School of Computing

Department of Computer Science and Engineering

UNIT - V

Problem Solving Techniques with C and C++ - SCSA1104

UNIT V Object Oriented Programming Concepts

Introduction-Procedure vs. object oriented programming-Concepts: Classes and Objects-Operator & Function Overloading-Inheritance-Polymorphism and Virtual Functions.

1. Introduction:

Programmers write instructions in various programming languages to perform their computation tasks such as:

(i) Machine level Language

(ii) Assembly level Language

(iii) High level Language

Machine level Language : Machine code or machine language is a set of instructions executed directly by a computer's central processing unit (CPU). Each instruction performs a very specific task, such as a load, a jump, or an ALU operation on a unit of data in a CPU register or memory. Every program directly executed by a CPU is made up of a series of such instructions.

Assembly level Language : An assembly language (or assembler language) is a low-level programming language for a computer, or other programmable device, in which there is a very strong (generally one-to-one) correspondence between the language and the architecture's machine code instructions. Assembly language is converted into executable machine code by a utility program referred to as an assembler; the conversion process is referred to as assembly, or assembling the code.

High level Language : High-level language is any programming language that enables development of a program in much simpler programming context and is generally independent of the computer's hardware architecture. High-level language has a higher level of abstraction from the computer, and focuses more on the programming logic rather than the underlying hardware components such as memory addressing and register utilization.

The first high-level programming languages were designed in the 1950s. Now there are dozens of different languages, including Ada , Algol, BASIC, COBOL, C, C++, JAVA, FORTRAN, LISP, Pascal, and Prolog. Such languages are considered high-level because they are closer to human languages and farther from machine languages. In contrast, assembly languages are considered lowlevel because they are very close to machine languages. The high-level programming languages are broadly categorized in to two categories:

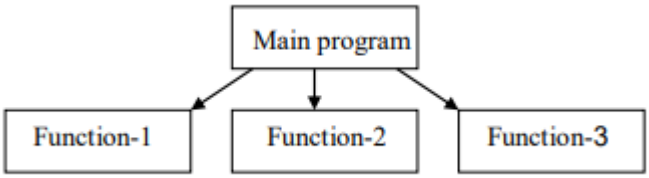
(i) Procedure oriented programming(POP) language.

(ii) Object oriented programming(OOP) language.

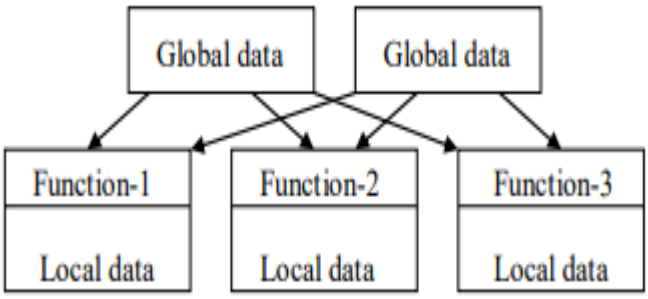
Procedure Oriented Programming (POP) :

The high level languages, such as BASIC, COBOL, C, FORTRAN are commonly known as Procedure Oriented Programming. Using this approach, the problem is viewed in sequence of things to be done, like reading, processing and displaying or printing. To carry out these tasks the function concepts must be used.

- ✓ This concept basically consists of number of statements and these statements are organized or grouped into functions.



- ✓ While developing these functions the programmer must care about the data that is being used in various functions.
- ✓ A multi-function program, the data must be declared as global, so that data can be accessed by all the functions within the program & each function can also have its own data called local data.
- ✓ The global data can be accessed anywhere in the program. In large program it is very difficult to identify what data is accessed by which function. In this case we must revised about the external data and as well as the functions that access the global data. At this situation there is so many chances for an error.

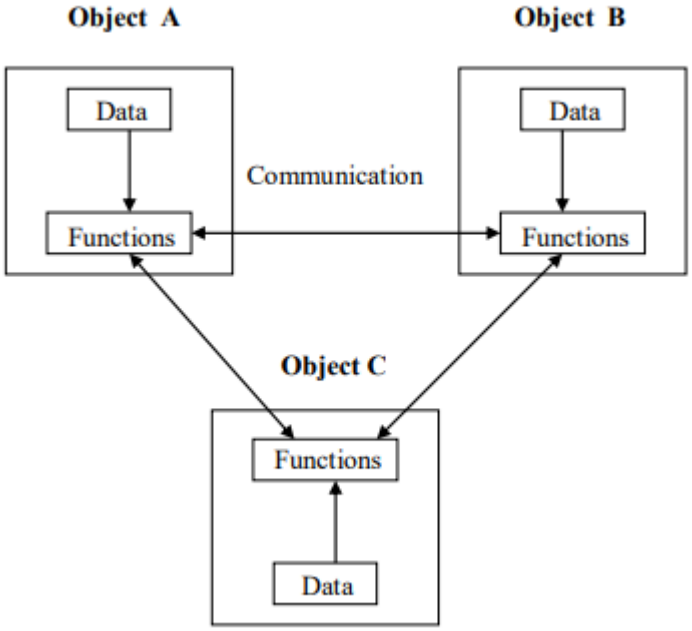


Characteristics of procedure oriented programming:

1. Emphasis is on doing things(algorithm)
2. Large programs are divided into smaller programs known as functions.
3. Most of the functions share global data
4. Data move openly around the system from function to function
5. Function transforms data from one form to another.
6. Employs top-down approach in program design

Object Oriented Programming (OOP) :

- ✓ This programming approach is developed to reduce the some of the drawbacks encountered in the Procedure Oriented Programming Approach.
- ✓ The OO Programming approach treats data as critical element and does not allow the data to freely around the program.
- ✓ It bundles the data more closely to the functions that operate on it; it also protects data from the accidental modification from outside the function.
- ✓ The object oriented programming approach divides the program into number of entities called objects and builds the data and functions that operates on data around the objects.
- ✓ The data of an object can only access by the functions associated with that object.



Features of the Object Oriented programming

1. Emphasis is on doing rather than procedure.
2. programs are divided into what are known as objects.
3. Data structures are designed such that they characterize the objects.
4. Functions that operate on the data of an object are tied together in the data structure.
5. Data is hidden and can't be accessed by external functions.
6. Objects may communicate with each other through functions.
7. New data and functions can be easily added.
8. Follows bottom-up approach in program design.

Comparison of Programming Paradigms:

S. No.	Procedure Oriented Programming (C)	Object Oriented Programming (C++)
1.	Programs are divided into smaller Sub-programs known as functions.	Programs are divides into objects & Classes.
2.	New data and functions can be added easily.	New data and functions can be added easily.
3.	It is a Top-Down Approach	It is a Bottom-Up Approach
4.	Data cannot be secured and available to all the function	Data can be secured and can be available in the class in which it is declared.
5.	Here, the reusability is not possible, hence redundant code cannot be avoided.	Here, We can reuse the existing one using the Inheritance concept.
6.	It does not have any access specifier.	It has access specifiers named Public, Private, Protected, etc.
7.	Data can move freely from function to function in the system.	Objects can move and communicate with each other through member functions.
8.	Overloading is not possible.	Overloading is possible in the form of Function Overloading and Operator Overloading.
9.	Example of POP are : C, VB, FORTRAN, Pascal.	Example of OOP are : C++, JAVA, Small talk.

BASIC CONCEPTS OF OBJECTS ORIENTED PROGRAMMING

The general concepts of OOPS comprises the following.
1. Object

2. Class

3. Data abstraction

4. Inheritance

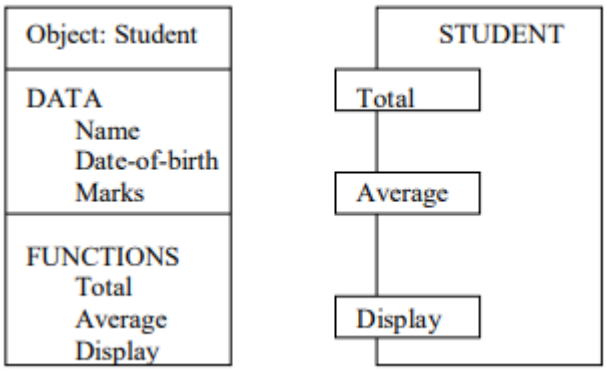
5. Polymorphism

6. Dynamic Binding

7. Message passing.

1. Object

Objects are the basic run-time entities in an object-oriented system. They may represent a person, a place, a bank account, a table of data or any item that the program must handle.
The term objects means a combination of data and program that represent some real word entity. For example: consider an example named Amit; Amit is 25 years old and his salary is 2500. The Amit may be represented in a computer program as an object. The data part of the object would be (name: Amit, age: 25, salary: 2500)
The program part of the object may be collection of programs (retrive of data, change age, change of salary). In general even any user –defined type-such as employee may be used. In the Amit object the name, age and salary are called attributes of the object.



2. Class:

A group of objects that share common properties for data part and some program part are collectively called as class. In C ++ a class is a user defined data type that contains member variables and member functions that operate on the variables.

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 will create an object mango belonging to the class fruit.

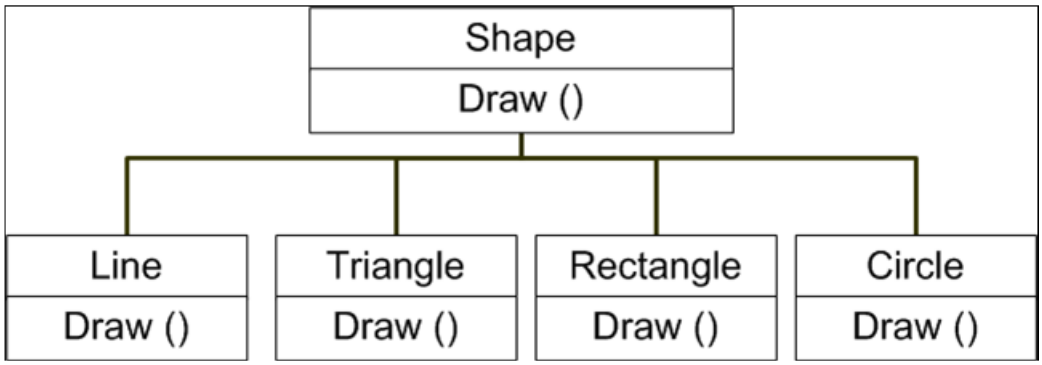
fruit mango;

3. Data Abstraction : Abstraction refers to the act of representing essential features without including the back ground details or explanations. Classes use the concept of abstraction and are defined as size, width and cost and functions to operate on the attributes.

Data Encapsalation : The wrapping up of data and function into a single unit (called 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 access it. These functions provide the interface between the objects data and the program.

4.Inheritence :
Inheritance is the process by which objects of one class acquire the properties of another class. In the concept of inheritance provides the idea of reusablity. This mean that we can add additional features to an existing class with out modifying it. This is possible by designing a new class will have the combined features of both the classes.

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.



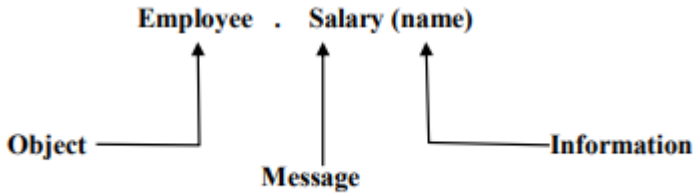
Polymorphism plays an important role in allowing objects having internal structures to share the same external interface. This means that a general class of operations may be accessed in the same manner even though specific actions associated with each operation may differ.

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 Passing

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.
A message for an object is a request for execution of a procedure and therefore will invoke a function (procedure) in the receiving object that generates the desired result. Message passing involves specifying the name of the object, the name of the function (message) and information to be sent.



BENEFITS OF OOP:
Oop offers several benefits to both the program designer and the user. Object-oriented contributes to the solution of many problems associated with the development and quality of software products. The principal advantages are :
1. Through inheritance we can eliminate redundant code and extend the use of existing classes.
2. We can build programs from the standard working modules that communicate with one another, rather than having to start writing the code from scratch. This leads to saving of development time and higher productivity.
3. This principle of data hiding helps the programmer to build secure programs that can't be invaded by code in other parts of the program.
4. It is possible to have multiple instances of an object to co-exist with out any interference.
5. It is easy to partition the work in a project based on objects.
6. Object-oriented systems can be easily upgraded from small to large systems.
7. Message passing techniques for communication between objects makes the interface description with external systems much simpler.
8. Software complexity can be easily managed.

APPLICATION OF OOP:
The most popular application of oops up to now, has been in the area of user interface design such as windows. There are hundreds of windowing systems developed using oop techniques.

Real business systems are often much more complex and contain many more objects with complicated attributes and methods. Oop is useful in this type of applications because it can simplify a complex problem. The promising areas for application of oop includes.

1. Real – Time systems.
2. Simulation and modelling
3. Object oriented databases.
4. Hypertext,hypermedia and expertext.
5. Al and expert systems.
6. Neural networks and parallel programming.
7. Dicision support and office automation systems.
8. CIM / CAM / CAD system.

Basics of C++
C ++ is an object oriented programming language, C ++ was developed by Jarney Stroustrup at AT & T Bell lab, USA in early eighties. C ++ was developed from c and simula 67 language. C ++ was early called 'C with classes'.

C++ Comments:
C++ introduces a new comment symbol //(double slash). Comments start with a double slash symbol and terminate at the end of line. A comment may start any where in the line and what ever follows till the end of line is ignored. Note that there is no closing symbol.
The double slash comment is basically a single line comment. Multi line comments can be written as follows:

```
// this is an example of
// c++ program
// thank you
```

The c comment symbols /* ….*/ are still valid and more suitable for multi line comments.

```
/* this is an example of c++ program */
```

Output Operator:

The statement cout <<"Hello, world" displayed the string with in quotes on the screen. The identifier cout can be used to display individual characters, strings and even numbers. It is a predefined object that corresponds to the standard output stream. Stream just refers to a flow of data and the standard Output stream normally flows to the screen display. The cout object, whose properties are defined in iostream.h represents that stream. The insertion operator << also called the 'put to' operator directs the information on its right to the object on its left.

Return Statement:
In C++ main ( ) returns an integer type value to the operating system. Therefore every main ( ) in C++ should end with a return (0) statement, otherwise a warning or an error might occur.

Input Operator:
The statement cin>> number 1; is an input statement and causes. The program to wait for the user to type in a number. The number keyed in is placed in the variable number1. The identifier cin is a predefined object in C++ that corresponds to the standard input stream. Here this stream represents the key board.
The operator >> is known as get from operator. It extracts value from the keyboard and assigns it to the variable on its right.

Cascading of I/O Operator:

```
cout<<<sum<<<<sum<<"\n"
cout<<< "sum="<<sum<<"\n"<<"average"<<<average<<"\n";
cin>>number1>>number2;
```

Tokens:

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

i. Keywords
ii. Identifiers
iii. Constants
iv. Strings
v. Operators

Keywords:

The keywords implement specific C++ language feature. They are explicitly reserved identifiers and can't be used as names for the program variables or other user defined program elements. The keywords not found in ANSI C are shown in red letter.

C++ Keywords

Asm	double	new	switch
Auto	else	operator	template
Break	enum	private	this
Case	extern	protected	throw
Catch	float	public	try
Char	for	register	typedef
Class	friend	return	union
Const	goto	short	unsigned
Continue	if	signed	virtual
Default	inline	sizeof	void
Delete	long	struct	while

Identifiers:

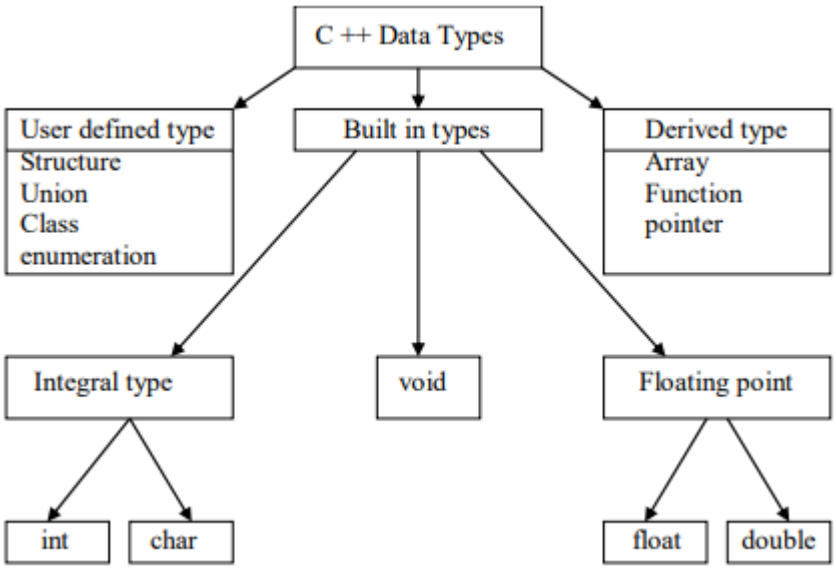
Identifiers refers to the name of variable , functions, array, class etc. created by programmer. Each language has its own rule for naming the identifiers.

The following rules are common for both C and C++.

1. Only alphabetic chars, digits and under score are permitted.
2. The name can't start with a digit.
3. Upper case and lower case letters are distinct.
4. A declared keyword can't be used as a variable name

In ANSI C the maximum length of a variable is 32 chars but in c++ there is no bar

Basic Data Types In C++



Both C and C++ compilers support all the built in types. With the exception of void the basic datatypes may have several modifiers preceding them to serve the needs of various situations. The modifiers signed, unsigned, long and short may applied to character and integer basic data types. However the modifier long may also be applied to double.

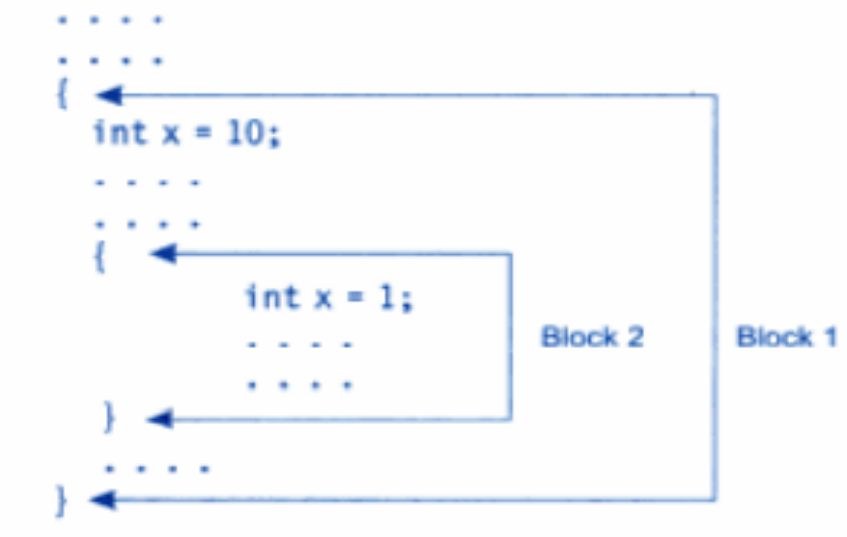
The type void normally used for:
1) To specify the return type of function when it is not returning any value.
2) To indicate an empty argument list to a function.
Example: Void function(void);

Operators in C++ :
C++ has a rich set of operators. All C operators are valid in C++ also. In addition. C++ introduces some new operators.

<<	insertion operator
>>	extraction operator
: :	scope resolution operator
: :*	pointer to member declarator
*	pointer to member operator
.*	pointer to member operator
Delete	memory release operator
Endl	line feed operator
New	memory allocation operator
Setw	field width operator

Scope Resolution Operator:

Like C,C++ is also a block-structured language. Block -structured language. Blocks and scopes can be used in constructing programs. We know same variables can be declared in different blocks because the variables declared in blocks are local to that function.
Blocks in C++ are often nested. Example:



Block2 contained in block l .Note that declaration in an inner block hides a declaration of the same variable in an outer block and therefore each declaration of x causes it to refer to a different data object . With in the inner block the variable x will refer to the data object declared there in.

In C, the global version of a variable can't be accessed from with in the inner block. C++ resolves this problem by introducing a new operator :: called the scope resolution operator .This can be used to uncover a hidden variable.

Syntax: : : variable –name;
Example:

```
#include <iostrcam.h>
int m=10;
main()
{
int m=20;
{
int k=m;
int m=30;
cout<<”we are in inner block”;
cout<<"k="<<k<<endl;
cout<<"m="<<m<<endl;
cout<<":: m="<<:: m<<endl;
}
cout<<”\n we are in outer block \n”;
cout<<"m="<<m<<endl;
cout<<":: m="<<:: m<<endl;
}
```

CLASS:-

Class is a group of objects that share common properties and relationships .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. It allows the data to be hidden, if necessary from external use. When we defining a class, we are creating a new abstract data type that can be treated like any other built in data type.

Generally a class specification has two parts:-

a) Class declaration
b) Class function definition

the class declaration describes the type and scope of its members. The class function definition describes how the class functions are implemented.

Syntax:-

class class-name

{

private:

variable declarations; function declaration ;

public:

variable declarations; function declaration;

};

The members that have been declared as private can be accessed only

from with in the class. On the other hand , public members can be accessed from outside the class also. The data hiding is the key feature of oops. The use of keywords private is optional by default, the members of a class are private.

The variables declared inside the class are known as data members and the functions are known as members mid the functions. Only the member functions can have access to the private data members and private functions. However, the public members can be accessed from the outside the class. The binding of data and functions together into a single class type variable is referred to as encapsulation.

Syntax:-

class item

{

public:

int member;

float cost;

void getldata (int a ,float b);

void putdata (void);

}

The class item contains two data members and two function members, the data members are private by default while both the functions are public by declaration. The function getdata() can be used to assign values to the member variables member and cost, and putdata() for displaying their values . These functions provide the only access to the data members from outside the class.

Creating Objects:

Once a class has been declared we can create variables of that type by using the class name.

Example:

item x;

creates a variables x of type item. In C++, the class variables are known as objects. Therefore x is called an object of type item.

Syntax for Object Declaration:

ClassName ObjectName;

Eg: car c1;

creates a variables c1 of type car. In C++, the class variables are known as objects. Therefore c1 is called an object of type item.

Declaring Objects:

When a class is defined, only the specification for the object is defined; no memory or storage is allocated.

Where should we create object?

Inside main function

Why should we create object?

To access the data and functions defined in the class, you need to create objects.

Accessing class members:

- ✓ The data members and member functions of class can be accessed using the dot('.') operator with the object. We can access data member either privately or publicly.

Accessing public data member:

Public data member can be accessed from anywhere.

Syntax for accessing public data member:

Objectname.data member=value;

Eg: c1.rollno=786;

Accessing private data member:

The private data of a class can be accessed only through the member functions of that class. The main() cannot contains statements that the access number and cost directly.

Syntax:

object name.function-name(actual arguments);

Example:- x. getdata(100,75.5);

It assigns value 100 to number, and 75.5 to cost of the object x by implementing the getdata() function .

similarly the statement

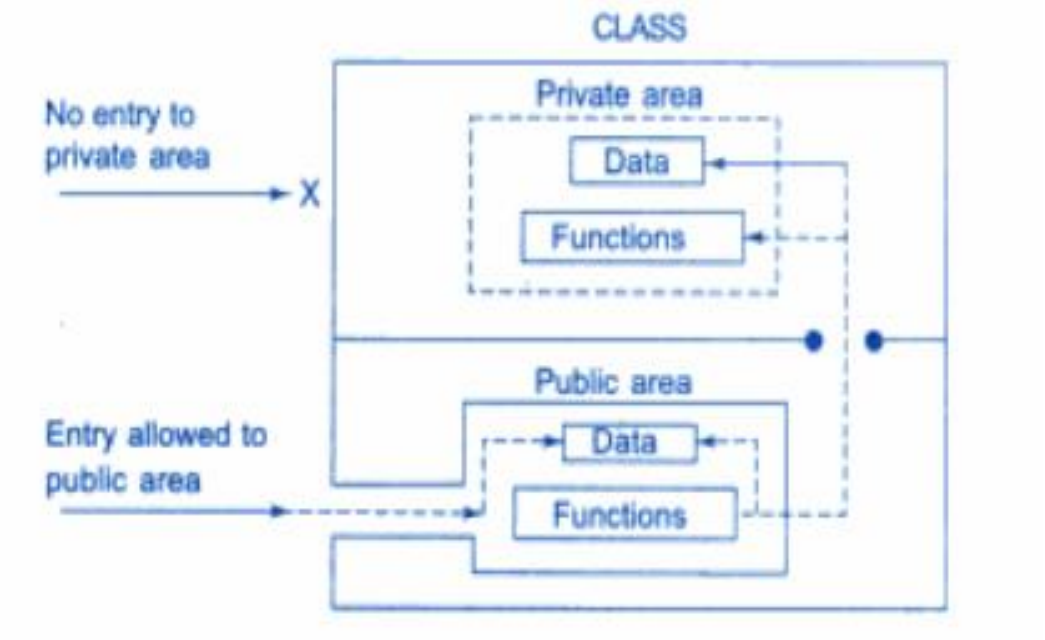
x. putdata ( ); //would display the values of data members.

x. number = 100 is illegal .Although x is an object of the type item to which number belongs ,

the number can be accessed only through a member function and not by the object directly.

Example:

```
class xyz
{
        Int x;
        Int y;
public:
        int z;
};

----------
----------

xyz p;
p. x =0;        error . x is private
p. z=10;        ok ,z is public
```



Defining Member Function:

Member can be defined in two places

- Outside the class definition
- Inside the class function

Outside the Class Deflnat1on;

Member function that are declared inside a class have to be defined separately outside the class.Their definition are very much like the normal functions.

An important difference between a member function and a normal function is that a member function incorporates a membership. Identify label in the header. The 'label' tells the compiler which class the function belongs to.

Syntax:

return type class-name::function-name(argument declaration)
{
function-body
}

The member ship label class-name :: tells the compiler that the function 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.

Example:

void item :: getdata (int a , float b)

```
{
number=a;
cost=b;
}
void item :: putdata ( void)
{
cout<<”number=:”<<number<<endl;
cout<<”cost=”<<cost<<endl;
}
```

The member function have some special characteristics that are often used in the program development.

- Several different classes can use the same function name. The "membership label" will resolve their scope, member functions can access the private data of the class .A non member function can't do so.
- A member function can call another member function directly, without using the dot operator.

Inside the Class Def1nation:

Another method of defining a member function is to replace the function declaration by the actual function definition inside the class .

Example:

```
class item
{
        Intnumber;
        float cost;
public:
        void getdata (int a ,float b);
        void putdata(void)
        {

        }
        };
```

Structure of C++ program

Header Files
Class Declaration
Member function definition
Main function

C++ Program With Class:

(member defining inside the class)

Write a simple program using class in C++ to input subject mark and prints it.

ans:

```
class marks
{
        private :
                int ml,m2;
                public:
                void      getdata();
                void displaydata();
};
void marks: :getdata()
{
        cout<<"enter 1st subject mark:";
        cin>>ml;
        cout<<"enter 2nd subject mark:";
        cin>>m2;
}
void marks: :displaydata()
```

```
{
    cout<<"1st subject mark:"<<ml<<endl ;
    cout<<"2nd subject mark:"<<m2;
}
void main()
{
clrscr();
marks     x;
x.getdata();
x.displaydata();

}
```

Defining member function outside class

```
# include< iostream. h>
  class item
        {
            int number;
            float cost;
public:
            void getdata ( int a , float b);
            void putdala ( void)
                {
                cout<<"number:"<<number<<endl;
                cout<<"cost :"<<cost<<endl;
                }
                };
        void item :: getdata (int a , float b)
        {
            number=a;
            cost=b;
        }
    main ( )
        {
            item                    x;
            cout<<"\nobjectx"<<endl;
            x. getdata( 100,299.95);
            x .putdata();
            item y;
            cout<<"\n object y"<<endl;
            y. getdata(200,175.5);
            y. putdata();
            }
```

Output: **object x**
number 100

cost=299.950012

object 4
cost=175.5

Function Overloading:

Overloading refers to the use of the same thing for different purposes . C++ also permits overloading functions .This means that we can use the same function name to creates functions that perform a variety of different tasks. This is known as function polymorphism in oops.

Using the concepts of function overloading , a family of functions with one function name but with different argument lists in the functions 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.

//Declaration

```
int add(int a, int b); //prototype 1

int add (int a, int b, int c); //prototype 2 double
add(double x, double y); //prototype 3 double
add(double p , double q); //prototype 4
```

//function call

```
cout<<add(5,10);       //uses      prototype      1
cout<<add(15,10.0);    //uses      prototype      4
cout<<add(12.5,7.5);   //uses      prototype      3
cout<<add(5,10,15);    //uses      prototype      2
cout<<add(0.75,5); //uses prototype 5
```

A function call first matches the prototype having the same no and type of arguments and then calls the appropriate function for execution.

The function selection invokes the following steps:-

b) The compiler first tries to find an exact match in which the types of actual arguments are the same and use that function .

c) If an exact match is not found the compiler uses the integral promotions to the actual arguments such as :

```
char   to   int
float to double
to find a match
```

d)When either of them fails ,the compiler tries to use the built in conversions to the actual arguments and them uses the function whose match is unique . If the conversion is possible to have multiple matches, then the compiler will give error message.

Example:

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

A function call such as :- square(lO)

Will cause an error because int argument can be converted to either long or double .There by

creating an ambiguous situation as to which version of square( )should be used.

Example Program:

```
#include<iostream.h>
class funoverloading
{
        int a, b, c;
        public:
        void add ( )
        {
                cin>>a>>b;
                c = a + b;
                cout << c;
        }
int add (int a, int b)
        {
                c = a + b;
                return c;
        }
void add (int a)
        {
                cin>>b;
                c = a + b;
                cout<<c;
        }

};
void main ( )
{
funoverloading f1;
int x;
f1. add ( );
x = f1. add (10, 5);
cout<<x;
f1. add (5);
}
```

Output:-

```
5 3
c = 8
c = 15
5
c = 10
```

OPERATOR OVERLOADING:-

C ++ has the ability to provide the operators with a special meaning for a data type. The mechanism of giving such special meanings to an operator is known as operator overloading.

The process of overloading involves the following steps:

1. Create a class that defines the data type that is to be used in the overloading operation.
2. Declare the operator function operator op()in the public part of the class
3. It may be either a member function or a friend function
4. Define the operator function to implement the required operations.

We can overload all the C++ operators except the following:

- Class members access operator (. , .*)
- Scope resolution operator (: :)
- Size operator(sizeof)
- Condition operator (? :)

Syntax:-
returntype classname:: operator op (argument list)
{
Function body
}.
Example:-
void space:: operator-( )
{
x=-x;
}
OVERLOADING UNARY OPERATORS:-
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 here how to 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 of its data items.

```
#include<iostream.h>
{
int x;
int y;
int z;
public:
void getdata(int a, int b, int c);
void display(void);
void operator-(); //overload unary minus
};
void space::getdata(int a, int b, int c)
{
x=a;
y=b;
z=c;
}
void space::display(void)
{
cout<<x<<” ”;
cout<<y<<” “;
cout<<z<<” “;
}
void space::operator-()
{
x=-x;
y=-y;
z=-z;
}
```

```
int main()
{
space S;
S.getdata(10,-20,30);
cout<<"S=";
S.display();
- S;
cout<<"S=";
S.display();
return 0;
}
```

Output:-

S= 10 -20 30

S= -10 20 -30

Note:

The function operator-() takes no argument. Then, what does this operator function do? It changes the sign of data members of the object S. Since this function is a member function of the same class, it can directly access the members of the object
which activated it.

OVERLOADING BINARY OPERATORS:-

The same mechanism which is used in overloading unary operator can be used
to overload a binary operator.

```
# include<iostream.h>
class complex
{
        float x;
        float y;
        public:
        complex()
{
}
complex(float real, float imag)
{
x=real;
y=imag;
}
complex operator+(complex);
void display(void);
};
complex complex::operator+(complex c)
{
complex temp;
temp.x=x+c.x;
temp.y=y+c.x;
return(temp);
}
void complex::display(void)
{
cout<<x<<"j"<<y<<"\n";
}
int main()
{
complex C1,C2,C3;
```

```
C1=complex(2.5,3.5)
C2=complex(1.6,2.7)
C3= C1 + C2;
cout<<"C1 = ";
C1.display();
cout<<"C2 = ";
C2.display();
cout<<"C3 = ";
C3.display();
return 0;
}
```

Output:-

```
C1=2.5 +j3.5
C2=1.6 + j2.7
C3= 4.1 +j6.2
```

INHERITANCE

Reaccessability is yet another feature of OOP's. C++ strongly supports the concept of reusability. The C++ classes can be used again in several ways. Once a class has been written and tested, it can be adopted by another programmers. This is basically created by defining the new classes, reusing the properties of existing ones. The mechanism of deriving a new class from an old one is called 'INHERTTENCE'. This is often referred to as IS-A' relationship because very object of the class being defined "is" also an object of inherited class. The old class is called 'BASE' class and the new one is called 'DERIEVED' class.

The derived class with only one base class is called single inheritance and one with several base classes is called multiple inheritance. On the other hand, the traits of one class may be inherited by more than one class. This process is known as hierarchical inheritance. The mechanism of deriving a class from another 'derived class' is known as multilevel inheritance.

Types of Inheritance:

1. Single inheritance
2. Multiple inheritance
3. Multilevel inheritance
4. Hierarchical inheritance(refer Assignment)
5. Hybrid inheritance

Defining derived classes:-

A derived class is defined by specifying its relationship with the base class in addition to its own details.
The general form:-

```
class derived-class-name: visibly-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. Visibility mode specifies whether the features of the base class are privately derived or publicly derived.
Examples:-

```
class ABC:private XYZ//private derivation
{
```

```
        members of ABC
};
class ABC:public XYZ//public derivation
{
        members of ABC
};
class ABC:XYZ//private derivation by default
{
        members of ABC
};
```

Making a Private Member Inheritable

Basically we have visibility modes to specify that in which mode you are deriving the another class from the already existing base class. They are:

Private: When base class is privately inherited by a derived class, 'public members' of the base class become 'private members' of the derived class and therefore the public members of the base class can only be accessed by the member functions of the derived class. They are inaccessible to the objects of the derived class.

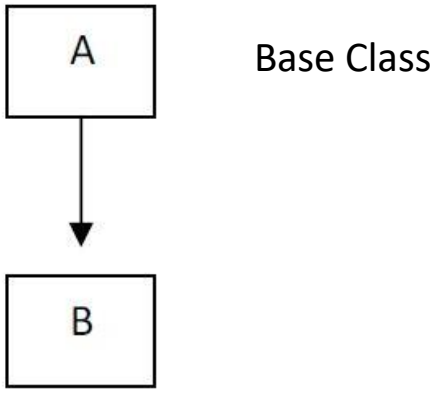
Remember, a public member of a class be accessed by its own objects using the dot operator. The result is that no member of the base class is accessible to the objects of the derived class

Public: On the other hand, when the base class is publicly inherited, 'public members' of the base class become 'public members' of derived class and therefore they are accessible to the objects of the derived class.

In both the cases, the private members are not inherited and therefore, the private members of the base class will never become the members of its derived class.

Single Inheritance:

The new class can be derived from only one base class is called single inheritance.



Base Class

Let us consider a simple example to illustrate inheritance. The following program shows the base class B and a derived class D. The class B contains one private data member, one public data member, and three public member functions. The class D

contains one private data member and two public member functions.

```
#include<iostream.h>
class B
{
int a; //private; not inheritable
public:
int b; //public; ready for inheritance
void get_ab()
{
a=5;
b=10;
}
int get_a(void)
{
return a;
}
void show_a(void)
{
cout<<”a=”<<a<<endl;
}
};
class D:public B //public derivation
{
int c;
public:
void mul(void)
{
c=b*get_ab();
}
void display()
{
cout<<”a=”<<get_ab()<<endl;
cout<<”b=”<<b<<endl;
cout<<”c=”<<c<<endl;
}
};
void main()
{
D d;
d.get_ab();
d.mul();
d.show_a();
d.display();
d.b=20;
d.mul();
d.display();
}
```

Output:-

```
a=5
a=5
b=10
c=50
a=5
```

b=20
c=100
The class D is a public derivation of the class B. Therefore, D inherits all the public members of B and retains their visibility. Thus a public member of the base class B is
also a public member of the derived class D. The private members of B cannot be inherited by D. The program illustrates that the objects of class D have access to all the public members of B. Let us have a look at the functions show_a() and mul().

```
void show_a()
{
cout<<"a="<<a<<endl;
}
void mul()
{
c=b*get_a();     //c=b*a
}
```

Although the data member a is private in B and cannot be inherited, objects of D are able to access it through an inherited member function of B. Let us now consider the case of private derivation.

```
#include<iostream.h>
class B
{
int a;
public:
int b;
void get_ab();
int get_a(void);
void show_a(void);
};
class D:private B
{
int c;
public:
void mul(void);
void display();
};
```

In private derivation, the public members of the base class become private members of derived class. Therefore, the objects of D can not have direct access to the public member functions of B. The statement such as d.get_ab(),d.get_a(),d.show_a() will not work.

Program:-

```
#include<iostream.h>
class B
{
int a;//private; not inheritable
public:
int b;//public; ready for inheritance
void get_ab()
{
a=5;
b=10;
}
int get_a(void)
{
return a;
```

```
}
void show_a(void)
{
cout<<"a="<<a<<endl;
}
};
class D:private B//private derivation
{
int c;
public:
void mul(void)
{
c=b*get_ab();
}
void display()
{
cout<<"a="<<get_ab()<<endl;
cout<<"b="<<b<<endl;
cout<<"c="<<c<<endl;
}
};
void main()
{
D d;
// d.get_ab(); it won't work
d.mul();
//d.show_a(); won't work
d.display();
//d.b=20; won't work
d.mul();
d.display();
}
```

Output:-

```
a=5
b=10
c=50
a=12
b=20
c=240
```

A private member of a base class cannot be inherited and therefore it is not available for derived class directly. A protected which serve a limited purpose in inheritance. A member declared as protected is accessible by the member functions within its class and any class immediately derived from it. It cannot be accessed by the functions outside these two classes. A class can now use all the three visibility modes as illustrated below:

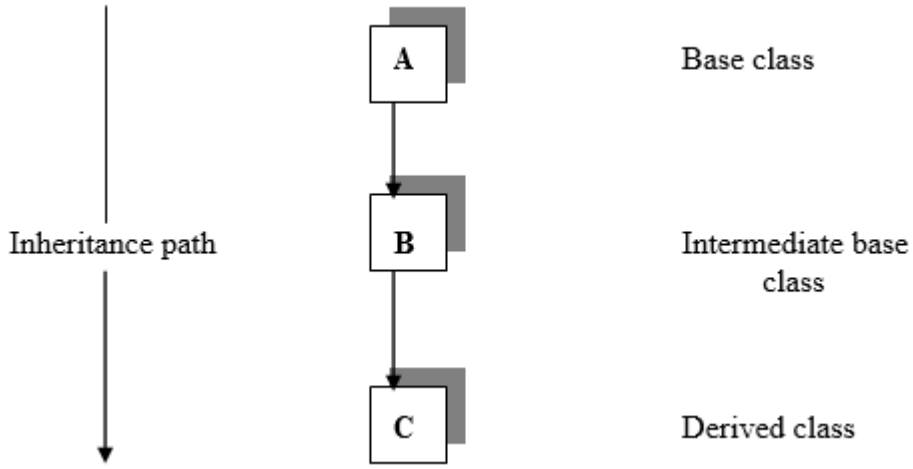
```
class alpha
{
private:
------- //optional
------- // visible to member functions
------- // within its class
protected:
---------- // visible to member functions
---------- // of its own and derived class
public:
```

---------- // visible to all functions
---------- //in the program
}

When a protected member is inherited in public mode, it becomes protected in the derived class too, and therefore is accessible by the member functions of the derived class. It is also ready for further inheritance. A protected member, inherited in the private mode derivation, becomes private in the derived class. Although it is available to the member functions of the derived class, it is not available for further inheritance.

Multilevel Inheritance

When the inheritance is such that, the class A serves as a base class for a derived class B which in turn serves as a base class for the derived class C. This type of inheritance is called 'MULTILEVEL INHERITENCE'. The class B is known as the 'INTERMEDIATE BASE CLASS' since it provides a link for the inheritance between A and C. The chain ABC is called 'ITNHERITENCE*PATH' for e.g.



Syntax for Multilevel Inheritance :

```
Class A
        {
        //body
        }
        Class B : public A
        {
        //body
        }
        Class C : public B
        {
        //body
        }
```

Following program exhibits the multilevel inheritance.

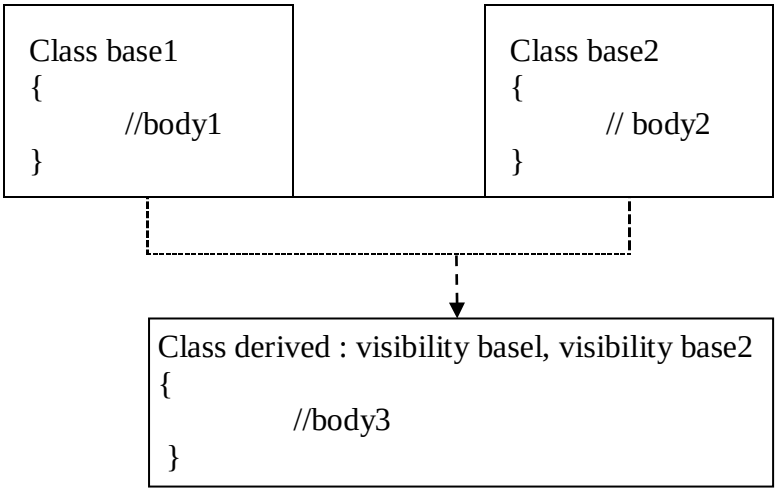
```
        #include<iostream.h>
        #include<conio.h>
        class student       // Base class declaration
        {
```

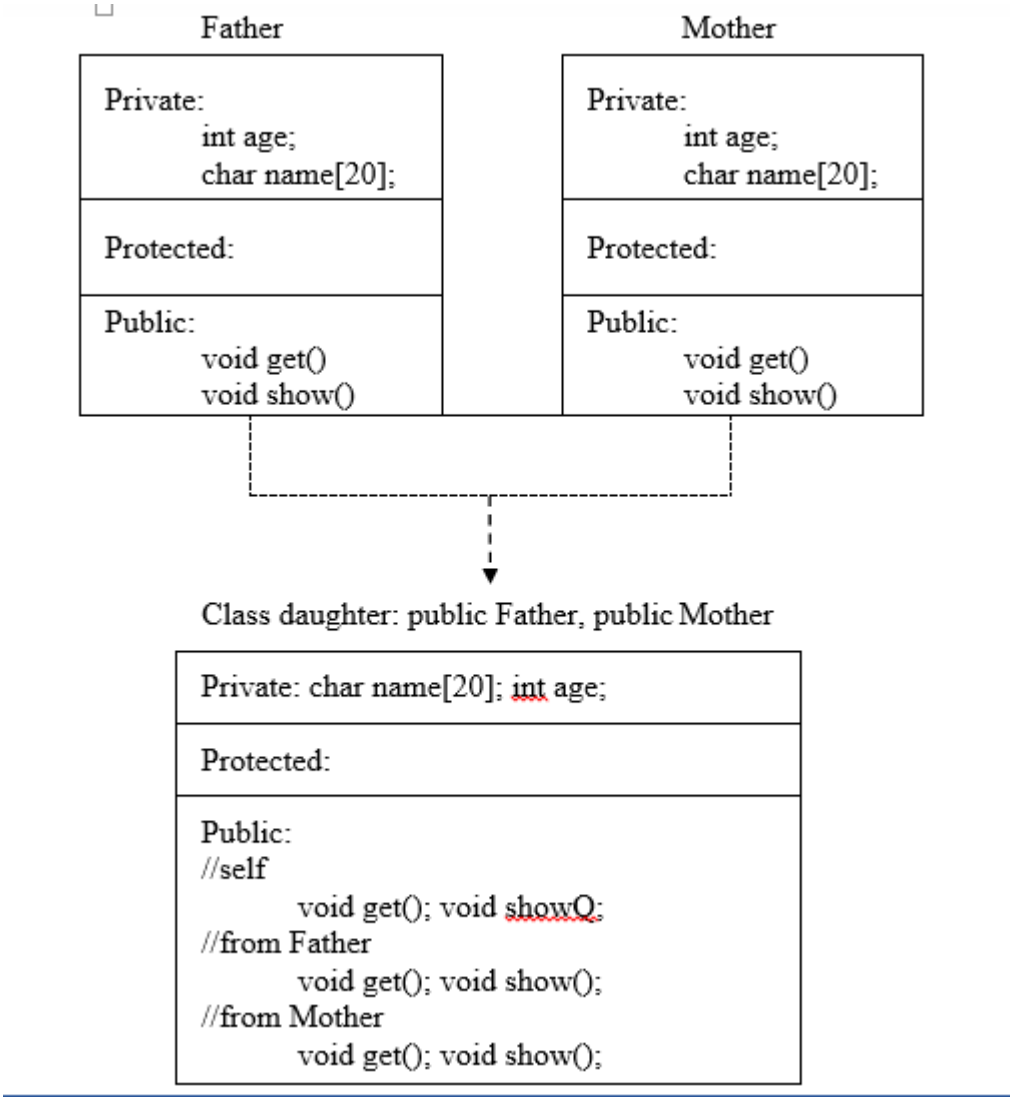
```
        int rollno;
        char name[20];
        public;
        void getstudentdetails( )
        {
        cout << "Enter  your name and rollno" ;
        cin name>>rollno;
        }
        void printstudentdetails( )
        {
        cout << " my name is : " <<name<< " my roll no is : " <<rollno;
        }
class marks : public student //Intermediate base class derived
{               //publicly from the base class
        protected:
        int m1,m2,m3;
        public:
        void getmarks ( )
  {
  cout<<"Enter three subject marks";
  cin>>m1>>m2>>m3;
  }
  void printmarks( )
  {
  cout<<"m1="<<m1;
  cout<<"m2="<<m2;
  cout<<"m3="<<m3;

};
class result: public marks      //declaration of derived class
{               //publicly inherited from the
        int total;
        float avg;      //intermediate base class
        public:
        void printresult( )
    {
            Total=m1+m2+m3;
            Avg=total/3.0;
            cout<<"Total="<<total;
            cout<<"Average="<<avg;
    }

};

void main ( )
{
        clrscr ( ) ;
                student s1;
                s1.getstudentdetails();
                s1.getmarks();
                s1.printstudentdetails();
                s1.printmarks();
```

```
        s1.printresult();
    }
```

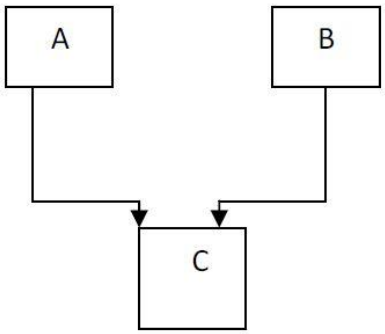
Multiple Inheritance:

A class can inherit the attributes of two or more classes. This mechanism is known as 'MULTIPLE INHERITENCE'. Multiple inheritance allows us to combine the features of several existing classes as a starring point for defining new classes. It is like the child inheriting the physical feature of one parent and the intelligence of another. The syntax of the derived class is as follows:





Syntax for Multiple Inheritance:



Class A
{
//body of the base class A
}
Class B
{
//body of the base class B
}
Class C:visibility mode class A, visibility class B
{
//body of the derived class C
}

Example:

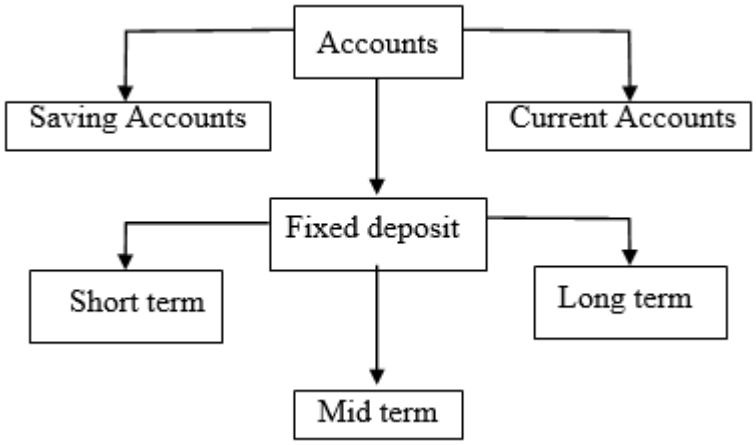
```
#include<iostream.h>
class A
{
```

```
        protected:
        int rollno;
        public:
        void getroll(int x)
        {
        rollno=x;
        }
};
class B
{
        protected:
        int sub1,sub2;
        public:
        void getmark(int y,int z)\
{
        sub1=y;
        sub2=z;
}
};
class C : public A, public B
{
        int total;
        public:
        void display()
        {
        total=sub1+sub2;
        cout<<"roll no:"<<rollno<<"sub1:"<<sub1<<"sub2:"<<sub2;
        cout<<"total"<<total;
}
};
void main()
{
C s;
s. getroll(435);
s.getmark(100,90);
s.display();
}
```

Run:

Roll no : 435
Sub1: 100
Sub2: 90

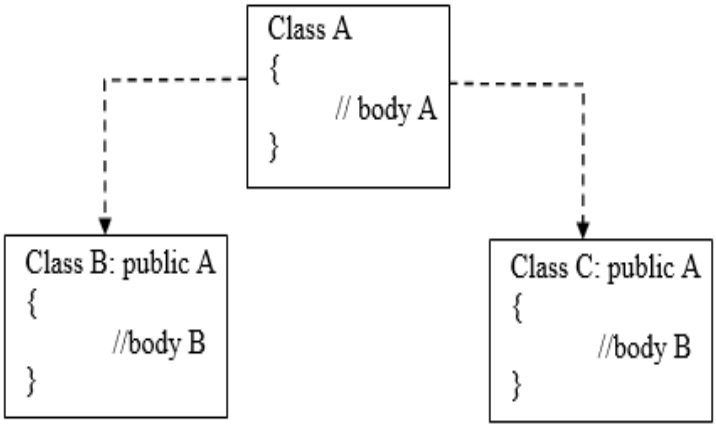
HIERARCHICAL INHERITANCE:-



Another interesting application of inheritance is to use is as a support to a hierarchical design of a class program. Many programming problems can be cast into a hierarchy where certain features of one level are shared by many others below that level for e.g.

In general the syntax is given as

```
Class A
{
//body of base class A
}
Class B:visibility mode class A
{
//body of derived class B
}
Class C:visibility mode class A
{
//body of derived class C
}
```



In C++, such problems can be easily converted into hierarchies. The base class will include all the features that are common to the subclasses. A sub-class can be constructed by inheriting the features of base class and so on. In this scheme the base class will include all the features that are common to the subclasses.

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

```
{
protected:
int x, y;
public:
void get ( )
{
cout<<"Enter two values"<<endl;
cin>> x>>y;
}
};
class B : public A
{
private:
int m;
public:
void add( )
{
m= x + y;
cout<<"The Sum is "<<m;
}
};
class C : public A
{
private:
int n;
public:
void mul( )
{
n= x * y;
cout << "The Product is "<<n;
}
};
class D : public A
{
private:
float l;
public:
void division( )
{
l = x / y;
cout <<"The Quotient is "<< l;
}
};
void main( )
{
B obj1;
C obj2;
D obj3;
B .get( );
B .add( );
```

```
C .get( );
C .mul( );
D .get( );
D .division( );
}
```

Output of the Program

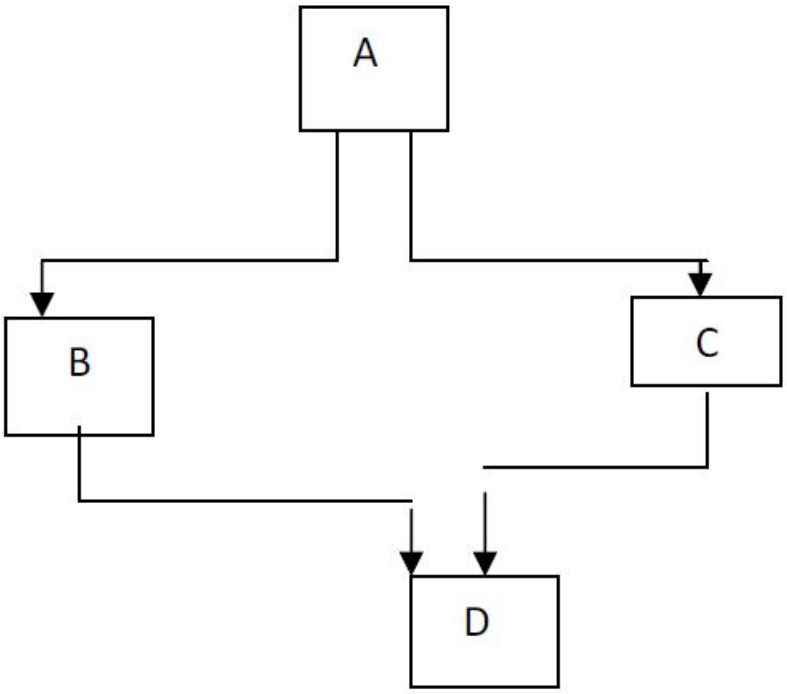
```
Enter two values
12 6
The Sum is 18
The Product is 72
The Quotient is 2.0
```

HYBRID INHERITANCE:-

There could be situations where we need to apply two or more types of inheritance to design a program. Basically Hybrid Inheritance is the combination of one or more types of the inheritance.

Hybrid inheritance = multiple inheritance + multi-level inheritance

This new class can be derived by either multilevel or multiple method or both.



Example program

In this program the derived class (result) object will be inheriting the properties of both test class and sports class.

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

```
protected:
int rollno;
public:
void getroll (int x)
{
rollno = x;
}
};
class test: public student
{
protected:
int sub1, sub2’
public:
void getmark (int y, int z)
{
sub1 = y; sub2 = z;
}
};
class sports
{
protected:
int score;
public:
void getscore (int a )
{
score=a;
}
};
class result: public test, public sports
{
int total;
public:
void display()
{
total= sub1+sub2+score;
cout<<”rollno:”<<rollno<< “total:”<<”Score:”<<total;
}
};
void main( )
{
result S;
s.getroll(101);
s.getmark(90,98);
s.getscore(2000);
s. display( );
}
```

Run:

Rollno: 101
Total: 188
Score: 2000

VIRTUAL FUNCTION

Virtual functions, one of advanced features of OOP is one that does not really exist but it« appears real in some parts of a program. This section deals with the polymorphic features which are incorporated using the virtual functions.

The general syntax of the virtual function

```
class use_detined_name
    {
      private:
      public:
      virtual return_type function_name1 (arguments);

      virtual return_type function_name2(arguments);

      virtual return_type function_name3( arguments);

----------------------

       };
```

To make a member function virtual, the keyword virtual is used in the methods while it is declared in the class definition but not in the member function definition. The keyword virtual precedes the return type of the function name. The compiler gets information from the keyword virtual that it is a virtual function and not a conventional function declaration.

The virtual function should be defined in the public section of a class. When one function is made virtual and another one is normal, and compiler determines which function to call at runtime.
For. example, the following declararion of the virtual function is valid.

```
class point
 {
        int  x;
        int y;
        public:
        virtual int length ( );
        virtual void display ( );
};
```

Remember that the keyword virtual should not be repeated in the definition if the definition occurs outside the class declaration. The use of a function specifier virtual in the function definition is invalid.

For example

```
class point
 {
         int x ;
        int y ;
        public:
        virtual void display ( );
};
```

virtual void point: : display ( ) //error
{
Function Body
}
In virtual function we have to create a ‘basepointer’.
If the basepointer paints to a base class object means it will execute the baseclass function.
If the basepointer points to a derived class object means it will execute the derived class function.

Example Program:

```
#include <iostream.h>
Class A
{
        public:
        void displayA( )
        {
        cout<<”Welcome”;
        }
        virtual void show( )
        {
        cout<<”Hello”;
        }
};
class B: public A
{
        public:
        void display( )
        {
        cout<<”Good Morning”;
        }
        void show( )
        {
        cout<<”Hai”;
        }
};
void main( )
{
A s1;
B s2;
A *bptr;
bptr = &s1;
bptr->show( ); //Calls base class show
bptr=&s2( );
bptr->show( ); //Calls derived class show
s2. displayA( ); //Calls base class displayA
s2.display( ); //Calls derived class display
}
RUN:
Hello
Hai
```

Welcome
Good Morning

Explanation:

The base pointer first holds the address of the base class object means it will run the base class member function display and show will be executed. Next the base pointer points to the derived class object, in this before executing the derived class function if will check whether the corresponding base class function is 'virtual' or not, if it is not virtual then execute the base class function otherwise it will execute the derived class function.

Rules for virtual function:

1. Virtual function must be a members of some class.
2. They cannot be static members.
3. They are accessed by using base pointer.
4. A virtual function in a base class must be defined eventhough it may not be used.
5. A prototype (or declaration) of a base class virtual function and all the derived class version must be identical.
6. We cannot we a pointer to a derived class to access an object of the base