the old class is referred to as the **base class** and the new one is called the **derived class.** the derived class inherits some of the traits (properties of the base class.

Defining derived class

A derived class is defined by specifying its relationship with the base class in addition to its own details. The general for m of defining a derived class is

```
Class derived_classname : visibility_ mode
base class name
{
members of derived class
};
```

The colon (**:**) indicates that the derived class name is derived from the base class name. The visibility mode is optional and , if present , may be either private or public. The default visibility mode is private.

Eg.

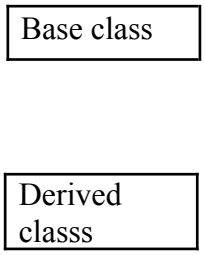
```
Class abc : private xyz
{
members of abc
};
```

the above class is derived from a base class xyz

Single inheritance

The derived class with with one base class is called Single inheritance

```
Class base
{
------------
------------
};

class derived:
public base
{
--------------
--------------
};

main()
{
```



Eg.

```
#include <iostream.h>
class abc
{

public:
int a;
  void getdata()
  {
  cin>>a;
  }
  void showdata()
  {
  cout <<a;
   }
};


clase xyz : public abc
{
   int b;
private:
   void get()
   {
   cin>>b;
   }
   void put()
   {
   cout <<b;
   }
};

main()
```

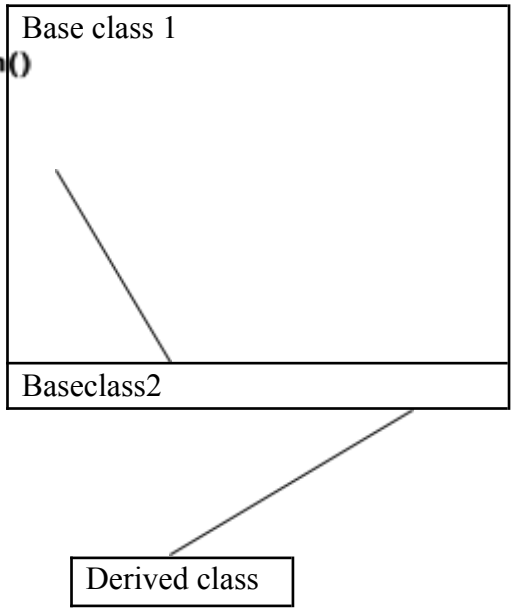
```
{
xyz  w;
w.getdata();
w.putdata();
w.get();
w.put();
}
```

Here in the above example we have defined a function abc as the base class and xyz is the derived class of in which the visibility mode is public so that the public data's can be inherited by the derived as the public data's of it's own class. here in the inheritance it is not necessary to create an object of the both base and derived instead of that we can create an object of the derived class and we can call all the member functions of the both the derived and the base class.

multiple inheritance

The derived class with with more than one base class is called multiple inheritance

```
Class base1
{
.......
.......
};

class base2
{
............
............
};

class derived:public
base1,public base2
{
..........
......
};

main()
{
...
....
}
```



The general syntax for derived class with multiple base class is as follows

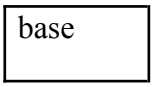
```
Class derived : visibility b1,visibility
b2,……..
{

body of derived
};
```

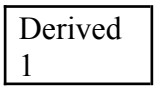
Eg.

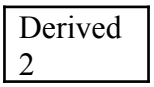
```
#include <iostream.h>
class abc
{

public:
int a;
  void getdata()
  {
  cin>>a;
  }

};


clase def
{
private:
int b;
   void get()
   {
   cin>>b;
   }
};

class  xyz : public abc, public def
{
public:
show()
{
cout <<a<<b;
}
};

main()
{
xyz  w;
w.getdata();
w.get();
w.show();
}
```

multilevel inheritance

The mechanism of deriving a class from derived class is known as multilevel inheritance








```
Class base
{
......
......
};

class derived1:public
base
{
.............
..........
};

class derived2:public
derived1
{
........
..........
};

main()
{
..
......
}
```

//program for multilevel inheritance///

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

class student
{
protected:
```

```
int rollno;
public:
void getno();
void putno(void);
};

void student : : getno( )
{
cin>>rollno;
}

void student : : putno()
{
cout<<rollno;
}


class test : public student                    //first lev el of derivation//
{
protected:
float mark1,mark2;
public:
void getmark();
void putmark();
};

void test : : getmark()
{
cin>>mark1>>mark2;
}

void test : : putmark()
{
cout<<mark1<<mark2;
}

class result : public test
{
float total;
public:
void display()
};

void result : : display()
{
total = mark1 + mark2;
putno();
putmark();
cout<<total;
}
```

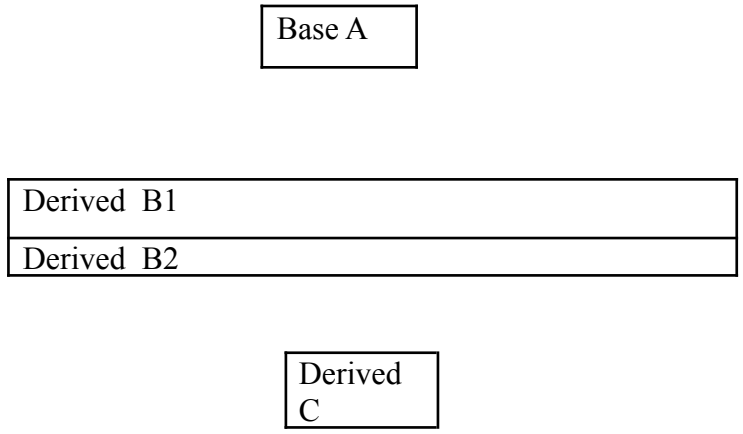
```
main()
{
result student1;
student1.getno()
student1.getmark();
student1.display();
}
```

in the above example we have a base class abc and two derived class def and xyz. Def is derived from the base class abc and xyz is derived from the former derived class i.e. is from def. So that def has all the traits of the base class and the derived class. so that the object of the base class is created i.e. (xyz) and all the member functions and the member data's of the abc and def class are accessed thru the object of xyz.

Deriving the properties from a single base class forming many derived class is called as hierarchical inheritance

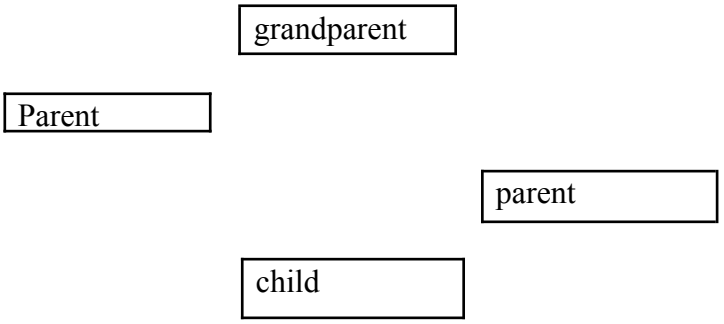
hybrid inheritance

Multiple + hierarchical is called as hybrid inheritance



Virtual base class

When three kind's of inheritance occurs the child will have two direct base class's parent1 and parent2 which they themselves have a common base class grand parent, which grandparent is sometimes referred to as indirect inheritance by the child. The child would have duplicate sets of the members inherited from grandparent and this should be avoided.
The duplication of member's can be avoided by making the common base class as **virtual base class** while declaring the direct or intermediate base class as shown below.



Eg.

```
Class grandparent
{
….
….
};


class  parent1 : virtual public grandparent
{
…….
…..
….
};

class  parent2 : virtual public grandparent
{
…….
…..
….
};

class child : public parent1, public parent2
{
…….
(only one copy of grand parent will be inherited)
…….
};
```

Note: give one example program

Pointers

pointers are variable which hold's the address of a variable and also the value of that variable.

A pointer variable is declared as

Int *ptr;

Pointers to objects

Here in C++ we can use pointer's to access the member's of the class. let us see how can we use these pointer's to create an object of the class.
For example consider the class example as written below;

```
#include<iostream.H>
Class example
{
  int a, b;
  public:
  getdata()
  {
  cin<<a<<b;
  }
  show()]
  {
  cout <<a<<b;
  }
};

main()
{
example *ptr, x;
ptr=&x;
ptr      getdata();
ptr     show();
}
```

here above I have written a class with two member functions one for reading the values of a and b and another function for printing the value of a and b. we can create an object of the class in the main program by using the pointer variable *pt. and using that variable we can call all the member functions of the class.

this pointer

C++ uses a unique keyword called this to represent an object that invokes a member function. **This** is a pointer that points to the object for which **this** functions was called.
This acts as an implicit argument to all the member functions.

Consider an example

```
Class abc
{
int a;
….
…..
};
```

in the above example we assign some values to a as

a=245
this above assignment can also be written as using **this** pointer as
this a=245
here in the above statement **this** function is written explicitly. But in the former it acts explicitly.

Virtual functions

As told earlier polymorphism refers to the property is which objects belonging to different classes are able to respond to the same message, but in different forms. When even when the object of the derived class is created it executes only the member functions of the base class. This drawback is over come using these virtual functions, by naming the base class functions using a keyword virtual followed by the normal declaration.
An example is given below

```
#Include<iostream.h>
Class base
{
Public:
display()
{
cout<<"display base";
}
virtual show()
{
cout<<"show base";
}
};

class derived : public base
{
display();
{
cout<<" display derived";
}
show()
{
cout<<"show derived";
}
};

main()
{
base b, *bptr;
derived d;
bptr=&b;
bptr    display(); //calls base function//
bptr    show();   // calls base function//

bptr=&d;
bptr     display(); // calls base version//
bptr    show() ;   // calls derived version//
}
```

here in the above program the member functions of base and derived class are same it show() and display()
in the main program object is created for both the base class (b and *bptr) and for derived class(d). first the address of the base object is assigned to the ptr var , the functions of the base class are called using the pointer variable. And next the pointer address is changed to the derived object as per the rule when the function is called by the pointer it calls the base class functions only. since the show () function is declared as virtual in the base class . the pointer variable of the derived class calls the functions of the deriv1ed class. But the object does not calls the member function display() of the derived class since it is not declared as virtual.

Rules for virtual functions

When virtual functions are created for implementing late binding, we should observe some basic rules that satisfy the compiler requirements.

The virtual functions must be members of some functions
They cannot be static members.
They are accessed by using object pointers.
The virtual function may be a friend of another class.
A virtual function in a base class must be defined even though it may not be used .
The prototype of the base class and derived class must be identical or else c++ considers them as overloaded functions, and the virtual function mechanism is ignored.
We cannot have virtual constructors, but we have virtual destructors.
While a base pointer can point to any type of the derived object, the reverse is not true. That is, we cannot use a pointer to a derived class to access any object of the base type.
When a base pointer points to a derived class, incrementing or decrementing it will not make is to point to the next object of the derived class. It is incremented or decrememnted only relative to its base type. Therefore we should not use this method to move the pointer to the next object.
If a virtual function is defined in the base class it need not be necessarily redefined in the derived class. In such cases, calls will invoke the base function.

Pure virtual functions
The function, which is declared in the base class as virtual, acts as a namesake function and it, and does not does any job. It simply serves as a placeholder.
We may call this function as “do – nothing “ function
E.g.;
Virtual display()=0;
Such functions are called pure functions. This pure virtual class is also called abstract base class.

FILES

A file is a collection of related data stored in a particular are on the disk.
A program typically involves either a both of the following kinds of data communication.
1.data transfer between console(keyboard) unit and program
2.data transfer between program and disk file.

In c++ file uses stream as an interface between the programs and files.

The stream that supplies data to the program is known as inputstream (reads data from file)
The stream that receives data from the program is known as outputstream (write data to the file)

File stream classes
The different file stream classes are

Filebuf – sets the file buffer to read and write, has open() and close() member functions.

Filestreambase – serves as a base class to fstream, ifstream and ofstream classes, contains open() and close() functions.

Ifstream – provides input operation, contains open() with i/p mode.
Inherits some function get(),getline(),read(),seekg(),and tellg functions from istream

Ofstream – provides o/p operation, contains open() with default o/p mode inherits put(),seekp(), tellp() and write() from ostream.

Fstream – i/o operation inherits from istream and ostream through iostream.

Opening and closing of file

When a program is written using files it must have an header file <fstream.h>

A file can be opened in two ways

3. using constructor function of the class
4. 2. Using member function of the class

1.opening a file using a constructor

this involves 2 steps

(a) create a file stream object of an appropriate class **ofstream** to create an o/p stream and **ifstream** to create an i/p stream.
(b) initialize the file onbject with desired filename. For example the following statement open's a file named "result " for output

ofstream outfile("results")

outfile is an object of the class ofstream
similarly

ifstream infile("data")

infile is an object of the class ifstream and file named "data1" is opened for reading

e.g.

```
#include<fstream.h>
main()
{
ofstream outfile ("item");
char name[30];
cin>>name;     //reads name from the user//
outfile<<name;          //puts into the file item//
```

```
outfile.close;
ifstream infile("item")
infile>>name; // reads from the file//
cout<< "name"<<name;        //prints on the screen//
infile.close();
}
```

2.opening files using open()

open() can be used to open many files when it is required. This can be done as follows.,

```
Filestream class stream object;
Stream object. Open("filename");
```

Example:

```
Ofstream outfile;
Outfile.open("data");
_____________
_____________
outfile.close();
outfile.open("data2")'
```

here is the above example first data1.file is opened after manipulating with that file it is closed . then again another file is opened ie. Data2.

For example program refer page no 220 c++ balaguruswamy.

Detecting end of file

Detection of EOF is necessary for preventing any further attempt to read data from the file

While(fin)

Fin returns a value 0 when it reaches end of file. And also returns 0 when any other eror occurs so they started using another statement

```
If(file.eof()! = 0)
{
exit(1);
}
```

to check the EOF.

File open modes

General syntax for opening a file using the modes is

```
Streamobject.open("filename", mode);
```

```
                //program for single level inheritance///
#include<iostream.h>
#include<conio.h>
class student
{
public:
   int rollno;
        void getno();
        void putno(void);
};
        void student : : getno( )
        {
        cin>>rollno;
        }

        void student : : putno()
        {
        cout<<rollno;
        }

class test : public student             //public inheritance//
{
private:
        float mark1,mark2;
public:
        void getmark();
        void putmark();
        void display();
};
        void test : : getmark()
```

```
        {
        cin>>mark1>>mark2;
        }

        void test : : putmark()
        {
        cout<<mark1<<mark2;
        }

        void test : : display()
        {
        putno();
        putmark();
        }
main()
{
test student1;
student1.getno()
student1.getmark();
student1.display();
}
```