

LECTURE NOTES

Object Oriented Programming Through C++

Name Of The Programme	B.Tech-CSE
Regulations	R-19
Year and Semester	II Year I Semester
Name of the Course Coordinator	Dr. B. Tarakeswara Rao
Name of the Module Coordinator	Dr Md. Umarkhan
Name of the Program Coordinator	Mr.N.Md.Jubair Basha



DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
KALLAM HARANADHAREDDY INSTITUTE OF TECHNOLOGY

NH-5, Chowdavaram Village, Guntur, Andhra Pradesh, India
Approved By AICET, New Delhi, Permanently Affiliated to JNTUK Kakinada
Accredited By NBA, Accredited By NAAC with A Grade

ACADEMIC YEAR 2020-21

UNIT-1

[INTRODUCTION TO C++]

Syllabus:

Difference between C and C++, Evolution of C++, The Object Oriented Technology, Disadvantage of Conventional Programming-, Key Concepts of Object Oriented Programming, Advantage of OOP, Object Oriented Language.

Introduction:

Program : A program is a set of instructions that a computer follows in order to perform a particular task.

Programmers write instructions in various programming languages to perform their computation tasks.

Various types of Languages are :

- (i) Machine level Language
- (ii) Assembly level Language
- (iii) High level Language

Machine level Language :

Machine Language is a collection of binary digits (01001101) or bits .

Machine Language is the only language a computer understands.

Most machine languages consist of binary codes for both data and instructions.

E.g., to add two numbers we would need a series of binary codes such as:

```
0010 0000 0000 0100
0100 0000 0000 0101
0011 0000 0000 0110
```

Assembly level Language :

An assembly language is a low-level programming language **designed for a specific type of processor**.

It uses mnemonic codes(MOV,ADD,JMP,...) that is English-like abbreviations to represent the machine-language instructions.

Assembly language has to be converted into executable machine code , a translator program called an assembler is used to convert each instruction from the assembly language to the machine language.

for E.g.: To add two numbers A and B in assembly lang
we use LOAD instruction, ADD instruction to add

```
LOAD R1, A
LOAD R2 , B
ADD    R1, R2
```

High level Language :

High-level language is more English-like.

Each high- level language has to be translated to low-level language .

We Use compilers to translate high-level language into machine language.

Compilers translate the whole program first, then execute the object program.

E.g., To add two numbers A and B simply we can use '+' operator

read two values in to A and B

```
Sum = A + B
```

A programming language such as C, enables to write programs which is understandable to programmer(human) and can perform any sort of task. such languages are considered as High-level because they are close to human languages. In contrast, assembly languages are considered low- level because they are very close to machine languages.

The first high-level programming languages were designed in the 1950s.

Examples : Ada , Algol, BASIC, COBOL, C, C++, JAVA, Python, FORTRAN, LISP, Pascal, Prolog.

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 Language

In the procedure oriented approach, the problem is viewed as sequence of things to be done such as reading , calculating and printing. A number of functions are written to accomplish this task.

Procedure oriented programming basically consist of writing a list of instruction or actions for the computer to follow and organizing these instruction into groups known as functions.

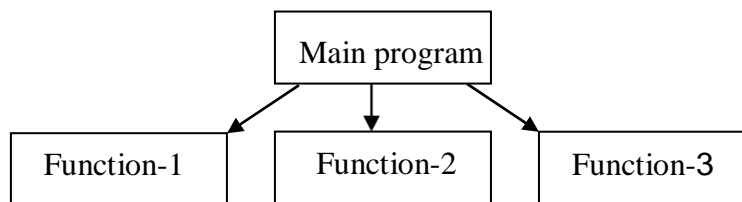


fig : structure of procedure oriented programming

The disadvantages of the procedure oriented programming languages is:

1. Global data access
2. It does not model real word problem very well
3. No data hiding

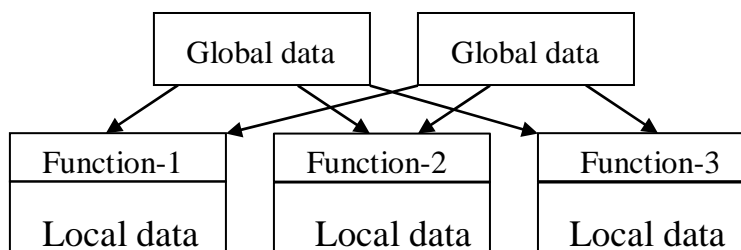
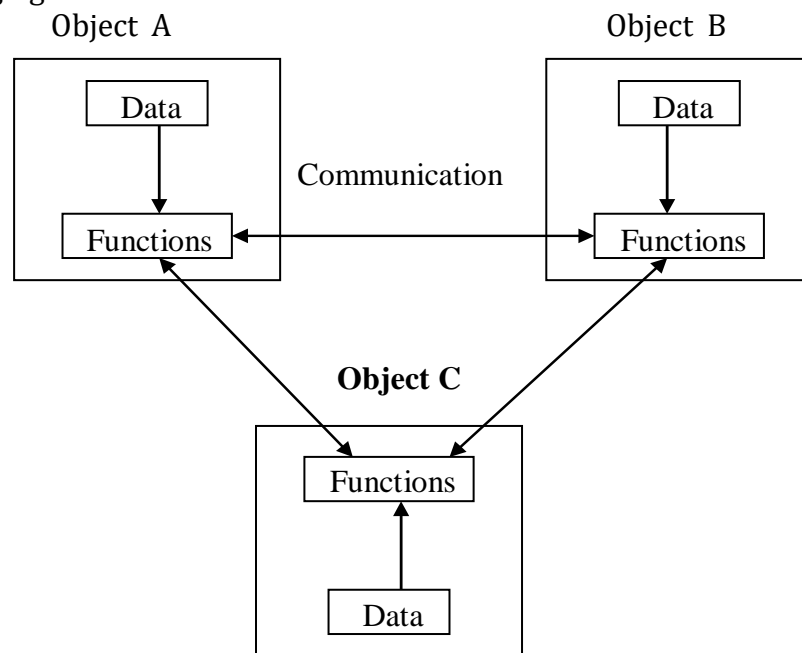


fig: Relationship of data and functions in procedure programming

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 / Object-oriented technology (OOT)**
- The major motivating factor in the invention of object oriented approach is to overcome the drawbacks of procedural oriented approach.
- The principal of Object Oriented Programming Approach is to combine both data and functions into a single unit. such a single unit is called object.
- object oriented programming treats data as critical element in the program development and does not allow it freely to flow around the system or application or program.
- object oriented programming combines both data and functions into a single unit this process is called Encapsulation.
- Ex : languages like C++, JAVA, PYTHON follow this approach.
- The organisation of data and functions in Object oriented programming is shown in the following figure.



Features of the Object Oriented programming

1. In OOP importance is given to data rather than procedure.
2. programs are divided into what are known as objects.
3. New data and functions can be easily added when ever necessary
4. Data of an object can be accessed only by functions associated with that object.
5. Data is hidden and can't be accessed by external functions.
6. Objects may communicate with each other through functions.
7. Follows bottom-up approach in program design.

Structure of C Program

Collection of functions

```

Void main() ← Entry point
{
    call the functions from here
}
  
```

Structure of C++ Program

```

class Example
{
    Collection of functions and variables
};
Void main() ← Entry point
{
    create the object of the class
    using the object invoke the functions in the class
}

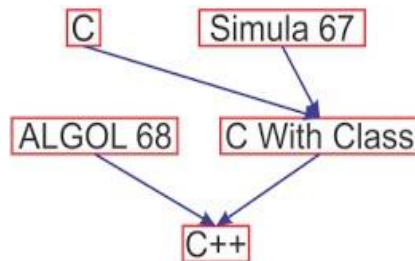
```

Differences between C and C++

	Procedure Oriented Programming	Object Oriented Programming
1	C is Procedural Oriented language.	C++ is an Object-Oriented Programming language.
2	program is divided into small parts called functions .	program is divided into parts called objects .
3	Structure in C does not provide the feature of function declaration.	Structure in C++ provides the feature of declaring a function as a member of the structure.
4	Importance is not given to data but to functions as well as sequence of actions to be done.	Importance is given to the data rather than procedures or functions because it works as a real world .
5	follows Top Down approach .	follows Bottom Up approach .
6	It does not have any access specifier.	OOP 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	To add new data and function in POP is not so easy.	OOP provides an easy way to add new data and function.
9	Most function uses Global data for sharing that can be accessed freely from function to function in the system.	In OOP, data can not move easily from function to function, it can be kept public or private so we can control the access of data.
10	It does not have any proper way for hiding data so it is less secure .	OOP provides Data Hiding so provides more security .
11	Overloading is not possible.	In OOP, overloading is possible in the form of Function Overloading and Operator Overloading.
12	Example of Procedure Oriented Programming are : C , VB , FORTRAN, PASCAL , ALGOL , COBOL , BASIC	Example of Object Oriented Programming are : C++, JAVA, VB.NET, C#.NET , PYTHON

Evolution Of C++ :

- C++ is an object oriented programming language and is considered as an extension of C.
- C++ was developed by Bjarne Stroustrup in 1979, at AT & T Bell labs.
- Initially stroustrup combined the features of C programming language as well as Simula67(a first object oriented language) for developing a most powerful Object oriented programming language it was called "C with classes" as it had all the properties of the C language.



- In 1982, Stroustrup renamed it as "C++" (++ being the increment operator in C)
- C++ introduced the concept of Class and Objects.
- It encapsulates high and low-level language features. So, it is seen as an intermediate level language.

Disadvantages of Conventional Programming

- Traditional Programming languages such as COBOL, FORTRAN and C are known as Conventional programming languages / procedural programming.
- programs written in these languages consists of sequence of instructions and these are executed by compiler or interpreter to perform a given task.
- In conventional Programming when the size of program is large it is difficult to manage the code, to over come this problem conventional programming uses procedures or functions.

Drawbacks :

- Large size programs are divided in to smaller programs known as functions these functions can call one another . Hence security is not provided.
- Another main problem was data security. Importance is not given to protection of data.
- Data passes globally from one function to another function.
- Most functions have access to global data.

Key Concepts of Object Oriented Programming

Key concepts of object oriented programming are :

1. class
 2. object
 3. Data Encapsulation
 4. Data Abstraction
 5. Inheritance
 6. Polymorphism
-

Class : Simply class is extension of 'C' Structure

Structure : Structure is a user defined data type .

It is a collection of heterogeneous variables.

Ex : struct stu

```
{
    int sid, sub[5] ;
    char sname[20];
};
s1, s2, s3, s4,..... s10
```

structure members

← structure variables

Here by default all the structure members are public .

The 'C' structure allows only variables(i.e., structure members) inside the structure.

Functions are not allowed. Structure data is not protected.

C++ Structure : Structure is available in C++ language.

Ex : struct stu

```
{
    int sid, sub[3] ;
    char sname[20];
    member_functions();
};
```

structure members

In C++ structures , member functions are introduced.

In C++ also structure data is public , it is not secured.

To avoid this problem they have introduced a concept called "class" in C++.

- 1. class :** It is a user defined data type, which consists of both data and functions into a single unit.

In OOP data must be represented in the form of a class . A class contains variables for storing data and functions to specify various operations that can be performed and hence it is only logical representation of data and not physical representation.

so class is collection of Data members and Member functions.

class is a blue print of an object.

Once a class has been defined we can create any number of objects.

Each object is associated with the data of type class which they were created.

class provides the concept of Encapsulation.

class provides the concept of Data hiding with private declarations.

Declaration of C++ class :

A class is collection of Data members and member functions which hide from other programs.

The key word 'class' is used to declare a class. Inside the class access specifiers are used to declare variables and methods.

Creating a class : To create class use the keyword class that has the following syntax :

```
[access specifier]   class < class name >
{
    Access specifier :
    variable declarations;
    Access specifier :
    method declarations & implementations
};
```

Example program :

```
#include<iostream>
using namespace std;
class temp
{
    int a;
    public :
    void read()
    {
        a=100;
        cout<<a;
    }
};
int main()
{
    temp obj;
    //obj.a=10;
    obj.read();
    return 0;
}
```

To access the members we have to create object .

NOTE: In c++ the private data should be accessed only by member functions of the same class.

Member functions : The functions which are declared inside the class are called as member functions.

Default access specifier in C++ is private.

so here 'a' is a private member .

private members are not directly accessible from outside the class. (this is called data hiding it is achieved with private declarations.)

main() is outside the class so 'a' is not accessible with object from main

obj.a =10 is an invalid statement.

2. Object :

A class is only logical representation of data. Hence to work with the data represented by the class you must create a variable for the class which is called as an object.

So object is a variable of type class.

Object is the basic unit of object-oriented programming.

In a class to access data members first memory have to be allocated.

No memory is allocated when a class is created.

Memory is allocated only when an object is created, i.e., when an instance of a class is created. Object is physical copy and class is logical copy.

Syntax for creating an object :

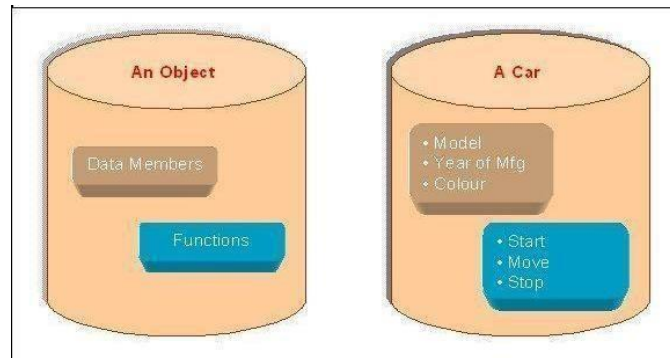
class name followed by variable name

Objects are identified by its unique name.

An object represents a particular instance of a class.

There can be more than one instance of an object.

Each instance of an object can hold its own relevant data.



An Object is a collection of data members and associated member functions also known as methods. Characteristics of an object are represented in a class as **Properties**.

The actions that can be performed by objects becomes functions of the class and is referred to as **Methods**.

For example consider we have a Class of *Cars* under which *Santro Xing*, *Alto* and *WaganR* represents individual Objects. In this context each *Car* Object will have its own, Model, Year of Manufacture, Colour, Top Speed, Engine Power etc., which form **Properties** of the *Car* class and the associated actions i.e., object functions like Start, Move, Stop form the **Methods** of *Car* Class.

3. **Data Encapsulation** : A class can contain variables for storing data and functions to specify various operations that can be performed on data. This process of wrapping up of data and functions that operate on data as a single unit is called as Encapsulation. When using Data Encapsulation, data is not accessed directly, it is only accessible through the functions present inside the class. Data Encapsulation enables the important concept of data hiding possible.
4. **Data Abstraction** : Abstraction refers to the act of representing essential features without including the back ground details or explanations. It represents a functionality of a program without knowing implementation details. It is an approach that speaks about hiding of the complexity and consume only the functionality.
5. **Inheritance**: Creating a new class from an existing class or base class is called Inheritance. The base class is also known as *parent class* or *super class*, The new class that is formed is called *derived class*. Derived class is also known as a *child class* or *sub class*. Inheritance helps in reducing the overall code size of the program, which is an important concept in object-oriented programming.

The concept of inheritance provides the idea of reusability. Instead of rewriting the code you can create the class from the existing class and extending the functionality of existing class. This mean that we can add additional features to an existing class without modifying it. This is possible by designing a new class that will have the combined features of both the classes.

6. **Polymorphism** : Polymorphism comes from the Greek words “poly” and “morphism”. “poly” means many and “morphism” means form i.e.. many forms.

Polymorphism means the ability to take more than one form. Advantage of this is you can make an object behave differently in different situations, so that no need to create different objects for different situations.

Polymorphism can be achieved with the help of Overloading and Overriding concepts and it is classified into compile time polymorphism and Runtime polymorphism.

Function with same name but different arguments .

Functioning is different.

Example : add(int a, int b)

add(float a, float b)

add(int a, int b, int c)

add(float a, float b, float c)

Advantages of Object Oriented Programming:

Object Oriented Programming provides several benefits to both program designer and user

- **Reusability:** In OOP programs, functions and modules that are written by a user can be reused by other users without any modification.
- **Inheritance:** Through this we can eliminate redundant code and extend the use of existing classes.
- **Data Hiding:** The programmer can hide the data and functions in a class from other classes. It helps the programmer to build the secure programs.
- **Reduced complexity of a problem:** The given problem can be viewed as a collection of different objects. Each object is responsible for a specific task. The problem is solved by interfacing the objects. This technique reduces the complexity of the program design.
- **Easy to Maintain and Upgrade:** OOP makes it easy to maintain and modify existing code as new objects can be created with small differences to existing ones. Software complexity can be easily managed.
- **Message Passing:** The technique of message communication between objects makes the interface with external systems easier.
- **Modifiability:** it is easy to make minor changes in the data representation or the procedures in an OO program. Changes inside a class do not affect any other part of a program, since the only public interface that the external world has to a class is through the use of methods.

Applications Of C++ :

- C++ is a general purpose, case-sensitive, free-form programming language that supports object-oriented, procedural and generic programming.
 - By the help of C++ programming language, we can develop different types of secured and robust applications:
 - Window application
 - Client-Server application
 - Commercial applications(shopping , Banking)
 - Operating systems(DOS, Windows, Linux)
 - Editors(Notepad, word pad)
 - Database(oracle)
 - Device drivers.
 - Compilers, interpreters.
 - Embedded firmware etc
-

UNIT II**[Classes and Objects & Constructors and Destructor]**

Classes and Objects & Constructors and Destructor: Classes in C++-Declaring Objects, Access Specifiers and their Scope, Defining Member Function-Overloading Member Function, Nested class, Constructors and Destructors, Introduction, Constructors and Destructor- Characteristics of Constructor and Destructor, Application with Constructor, Constructor with Arguments (parameterized Constructor, Destructors- Anonymous Objects.

STRUCTURE IN C

In C , it is possible to combine dissimilar data types using structure but it has the following limitations :

- a) Functions are not allowed as members of structure.
- b) By default all the structure members are public and direct access to data members is possible. Hence , security to data or data hiding is not provided.
- c) The struct data type is not treated as built -in type , that is the use of struct keyword is necessary to declare objects.
- d) The data members cannot be initialized inside the structure.

The syntax of structure declaration is as follows :

syntax :

```
struct <struct name> {  
    variable 1;  
    variable 2;  
};
```

Note : Structure declaration should start with a key word struct.

Example:

```
struct student {  
    int sid;  
    char sname[20];  
    float avg;  
};
```

here in this example structure name is student

data members : sid, sname , avg

thus, a custom data type is created by combination of one or more data members.

The object of structure item can be declared as follows :

```
struct student s1 , *s2;
```

The object declaration is same as the declaration of variables of built-in data types.

The object s1 and pointer *s2 are variables of type student, s1,*s2 can access the data members of struct student.

Access to structure members :

The data members of a structure are accessed by using object name and operators such as dot(.) or arrow(->). The dot(.) or arrow(->) operators are known as referencing operators.

The dot operator is used when simple object is declared .

The arrow operator is used when object is pointer to structure.

Accessing of members can be accomplished by using the following syntax :

[object name][operator][member variable name]

When an object is a simple variable, access of members is done as follows:

object name dot(.) member variable name

s1.sid , s1.sname , s1.avg

When an object is a pointer to structure then members are accessed as follows:

object name arrow(->) member variable name

s1->sid

s1->sname

s1->avg

Write a program to access data members of a structure.(Using dot(.) operator for simple variable)

```
#include<stdio.h>
#include<conio.h>
struct student {
    int sid;
    char sname[20];
    float avg;
};
void main() {
    struct student s1 = {1901,"RAMA",72.8};
    printf("\n s1 details With simple variable");
    printf("\n Student id = %d", s1.sid);
    printf("\n Student Name= %s", s1.sname);
    printf("\n Student average = %f", s1.avg);
}
```

Write a program to access data members of a structure.(Using arrow(->) operator for pointer variable)

```
#include<stdio.h>
#include<conio.h>
struct student {
    int sid;
    char sname[20];
    float avg;
}*s2;
void main() {
    struct student s = {1902,"KAMALA",82.6};
    s2=&s;
    printf("\n s2 details With pointer variable");
    printf("\n Student id = %d", s2->sid);
    printf("\n Student Name= %s", s2->sname);
    printf("\n Student average = %f", s2->avg);
}
```

C++ Structure : Structure is available in C++ language.

```
Ex : struct student
{
    int sid, sub[3] ;
    char sname[20];           structure members
    member_functions();
};
```

In C structure function declaration is not allowed.

In C++ structures , member functions are introduced.

In C++ also structure data is public , it is not secured.

To avoid this problem they have introduced a concept called "class" in C++.

CLASSES AND OBJECTS

1. classes in C++ :

In OOP data must be represented in the form of a class .

A class contains variables(data members) for storing data and functions(member functions) to specify various operations that can be performed and hence it is only logical representation of data and not physical representation.

A class is a user defined data type, which consists of both data members and member functions into a single unit.

class is a blue print of an object.

Once a class has been defined we can create any number of objects.

Each object is associated with the data of type class which they were created.

class provides the concept of Encapsulation.

class provides the concept of Data hiding with private declarations.

Declaration of C++ class :

A class is collection of Data members and member functions which hide from other programs.

The key word 'class' is used to declare a class. Inside the class access specifiers are used to declare variables and methods.

Creating a class : To create class use the keyword class that has the following syntax :

```
[access specifier]  class < name of class >
{
    private :
        declaration of variables;
        declaration of functions;
    public :
        declaration of variables;
        declaration of functions;
};
```

example:

```
class student {  
    private:  
        int sid;  
        char sname[20];  
        float avg;  
        void readvalues();  
    public :  
        void printvalues();  
};
```

The declaration of a class is enclosed with curly braces and terminated with a semicolon.

The class body contains declarations of variables and functions and these are called members of class.

The key words private and public are known as labels that are followed by colon(:).

The class members that have been declared as private can be accessed only within the class. The class members that have been declared as public can be accessed outside the class also.

```
#include<iostream>  
using namespace std;  
class student  
{  
    //private:  
    int sid;  
    char sname[20];  
    public:  
    void getdata()  
    {  
        cout<<"Enter STUDENT ID : ";  
        cin>>rollno;  
        cout<<"Enter STUDENT NAME : ";  
        cin>>sname;  
    }  
    void putdata()  
    {  
        cout<<"\n Student ID : "<<sid;  
        cout<<"\n student Name : "<<sname;  
    }  
};  
int main()  
{  
    student s;  
    s.getdata();  
    s.putdata();  
    return 0;  
}
```

Declaring objects :

The declaration of object is same as declaration of variables of basic data types.

Defining objects of class data type is known as class instantiation.

Memory is allocated only when objects are created.

syntax :

class_name variable_name;

Ex: student s;

Example:

- a) int x, y, z; // declaration of integer variables
 x, y, z are variables of type integer.
- b) char a, b, c; //declaration of character variables
 a, b, c are variables of type character.
- c) student s1,s2, *s3; // declaration of object or class type variables
 s1, s2, s3 are three objects of class student . *s3 is a pointer to class student.

Accessing class members

- A object can be declared in the main(), and member functions are declared in class in public section so always a member function can be called by using object.

The object can access the public data member and member functions of a class by using dot(.) and arrow(->) operators.

syntax:

[object name] [operator] [Member name]

Access to class members :

s1.getdata();

accessing data members of a class

where s1 is an object and getdata() is a member function .

The dot(.) operator is used because s1 is a simple object.

s3 -> sid

where s3 is a pointer and sid is a data member .

The arrow(->) operator is used because s3 is a pointer.

- if data member is declared as public, we can access the data member by using object .
- If data member is declared private then it is not accessible with object.

Access specifiers :

Access specifiers are used to identify access rights for the data and member functions of the class. There are three main types of access specifiers in C++ programming language:

- public
 - private
 - protected
-

- **public :** *Public* members are accessible from outside the class. The keyword `public` is used to allow objects to access the member variables of a class directly. The member variables and functions declared as `public` can be accessed directly by the object.

Write a program to declare all members of a class as `public`. Access the elements using object.

```
#include<iostream>
using namespace std;
class item {
    public:    // public section begins
    int codeno;
    float price;
    int qty;
};           // end of class

int main() {
    item one; // object declaration
    one.codeno=123; // member initialization
    one.price=123.45;
    one.qty=150;
    cout<<"\n Codeno = "<<one.codeno;
    cout<<"\n Price = "<<one.price;
    cout<<"\n Quantity = "<<one.qty;
    return 0;
}
```

- **private :**

A *private* member within a class denotes that only members of the same class have accessibility. The *private* member is inaccessible from outside the class.

To prevent member variables and functions from direct access the `private` keyword is used.

```
#include<iostream>
using namespace std;
class item {
    private: // private section begins
    int codeno;
    float price;
    int qty;
};           // end of class

int main() {
    item one; // object declaration
    one.codeno=123; // member initialization
    one.price=123.45;
    one.qty=150;
    cout<<"\n Codeno = "<<one.codeno;
```

```
cout<<"\n Price = "<<one.price;
cout<<"\n Quantity = "<<one.qty;
return 0;
}
```

when the above program is compiled, the compiler will display the error message
codeno, price, qty are private in this scope.

so, the private members are not accessible by the object directly.

Now how to access the private members.

To access private data members a member function must be declared in the class in public section.

**** Member functions are used to initialize the data members.**

```
#include<iostream>
using namespace std;
class item {
private: // private section begins
    int codeno;
    float price;
    int qty;

public : // public section starts
    void display(){ // member function
        codeno=123; // member initialization
        price=123.45;
        qty=150;
        cout<<"\n Codeno = "<<codeno;
        cout<<"\n Price = "<<price;
        cout<<"\n Quantity = "<<qty;
    }
}; // end of class

int main() {
    item one; // object declaration
    one.display();
    return 0;
}
```

Note : By default structure members are public(accessible).

By default class members are private(not accessible).

- **protected:** A protected access specifier is a stage between *private* and *public* access. If member functions defined in a class are *protected*, they cannot be accessed from outside the class but can be accessed from the derived class.

We can not access protected section members from outside the class by any object.

protected keyword is used in inheritance concepts.

Defining Member Function :

- Functions declared inside the class are called member functions.
- The member function must be declared inside the class. They can be defined as
 - a) private or public section and
 - b) inside or outside the class
- In C++ the private data should be accessed only with the member functions of same class.
- private data is not visible to outside classes, this is called data hiding.

Access Specifiers	Member function declaration	
	Inside the class	Outside the class
Private	✓	×
Public	✓	✓

Declaration of member function inside the class :

The member functions defined inside the class are treated as inline function.

If the member function is small then it should be defined inside the class .

Other wise it should be defined outside the class.

Member function inside the class can be declared in public or private section.

The following program illustrates the use of a member function inside the class in public section.

Write a program to access private members of a class using member function.

```
#include<iostream>
using namespace std;
class employee {
private: // private section starts
    int eno;
    float esal;
public: // public section starts
    void display() {
        eno=1023; // Access to private members
        esal=15000;
        cout<<"\n Employee Number : "<<eno;
        cout<<"\n Employee salary : "<<esal;
    }
};
int main() {
    employee e1;
    e1.display();
    return 0;
}
```

Explanation : In the above program display() is the member function defined inside the class in public section. We know that object has permission to access the public members of the class. Under main() function object e1 is declared. The public member function can access the private members of the same class. The function display() initializes the private member variables and display the contents on the console.

Defining private member function inside the class:

It is also possible to declare a function in private section like data variables. To execute private member function, it must be invoked by public member function of the same class.

A member function of a class can invoke any other member function of its own class. This method of invoking function is known as nesting of member function.

Write a program to declare private member function and access it using public member function.

```
#include<iostream>
using namespace std;
class employee {
private: //private section starts
    int eno;
    char ename[20];
    float esal;
    void readvalues() { //private member function
        cout<<"Enter Emp No : ";
        cin>>eno;
        cout<<"Enter Emp Name : ";
        cin>>ename;
        cout<<"Enter Emp sal : ";
        cin>>esal;
    }
public: //public section starts
    void display() { //public member function
        readvalues(); //call to private member function
        cout<<"\n Employee Number : "<<eno;
        cout<<"\n Employee Name : "<<ename;
        cout<<"\n Employee salary : "<<esal;
    }
};
int main() {
    employee e1; // object declaration
    //e1.readvalues(); //not accessible
    e1.display(); // call to public member function
    return 0;
}
```

Member function outside the class

If the function is defined outside the class following care must be taken.

- a) The prototype declaration(function declaration) must be done inside the class.
- b) The function name must be preceded by class name and its return type separated by scope access operator. The following is syntax:

```
return type class_name :: function(args...)
{
    ----
    ----
}
```

Write a program to define member function of a class outside the class.

```
#include<iostream>
using namespace std;
class employee {
private:
    int eno;
    char ename[20];
    float esal;

public:
    void display();    // prototype declaration
};
void employee :: display()    // definition outside the class
{
    cout<<"Enter Emp No : ";
    cin>>eno;
    cout<<"Enter Emp Name : ";
    cin>>ename;
    cout<<"Enter Emp sal : ";
    cin>>esal;
    cout<<"\n Employee Number : "<<eno;
    cout<<"\n Employee Name : "<<ename;
    cout<<"\n Employee salary : "<<esal;
}

int main() {
    employee e1; // object declaration
    e1.display(); // call to public member function
    return 0;
}
```

// to demonstrate invoking of member functions inside and outside the class

```
#include<iostream>
using namespace std;
class employee
{
private:
    int eno;
    char ename[20];
    float esal;
public:
    void readvalues() {
        cout<<"Enter Emp No : ";
        cin>>eno;
        cout<<"Enter Emp Name : ";
        cin>>ename;
        cout<<"Enter Emp sal : ";
        cin>>esal;
    }
    void display();
};

void employee :: display() {
    cout<<"\n Employee Number : "<<eno;
    cout<<"\n Employee Name : "<<ename;
    cout<<"\n Employee salary : "<<esal;
}

int main() {
    employee e1;
    e1.readvalues();
    e1.display();
    return 0;
}
```

Overloading Member Function

Overloading is nothing but two or more functions is defined with same name but different parameters

Function overloading can be considered as an example of polymorphism feature in C++.

Function overloading is a compile-time polymorphism.

Rules for Function overloading :

An overloaded function must have :

- Different type of parameters
ex: void test(int a); void test(float a);
- Different number of parameters
ex: void test(); void test(int a);
- Different sequence of parameters
ex: void test(int a , float b); void tes t(float a , int b);

// to demonostrate function over loading

```
#include<iostream>
using namespace std;
class addition {
public:
int add(int a, int b);
int add(int a, int b,int c);
double add(int a, double b);
double add(double a , int b);
};

int addition::add(int a, int b) {
cout<<"Function 1"<<endl;
return (a+b);
}

int addition::add(int a, int b, int c) {
cout<<"Function 2"<<endl;
return (a+b+c);
}

double addition::add(int a, double b) {
cout<<"Function 3"<<endl;
return (a+b);
}

double addition::add(double a,int b) {
cout<<"Function 4"<<endl;
return (a+b);
}
```

```
int main() {
    addition obj;
    cout<<"Addition of two integers : "<<obj.add(1,2)<<endl;
    cout<<"Addition of three integers : "<<obj.add(3,4,5)<<endl;
    cout<<"Addition of integer and double : "<<obj.add(4,5.6)<<endl;
    cout<<"Addition of double and integer : "<<obj.add(7.4,5)<<endl;
    return 0;
}
```

Nested class

A nested class is a class that is declared in another class.

The outside class is called enclosing class and inside class is called nested class.

The nested class is also a member variable of the enclosing class and has the same access rights as the other members.

The member functions of the enclosing class have no special access to the members of a nested class.

Example :

```
#include<iostream>
using namespace std;
class A {
    public:
    class B { // Member variable of class A
        private:
        int a,b;
        public:
        void getdata() { // Member function of class B
            cout<<"Enter values for a & b : ";
            cin>>a>>b;
        }
        void putdata() {
            cout<<"a ="<<a<<endl<<"b ="<<b;
        }
    };
};
int main() {
    cout<<"Nested classes in C++"<< endl;
    A :: B obj;
    obj.getdata();
    obj.putdata();
    return 0;
}
```

Constructors and Destructors :

Constructors and Destructors are special class operations.

Constructor :

If we want to automatically initialize data members i.e., without calling member function we want to initialize data members. It is possible only through Constructors.

A constructor is a special member function.

Constructors are used to initialize the objects of its class and this is called automatic initialization of object.

Constructors construct the values of data members of class i.e., when constructor is used data members are automatically initialized, when the object of the class is created.

Characteristics of constructors :

1. Constructor name and class name should be same.
2. Constructors should be declared in public section.
3. Constructors never have any return value (or) data type including void.
4. The main function of Constructor is to initialize objects and allocate appropriate memory to objects.
5. Constructors can have default values and can be overloaded.
6. Constructor may or may not have arguments (or) parameters.
7. Constructor is executed only once when the object is created.
8. Constructors are automatically invoked they are not invoked with dot(.) operator.
9. Constructor is invoked automatically when object of class is created.

Syntax of defining a constructor in a class

```
class A {  
    public:  
    int x;  
    A() // constructor  
    {  
        // object initialization  
    }  
};
```

Constructors can be defined either inside the class definition or outside class definition using class name and scope resolution :: operator.

```
class A {  
    public:  
    int i;  
    A(); // constructor declared  
};  
A::A() // constructor definition  
{  
    i = 1;  
}
```


APPLICATIONS WITH CONSTRUCTORS

The initialization of member variables of class is carried out using constructors.

The constructor also allocates required memory to the object.

example :

```
class num {
private:
int a,b,c;
public:
num(void); // declaration of constructor
-----
};

num::num(void) { //definition of constructor
a=0; b=0; c=0; // value assignment
}
```

```
main() {
class num x;
}
```

In the above example, class num has three integer variables a, b, and c

In definition, the member variables of a class num are initialized to zero.

When an object is created, its member variables(private and public) are automatically initialized to the given value. By this we understood that with out calling a member function explicitly to initialize member variables The compiler automatically calls the constructor when an object is created and member variables are initialized.

The compiler automatically calls the constructors.

In case there is no constructor in the program, the compiler calls a dummy constructor.

Write a program to define a constructor and initialize the class data member variables with constants.

```
#include<iostream>
using namespace std;
class num {
private:
int a,b,c;
public:
int x;
num(void); //declaration of constructor
void show()
{
cout<<"\n x = " <<x<<" a= "<<a<<" b= "<<b<<" c= "<<c;
}
};
```

```
num::num(void) //definition of constructor
{
    cout<<"\n Constructor called";
    x=5; a=0; b=1; c=2;
}
main()
{
    num x;
    x.show();
    return 0;
}
```

C++ supports three types of Constructors

1. Default Constructor.
2. Parameterized Constructor.
3. Copy Constructor.

1. Default Constructor :

Constructor with out arguments are called Default Constructor.

There are two types Default Constructors

One is System written which is created by the compiler at compile time(When the class is not having a Constructor.

second one, we have to define that is called user defined Constructor.

Write a program to read values through the key board. Use Constructor.

```
#include<iostream>
using namespace std;
class num {
private:
    int a,b,c;
public:
    num(void); //declaration of constructor
    void show() {
        cout<<"\n"<<"a= "<<a<<" b= "<<b<<" c= "<<c;
    }
};

num::num(void) { //definition of constructor
    cout<<"\n Constructor called";
    cout<<"\n Enter values for a,b and c :";
    cin>>a>>b>>c;
}
```

```
main() {  
    num x;  
    x.show();  
    return 0;  
}
```

Program to demonstrate user defined (or) Default constructor.

```
#include<iostream>  
using namespace std;  
class test {  
private:  
    int a,b;  
public:  
    test() { //default constructor  
        cout<<"Constructor called"<<endl;  
        a=10;  
        b=20;  
    }  

```

```
    void show() {  
        cout<<"\n a= "<<a<<endl;  
        cout<<" b= "<<b<<endl;  
    }  
};  
main() {  
    test t;  
    t.show();  
    return 0;  
}
```

2. Constructor with arguments:

A constructor initialize the member variables with given values, It is also possible to create a constructor with arguments and such constructors are called Parameterized Constructors.

```
#include<iostream>  
using namespace std;  
class test {  
private:  
    int a,b,c;  
public:  
    test(int x, int y, int z) { //parameterized constructor  
        a=x;  
        b=y;  
        c=z;  
    }  

```

```

void show() {
    cout<<"\n a= "<<a<<endl;
    cout<<" b= "<<b<<endl;
    cout<<"c= "<<c<<endl;
}
};

main() {
    test t1(30,20,10);
    t1.show();
    return 0;
}

```

Program : Constructor with default arguments

```

#include<iostream>
using namespace std;
class test {
private:
    int a,b,c;
public:
    test(int x=10, int y=20, int z=30) { //parameterized constructor
        a=x;
        b=y;
        c=z;
    }
}

```

```

void show() {
    cout<<" a= "<<a<<endl;
    cout<<" b= "<<b<<endl;
    cout<<"c= "<<c<<endl<<endl;
}
};

```

```

int main() {
    test t1(1,2,3);
    test t2(1,2);
    test t3(1);
    test t4;
    t1.show();
    t2.show();

    t3.show();
    t4.show();
    return 0;
}

```

t1	t2
a = 1	a = 1
b = 2	b = 2
c = 3	c = 30
t3	t4
a = 1	a = 1
b = 20	b = 2
c = 30	c = 3

Missing arguments are replaced with default values.

3. Copy constructor:

The process of copying one object data in to another object is called as Copy Constructor.

A copy constructor is a constructor which is used to initialize the current values with another object value.

Copy constructor is used to create another copy or xerox of one object.

Copy constructors are having reference type parameters.

Copy constructors are having class type parameters means object.

Copy constructors receives another object to initialize current object.

```
#include<iostream>
using namespace std;
class sample {
int a,b;
public:
sample(int x,int y) // parameterized constructor
{
    a=x;
    b=y;
}

sample(sample &obj) // copy constructor
{
    a=obj.a;
    b=obj.b;
}

void display( ) {
cout<<"a= "<<a<<endl;
cout<<"b= "<<b<<endl;
}
};

int main( ) {
    sample s1(10,20);
    sample s2(s1);
    cout<<"Parameterized constructor"<<endl;
    s1.display();
    cout<<"copy constructor"<<endl;
    s2.display();
    return 0;
}
```

Destructors :

- When ever constructor is executed object is defined and memory is allocated.
- Now to release (or) delete this memory we use destructor.
- Destructor is also one special member function, which is used to delete the memory created by the constructor.
- Constuctor constructs the memory (or) Constructor constructs the object data.
- Destructor is used to delete this memory created by constructor.
- Using destructors is a good practice because it automatically release the memory occupied by the constructor.

Rules:

1. Destructor name should be similar to class name and preceded by 'tilde' operator(~).

```
ex: class sample {
        sample() //constructor
        {
            -----
            -----
        }

        ~sample() //Destructor
        {
            -----
            -----
        }
    };
```

2. Destructor should be declared in public section.
3. Destructor doesn't have any arguments, so overloading is not possible.
4. Destructor never returns a value, it makes a implicit call new and delete operator.
5. Destructor necer called with object followed by dot(.) operator.
6. Destructor never participate in inheritance.

Program to demonstrate destructor

```
#include<iostream>
using namespace std;
class sample {
int a,b;
public:
sample() { //constructor
a=10;
b=20;
}

~sample() { //destructor
cout<<"\n a = "<<a<<endl;
cout<<" b = "<<b<<endl;
}
};
```

```
main() {  
    sample s;  
    return 0;  
}
```

Note: Destructor is invoked when object is delete. Before going to delete data it prints the data.

Anonymous Objects:

Objects are created with names. It is possible to declare objects without name.

Such objects are called anonymous objects.

With out specifying object name we can initialize and destroy the members of the class.

We can also assume that the operations are carried out using an anonymous object , which exists but is hidden.

write a program to create anonymous object initialize and display the contents of member variables.

```
#include<iostream>  
using namespace std;  
class anonymous {  
private:  
    int rollno;  
    float marks;  
public:  
    anonymous() {  
        rollno=1201;  
        marks=76.8;  
        display();  
    }  
    ~anonymous() {  
        cout<<" In destructor";  
    }  
    void display() {  
        cout<<"Roll no = "<<rollno<<endl;  
        cout<<"Marks = "<<marks<<endl;  
    }  
};  
  
int main() {  
    anonymous();  
    return 0;  
}
```

UNIT III**[Operator Overloading and Type Conversion & Inheritance]**

The Keyword Operator, Overloading Unary Operator, Operator Return Type, Overloading Assignment Operator (=), Rules for Overloading Operators, Inheritance, Reusability, Types of Inheritance, Virtual Base Classes, Object as a Class Member, Abstract Classes, Advantages of Inheritance-Disadvantages of Inheritance.

FRIEND FUNCTIONS: The private members cannot be accessed from outside the class. i.e., a non member function cannot have an access to the private data of a class. In C++ a non member function can access private members by making the function friendly to a class.

A friend function is not a member of any class, but it is able to access the private members of those where it is introduced as a friend.

when a data is declared as private inside a class, then it is not accessible from outside the class. A friend function is used for accessing the non-public members of a class. A class can allow non-member functions and other classes to access its own private data, by making them friends. Thus, a friend function is an ordinary function or a member of another class.

Definition:

A friend function is a function which is declared within a class and is defined outside the class. It does not require any scope resolution operator for defining . It can access private members of a class. It is declared by using keyword "friend".

How to declare Friend Function in C++:

Friend function declaration should be inside the class using the Friend key word.

Syntax: friend ret_type func_name(arguments);

Definition of friend function is specified outside the class .

Function definition must not use keyword friend.

Friend function characteristics:

It is not in scope of class.

–It cannot be called using object of that class.

–It can be invoked like a normal function.

–It should use a dot operator for accessing members that is inside the friend function to access the object data member we have to use obj name(dot operator).

It can be public or private.

–It has objects as arguments.

Friend Functions are majorly used in operator overloading.

Example to understand the friend function:

```
#include<iostream>
```

```
using namespace std;
```

```
class sample {
```

```
int a,b;
```

```
public :
```

```
void get() {
```

```
a=100;
```

```
b=200;
```

```
}
```

```
friend int compute(sample s1); //Friend Function Declaration with keyword friend and with
the object of class sample to which it is friend passed to it
};
```

```
int compute(sample s1) { //Friend Function Definition which has access to private data
return int(s1.a+s1.b)-5;
}
```

```
int main() {
sample s ;
s.get();
cout<<"The result is:"<<compute(s)<<endl; //Calling of Friend Function with object as
argument.
return 0;
}
```

The output of the above program is

The result is:295

The function compute() is a non-member function of the class sample.

In order to make this function have access to the private data a and b of class sample, it is created as a friend function for the class sample. As a first step, the function compute() is declared as friend in the class sample as:

```
friend int compute (sample s1)
```

The keyword friend is placed before the function. The function definition is written as a normal function and thus, the function has access to the private data a and b of the class sample. It is declared as friend inside the class,

In function compute the private data values a and b are added, 5 is subtracted from the result, giving 295 as the result. This is returned by the function and thus the output is displayed as shown .

// Program to access private data using non-member function, by using Friend function.

```
#include<iostream>
using namespace std;
class account {
char name[20];
int acno;
float bal;
public :
void read() {
cout<<"\n Enter Name : ";
cin>>name;
cout<<"\n Enter Account No : ";
cin>>acno;
cout<<"\n Enter Balance : ";
cin>>bal;
}
```

```
friend void showbal(account); //Friend function declaration
};
void showbal(account a) //Friend Function Definition which has access to private data
{
    cout<<"\n Balance of A/c no."<<a.acno<<" is Rs." <<a.bal<<endl;
}
int main() {
    account ac ;
    ac.read();
    showbal(ac);
    return 0; //Calling of Friend Function with object as argument.
}
```

Explanation : In the above program class account is declared. It has three member variables name,acno and bal and one member function read() . Also, inside the class account , showbal() is a function, which is declared as friend of the class account. Once the outside function is declared as friend to any class it gets an authority to access the private data of that class. The function read() reads the data through the keyboard. The friend function showbal() display the balance and accno. inside the friend function to access the object data member we have to use obj name(dot operator)

write a program to find max of two numbers using friend function for two different classes.

```
#include<iostream>
using namespace std;
class sample2;
class sample1 {
    int x;
    public:
    void getx(){
        cout<<"Enter X : ";
        cin>>x;
    }
    friend void max(sample1 s1,sample2 s2); // max is a friend function with 2 class type objects.
};
class sample2 {
    int y;
    public:
    void gety() {
        cout<<"Enter Y : ";
        cin>>y;
    }
    friend void max(sample1 s1,sample2 s2);
};
void max(sample1 s1,sample2 s2) {
    if(s1.x>s2.y)
        cout<<"Data member of class sample1 is larger"<<endl;
```

```
else if(s2.y>s1.x)
cout<<"Data member of class sample2 is larger"<<endl;
else cout<<"both are equal"<<endl;
}
```

```
int main() {
sample1 obj1;
sample2 obj2;
obj1.getx();
obj2.gety();
max(obj1,obj2);
return 0;
}
```

//program to declare friend function in two classes. Calculate the sum of integers of both the classes using friend sum() function.

When a function is friend of more than one class then forward declaration of class is needed so we are declaring class first;

```
#include<iostream>
using namespace std;
class first;
class second {
int s;
public:
void getvalue(){
cout<<"Enter S : ";
cin>>s;
}
friend void sum(second , first);
};
class first {
int f;
public:
void getvalue(){
cout<<"Enter F : ";
cin>>f;
}
friend void sum(second , first);
};
void sum(second d, first t) {
cout<<"Sum of two numbers : "<<t.f+d.s<<endl;
}
int main() {
first a;
```

```
second b;  
a.getvalue();  
    etvalue();  
sum(b,a);  
return 0;  
}
```

Explanation : In the above program, two classes first and second are declared with one integer and one member function in each . The member function getvalue() of both classes reads integers through keyboard. In both the classes the function sum() is declared as friend. Hence this function has an access to the members of both the classes. Using sum() function addition of two integers is calculated and displayed.

Friend Class

It is possible to declare one or more functions as friend functions or an entire class can also be declared as friend class. A class can also be declared to be the friend of some other class. When we create a friend class then all the member functions of the friend class also become the friend of the other class. This requires the condition that the friend becoming class must be first declared or defined (forward declaration).

// program to demonstrate friend classes.

//declare two friend classes and access the private data.

```
#include <iostream>  
using namespace std;  
class sample_1 {  
int a,b;  
public:  
friend class sample_2; //declaring friend class  
void getdata_1() {  
cout<<"Enter A & B values in class sample_1: ";  
cin>>a>>b;  
}  
void display_1() {  
cout<<"A="<<a<<endl;  
cout<<"B="<<b<<endl;  
}  
};  
class sample_2 {  
//public:  
int c,d,sum;  
sample_1 obj1;  
public:  
void getdata_2() {  
obj1.getdata_1();  
obj1.display_1();  
cout<<"Enter C & D values in class sample_2 : ";  
cin>>c>>d; }  
};
```

```
void sum_2() {
sum=obj1.a+obj1.b+c+d;
}

void display_2() {
cout<<"A="<<obj1.a<<endl;
cout<<"B="<<obj1.b<<endl;
cout<<"C="<<c<<endl;
cout<<"D="<<d<<endl;
cout<<"SUM="<<sum<<endl;
}
};

int main() {
//sample_1 s1;
//s1.getdata_1();
//s1.display_1();
sample_2 s2;
s2.getdata_2();
s2.sum_2();
s2.display_2();
}
```

Container class : The primary class is called container class.

Contained class : The class which is declared as friend is called contained class.

In the above program sample_1 is container class.

sample_2 has object of sample_1 , so sample_2 is contained class.

Declaring the object of one class in another class is called composition relation.

Composition allows the concept of reusability, i.e., one class data is reused in another class.

RESTRICTIONS ON FRIEND CLASSES

- Friend functions and classes are not inherited

- Friend function cannot have storage-class specifier i.e. cannot be declared as static or extern

- Friend classes are not corresponded i.e. if class A is a friend of B it does not imply that class B is a friend of A

- Friend classes are not transitive: i.e. friend of a friend is not considered to be a friend unless explicitly specified

- for ex :If class A declares class B as a friend, and class B declares class C as a friend, class C doesn't necessarily have any special access rights to class A.

OPERATOR OVERLOADING

A symbol that is used to perform an operation is called an operator. It is used to perform an operation with constants and variables. A programmer cannot build an expression without an operator. The compiler knows how to perform various operations using operators for the built-in types, however, for the objects those are instance of the class, the operation routine must be defined by the programmer.

Keyword operator

The keyword operator defines a new operation (or) action to the operator.

In C++, we can change the way operators work for user-defined types like objects and structures. This is known as operator overloading.

The process of overloading involves following steps:

1. Create a class which is to be used with overloading operation.
2. Declare the operator function **operator symbol()** in the public part of the class.
3. It may be a member function or a friend function.
4. Here symbol is the operator to be overloaded.
5. Define the operator function to implement the required operations.

syntax:

```
return-type operator operatorsymbol(parameter list)
{
    statement 1;
    statement 2;
    .
    .
    statement n;
}
```

Here,

returnType is the return type of the function.

operator is a keyword.

operatorsymbol is the operator we want to overload. Like: +, <, -, ++, etc.

parameters is the arguments passed to the function.

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

Note: We cannot use operator overloading for fundamental data types like int, float, char and so on.

Example :

```
void operator ++();
void operator --();
void operator +();
void operator -();
void operator ++(num);
void operator -(num,num);
```

Concept of Operator Overloading

One of the unique features of C++ is Operator Overloading. Applying overloading to operators means, same operator is responding in different manner. For example operator + can be used as concatenate operator as well as addition operator.

That is 2+3 means 5 (addition), where as "2"+"3" means 23 (concatenation).

Performing many actions with a single operator is operator overloading. We can assign a user defined function to an operator. We can change function of an operator, but it is not recommended to change the actual functions of operator. We can't create new operators using this operator overloading.

The keyword operator followed by an operator symbol defines a new action of the given operator.

- Operator function should be either member function or friend function.
- A friend function requires one parameter for unary operation and two parameters for binary operation.
- The member function requires no parameter for unary operation and one parameter for binary operation.

When a member function is called the calling object is passed implicitly to that function and it is available for that member function.

//Write a program to perform addition of two objects by using key word operator.

```
#include<iostream>
using namespace std;
class number {
public:
int x,y;
number() //zero argument constructor
{
}
number(int j, int k) //two argument constructor
{
x=j;
y=k;
}
number operator +(number D) {
number T;
T.x=x+D.x;
T.y=y+D.y;
return T;
}
void show() {
cout<<"\n X= "<<x<<" Y= "<<y;}
};
```

```
int main() {  
    number A(2,3),B(4,5), C;  
    A.show();  
    B.show();  
    C=A+B;  
    C.show();  
}
```

Explanation :

In the above program, A,B,C are objects of class number. Using constructor, objects are initialized.

here, we performed addition using stmt C=A+B.

Whenever the stmt C=A+B is executed, the compiler searches for definition of operator +() function. The object A invokes the operator function and object B is passed as argument. The copy of object B is stored in the formal argument D. The member variables of A are directly available in operator function as the function is invoked by the same object. The addition of individual members are carried out and stored in member variable of object T. The return type of operator function is same as that of its class. The function returns object T and it is assigned to variable C.

//Program to perform subtraction of two objects using operator keyword.

```
#include<iostream>  
using namespace std;  
class number {  
public:  
    int x,y;  
    number() //zero argument constructor  
    {  
    }  
    number(int i, int j) //two argument constructor  
    {  
        x=i;  
        y=j;  
    }  
    number operator -(number D) {  
        number T;  
        T.x=x-D.x;  
        T.y=y-D.y;  
        return T;  
    }  
    void show() {  
        cout<<"\n X= "<<x;  
        cout<<"\n Y= "<<y;  
    }  
};
```

```

int main() {
    number A(2,3),B(4,5), C;
    A.show();
    B.show();
    C=A+B;
    C.show();
}

```

➤ **Rules for operator overloading**

- Only existing operators can be overloaded.
- We cannot change basic meaning of an operator.
- Overloaded operator must follow minimum characteristics that of original operator.
- Precedence of an operator cannot be changed
- Associativity of an operator cannot be changed
- No new operators can be created.
- No overloading operators for built-in types

➤ Following operators can't be overloaded:

- Class member access operators (.,*)
- Scope resolution operator (::)
- Size operator (sizeof)
- Conditional operator (?:)

All other operators can be overload

Operators that can be overloaded							
+	-	*	/	%	^	&	
~	!	=	<	>	+=	-=	*=
/=	%=	^=	&=	=	<<	>>	>>=
<<=	=	!=	<=	>=	&&		++
--	->*	,	->	[]	()	new	delete
new[]	delete[]						

Overloading Unary operator:

An unary operator means, an operator which works on single operand. For example, ++ is an unary operator, it takes single operand (c++). So, when overloading an unary operator, it takes no argument (because object itself is considered as argument).

Syntax for Unary Operator (Inside a class)

```
return-type operator operatorsymbol()  
{  
  //body of the function  
}
```

Syntax for Unary Operator (Outside a class)

```
return-type classname::operator operatorsymbol()  
{  
  //body of the function  
}
```

Example 1:-

```
void operator++()  
{  
  counter++;  
}
```

Example 2:-

```
void complex::operator-()  
{  
  real=-real;  
  img=-img;  
}
```

As the name implies it operates on only one operand.

Ex: (++) Increment operator, (--) Decrement Operator are called Unary operators.

The unary operators can be used as prefix (or) postfix notation.

The unary operators have only one operand.

- Consider unary ++ operator(it increments the operand)
- The unary ++ when applied to an object should increment each of its data items.

The unary operators can be used as prefix (or) postfix notation.

The unary operators have only one operand.

// program to increment member variables of an object by using unary ++ operator.

```
#include<iostream>  
using namespace std;  
class number {  
  int a,b,c,d;
```

```
public:
number(int j, int k, int m, int l) {
    a=j;
    b=k;
    c=m;
    d=l;
}
void show(void);
void operator ++();
};
void number::operator ++() {
    ++a; ++b; ++c; ++d;
}
void number::show() {
    cout<<"\n A= "<<a<<" B= "<<b<<" C= "<<c<<" D= "<<d<<endl;
}
int main() {
    number X(3,2,5,7);
    cout<<"\n Before Increment of X : ";
    X.show();
    ++X;
    cout<<"\n After Increment of X : ";
    X.show();
    return 0;
}
```

Explanation : In the above prog the class num contains four integer variables a,b,c, and d.

The class also has two member functions show() and operator++() and one parameterized constructor. The constructor is used to initialize object. The show() displays the contents of the member variables. The operator ++() overloads the unary operator ++. when this operator is used with integer or float variables, its value is increased by one. In this function, ++ operator precedes each member variable of class. This operation increments the value of each variable by one.

In function main() ,the stmt ++X calls the function operator ++(), where X is an object of the class num. The function can also be called using stmt X. operator ++().

In the output, values of member variables before and after increment operations are displayed.

OPERATOR RETURN TYPE

We declared operator() of void types, that is , it will not return any value. However, it is possible to return a value and assign it to other object of the same type.

The return type of operator is always of class type, because the operator overloading is only for objects.

An operator cannot be overloaded for basic data type. Hence if operator returns any value, it will be always of class type.

// Write a program to return values from operator() function.

```
#include<iostream>
using namespace std;
class plusplus {
int num;
public:
plusplus() {
num=0;
}
int getnum() {
return num;
}
plusplus operator ++(int) {
plusplus tmp;
num=num+1;
tmp.num=num;
return tmp;
}
};

int main() {
plusplus p1,p2;
cout<<"\n p1 = "<<p1.getnum();
cout<<"\n p2 = "<<p2.getnum();
p1=p2++;
cout<<"\n p1 = "<<p1.getnum();
cout<<"\n p2 = "<<p2.getnum();
p1++;
cout<<endl<<" p1 = "<<p1.getnum();
cout<<endl<<" p2 = "<<p2.getnum();
return 0;
}
```

Explanation: In this program class plusplus is declared with one private integer num. The class constructor initializes the object with zero. The member function getnum() returns current increment of the objects. p1 and p2 are objects of the class plusplus.

Overloading unary operator using Friend Function:

While overloading an unary operator using friend function argument list must have 1 argument as reference to the object.

// Write a C++ program to overload unary operator ++ by using friend function

```
#include<iostream>
using namespace std;
class number {
int a,b,c,d;
public:
    number(){a=b=c=d=0;}
number(int j, int k, int m, int l) {
a=j;
b=k;
c=m;
d=l;
}
void show(void);
friend number operator ++(number);
};
number operator ++(number n) {
n.a=++n.a;
n.b=++n.b;
n.c=++n.c;
n.d=++n.d;
return n;
}
void number::show() {
cout<<"\n A= "<<a<<" B= "<<b<<" C= "<<c<<" D= "<<d<<endl;
}

int main() {
number X(3,-2,5,-7),N;
cout<<"\n Before Increment of X : ";
X.show();
N=++X;
//X.operator++();
cout<<"\n After Increment of X : ";
N.show();
return 0;
}
```

// Write a C++ program to overload unary operator in complex numbers by using friend function

```
#include<iostream>
using namespace std;
class complex {
float real,imag;
public:
    complex(){
        real=imag=0;
    }
    complex(float r, float i) //two argument constructor
    {
        real=r;
        imag=i;
    }
    void show(void){
        cout<<"\n Real= "<<real;
        cout<<"\n Imag= "<<imag<<"i";
    }
    friend complex operator -(complex);
};
complex operator -(complex c) {
    c.real=-c.real;
    c.imag=-c.imag;
    return c;
}
//void number::show() {}

int main()
{
    complex c1(3.2 , 5.7) , c2;
    cout<<"\n Before Negation : ";
    c1.show();
    c2=-c1;
    cout<<"\n After Negation of X : ";
    c2.show();
    return 0;
}
```

Binary Operator Overloading

A binary operator means, an operator which works on two operands. For example, + is an binary operator, it takes single operand (c+d). So, when overloading an binary operator, it takes one argument (one is object itself and other one is passed argument).

A binary operator + can be overloaded to add two objects rather than adding two variables.

While overloading binary operators using member function, the arg-list will contain one parameter.

Syntax for Binary Operator (Inside a class)

```
return-type operator operatorsymbol(argument)
{
    //body of the function
}
```

Syntax for Binary Operator definition (Outside a class)

```
return-type classname::operator operatorsymbol(argument)
{
    //body of the function
}
```

Overloading binary operator using Member function

Write a C++ program to add two complex numbers using operator overloading by a member function.

Answer: Following program is demonstrating operator overloading by using member function.

```
#include<iostream>
using namespace std;
class Complex {
    int num1, num2;
public:
    void input() {
        cout<<"\n Enter Two Complex Numbers : ";
        cin>>num1>>num2;
    }
    void display() {
        cout<<num1<<"+"<<num2<<"i"<<"\n";
    }
    Complex operator+(Complex c2);
};
Complex Complex :: operator+( Complex c2) {
    Complex c;
    c.num1=num1+c2.num1;
    c.num2=num2+c2.num2;
    return(c); }
```

```
int main() {
    Complex c1,c2, sum;    //Created Object of Class Complex i.e c1 and c2
    c1.input(); //reading the values
    c2.input();
    sum = c1+c2; //Addition of object
    cout<<"\n Entered Values : \n";
    cout<<"\t";
    c1.display(); //Displaying user input values
    cout<<"\t";
    c2.display();
    cout<<"\n Addition of Real and Imaginary Numbers : \n";
    cout<<"\t";
    sum.display(); //Displaying the addition of real and imaginary numbers
    return 0;
}
```

Explanation :

In the above program the class name is Complex it has two private data members num1 ,num2 and three member functions , the input() member function reads the values and assign it to num1 and num2. display() member function is used to display the user input values. here we are over loading binary + operator and inside the operator +() function

the stmt sum = c1+c2 invokes the operator function , in this stmt object c2 is assigned to formal parameter c2 of operator function and member variables of c1 are accessed directly.

The object c is used for holding the result of addition, and it is returned to object sum. and the function display() displays the values of c1,c2 and sum.

Overloading binary operator using friend function

Write a C++ program to add two complex numbers using operator overloading by a friend function.

Answer: Following program is demonstrating operator overloading by using member function.

- While overloading a binary operator using friend function, argument list must take 2 arguments.

```
#include<iostream>
using namespace std;
class Complex {
    int num1, num2;
public:
    void input() {
        cout<<"\n Enter Two Complex Numbers : ";
        cin>>num1>>num2;
    }
}
```

```

//Overloading '+' operator using Friend function
friend Complex operator+(Complex c1, Complex c2);
void display() {
    cout<<num1<<"+"<<num2<<"i"<<"\n";
}
};
Complex operator+(Complex c1, Complex c2) {
    Complex c;
    c.num1=c1.num1+c2.num1;
    c.num2=c1.num2+c2.num2;
    return(c);
}
int main() {
    Complex c1,c2, sum;    //Created Object of Class Complex i.e c1 and c2
    c1.input(); //Accepting the values
    c2.input();
    sum = c1+c2; //Addition of object
    cout<<"\n Entered Values : \n";
    cout<<"\t";
    c1.display(); //Displaying user input values
    cout<<"\t";
    c2.display();
    cout<<"\n Addition of Real and Imaginary Numbers : \n";
    cout<<"\t";
    sum.display(); //Displaying the addition of real and imaginary numbers
    return 0;
}

```

Explanation :

- In the above program the class name is Complex it has two private data members num1 ,num2 and three member functions , the input() member function reads the values and assign it to num1 and num2. display() member function is used to display the user input values. here we are over loading binary + operator using friend function so in side the class complex the operator +() function is declared as friend While overloading a binary operator using friend function, argument list must take 2 arguments , friend function definition must be outside the class.

inside the operator +() function , addition of member variables of two objects is performed and results are assigned to member variables of third object. In this program c1,c2,sum are objects of class complex.

the stmt sum = c1+c2 invokes the operator function , in this stmt object c1 is assigned to formal parameter c1 ,object c2 is assigned to formal parameter c2 of operator function to access the member variables of class.

The object c is used for holding the result of addition, and it is returned to object sum. and the function display() displays the values of c1,c2 and sum.

Overloading assignment operator:

Assignment operator is used to assign values from right side to left side.

Assignment operator is denoted by equal "=" symbol.

Using Assignment operator data members of one object can be assigned to another object.

The **assignment operator** (operator=) is used to copy values from one object to another *already existing object*.

Assignment operator can be overloaded in two ways :

1. Implicit overloading.
2. Explicit overloading.

//Program to overload the assignment operator(=) implicitly

```
#include<iostream>
```

```
using namespace std;
```

```
class num {
```

```
    int x;
```

```
    public:
```

```
        num(int a);
```

```
        void show();
```

```
};
```

```
num::num(int a) {
```

```
    x=a;
```

```
}
```

```
void num :: show(){
```

```
    cout<<x<<" ";
```

```
}
```

```
int main() {
```

```
    num a1(2) , a2(8);
```

```
    cout<<"\n Before overloading assignment operator:";
```

```
        cout<<"\n A = ";
```

```
        a1.show();
```

```
        cout<<"\n B = ";
```

```
        a2.show();
```

```
        a2=a1;
```

```
        cout<<"\n After overloading assignment operator implicitly(b=a):";
```

```
        cout<<"\n A = ";
```

```
        a1.show();
```

```
        cout<<"\n B = ";
```

```
        a2.show();
```

```
        return 0;
```

```
}
```

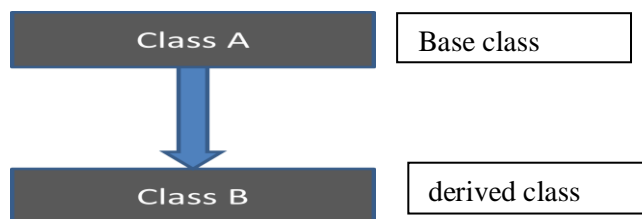
```
//Program to overload the assignment operator(=) Explicitly
#include<iostream>
using namespace std;
class num {
    int x;
public:
    num(int a);
    void show();
    void operator =(num b);
};
num::num(int a) {
    x=a;
}
void num :: show() {
    cout<<x<<" ";
}
void num :: operator =(num b) {
    x=b.x;
}
int main() {
    num a1(100) , a2(200);
    cout<<"\n Before overloading assignment operator:";
    cout<<"\n A = ";
    a1.show();
    cout<<"\n B = ";
    a2.show();
    a2.operator =(a1);
    cout<<"\n After overloading assignment operator Explicitly :";
    cout<<"\n A = ";
    a1.show();
    cout<<"\n B = ";
    a2.show();
    return 0;
}
```

INHERITANCE

Inheritance is one of the most important and useful characteristics of OOP. Literally inheritance means adopting features by newly created thing from the existing one.

The process of obtaining data members and methods from one class to another class i.e., one class to acquire properties and characteristics from another class is known as inheritance.

In OOP inheritance can be defined as the as the process of creating a new class from one or more existing classes. The existing class is known as base class or parent class or super class where as newly created class is known as derived class or child class or sub class. Inheritance provides a significant advantage in terms of code reusability. Consider the diagram shown below



In the above ,a new class B is derived from an existing class A using the feature inheritance. Due to this the derived class B inherits the features of base class A. The derived classes has all the features of the base class and the programmer can choose to add new features specific to the newly created derived class.

However, the derived class can also have its own features.

Inheritance makes the code reusable. Now the term reusability means, When we inherit an existing class, all its methods and fields become available in the new class, hence code is reused. that is the derived class object can access the base class members also. However, the reverse of this is not true I,e the base class object can not access the derived class members as the base class is not aware of derivation of derived class.

(All the members of base class are inherited except the private members.)

Base class and Derived class

The existing class from which the derived class gets inherited is known as the base class. It acts as a parent for its child class and all its properties I,e public and protected members get inherited to its derived class.

A derived class can be defined by specifying its relationship with the base class in addition to its own details i.e., members

The general form of defining a derived class is

```
class derived-class-name: access-specifier base-class-name
{
//members of the derived class
};
```

The access specifier determines how elements of the base class are inherited by the derived class. (or)access specifier define how can the derived class access the Base class members .

Here access is one of the three keywords: public, private and protected.

Base class accesibility is dependent on the visibility mode of derived class.

Visibility modes can be classified into three categories



Modes of inheritance :

1. **Public Mode:** If we derive a sub class in public mode. Then the public member of the base class will become public in the derived class and the protected members of the base class will become protected in derived class.
2. **Protected Mode:** If we derive a sub class in protected mode. Then both public member and protected members of the base class will become protected in derived class.
3. **Private Mode:** If We derive a sub class in private mode. Then both public member and protected members of the base class will become private in derived class

In all cases, private members of a base class remain private to that base class.

It is important to understand that if the access specifier is private, public members of the base become private members of the derived class. If access specifier is not present, it is private by default.

List out visibility of inherited members in various categories of inheritance.

Base class visibility	Derived class visibility		
	Public derivation	Private derivation	Protected derivation
Private	Not inherited	Not inherited	Not inherited
Protected	Protected	Private	Protected
Public	Public	Private	Protected

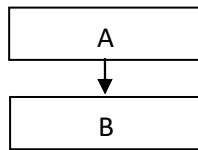
Explain protected access modifier for class members.

Protected:

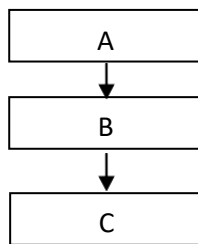
- This access modifier plays a key role in inheritance.
- Protected members of the class can be accessed within the class and from derived class but cannot be accessed from any other class or program.
- It works like public for derived class and private for other programs

Explain types of inheritance with example.

- Inheritance is the process, by which class can acquire the properties and methods of another class.
- The mechanism of deriving a new class from an old class is called inheritance.
- The new class is called derived class and old class is called base class.
- The derived class may have all the features of the base class and the programmer can add new features to the derived class.

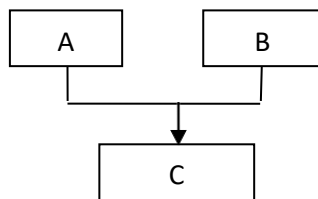
Types of Inheritance:**Single Inheritance**

- If a class is derived from a single class then it is called single inheritance.
- Class *B* is derived from class *A*

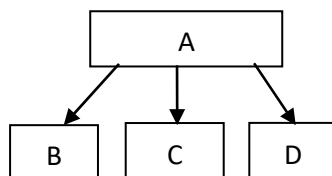
Multilevel Inheritance

- A class is derived from a class which is derived from another class then it is called multilevel inheritance
- Here, class *C* is derived from class *B* and class *B* is

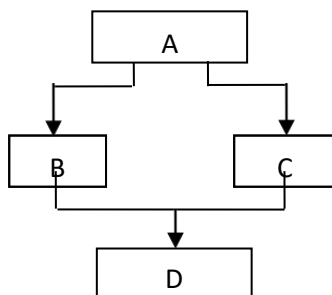
derived from class *A*, so it is called multilevel inheritance.

Multiple Inheritance

- If a class is derived from more than one class then it is called *multiple inheritance*.
- Here, class *C* is derived from two classes, class *A* and class *B*.

Hierarchical Inheritance

- If one or more classes are derived from one class then it is called hierarchical inheritance.
- Here, class *B*, class *C* and class *D* are derived from class *A*.

Hybrid Inheritance

- It is a combination of any above inheritance types. That is either multiple or multilevel or hierarchical or any other combination.
 - Here, class *B* and class *C* are derived from class *A* and class *D* is derived from class *B* and class *C*.
 - class *A*, class *B* and class *C* is example of Hierarchical Inheritance and class *B*, class *C* and class *D* is example of Multiple Inheritance so this hybrid inheritance is combination of Hierarchical and Multiple Inheritance.
-

Single inheritance: One derived class inherits from only one base class. The process in which a derived class inherits traits from only one base class, is called single inheritance. In single inheritance, there is only one base class and one derived class. The derived class inherits the behavior and attributes of the base class. The derived class can add its own properties, ie data members(variables) and functions. It can extend or use properties of the base class with out any modification to the base class.

Syntax:

```
class base_class
{
};
class derived_class:visibility-mode base_class
{
};
```

Program to illustrate the concept of single inheritance

```
#include<iostream>
using namespace std;
class person
{
protected:
char name[15];
int age;
};
class physique : public person
{
float height;
float weight;
public:
void getdata() {
cout<<"Enter Name and Age :";
cin>>name>>age;
cout<<"Enter height and weight :";
cin>>height>>weight;
}
void show()
{
cout<<"\n Name: "<<name<<"\n";
cout<<" Age: "<<age<<" years"<<endl;
cout<<" Height: "<<height<<" Feet"<<endl;
cout<<" Weight: "<<weight<<" kg."<<endl;;
}
};

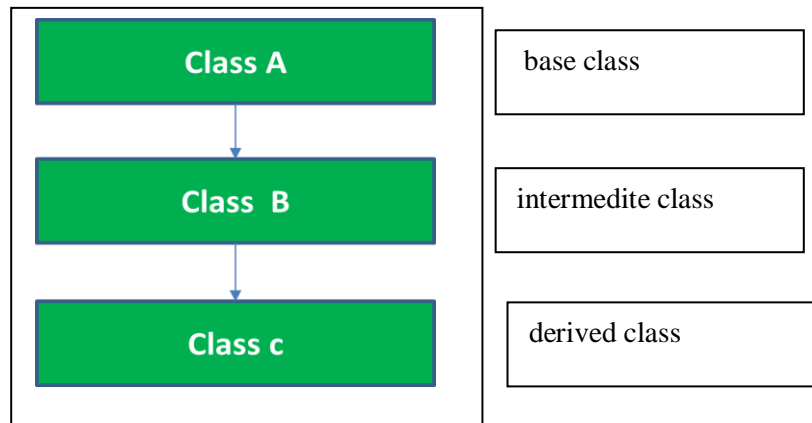
int main() {
physique p;
p.getdata();
p.show();
return 0; }
```

Multilevel inheritance

In this type of inheritance a single derived class may inherit from two or more than two base classes. The process in which a derived class inherits traits from another derived class, is called multilevel inheritance.

syntax:

```
class A
{
.....
};
class B:public A
{
.....
};
class C:public B
{
.....
};
```



Here, class B is derived from class A. Due to this inheritance, class B adopts the features of base class A. Additionally, class B also contains its own features, Further, a new class, class c is derived from class B, Due to this ,class c adopts the features of class B, as well as features of class A.

Program to illustrate the concept of multilevel inheritance

```
#include<iostream>
using namespace std;
class ONE {
protected:
int n1;
};
class TWO: public ONE {
protected:
int n2;
};
class THREE:public TWO {
public:
void input() {
cout<<"Enter two integer numbers :";
cin>>n1>>n2;
}
void sum() {
int sum=0;
sum=n1+n2;
cout<<"Sum of "<<n1<<" + "<<n2<<" = "<<sum;
}
};
int main()      {
THREE t1;
t1.input();
t1.sum();
return 0; }
```


Multiple inheritance

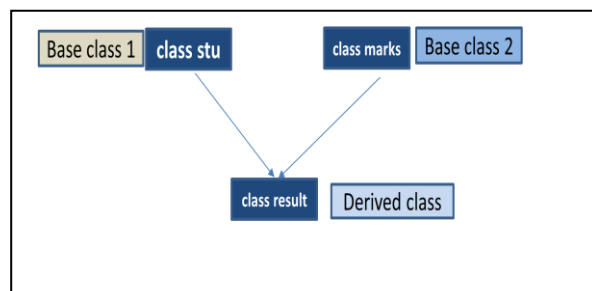
The process in which derived class inherits traits from several base classes, is called multiple inheritance. In multiple inheritance, there is only one derived class and several base classes. we declare the base classes and derived class as given.

Syntax:

```
class base_class1{
};
class base_class2{
};
class derived_class: visibility-mode base_class1,visibility-mode base-class2 {
};
```

Program to illustrate the concept of multiple inheritance

```
#include<iostream>
using namespace std;
class student {
int id;
char name[20];
public:
void getstudent() {
cout<<"Enter Student id and name : ";
cin>>id>>name;
}
void putstudent() {
cout<<"ID = "<<id<<endl;
cout<<"NAME = "<<name<<endl;
}
};
class marks {
protected:
int m1,m2,m3;
public:
void getmarks(){
cout<<"Enter marks of 3 subjects";
cin>>m1>>m2>>m3;
}
void putmarks() {
cout<<"M1= "<<m1<<endl;
cout<<"M2= "<<m2<<endl;
cout<<"M3= "<<m3<<endl;
}
};
class result:public student,public marks {
int total;
float avg;
public:
void show() {
total=m1+m2+m3;
avg=float(total)/3;
cout<<"TOTAL= "<<total<<endl;
cout<<"AVERAGE= "<<avg<<endl; } };
```



```
int main() {
result r;
r.getstudent();
r.getmarks();
r.putstudent();
r.putmarks();
r.show();
return 0;
}
```

Hierarchical Inheritance

In this type of inheritance, more than one sub class is inherited from a single base class, i.e. more than one derived class is created from a single base class.

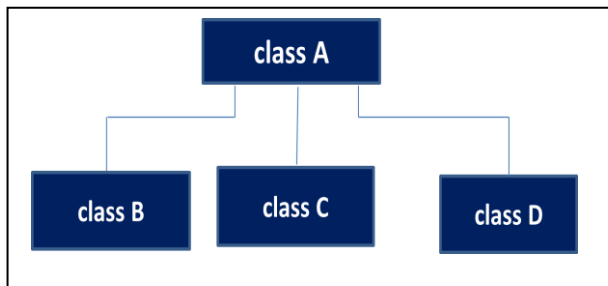
syntax:

```
class base-class-name {  
    data members;  
    member functions;  
};  
class derived-class-name1:visibility mode base-class-name  
{  
    data members;  
    member functions;  
};  
class derived-class-name2:visibility mode base-class-name  
{  
    data members;  
    member functions;  
}
```

Note: visibility mode can be either private, public or protected

Program to illustrate the concept of hierarchical inheritance

```
#include<iostream>  
using namespace std;  
class A {  
public:  
    int x,y;  
    void getdata() {  
        cout<<"\n Enter the value for x and y:\n";  
        cin>>x>>y;  
    }  
};  
class B:public A {  
public:  
    void product() {  
        cout<<"\n product= "<<x*y;  
    }  
};  
class C:public A {  
public:  
    void sum() {  
        cout<<"\n sum= "<<x+y;  
    }  
};  
int main() {  
    B obj1;  
    C obj2;  
    obj1.getdata();  
    obj1.product();  
    obj2.getdata();  
    obj2.sum(); }
```

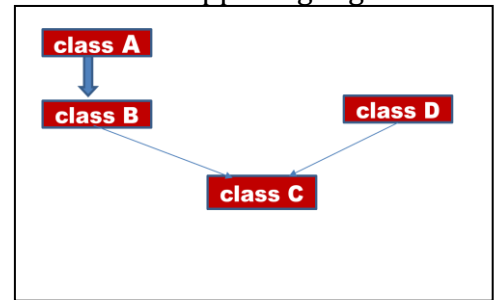


Hybrid inheritance: The combination of one or more type of inheritance happening together is known as hybrid inheritance.

The following diagram explains the concept in better way.

In the diagram, the derivation of class B from class A is single inheritance.

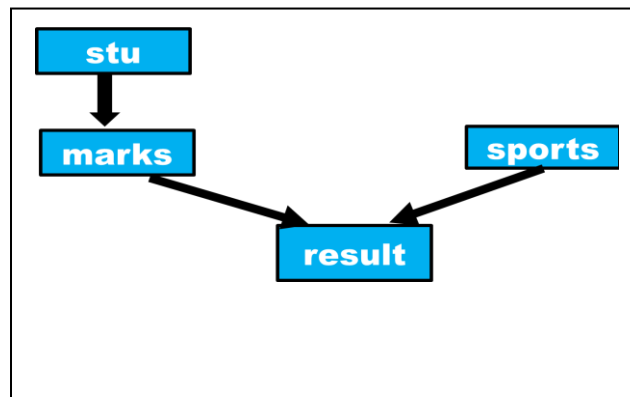
The derivation of class C from class B, class D is multiple inheritance.



Program to illustrate the concept of hybrid inheritance

```

#include<iostream>
using namespace std;
class stu {
int id;
char name[20];
public:
void getstu() {
cout<<"Enter student id and Name:";
cin>>id>>name;
}
};
class marks:public stu {
protected:
int m,p,c;
public:
void getmarks() {
cout<<"Enter 3 subjects marks:";
cin>>m>>p>>c;
}
};
class sports {
protected:
int spmarks;
public:
void getsports() {
cout<<"Enter sports marks: ";
cin>>spmarks;
}
};
class result:public marks,public sports {
int tot;
float avg;
public:
void show() {
tot=m+p+c;
avg=tot/3.0;
cout<<"Total= "<<tot<<endl;
cout<<"Average= "<<avg<<endl;
cout<<"Avg+sportsmarks= "<<avg+spmarks;
}
};
  
```



```

int main()
{
result r;
r.getstu();
r.getmarks();
r.getsports();
r.show();
}
  
```

Explain virtual base class with example.

- It is used to prevent the duplication/ambiguity.
- In hybrid inheritance child class has two direct parents which themselves have a common baseclass.
- So, the child class inherits the grandparent via two seperate paths. it is also called as indirect parent class.
- All the public and protected member of grandparent are inherited twice into child.
- We can stop this duplication by making base class virtual.
- The keywords virtual and public may be used in either order.
- If we use virtual base class, then it will inherit only single copy of member of base class to child class.

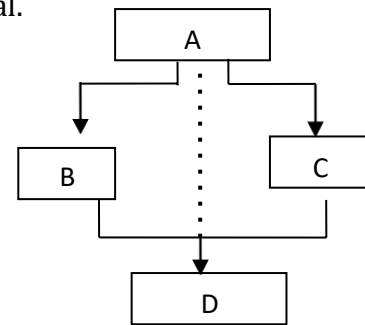


Figure: Multipath Inheritance

//VIRTUAL BASE CLASSES

```

#include<iostream>
using namespace std;
class A {
protected:
int a;
};
class B : public virtual A {
protected:
int b;
};
class C : public virtual A {
protected:
int C;
};
class D : public B,C {
int d;
public:
void getdata() {
cout<<"Enter values for a,b,c and d:";
cin>>a>>b>>c>>d;
}
void putdata(){
cout<<"a ="<<a<<"\t b ="<<b<<"\t c ="<<c<<"\t d ="<<d;
}
};
  
```

```

int main() {
D obj;
obj.getdata();
obj.putdata();
return 0;
}
  
```

OBJECT AS A CLASS MEMBER

Properties of one class can be used in another class using inheritance or using the object of a class as a member in another class.

Declaring the object as a class data member in another class is also known as delegation.

When a class has an object of another class as its member, such a class is known as a container class.

In delegation, the class consists of objects from other classes. The composed class uses the properties of other classes through their objects. This kind of relationship is known as has-a-relationship or containership.

//Write a program use object of one class in another class as a member.

```
#include<iostream>
using namespace std;
class A {
public:
    int x;
    A() {
        cout<<"\n Constructor of class A";
        x=20;
    }
};
class B {
public:
    int y;
    A one;
    B() {
        y=30;
        cout<<"\n Constructor of class B";
    }
    void show() {
        cout<<"\n x= "<<one.x<<" y= "<<y;
    }
};

int main() {
    B two;
    two.show();
    return 0;
}
```

Explanation: In the above program, the class B contains integer y and object one of class A. in function main(), object two is an object of class B. The constructor of class A is executed first, because when the compiler reaches the class B, it finds an object of class A. We know that the object declaration always executes the constructor of that class. Thus, the object of class A is declared, and the constructor of class A is executed. The constructor of class B is executed, and the variables of class B is initialized. The member function show() displays the contents of x and y. the content of class A member is obtained from object one.

ABSTRACT CLASS

When a class is not using for creating an object such class is known as abstract class.

The abstract class can act as a base class only.

An abstract class gives a skeleton or a structure, using this other classes are shaped.

A class which contains pure virtual function is called abstract base class.

```
// Abstract class
#include<iostream>
using namespace std;
class base {
public:
    virtual void display()=0;
    int x;

};
class derived : public base {
public:
    void display()
    {
        cout<<"\n In derived class ";
    }

};

int main() {
    base *b;
    derived d;
    b = &d;
    b->display();
    return 0;
}
```

ADVANTAGES OF INHERITANCE

When a class inherits from another class, there are three benefits:

1. You can reuse the methods and data of the existing class .

The most frequent use of inheritance is for deriving classes using existing classes, which provides reusability. The existing classes remain unchanged. By reusability, the development time of software is reduced.

The same base classes can be used by a number of derived classes in class hierarchy.

2. You can extend the existing class by adding new data and new methods.

The derived classes extend the properties of base classes to generate more dominant objects.

When a class is derived from more than one class, all the derived classes have similar properties to those of base classes.

3. You can modify the existing class by overloading its methods with your own implementations

DISADVANTAGES OF INHERITANCE

1. Though object -oriented programming is frequently propagandized as an answer for complicated projects, inappropriate use of inheritance makes programs more complicated.
 2. Invoking member functions using objects creates more compiler overheads.
 3. In class hierarchy, various data elements remain unused, and the memory allocated to them is not utilized.
-

TYPE CONVERSION

Converting data from one type into another type is called type conversion.

Example: - int m;

```
float x=3.141;
```

```
m = x ; // conversion of float type data into integer.
```

here m is integer x is float and the stmt m = x , here we want to store float value in to an integer data type, compiler will automatically convert float value in to integer after that it will store in to integer value m. This is done by compiler there fore it is called implicit type conversion or automatic type conversion.

Different types of Type conversion:

There are four types of type conversion:

1. Conversion from basic type to basic type.
2. Conversion from basic type to class type.
3. Conversion from class type to basic type.
4. Conversion from class type to class type.

1. Conversion from basic type to basic type:

This type of conversion can be done in two ways :

Implicit (Automatic)

Explicit(by Programmer)

Automatic(implicit) Type conversion : This is done by the compiler from the type that doesn't fit, to the type it wants.

Example : - int i;

```
float f=3.141;
```

```
i = f ; // implicit type conversion of float type data into integer.
```

Explicit Type Conversion :

Explicit type conversion done by the programmer from the type that doesn't fit, to the type it wants.

Example : - int i=3;

```
float f;
```

```
f = float(i)/2; //Explicit type conversion of integer to float.
```

2. Conversion from Basic to class type :

In c++ conversion of basic type to class type can be done using constructor. By creating a constructor the basic data type integer, float can be converted in to an object. conversion is automatically done by the compiler, by applying type casting.

In this , the left-hand operand of = sign is always class type and the right - hand operand is always basic type.

```
#include<iostream>
```

```
using namespace std;
```

```
class hours {
```

```
    int hrs;
```

```
    float f;
```

```
public:
```

```
    hours(int t) {
```

```
        hrs=t/60;
```

```
    }
```

```
void show() {  
    cout<<hrs<<" hours "<<endl;  
}  
};  
int main() {  
hours t1 = 125;  
t1.show();  
return 0;  
}
```

Explanation: The class name is hours . inside class we have declared a private variable hrs. we have declared a parameterized constructor by passing an integer value and this value is assigned to data member hrs. here actually we are converting a basic data type to object . The stmt hours t1 = 125, here t1 is object, 125 is an integer value, this integer value is going to be stored in to object. and this done by calling a parameterized constructor , so when the stmt hour t1 = 125 is executed the type casting is performed. i.e, a type conversion is performed with the help of constructor. at this step constructor is called 125 is passed as argument to constructor and inside the constructor hrs = t/60; hrs have integer value and using the object t1 we are printing the result. so here stmt hours t1 = 125 is representing the basic type to class type conversion . t1.show() will display o/p.

3. Conversion from class type to Basic type

The compiler does not have any knowledge about user-defined data type built using classes. In this type of conversion , the programmer, needs to explicitly tell the compiler how to perform conversion from class to basic type. These instructions are written in member function. Such type of conversion is also known as overloading of type cast operators, which is also known as conversion function. With the help of this conversion function a class data type can be converted in to basic types like integer, char, float.

The compiler first searches for the operator key word followed by data type .

In this type , the left- hand operand is always of basic data type and the right-hand operand is always of class type.

Syntax of conversion function:

It has the keyword operator followed by basic data type in which you want to convert the object and in body of function we have to return a particular data type in which we have to convert our object.

```
operator float()  
{  
    return (basic_type data);  
}
```

The casting operator function should satisfy following conditions :

1. It should be a class member function.
 2. It must not specify return type.
 3. The conversion function should not have any arguments.
-

Example: Program to convert class type data in to basic type data.

```
#include<iostream>
using namespace std;
class hours {
    int hrs;
    float f;
public:
    hours(int t) {
        hrs=t/60;
    }
    operator float() // type conversion from class to basic
    {
        return float(hrs)/2; // type conversion int to float(basic to basic)
    }
    void show() {
        cout<<hrs<<" hours "<<endl;
    }
};

int main() {
    float f;
    hours t1 = 185; //integer 185 is converted into object of class hrs
    t1.show();
    f=t1; // type conversion from class object "t1" to float "f"
    cout<<"time = "<<f<<endl;
    return 0;
}
```

Explanation : The class name is hours . inside class we have declared a private variable hrs. we have declared a parameterized constructor by passing an integer value and this value is assigned to data member hrs. here we have written a casting operator function , you can see there is no return type , and no parameters passed to it. in side the body of this casting function we are returning a float value so, our object will be converted to float value. so we can say our class type can be converted to basic data type. i.e, hours object is converted in to a float variable. In main() function the stmt hrs t1 = 185; is basic to class type conversion the stmt f=t1; here you can see class type to basic type , t1 is object of class and f is a float variable.

when the stmt f=t1 is executed the casting operator is called. in side this casting operator function the value of hrs/2 is converted to float and it is returned, the float value is returned by object t1 and it is stored in float variable f.

this is how a class type is converted to basic type with the help of casting operator function.

4. Conversion from one class type to another class type

When an object of one class is assigned to object of another class, it is necessary to instruct the compiler about how to make conversion between these two - user defined data types. there are two ways to convert object data type from one class to another class.

1. Define a conversion operator function in source class
 2. Define one argument constructor in a destination class.
- Consider the statement:
objX = objY;
 - The class Y type data is converted to the class X type data and the converted value is assigned to the objX.
 - The **class Y** is known as **source class** and **class X** is known as **destination class**.

Example : Program to convert class type to class type using conversion operator function in source class. (Using constructor)

```
#include<iostream>
using namespace std;
class minutes //source class
{
    int m;
public:
    minutes(int ms) {
        m=ms;
    }
    void show() {
        cout<<"Minutes= "<<m<<endl;
    }
    int getdata() {
        return m;
    }
};
class hours { //destination class
    int h;
public:
    hours() {
        h=0;
    }
    void show() {
        cout<<"Hours= "
            <<h<<endl;
    }
    hours(minutes mm) {
        h=mm.getdata()/60;
    }
};
```

```
int main() {
    minutes min(70);
    hours hr;
    hr=min; // class minutes to class hours
    min.show();
    hr.show();
    return 0;
}
```

Example: PROGRAM TO CONVERT CLASS TO CLASS TYPE(USING CONVERSION FUNCTION)

```
#include<iostream>
using namespace std;
class hours { //destination class
    public:
    int h;
    hours() {
        h=0;
    }
    void show() {
        cout<<"Hours= "<<h<<endl;
    }
};
class minutes { //source class
    int m;
    public:
    minutes(int ms) {
        m=ms;
    }
    operator hours() {
        hours h1;
        h1.h=m/60;
        return(h1);
    }
    void show() {
        cout<<"Minutes= "<<m<<endl;
    }
    int getdata() {
        return m;
    }
};
int main() {
    minutes min(60);
    hours hr;
    hr=min; // class minutes to class hours
    min.show();
    hr.show();
    return 0; }
```

UNIT IV**[Pointers & Binding Polymorphisms and Virtual Functions]**

Pointer, Features of Pointers, Pointer Declaration, Pointer to Class, Pointer Object, The this Pointer, Pointer to Derived Classes and Base Class, Binding Polymorphisms and Virtual Functions, Binding in C++, Virtual Functions, Rules for Virtual Function, Virtual Destructor.

POINTER: Pointer is a variable it holds address of another variable.

Each variable occupies certain memory location at the time of executing program and it is possible to access the address of memory location by using pointers.

Memory is arranged in series of bytes these bytes are numbered from zero onwards .

The number specified to a specific location is known as memory address.

A pointer variable stores the memory address of any type of variable.

The pointer variable and normal variable must be of same data type.

Pointer is denoted by ' * ' asterisk symbol.

The memory address is an unsigned integer(positive only) starting from zero to upper most addressing capacity of microprocessor.

The number of memory locations pointed by a pointer depends on the type of the pointer.

The pointers are either 16 bits or 32-bits long

The process of allocating memory at run time is called as dynamic memory allocation.

Pointers are used in array concept and functions concept.

FEATURES OF POINTER :

pointers save memory space.

Execution time with pointers is faster, because data are manipulated with the address, that is direct access to memory.

Memory is accessed efficiently with the pointers.

The pointer assigns as well as releases the memory space.

Memory is dynamically allocated.

Pointer are used with data structures.

They are useful for representing two-dimensional and multi - dimensional array.

In c++ a pointer declared to a base class could access the object of a derived class, how ever , a pointer to a derived class cannot access the object of base class.

POINTER DECLARATION

Pointer variables can be declared as follows:

Example:

```
int *x;  
float *f;  
char *y;
```

in the first statement 'x' is an integer pointer, and it informs the compiler that it holds the address of any integer variable. In the same way 'f' is a float pointer that stores the address of any float variable, in the same way character pointer which stores the address of any character variable.

Normal variables provide direct access to their own values, where as a pointer indirectly accesses the value of a variable to which it points.

The indirection operator(*) is used in two different ways:

- 1) When a pointer is declared the asterisk (*) symbol indicates it is a pointer variable, and it holds address of another variable.
- 2) When a pointer is dereference the asterisk symbol (*) indicates that the value at memory location stored in pointer is to be accessed.

// Write a program to display the address of the variable.

```
#include<iostream>
using namespace std;
int main() {
    int n;
    cout<<"\n ENTER A NO: ";
    cin>>n;
    cout<<"\n value of N: "<<n<<endl;
    cout<<"\n address of n = "<<(unsigned)&n;
    return 0;
}
```

// Write a program to display the value and address of the variable using pointer.

```
#include<iostream>
using namespace std;
int main() {
    int *p;
    int x=10;
    p=&x;
    cout<<"\n x= "<<x<<" &x= "<< &x;
    cout<<"\n *p= "<<*p<<"\t &p= "<<p<<"\t(Contents of pointer)"<<endl;
    return 0;
}
```

POINTER TO CLASS

Pointer is a variable it holds address of another variable and it may be of any data type i.e., int , float , double

Similarly we can also define a pointer to a class.

Example:-

```
class student {
    int id;
    int age;
};
class student *ptr;
```

In the above example ptr is a pointer to class student.

The syntax for using pointer with member is given below

```
ptr -> id
ptr -> age
```

Write a program to declare a class and declare pointer to class and display the contents of class members.

```
//Pointer to class
#include<iostream>
using namespace std;
class student{
public:
    char name[20];
    int id;
    int age;
};
int main() {
    class student s1={"Ravi",1210,21};
    student *ptr;
    ptr=&s1;
    cout<<"\n Student Name= "<<ptr->name<<endl;
    cout<<" Student id= "<<ptr->id<<endl;
    cout<<" Student Age= "<<ptr->age<<endl;
    return 0;
}
```

POINTER OBJECT

Similar to variables, objects also have an address.

A pointer can point to a specified object.

When accessing members of a class given a pointer to an object, use the arrow(->) operator instead of the dot operator.

Write a program to declare an object and pointer to the class. Invoke the member function using pointer.

```
#include<iostream>
using namespace std;
class student {
    char name[20];
    int id;
    int age;
public:
    void getstudent() {
        cout<<"Enter Student Name: ";
        cin>>name;
        cout<<"Enter Student ID: ";
        cin>>id;
        cout<<"Enter Student Age: ";
        cin>>age;
    }
}
```

```
void putstudent() {
    cout<<"\n Student Name= "<<name<<endl;
    cout<<" Student id= "<<id<<endl;
    cout<<" Student Age= "<<age<<endl;
}
};
int main() {
    class student s1;
    student *ptr;
    ptr=&s1;
    ptr->getstudent();
    ptr->putstudent();
    return 0;
}
```

Explain 'this' pointer with example.

- **'this'** pointer represent an object that invoke or call a member function.
- It will point to the object for which member function is called.
- It is automatically passed to a member function when it is called.
- It is also called as implicit argument to all member function.
For example: S.getdata();
- Here **S** is an object and **getdata()** is a member function.
So, **'this'** pointer will point or set to the address of object **S**.
To know the current object address we use this pointer.
- It is used to distinguish the data members from local variables when both are declared with name.
Every non-static member of C++ have this local variable 'this'.
- Suppose **'a'** is private data member, we can access it only in public member function like as follows
a=50;
- We access it in public member function by using **'this'** pointer like as follows
this->a=50;
Both will work same.

Example 1:

```
//this pointer
#include<iostream>
using namespace std;
class sample {
    int a,b;
public:
    show() {
        a=10;
        b=20;
```

```

        cout<<"Obj addr = "<<this<<endl;//this will display the current obj addr in hexa decimal
        cout<<"a= "<<this->a<<endl;
        cout<<"b= "<<this->b<<endl;;
    }
};

```

```

int main() {
    sample s;
    s.show();
    return 0;
}

```

output:

```

obj addr = 0x28ff08
a=10
b=20

```

Example 2:

```

//this pointer
#include<iostream>
using namespace std;
class sample {
    int a,b;
public:
    get(int a, int b) {
        this->a=a;
        this->b=b;
    }
    void display() {
        cout<<"A= "<<a<<endl;
        cout<<"B= "<<b<<endl;
    }
};
int main() {
    sample s;
    s.get(10,20);
    ispl
    ay()
    ;
    retu
    rn
    0;
}

```

out put:

A=10

B=20

Example 3:

```
//this pointer
#include<iostream>
using namespace std;
class sample {
int a;
    public: sample() {
        a=10;
    }
void disp(int a) {
    cout<<"The value of argument a="<<a;
    cout<<"\nThe value of data member a="<<this->a;
}
};
int main() {
    sample S;
    S.disp(20);
    return 0;
}
```

out put:

The value of argument a= 20

The value of data member a= 10

Explain pointer to derived classes and base class.

- We can use pointers not only to the base objects but also to the objects of derived classes.
- A single pointer variable can be made to point to objects belonging to different classes.

For example:

```
B *ptr          //pointer to class B type
variable B b;    //base object
D d;             // derived object
ptr = &b;        // ptr points to object b
```

- In above example **B** is base class and **D** is a derived class from **B**, then a pointer declared as a pointer to **B** and point to the object b. •
 - We can make **ptr** to point to the object **d** as follows
ptr = &d;
 - We can access those members of derived class which are inherited from base class by base class pointer.
 - But we cannot access original member of derived class which are not inherited by base classpointer.
 - We can access original member of derived class which are not inherited by using pointer of derivedclass.
-

// Write a program to declare a pointer to the base class and access the member variable of base and derived class.

//pointer TO BASE CLASS

```
#include<iostream>
```

```
using namespace std;
```

```
class base {
```

```
public: int b;
```

```
void show() {
```

```
cout<<"\nThe value of b : "<<b;
```

```
}};
```

```
class derived:public base {
```

```
public: int d;
```

```
void show() {
```

```
cout<<"\nThe value of b="<<b
```

```
<<"\nThe value of d="<<d;
```

```
}};
```

```
int main() {
```

```
base B;
```

```
base *bptr;
```

```
bptr=&B;
```

```
cout<<"\nBase class pointer assign address of base class object";
```

```
bptr->b=100;
```

```
//bptr->d=200;
```

```
bptr->show();
```

```
derived D;
```

```
bptr=&D;
```

```
cout<<"\nBase class pointer assign address of derived class object";
```

```
bptr->b=200;
```

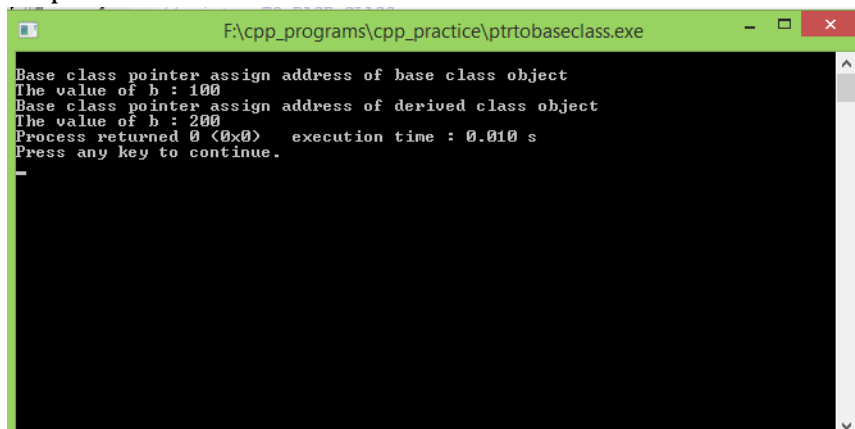
```
//bptr->d=300;
```

```
bptr->show();
```

```
return 0;
```

```
}
```

output :

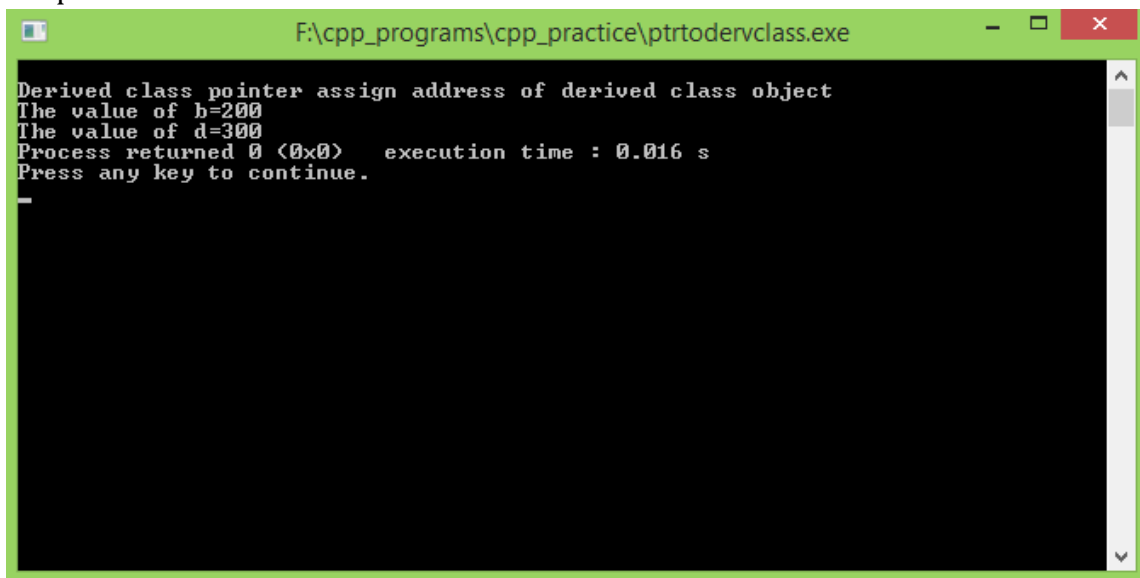


The screenshot shows a Windows command prompt window titled "F:\cpp_programs\cpp_practice\ptrtobaseclass.exe". The output text is as follows:

```
Base class pointer assign address of base class object
The value of b : 100
Base class pointer assign address of derived class object
The value of b : 200
Process returned 0 (0x0)   execution time : 0.010 s
Press any key to continue.
```

```
//pointer TO DERIVED CLASS
#include<iostream>
using namespace std;
class base {
public: int b;
void show() {
cout<<"\nThe value of b : "<<b;
}
};
class derived:public base {
public: int d;
void show() {
cout<<"\nThe value of b="<<b
<<"\nThe value of d="<<d;
}
};

int main() {
derived D;
derived *dptr;
dptr=&D;
dptr->b=200;
dptr->d=300;
cout<<"\nDerived class pointer assign address of derived class object";
dptr->show();
return 0;
}
out put:
```

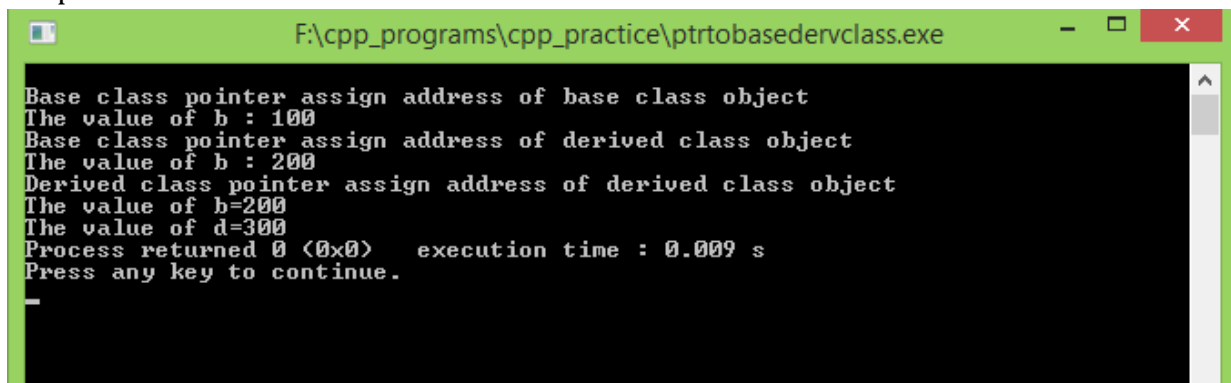


```
F:\cpp_programs\cpp_practice\ptrtodervclass.exe
Derived class pointer assign address of derived class object
The value of b=200
The value of d=300
Process returned 0 (0x0) execution time : 0.016 s
Press any key to continue.
```

// Write a program to declare a pointer to the derived class and access the member variable of base and derived class.

```
#include<iostream>
using namespace std;
class base {
public: int b;
void show() {
cout<<"\nThe value of b : "<<b;
}
};
class derived: public base {
public: int d;
void show() {
cout<<"\nThe value of b="<<b
cout<<"\nThe value of d="<<d;
}
};
int main() {
base B, *bptr;
derived D;
bptr=&B;
cout<<"\nBase class pointer assign address of base class object";
bptr->b=100;
bptr->show(); bptr=&D;
bptr->b=200;
cout<<"\nBase class pointer assign address of derived class object";
bptr->show();
derived *dptr;
dptr=&D;
cout<<"\nDerived class pointer assign address of derived class object";
dptr->d=300;
dptr->show();
return 0;
}
```

out put:

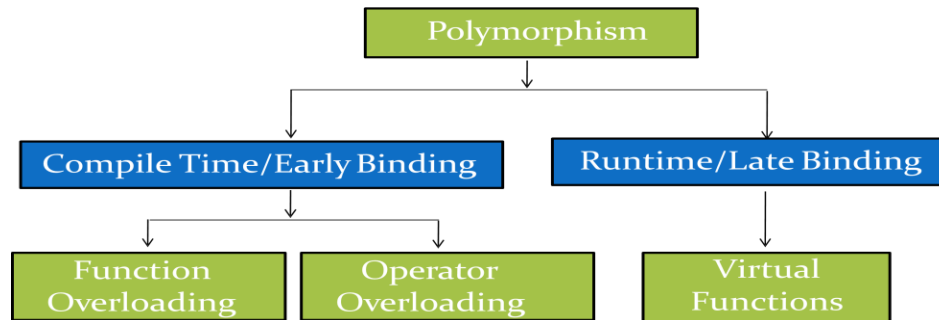


```
F:\cpp_programs\cpp_practice\ptrtobasedervclass.exe
Base class pointer assign address of base class object
The value of b : 100
Base class pointer assign address of derived class object
The value of b : 200
Derived class pointer assign address of derived class object
The value of b=200
The value of d=300
Process returned 0 (0x0)   execution time : 0.009 s
Press any key to continue.
```

Binding Polymorphism and Virtual Functions

Virtual functions belongs to the branch of Runtime Polymorphism in C++
Polymorphism means one name multiple forms.

- Polymorphism is classified into 2 branches
 Compile Time Polymorphism/Early Binding/Static Binding
 Runtime Polymorphism/Late Binding/Dynamic Binding



What is Binding?

- For every function call, compiler binds or links the call to one function definition. This linking can happen at 2 different time
 - At the time of compiling program, or
 - At Runtime

Deciding a function call at compile time is called as compile time (or) static (or) early binding. Deciding a function call at run time is called as run time (or) dynamic (or) late binding.

➤ COMPILE TIME POLYMORPHISM/EARLY BINDING

- When a function call encountered the compiler will bind the appropriate function with the object based on the type of the function call.
- The overloaded member functions are selected for invoking by matching arguments, both type and number. This information is known to the compiler at the compile time and compiler is able to select the appropriate function for a particular call at the compile time itself. This is called **Early Binding** or **Static Binding** or **Static Linking**. Also known as **compile time polymorphism**. Early binding means that an object is bound to its function call at the compile time.
- Compile time polymorphism is function overloading and operator overloading.

FUNCTION OVERLOADING:

- Function overloading is declaring the same function with different signatures. that is defining several functions with same name, by changing no.of arguments, type of arguments(data types) and order of arguments,
- The same function name will be used with different number of parameters and parameters of different type.
- Overloading of functions with different return type is not allowed.

OPERATOR OVERLOADING:

- Operator overloading is the ability to tell the compiler how to perform a certain operation based on its corresponding operator's data type.
- Like + performs addition of two integer numbers, concatenation of two string variables and works totally different when used with objects of type class.

➤ RUNTIME TIME POLYMORPHISM/LATE BINDING

- It would be nice if the appropriate member function could be selected while the program is running. This is known as **runtime polymorphism**. C++ supports a mechanism known as **virtual function** to achieve run time polymorphism.
- At the runtime, when it is known what class objects are under consideration, the appropriate version of the function is invoked. Since the function is linked with a particular class much later after the compilation, this process is termed as late binding. It is also known as **dynamic binding** because the selection of the appropriate function is done dynamically at run time.
- In late binding, call to a function is resolved at Runtime, the compiler determines the type of object at execution time and then binds the function call to a function definition.
- It means that the code associated with a given procedure call is not known until the time of the call.

At run-time, the code matching the object under current reference will be called.

Late binding is also called as Dynamic Binding or Runtime Binding.

Virtual Functions are example of Late Binding in C++

Runtime polymorphism is achieved using pointers.

VIRTUAL FUNCTIONS

Polymorphism refers to the property by which objects belonging to different classes are able to respond to the same message, but different forms. An essential requirement of polymorphism is therefore the ability to refer to objects without any regard to their classes.

When we use the same function name in both the base and derived classes, the function in the base class is declared as virtual using the keyword virtual preceding its normal declaration.

When a function is made virtual, C++ determines which function to use at runtime based on the type of object pointed to by the base pointer, rather than the type of the pointer. Thus, by making the base pointer to point to different objects, we can execute different versions of the virtual function.

```
#include<iostream>
using namespace std;
class Base {
public:
void display() {
cout<<"Display Base"<<endl;
}
virtual void show() {
cout<<"Show Base"<<endl;;
}
};
class Derived : public Base {
public:
void display() {
cout<<"Display Derived"<<endl;;
}
```

```
void show() {  
    cout<<"show derived"<<endl;  
}  
};  
int main() {  
    Base b;  
    Derived d;  
    Base *ptr;  
    cout<<"ptr points to Base"<<endl;  
    ptr=&b;  
    ptr->display(); //calls Base  
    ptr->show(); //calls Base  
    cout<<"ptr points to derived"<<endl;  
    ptr=&d;  
    ptr->display(); //calls Base  
    ptr->show(); //class Derived  
    return 0;  
}
```

Output:

```
ptr points to Base  
Display Base  
Show Base  
ptr points to Derived  
Display Base  
Show Derived
```

When ptr is made to point to the object d, the statement ptr->display(); calls only the function associated with the Base i.e.. Base::display()

where as the statement ptr->show(); calls the derived version of show().

This is because the function display() has not been made virtual in the Base class.

- By making the function 'virtual' in base class C++ determines which function to use at run time based on the type of object pointed to by the base pointer, rather than the type of the pointer.
 - Runtime polymorphism is achieved only when a virtual function is accessed through a pointer to the base class.
 - When a class containing virtual function is inherited, the derived class can redefine (Override) the virtual function to suit its own unique needs.
 - The method name and type signature should be same for both base and derived version of function.
-

Example for virtual function/method over riding

in this program we are over riding the area() function of base class in derived class.

```
#include<iostream>
using namespace std;
class circle
{
protected:
float radius;
public:
circle(float r)
{
radius=r;
}
virtual float area() {
float a;
a=3.14*radius*radius;
return a;
}
};
class cylinder : public circle
{
private:
float height;
public:
cylinder(float r, float h): circle(r)
{
height =h;
}
float area() {
float a;
a=(2*3.14*radius*radius)+(2*3.14*radius*height);
return a;
}
};
int main() {
circle *ptr;
circle ciobj(8);
cylinder cyobj(8,4);
ptr=&ciobj;
cout<<"Area of circle : "<<ptr->area()<<endl;
ptr=&cyobj;
cout<<"Area of cylinder : "<<ptr->area();
return 0;
}
out put:
```

Area of circle : 200.96

Area of cylinder :602.88

- **Method overriding** allows a derived class to provide a specific implementation of a method that is already provided by one of its base class.
- The implementation in the derived class overrides (replaces) the implementation in the base class by providing a method that has same name, same parameters (signature), and same return type as the method in the parent class has.
- Using a base class pointer variable, we can point to object of any child class, depending upon the type of object at runtime a particular method will be called if that method is made virtual in base class & overridden in derived class.

overloading Vs over riding

- **Overloading** occurs when two or more methods in one class have the same method name but different parameters (signature), **Overriding** means having two methods with the same method name and parameters (*method signature*), one of the method is in the parent class and the other is in child class.
- Call to an **Overloaded** method is resolved at compile time, while call to an **Overridden** method is resolved at runtime depending upon the type of object.

	Overloading	Overriding
Definition	Methods having same name but each must have different number of parameters or parameters having different types & order.	Sub class have method with same name and exactly the same number and type of parameters and same return type as super class method.
Meaning	More than one method shares the same name in the class but having different signature.	Method of base class is re-defined in the derived class having same signature.
Behaviour	To Add/Extend more to method's behaviour.	To Change existing behaviour of method.
Polymorphism	Compile Time	Run Time
Inheritance	Not Required	Always Required
Method Signature	Must have different signature	Must have same signature.
Method Relationship	Relationship is between methods of same class.	Relationship is between methods of super class and sub class.
No of Classes	Does not require more than one class for overloading.	Requires at least 2 classes for overriding.
Example	<pre> class Sample { public: void MyFunction() { cout << "MyFunction Called"; } void MyFunction(int param) { cout << "MyFunction Called : " << param; } }; </pre>	<pre> class Base { public: virtual void MyFunction() { cout << "Base MyFunction"; } }; class Derived : public Base { public: void MyFunction() { cout << "Derived MyFunction"; } }; </pre>

Rules For Virtual Functions:

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

1. The virtual functions must be members of some class.
2. They cannot be static members.
3. They are accessed by using object pointers.
4. A virtual function can be a friend of another class.
5. A virtual function in a base class must be defined, even though it may not be used.
6. The prototypes of the base class version of a virtual function and all the derived class versions must be identical. C++ considers them as overloaded functions, and the virtual function mechanism is ignored.
7. We cannot have virtual constructors, but we can have virtual destructors.
8. While a base pointer points to any type of the derived object, the reverse is not true. i.e. we cannot use a pointer to a derived class to access an object of the base class type.
9. When a base pointer points to a derived class, incrementing or decrementing it will not make it to point to the next object of the derived class. It is incremented or decremented only relative to its base type. Therefore we should not use this method to move the pointer to the next object.
10. 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.

Explain pure virtual functions.

- A pure virtual function means "do nothing" function.
- We can say empty function. A pure virtual function has no definition relative to the base class.
- Programmers have to redefine pure virtual function in derived class, because it has no definition in base class.
- A class containing pure virtual function cannot be used to create any direct objects of its own.
- This type of class is also called as abstract class.

Syntax:

```
virtual void display() = 0;
```

(OR)

```
virtual void display() {}
```

```
#include <iostream>
using namespace std;
class employee {
int code;
char name [20];
public:
virtual void getdata ()=0;
virtual void display ()=0;
};

class grade: public employee {
char grd [10];
float salary;
public: void getdata ();
void display ();
};

void employee :: getdata (){
}
void employee:: display () {
}
void grade:: getdata () {
cout<<"enter employee's grade ";
cin>> grd;
cout<<"\n enter the salary ";
cin>> salary;
}
void grade:: display () {
cout<<"\tGrade \tsalary \n";
cout<<"\t"<<grd<<" \t"<<salary<< endl;
}
int main () {
employee *ptr;
grade obj;
ptr = &obj;
ptr->getdata ();
ptr->display ();
return 0;
}
```

VIRTUAL DESTRUCTORS

Destructors can be declared as virtual, but constructors can't be declared as virtual.

Virtual destructors are implemented just like normal virtual functions.

Destructors of base class and derived class are called when base class pointer is deleted.

'delete' is the operator which is used to delete the pointer.

Example: A derived class object is constructed using a new operator. The base class pointer object holds the address of the derived object. when the base class pointer is destructed using the delete operator, the destructor of the base and derived class is executed.

The following program illustrates this

```
#include<iostream>
using namespace std;
class Base {
public:
    Base() {
        cout<<"Base class constructor"<<endl;
    }
    virtual ~Base () {
        cout<<"Base class destructor"<<endl;
    }
};

class Derived : public Base {
public:
    Derived()
    {
        cout<<"Derived class constructor"<<endl;
    }
    ~Derived()
    {
        cout<<"Derived class destructor"<<endl;
    }
};

int main() {
    Base *bptr;
    bptr=new Derived;
    delete bptr;
    return 0;
}
```

output:

```
Base class constructor
Derived class constructor
Derived class destructor
Base class destructor
```

UNIT V**[Generic Programming with Templates]**

Need for Templates, Definition of class Templates, Normal Function Templates, Overloading of Template Function, Bubble Sort Using Function Templates, Difference Between Templates and Macros, Linked Lists with Templates, Exception Handling, Principles of Exception Handling, The Keywords try throw and catch, Multiple Catch Statements – Specifying Exceptions.

TEMPLATES in C++ (GENERIC):

The template is one of the most useful characteristics of the C++.

The template provides generic programming by defining generic classes.

In templates, generic data types are used as arguments, and they can handle a variety of data types.

A function that works for all C++ data types is called a generic function.

NEED FOR TEMPLATES:

A template is a technique that allows a single function or class to work with different data types.

Using a template, we can create a single function that can process any type of data, that is the formal arguments of a template function are of template(generic) type. They can accept data of any type, such as int, float, and long. Thus a single function can be used to accept values of a different data type.

Example

Function overloading: It is a concept of defining several functions with same name, by changing number of arguments, data types and order of arguments.

Usually we overload a function, when we need to handle different data types,

Ex: sum of int and sum of float.

```
int sum(int x, int y)
{
    return x+y;
}
```

```
int sum(float x, float y)
{
    return x+y;
}
```

In this example the function names are identical, one performs integer addition, and another performs float addition. Here we are using same functions for different purposes, but the length of the program is not reduced.

We have to reduce the program length that is we have to declare one function which performs all the operations.

A template safely overcomes the limitations occurring during overloading function and allows better flexibility to the program.

By using templates with one function we can perform different task, by this the program size is reduced.

Template: It allows to design several function declarations.

1. We can design functions by using templates.
2. We can design classes by using templates.

Templates are classified in to two types:

1. Function templates(Generic functions).
2. Class templates (Generic class).

The key word template is used to define Function template and class template.

FUNCTION TEMPLATES

The templates declared for functions are called as function templates.

A function template defines how an individual function can be constructed.

Normal Function Templates

The normal function(not a member function) can also use template arguments.

The difference between normal and member function is that normal functions are defined outside the class. They are not members of any class and hence, can be invoked directly without using the object of a dot operator. The member functions are class members, and they can be invoked using the object of the class they belong to.

Syntax for Normal template function declaration :

```
template < class T>
ret_type fun_name(arguments)
{
    -----// body of the function
    -----
}
```

Write a program to define normal template function.

```
#include<iostream>
using namespace std;
template<class T>
void show(T x)
{
    cout<<"\n x= "<<x;
}
int main() {
char c='A';
int i=65;
double d=65.254;
show(c);
show(i);
show(d);
return 0;
}
```

Program to illustrate a function which holds same type of data type.

```
#include<iostream>
using namespace std;
template<class t>
t sum(t a, t b) {
    return a+b;
}
int main() {
    cout<<"Sum of Integer values = "<<sum(5,9)<<endl;
    cout<<"Sum of Float values = "<<sum(5.5,9.6);
    return 0;
}
```

Function Templates with Multiple Parameters:

- We can use more than one generic data type in the template statement, using comma seperated list as shown below.
- For example:

```
template <class T1, class T2>
```

Program to illustrate a function which holds arguments of different type or arguments of different data type.

```
#include<iostream>
using namespace std;
template<class T1, class T2>
float sum(T1 a,T2 b) {
    return a+b;
}
int main() {
    cout<<"Sum of Integer values = "<<sum(15,19)<<endl;
    cout<<"Sum of Float values = "<<sum(5.5,9.6)<<endl;
    cout<<"Sum of Integer and float = "<<sum(15,19.9)<<endl;
    cout<<"Sum of Float and Integer = "<<sum(5.5,9)<<endl;
    return 0;
}
```

Here in the above program we have defined one function and used it in 4 different ways. By this the program size is reduced.

Overloading of Template Function:

This is also called as overloading Generic Functions.

Overloading is a mechanism of defining several functions with same name by changing number of arguments order of arguments and data types.

A template function also supports overloading mechanism.

It can be overloaded by a normal function or template function.

The compiler follows the following rules for choosing appropriate function when program contains overloaded functions.

i) Searches for an accurate match of functions, if found, it is invoked.

ii) In case no match is found, an error will be reported.

Program to illustrate overloading of function templates

```
#include<iostream>
using namespace std;
template<class T>
T sum(T a, T b)
{
    return a+b;
}
template<class T>
T sum(T a, T b, T c)
{
    return a+b+c;
}
int main() {
    cout<<"Sum of two Integers = "<<sum(15,19)<<endl;
    cout<<"Sum of two Floats = "<<sum(5.5,9.6)<<endl;
    cout<<"Sum of three Integers = "<<sum(15,19,23)<<endl;
    cout<<"Sum of three floats = "<<sum(5.5,9.8,7.5)<<endl;
    return 0;
}
```

Templates in array:

program to find the sum of integer array, float array.

```
#include<iostream>
using namespace std;
template<class T>
T sum(T a[], int size)
{
    T s=0;
    for(int i=0;i<size;i++)
    {
        s=s+a[i];
    }
    return s;
}

int main()
{
    int x[5]={10,20,30,40,50};
    float y[3]={1.1,2.2,3.3};
    cout<<"Sum of Integer array = "<<sum(x,5)<<endl;
    cout<<"Sum of Float array = "<<sum(y,3)<<endl;
    return 0;
}
```

CLASS TEMPLATES

The templates declared for classes are called class templates. A class template specifies how individual classes can be constructed similar to the normal class specification. These classes model a generic class which support similar operations for different data types.

General Form of a Class Template

```
template <class T>
class class-name
{
.....
.....
};
```

A class created from a class template is called a template class.

The syntax for defining an object of a template class is:

```
classname<type> objectname(arglist);
```

The statement `template<class T>` tells the compiler that the following class declaration can use the template data type. The T is a variable of template type.

Templates cannot be declared inside the classes or functions. They should be global and should not be local.

Write a program to show values of different data types using overloaded constructor.

```
#include<iostream>
using namespace std;
class data {
public :
    data(char c)
    {
        cout<<"\n C= "<<c<<" Size in bytes: "<<sizeof(c);
    }
    data(int c)
    {
        cout<<"\n C= "<<c<<" Size in bytes: "<<sizeof(c);
    }
    data(double c)
    {
        cout<<"\n C= "<<c<<" Size in bytes: "<<sizeof(c);
    }
};
int main() {
    data h('A'); //passes character type data
    data i(100); //passes integer type data
    data j(68.2); //passes double type data
    return 0;
}
```

Write a program to show values of different data types using constructor and template

```
#include<iostream>
using namespace std;
template<class T>
class data {
public :
    data(T c)
    {
        cout<<"\n C= "<<c<<" Size in bytes: "<<sizeof(c);
    }
};

int main() {
    data <char> h('A');
    data <int> i(100);
    data <float> j(68.2);
    return 0;
}
```

Write a program to define data members of template type.

```
#include<iostream>
using namespace std;
template<class T>
class data {
    T x;
public:
    data(T u)
    {
        x=u;
    }
    void show(T y)
    {
        cout<<"X="<<x;
        cout<<"\tY="<<y<<endl;
    }
};

int main() {
    data <char> c=('B');
    data <int> i(100);
    data <double> d(48.25);
    c.show('A');
    i.show(65);
    d.show(68.25);
    return 0;
}
```

Write a program to find square of a given value by using template.

```
#include<iostream>
using namespace std;
template<class S>
class sqr {
    public:
    sqr(S c) {
        cout<<"\n Square of given number "<<c<<" is "<<c*c<<endl;
    }
};

int main() {
    sqr <int> i(25);
    sqr <double> d(15.2);
    return 0;
}
```

Another Example of class template.

Class Template:

- Suppose you want to perform addition of two integer number in class

```
class add {
    int a,b,c;
    public:
    add() {
        a = 10;
        b = 20;
        c = a + b;
    }
    void putdata() {
        cout<<"\nThe sum="<<c;
    }
};
```

- If you later decide you also want to perform addition of two float numbers in class then you have to write separate class for addition of two numbers.

```
class add {
    float a,b,c;
    public:
    add() {
        a = 10.45;
        b = 20.67;
        c = a + b;
    }
    void putdata() {
        cout<<"\nThe sum="<<c;
    }
};
```

write a C++ program for class template .

In the above code a class is declared with integer data type or data members, then we have to use that class only for integer type.

If it is float we have to use it only for float.

- It means we have to write same class two times because of only different data type.
- If we want to use same class for different types then it is done by using Template class or Generic class.

```
#include<iostream>
using namespace std;
template<class T>
class Sample {
    T a,b,c;
public:
    void getdata() {
        cout<<"Enter the value of a & b\n";
        cin>>a>>b;
    }
    void putdata() {
        cout<<"\nThe sum="<<c;
    }
    T sum();
};
template<class T>
T Sample<T> :: sum()
{
    c=a+b;
}
int main(){
    Sample <int> S1;
    cout<<"Enter Integer values"<<endl;
    S1.getdata();
    S1.sum();
    Sample <float> S2;
    cout<<"Enter Float values"<<endl;
    S2.getdata();
    S2.sum();
    cout<<"Integer values sum ";
    S1.putdata();
    cout<<"\nFloat values sum ";
    S2.putdata();
    return 0;
}
```

- We will pass data type when we create object of class. So, now template variable 'T' is replaced by data type which we passed with object.
-

Note: When template class member function definition is outside it should once again start with
template <class T>
return type class name<T> :: functionname

Class Templates with Multiple Parameters:

- We can use more than one generic data type in a class template. They are declared as a comma separated list within the template specification as shown below.

```
template <class T1, class T2>
class classname
{
/*Statement 1;
Statement 2;
.
.
Statement n; */
};
```

write a C++ program for class template with multiple parameters.

```
#include<iostream>
using namespace std;
template<class T1, class T2>
class Sample {
T1 a;
T2 b;
public:
Sample(T1 x, T2 y) {
a=x; b=y;
}
void display() {
cout<<"\na="<<a<<"\tb="<<b;
}
};
int main() {
Sample <int, float> S1(12,23.3);
Sample <char, int> S2('N',12);
S1.display();
S2.display();
return 0;
}
```

Bubble sort using function templates

The process of arranging the given elements either in ascending or descending order is called as sorting.

- It is a simple sorting algorithm that works by repeatedly stepping through the list to be sorted, comparing each pair of adjacent items and swapping them if they are in the wrong order.
- The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted.
- The algorithm gets its name from the way smaller elements "bubble" to the top of the list.
- As it only uses comparisons to operate on elements, it is a comparison sort.
- Although the algorithm is simple, it is too slow for practical use, even compared to insertion sort.

Example

Consider an array A of 5 element

A[0]	45
A[1]	34
A[2]	56
A[3]	23
A[4]	12

Pass-1: The comparisons for pass-1 are as follows.

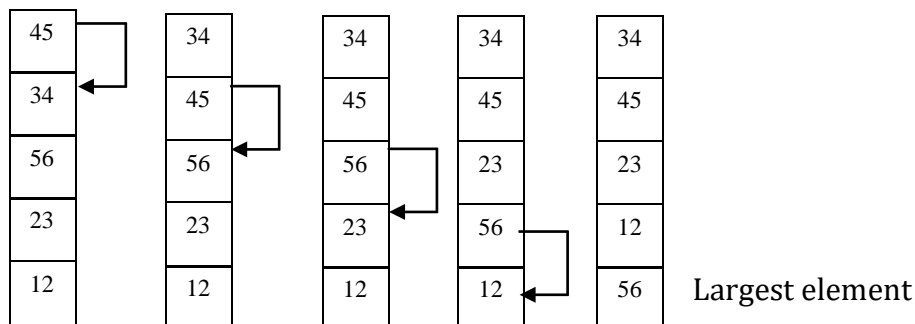
Compare A[0] and A[1]. Since $45 > 34$, interchange them.

Compare A[1] and A[2]. Since $45 < 56$, no interchange.

Compare A[2] and A[3]. Since $56 > 23$, interchange them. Compare A[3] and A[4].

Since $56 > 12$ interchange them.

At the end of first pass the largest element of the array, 56, is bubbled up to the last position in the array as shown.

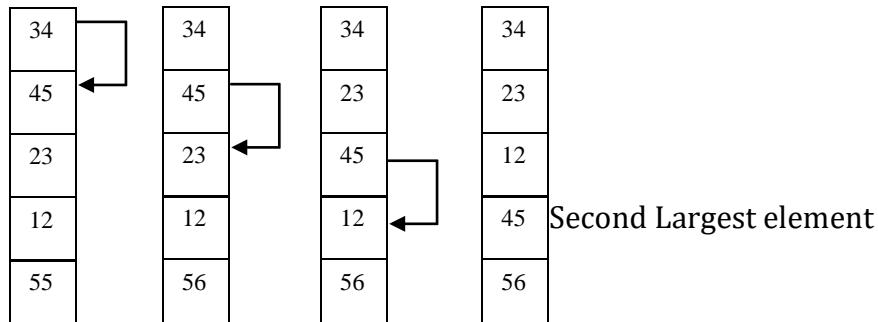


Pass-2: The comparisons for pass-2 are as follows.

Compare A[0] and A[1]. Since $34 < 45$, no interchange.

Compare A[1] and A[2]. Since $45 > 23$, interchange them.

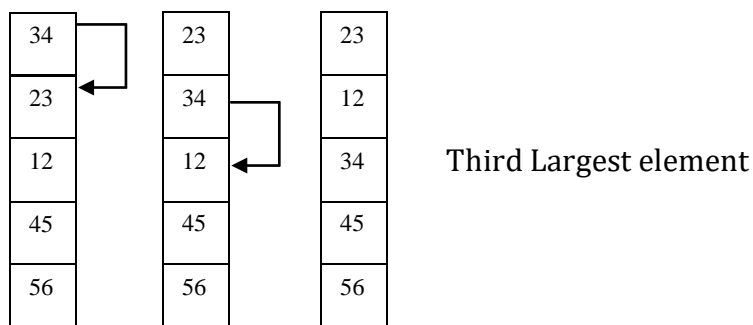
Compare A[2] and A[3]. Since $45 > 12$, interchange them.



Pass-3: The comparisons for pass-3 are as follows.

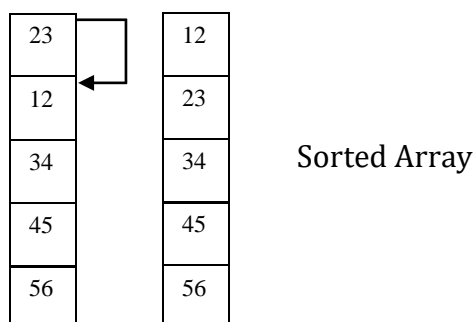
Compare A[0] and A[1]. Since $34 > 23$, interchange them.

Compare A[1] and A[2]. Since $34 > 12$, interchange them.



Pass-4: The comparisons for pass-4 are as follows.

Compare A[0] and A[1]. Since $23 > 12$, interchange them.



Write a program to sort the given elements by using bubble sort using template function

```
#include<iostream>
using namespace std;
//Declaration of template class
template<class B>
B bsort(B a[], int n) {
int i, j;
//bubble sort algorithm inside template
for(i=0;i<n-1;i++)
{
    for(j=0;j<(n-i-1);j++)
    {
        if(a[j]>a[j+1])
        {
            B b;
            b=a[j];
            a[j]=a[j+1];
            a[j+1]=b;
        }
    }
}
}

int main() {
int arr[20],m,k,i;
char ch[20];
cout<<"Enter what type of elements you want to sort :\n1. int\n2. char\n";
cin>>m;
if(m==1) {
cout<<"\nEnter the number of elements you want to enter:";
cin>>k;
cout<<"\nEnter Integer elements:";
for(i=0;i<k;i++)
cin>>arr[i];
bsort(arr,k);    //call to bubble for integer sorting
cout<<"\nSorted Order Integers: ";
for(i=0;i<k;i++)
cout<<arr[i]<<"\t";
}
else {
cout<<"\nEnter the number of characters you want to enter:";
cin>>k;
cout<<"\nEnter elements:";
for(i=0;i<k;i++)
cin>>ch[i];
bsort(ch,k);    //call to bubble for character sorting
cout<<"\nSorted Order Characters: ";
for(i=0;i<k;i++)
cout<<ch[i]<<"\t";
}
return 0; }
```

Difference Between Templates and Macros

Macros are preprocessing statements and are not type safe, that is a macro defined for integer operation cannot accept float data.

It is difficult to find errors in macros.

Write a program to perform increment operation by using macros

```
#include<iostream>
using namespace std;
#define mac(x) x+++x
int main()
{
    int a=10,c;
    c=mac(a);
    cout<<"After Increment a="<<c;
return 0;
}
```

Write a program to perform increment operation by using templates

```
#include<iostream>
using namespace std;
template<class T>
T mac(T k)
{
    ++k;
    return k;
}
int main()
{
    int a=10,c;
    c=mac(a);
    cout<<"After Increment a="<<c;
return 0;
}
```

Exception Handling:

- In programming two most common types of errors are
 1. Compile time errors (or) syntax errors.
 2. Run time error (or) logical error
- The syntax error is detected during compilation of program, but the logical error will detect during execution of program. So, it is very difficult to handle logical error. These logical errors are also known as exceptions.

Principles of Exception Handling:

When a program encounters an exception condition, it must be identified and handled.

Exceptions provide a way to transfer control from one part of a program to another. C++ exception handling is built upon three keywords: try, catch, and throw.

Types of exceptions: There are two kinds of exceptions

1. Synchronous exceptions
2. Asynchronous exceptions

1. **Synchronous exceptions:** The exceptions which occur during the program execution due to some fault in the input data are known as synchronous exceptions.

Example: Errors such as "Out-of-range index" and "over flow" are synchronous exceptions

2. **Asynchronous exceptions:** The errors that are generated by any event beyond the control of the program are called asynchronous exceptions

The purpose of exception handling is to provide a means to detect and report an exceptional circumstance

- The exception handling provides mechanism to handle logical error during execution of program. Steps to handle logical error:

1. Find the problem (Hit the exception)
2. Inform that an error has occurred (Throw the exception)
3. Receive the error information (Catch the exception)
4. Take corrective actions (Handle the exception)

Exceptions were considered to be dangerous because when ever they occur the program terminates abnormally on the line where the exception occur, with out executing the next lines of code.

Exception Handling Mechanism:

It is an approach which stops the abnormal termination of a program when ever an exception raises and executes the code which was not related with the exception.

The Keywords try throw and catch:

The exception-handling technique passes the control of a program from a location of exception in a program to an exception-handler routine linked with try block.

An-exception-handler routine can only be called by the throw statement.

(a) try : The try keyword is followed by a series of statements enclosed in curly braces.

Syntax of try statement:

```
try
{
    statement 1;
    statement 2;
}
```

(b)throw: The function of the throw statement is to send the exception found. The declaration of the throw statement is as follows :

Syntax of throw statement :

```
throw (excep);
throw excep;
throw // re-throwing of an exception
```

The argument excep is allowed to be of any type, and it may be a constant. The catch block associated with the try block catches the exception thrown. The throw statement can be placed in a function or in a nested loop, but it should be in the try block. After throwing the exception, control passes to the catch statement.

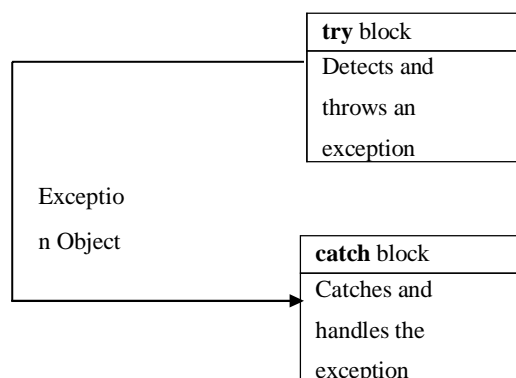
(c) catch: Similar to the try block, the catch block also contains a series of statements enclosed in curly braces. It also contains an argument of an exception type in parantheses.

Syntax of catch statement:

```
try
{
    statement 1;
    statement 2;
}
catch (argument)
{
    statement 3; //Action to be taken
}
```

When an exception is found, the catch block is executed. The catch statement contains an argument of exception type, and it is optional. When an argument is declared, the argument can be used in the catch block. After the execution of the catch block, the statements inside the blocks are executed. In case no exception is caught, the catch block is ignored, and if a mismatch is found, the program is terminated.

- Exception handling mechanism is built upon three keyword: try, throw and catch.
- try is used to preface a block of statement which may generate exceptions.
- When an exception is detected, it is thrown using a throw statement.
- A catch block is used to catch the exceptions thrown by throw statement and take appropriate action. The relationship between try, throw and catch block is as shown in below figure.



try, throw, and catch blocks :

```
try    {
    // Set of Statments;
    throw exception;
    // Set of Statements;
    catch(type arg)
    {
        // Set of Statements;
    }
}
```

Program to throw an exception

```
#include<iostream>
using namespace std;
int main()
{
int a,b;
cout<<"Enter the value of a and b\n";
cin>>a>>b;
try
{
if(b != 0)
cout<<"The result(a/b)="<<a/b;
else
throw(b);
}
catch(int x)
{
cout<<"Exception caught b="<<x;
}
return 0;
}
```

1) Explain multiple catch statements with appropriate example.

We can also define multiple catch blocks, in the try block.

The format of multiple catch statements is as follows:

```
try
{
    // try section
}

catch (object 1)
{
    // catch section1
}
catch (object 2)
{
    // catch section2
}
. . . . .
catch (type n object)
{
    // catch section - n
}
```

- It is possible that a program segment has more than one condition to throw an exception.
- For these situations we can associate more than one catch statement with a try block.
- When an exception is thrown, the exception handlers are searched in order for an appropriate match.
- The first handler that yields a match is executed.
- After executing the handler, the control goes to the first statement after the last catch block for that try.

Note: It is possible that arguments of several catch statements match the type of an exception.

In such cases, the first handler that matches the exception type is executed.

Program to throw multiple exceptions and define multiple catch statement.

```
#include<iostream>
using namespace std;
void num(int k) {
    try
    {
        if(k==0) throw k;
        else
        if(k>0) throw 'P';
        else
        if(k<0) throw .0;
        cout<<"*** try block ***\n";
    }
    catch(char g)    {
        cout<<"Caught a positive value \n";
    }
    catch(int j)     {
        cout<<"Caught a null value \n";
    }
    catch(double f) {
        cout<<"Caught a negative value \n";
    }
    cout<<"*** try catch ***\n \n";
}
int main() {
    cout<<"Demo of Multiple catches\n";
    num(0);
    num(5);
    num(-1);
    return 0;
}
```

Catching multiple exceptions

- ◆ Sometimes it is not possible to predict all possible types of exceptions and therefore not able to design independent catch handlers to catch them.
- ◆ In such situation a single catch block is used to catch the exception thrown by the multiple throw statements

Syntax:

```
catch (...)  
{  
    // statements for processing all exceptions.  
}
```

```
#include<iostream>  
using namespace std;  
void num(int k)  
{  
    try  
    {  
        if(k==0) throw k;  
        else  
        if(k>0) throw 'P';  
        else  
        if(k<0) throw .0;  
        cout<<"*** try block ***\n";  
    }  
    catch(...)  
    {  
        cout<<"Caught an exception \n";  
    }  
}  
int main()  
{  
    cout<<"Demo of Multiple catches\n";  
    num(0);  
    num(5);  
    num(-1);  
    return 0;  
}
```

here we have num class if k=0 we are going to throw null exception

if k>0 we are going to throw a character exception

if k<0 we are going to throw a double exception

Re-Throwing exception:

An exception is thrown from the catch block is known as the re-throwing exception.

We have seen a throw statement in try block, if we write throw statement in catch block then it is called re-throwing exception.

It can be simply invoked by throw without arguments.

Rethrown exception will be caught by newly defined catch statement.

```
#include<iostream>
using namespace std;
void divide(double x, double y) {
try{
if(y==0)
    throw y;
else
    cout<<"Division= "<<x/y;
}
catch(double) {
    cout<<"\nException inside function\n";
    throw;
}
}

int main() {
try
{
    divide(10.5,2.0);
    divide(20.5,0.0);
}
catch(double)
{
    cout<<"Exception inside main function";
}
return 0;
}
```

in main divide func is called with 2 args so it check y=0 if not it performs division operation. next divide func is called with(20.5, 0.0) now y=0 ,it throws y as exception control will come to catch block it displays Exception inside function and next stmt is throw and this throw stmt throws exception to function call for that it needs catch block in main function and it gets executed.
