

OBJECT ORIENTED PROGRAMMING IN C++

Content

UNIT-1

1. Introduction to different programming paradigm	2-7
2. Characteristics of OOPs	7-10
3. Class and objects	10-11
4. Data member and member function	11-13
5. Structure in C++	13-16
6. Access specifier	16-17
7. Defining member function inside and outside of class	17-18
8. Array of object	19-24

UNIT-2

1. Concept of reference	25-27
2. Dynamic memory allocation using new and delete operators	27-31
3. Inline functions	31-35
4. Function overloading	35-40
5. Function with default arguments	40-41
6. Constructors and destructors	41-53
7. Friend function and classes	53-55
8. Using this pointer	55-57

UNIT-3

1. Inheritance	58-60
2. Types of inheritance	61-68
3. Multiple inheritance	68-71
4. Virtual base class	72-74
5. Function overriding	74-78
6. Abstract class and pure virtual function	78-81

UNIT-4

1. Constant data member and member function	82-86
2. Static data member and member function	86-88
3. Polymorphism	88-92
4. Operator overloading	92-99
5. Dynamic binding and virtual function	99-104

UNIT-5

1. Exception handling	105-113
2. Template(Generics in c++)	113-124
3. Stream class	124-130
4. File handling	130-133

UNIT – 1

❖ Introduction to different programming paradigm:

1. Programming Approches /(Methodology)

2. Programming Langauages

➤ *Sw Development Life Cycle (SDLC) Steps/Phase*

1. Requirements Gethring/ Understandings inputs values
2. Design /Blue Print-Archict(Flow Charts,DFDs,Algorithms, etc)
3. Implementation (Usage of Programming Languages like C , C++, Java, Python, ML etc)
4. Testing
5. delevery(end user)

➤ *Programming Approches /(Methodology)*

Approch/(Methodology): - It is a way to prepare blue print for solution of a give Problem.

Blueprint:- Path (Design ---Archict)--->Approches

Examples: -

1. Procedural Oriented Programming Approach (PoA): EX-- C , C++ Language only

2. Object Oriented Programming Approach (Oops): EX---C++, Java, ML, Pyhton...etc

1. Procedural Oriented Programming Approach (PoA):-

Procedural Oriented Programming is a **programming** language that follows a step-by-step approach to break down a task into a collection of variables and routines (or subroutines) through a sequence of instructions

```
int getFactorial(int x, int y)    // A procedure or Sub Function
{
    int sum = Logic of Factorial; //10 Lines
    return sum;
}

void main()                    // Main Procedure/function
{
    //logic of problem
    ..
    ..
    ..
    int num;
    printf("Enter the Number");
    scanf("%d",&num);           // get the Number from key board;

    int result = getFactorial(num); // procedure Calling
    // result var is Used in other Logic
    // result Var is used to complete the task of Any Other Logic
    .....
    .....
    //logic of program(more lines)
```

```

int result2 = getFactorial(num); // procedure Calling
    // result var is Used in other Logic
    // result Var is used to complete the task of Any Other Logic
    .....
    .....
    //logic of program(more lines)
}

```

Drawbacks of Procedural Approach :-

1. In procedural Approach focus on "how to do? ie. (Writing functions/procedures)" instead of Data/Information;
2. Data is not secured. ie. All data are publicly Accessible.
3. Data can not reused ie. write once use any where.

2. OOPS Approach(Object Oriented Programming Approach):-

In the object-oriented approach, the focus is on capturing the structure and behavior of information systems into small modules that combines both data and process. The main aim of Object Oriented Design (OOD) is to improve the quality and productivity of system analysis and design by making it more usable

Its A collection of concepts:

- 1) Class and Object
- 2) Encapsulation
- 3) Abstraction
- 4) inheritance
- 5) Polymorphism

➤ Programming Language

It is a medium to Implement a Solution for a Given Problem.

Example:

1. **C Programming :-** it prepares a blue print using Procedural Approach
2. **C++ programming :-** it prepares a blue print using Procedural Approach and OOPS Approach.
 - C compiler and C++ compiler/ **IDE(Integrated Development Environment)** is used for Develop C and C++ programs

Source File:- In which we write the Source Code in a particular Language.

eg.

XYZ.c with C extension is called C Source File

XYZ.cpp with cpp extension is called C++ Source file.

XYZ.java with java extension is called Java Source file.

xyz.html with Html extension is called Html source file.

C source file structure:

```
// header files inclusion
```

```
#include<stdio.h>
```

```
.....
```

```

.....
.....
//globle Data Var Declaretion;
....
....
void main()
{
    // executable statements
    printf("Enter the value of x :");
    scanf("%d",&x);

    printf("Value of x = %d",x);
}

// sub function/sub-procedure
returnType funName(parameter list 0 or more)
{
    ---
    ---
    ---
}

```

Save this as Demo.c

C++ Source file structure:

```

// Header files
#include<iostream>      // #include<stdio.h> in C source file
#include<conio.h>
...
...
...
using namespace std;    // Must to write This Line
int main()
{
    int x,y;             // Var Declaration

    //input from keyboard
    cout<<"Enter the value of x:"<<endl;    // endl is used to enter a new line
    cin>>x;// 10
    cout<<"Enter the value of y:";
    cin>>y;// 15

    cout<<"Value of x ="<<x;

```

```

cout<<"Value of y ="<<y;

int cTotal = getTotal(x,y);          // procedure Call/Function Call
getTotal(x,y);                      // procedure Call/Function Call
cout<<"Total of two Vars ="<<cTotal<<endl;
return 0;                          // this return line must be the Last line in Main Function always
}

void getTotal(int x,int y)
{
    int sum = x+y;
    cout<<"Total of two Var ="<<sum<<endl;
}

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

```

Save this as Demo.c

➤ *Programming constrains*

1. tokens (Keywords,var.idendifiers,constants..etc)
2. Data Types (Which defines the Type of Data eg. int long int, float...etc)
3. Operators (Arithmetic,relational,conditional,ternary ...etc)
4. Control Stmts (selection(If else, Switch Case..), iteration (For, while, do While), jump stmt (return,goto,break, Continue) etc)
5. Arrays/ functions
6. User Define Data types (Structure/ Union)
7. pointers
8. OOps Concepts(Later Discuss in Detail)

➤ *Arrays and Functions*

Array:- Array is a indexed based fixed length data structure or container .which hold similer kinds of data values in linear way.

characteristics of an Array:

Limitation:

1. Fixed Length (Starting index= 0 and Last index = Size-1)
2. Contains Homogenous Elements/Values

Difference between Var and an Array? (in concern of Memory)

eg. int x1 = 10; int x2 = 20;..... 5 Values (Low Performance)
int x[5]; (High Performance)

syntax:

dataType ArrayName[size]; // Array Declaration ad Definition

Note:-

Data Types: - Which define the type of data;

1. Inbuilt Data Types (Primitive):- (int short long float, double, char, boolean etc)
2. User Define Data Type(non-primitive):- (Array, string ,structure,Union, class..etc)

Eg:-

```
int x[10];           // Array Declaration ad Definition of primitive int Values
string names[5] ;    // Non-Primitive Values Array
```

Array of structurs:

```
struct student
{
    char name[10];
    int RollNo;

} s1, s2 s3, s4,s5.....n;    // Objects

student abc[5];           // Non primitive values Array
```

Operations on Arrays:

1. Insertion
2. Update
3. Delete
4. Retrival (Get the Values from an Array)

Eg:

```
int x[5];
1. way of initialization
    x[0] = 10;
    x[1] = 20;
    x[2] = 50;
    x[3] = 40;
    x[4] = 30;
    //Hard Coding Way
2 Way: By using Loop (For , while, Do while)
for(int i = 0; i < 5; i++)
{
    x[i] = Value;    //you can use Keyboard
}

cout<<x[3];    // 40 retrival Operation
```

```
x[3] = new Value;  
x[3] = 100;      // update Operation  
cout<<x[3];      // 100
```

```
x[5] = 200;      // error
```

Array can Classified as Follows:

1- D Array

eg:- int x[5];

2-D Array

eg :- int x[m][n]; //total no of elements = m*n

```
int x[3][3];  
/* x[0][0] x[0][1] x[0][2]  
   x[1][0] x[1][1] x[1][2]  
   x[2][0] x[2][1] x[2][2] */
```

```
//initialization 2D array  
for(int i = 0; i < 3; i++)  
{  
    for(int j = 0; j < 3; j++)  
    {  
        x[i][j] = Value; youcan use Keyboard;  
    }  
}
```

❖ Characteristics of OOPs:

- 1. Class**
- 2. Objects**
- 3. Encapsulation**
- 4. Abstraction**
- 5. Polymorphism**
- 6. Inheritance**
- 7. Dynamic Binding**
- 8. Message Passing**

Class:-The building block of C++ that leads to Object-Oriented programming is a Class. It is a user-defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class. A class is like a blueprint for an object.

For Example: Consider the Class of Cars. There may be many cars with different names and brand but all of them will share some common properties like all of them will have 4 wheels, Speed Limit, Mileage range etc. So here, Car is the class and wheels, speed limits, mileage are their properties.

- A Class is a user-defined data-type which has data members and member functions.

- Data members are the data variables and member functions are the functions used to manipulate these variables and together these data members and member functions define the properties and behaviour of the objects in a Class.
- In the above example of class Car, the data member will be speed limit, mileage etc and member functions can apply brakes, increase speed etc.

We can say that a **Class in C++** is a blue-print representing a group of objects which shares some common properties and behaviours.

Object:- An Object is an identifiable entity with some characteristics and behaviour. An Object is an instance of a Class. When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated.

```
class person
{
    char name[20];
    int id;
public:
    void getdetails(){}
};

int main()
{
    person p1; // p1 is a object
}
```

Object take up space in memory and have an associated address like a record in pascal or structure or union in C.

When a program is executed the objects interact by sending messages to one another.

Each object contains data and code to manipulate the data. Objects can interact without having to know details of each other's data or code, it is sufficient to know the type of message accepted and type of response returned by the objects.

Encapsulation:- In normal terms, Encapsulation is defined as wrapping up of data and information under a single unit. In Object-Oriented Programming, Encapsulation is defined as binding together the data and the functions that manipulate them.

Consider a real-life example of encapsulation, in a company, there are different sections like the accounts section, finance section, sales section etc. The finance section handles all the financial transactions and keeps records of all the data related to finance. Similarly, the sales section handles all the sales-related activities and keeps records of all the sales. Now there may arise a situation when for some reason an official from the finance section needs all the data about sales in a particular month. In this case, he is not allowed to directly access the data of the sales section. He will first have to contact some other officer in the sales section and then request him to give the particular data. This is what encapsulation is. Here the data of the sales section and the employees that can manipulate them are wrapped under a single name "sales section".



Encapsulation also leads to *data abstraction or hiding*. As using encapsulation also hides the data. In the above example, the data of any of the section like sales, finance or accounts are hidden from any other section.

Abstraction:-Data abstraction is one of the most essential and important features of object-oriented programming in C++. Abstraction means displaying only essential information and hiding the details. Data abstraction refers to providing only essential information about the data to the outside world, hiding the background details or implementation.

Consider a real-life example of a man driving a car. The man only knows that pressing the accelerators will increase the speed of the car or applying brakes will stop the car but he does not know about how on pressing accelerator the speed is actually increasing, he does not know about the inner mechanism of the car or the implementation of accelerator, brakes etc in the car. This is what abstraction is.

- **Abstraction using Classes:** We can implement Abstraction in C++ using classes. The class helps us to group data members and member functions using available access specifiers. A Class can decide which data member will be visible to the outside world and which is not.
- **Abstraction in Header files:** One more type of abstraction in C++ can be header files. For example, consider the pow() method present in math.h header file. Whenever we need to calculate the power of a number, we simply call the function pow() present in the math.h header file and pass the numbers as arguments without knowing the underlying algorithm according to which the function is actually calculating the power of numbers.

Polymorphism:-The word polymorphism means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form.

A person at the same time can have different characteristic. Like a man at the same time is a father, a husband, an employee. So the same person posses different behaviour in different situations. This is called polymorphism.

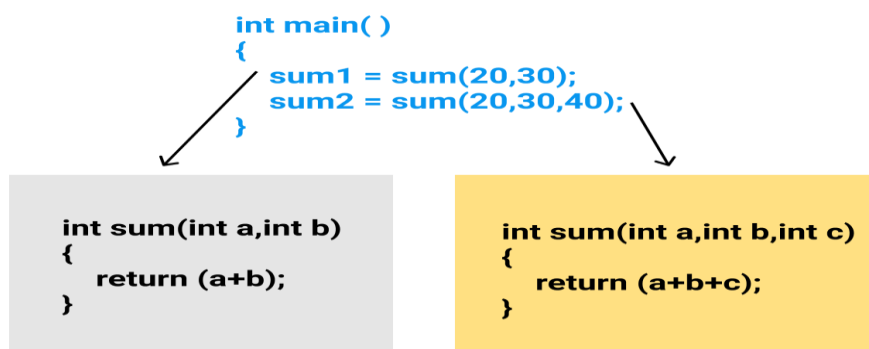
An operation may exhibit different behaviours in different instances. The behaviour depends upon the types of data used in the operation.

C++ supports operator overloading and function overloading.

- **Operator Overloading:** The process of making an operator to exhibit different behaviours in different instances is known as operator overloading.
- **Function Overloading:** Function overloading is using a single function name to perform different types of tasks.

Polymorphism is extensively used in implementing inheritance.

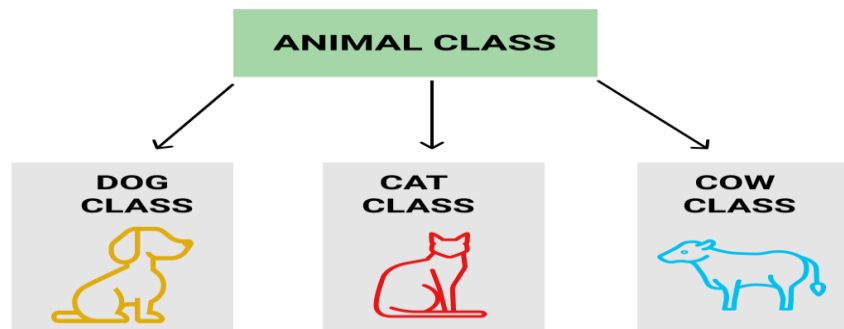
Example: Suppose we have to write a function to add some integers, some times there are 2 integers, some times there are 3 integers. We can write the Addition Method with the same name having different parameters, the concerned method will be called according to parameters.



Inheritance:- The capability of a class to derive properties and characteristics from another class is called Inheritance. Inheritance is one of the most important features of Object-Oriented Programming.

- **Sub Class:** The class that inherits properties from another class is called Sub class or Derived Class.
- **Super Class:** The class whose properties are inherited by sub class is called Base Class or Super class.
- **Reusability:** Inheritance supports the concept of "reusability", i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.

Example: Dog, Cat, Cow can be Derived Class of Animal Base Class.



Dynamic Binding: In dynamic binding, the code to be executed in response to function call is decided at runtime. C++ has virtual functions to support this.

Message Passing: Objects communicate with one another by sending and receiving information to each other. A message for an object is a request for execution of a procedure and therefore will invoke a function in the receiving object that generates the desired results. Message passing involves specifying the name of the object, the name of the function and the information to be sent.

❖ Class and Object:

Class:-

- Its a Redefind Vesion of Structure in C++.
- we use "class" Keyword to define a class instead of "Struct".
- Every member in a class is "private" by default instead of "public".
- in a class we can define behaviour as a function but not in structure.
- it's A real world Entity and also represent to a Catagory, which about we can think about it only.
- Its A blue Print of a real world Entity.

Object:-

- whatever we are seeing by our eyes that all entities are Objects...we can feel their behaviour and can also touch them.
- Objects are real actors.
- They have their existance.

Technical Definition of class:

its a collection of properties/attributes/Vars and Function/Behaviour/Methods.

Properties are refered as "Data Members" and Functions are refered as "Member Function" in side a class Boundry.

ie. class a collection of Data Members and Member Functions.

Technical Definition of object:

An Object is an identifiable entity with some characteristics and behaviour. An Object is an instance of a Class. When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated.

eg :-

Student --- class ---but we can not touch but we can feel & imagin only about student Entity.

Class:

class Student

```
{
    // Data Members
    int rollNo;
    String Name;
    String Section
    int year;
    int age;
    string address;
    int mobile;
    ....        // As per Requirement.
    ....

    // Member Functions
    void getInformation()
    {
        // Logic to get values from key board
    }

    void printStdInformation()
    {
        // Logic for print information
    }
};    // end of class
```

objects:

Student s1, s2, s3,.....sn ; //Number of Objects as per Requirement
Shubham -----Object of Student

❖ Data member and member function:

"Data Member" and "Member Functions" are the new names/terms for the members of a class, which are introduced in C++ programming language.

The variables which are declared in any class by using any (like int, char, float etc) or derived data type (like class, structure, pointer etc.) are known as **Data Members**. And the functions which are declared either in private section or public section are known as **Member functions**.

There are two **types of data members/member functions in C++**:

1. Private members
2. Public members

1) Private members

The members which are declared in private section of the class (using private access modifier) are known as private members. Private members can also be accessible within the same class in which they are declared.

2) Public members

The members which are declared in public section of the class (using public access modifier) are known as public members. Public members can access within the class and outside of the class by using the object name of the class in which they are declared.

Consider the example:

```
class Test
{
    private:
        int a;
        float b;
        char *name;

        void getA() { a=10; }
        ...;

    public:
        int count;
        void getB() { b=20; }

        ...;
};
```

Here, **a**, **b**, and **name** are the private data members and **count** is a public data member. While, **getA()** is a private member function and **getB()** is public member functions.

C++ program that will demonstrate, how to declare, define and access data members and member functions in a class?

```
#include <iostream>
#include <string.h>
using namespace std;

#define MAX_CHAR 30

//class definition
```

```

class person
{
    //private data members
    private:
        char name [MAX_CHAR];
        int age;

    //public member functions
    public:
        //function to get name and age
        void get(char n[], int a)
        {
            strcpy(name , n);
            age = a;
        }

        //function to print name and age
        void put()
        {
            cout<< "Name: " << name <<endl;
            cout<< "Age: " <<age <<endl;
        }
};

//main function
int main()
{
    //creating an object of person class
    person PER;

    //calling member functions
    PER.get("Manju Tomar", 23);
    PER.put();

    return 0;
}

```

Output:

Name: Manju Tomar
Age: 23

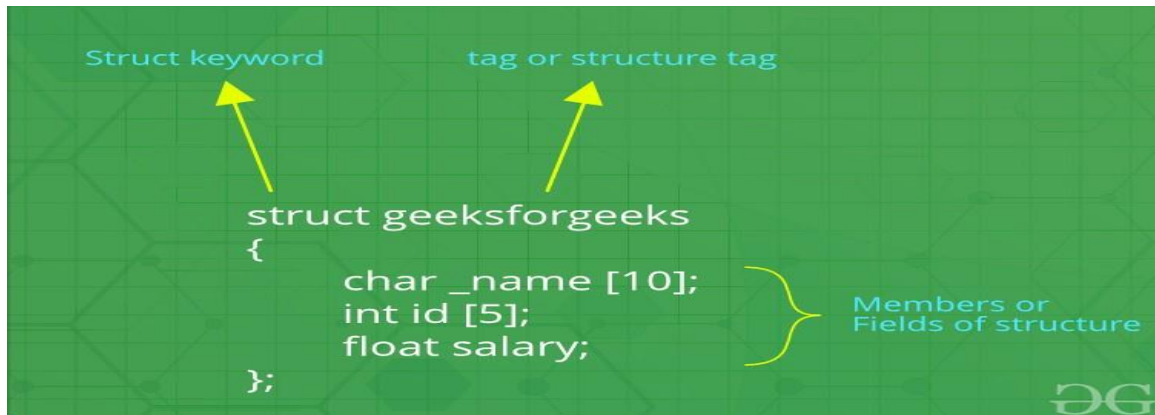
❖ Structure in C++ :

We often come around situations where we need to store a group of data whether of similar data types or non-similar data types.

Unlike Arrays, **Structures in C++** are user defined data types which are used to store group of items of non-similar data types.

What is a structure?

A structure is a user-defined data type in C/C++. A structure creates a data type that can be used to group items of possibly different types into a single type.



Structures in C++

How to create a structure?

The 'struct' keyword is used to create a structure. The general syntax to create a structure is as shown below:

```
struct structureName{
    member1;
    member2;
    member3;
    .
    .
    .
    memberN;
};
```

Structures in C++ can contain two types of members:

- **Data Member:** These members are normal C++ variables. We can create a structure with variables of different data types in C++.
- **Member Functions:** These members are normal C++ functions. Along with variables, we can also include functions inside a structure declaration.

Example:

```
// Data Members
int roll;
int age;
int marks;

// Member Functions
void printDetails()
{
    cout<<"Roll = "<<roll<<"\n";
    cout<<"Age = "<<age<<"\n";
    cout<<"Marks = "<<marks;
}
```

In the above structure, the data members are three integer variables to store *roll number, age and marks* of any student and the member function is *printDetails()* which is printing all of the above details of any student.

How to declare structure variables?

A structure variable can either be declared with structure declaration or as a separate declaration like basic types.

// A variable declaration with structure declaration.

```
struct Point
{
    int x, y;
} p1; // The variable p1 is declared with 'Point'
```

// A variable declaration like basic data types

```
struct Point
{
    int x, y;
};
```

```
int main()
```

```
{
    struct Point p1; // The variable p1 is declared like a normal variable
}
```

Note: In C++, the struct keyword is optional before in declaration of a variable. In C, it is mandatory.

How to initialize structure members?

Structure members **cannot be** initialized with declaration. For example the following C program fails in compilation.

But is considered correct in C++11 and above.

```
struct Point
{
    int x = 0; // COMPILER ERROR: cannot initialize members here
    int y = 0; // COMPILER ERROR: cannot initialize members here
};
```

The reason for above error is simple, when a datatype is declared, no memory is allocated for it. Memory is allocated only when variables are created.

Structure members **can be** initialized using curly braces '{}'. For example, following is a valid initialization.

```
struct Point
{
    int x, y;
};
```

```
int main()
{
    // A valid initialization. member x gets value 0 and y
    // gets value 1. The order of declaration is followed.
    struct Point p1 = {0, 1};
}
```

How to access structure elements?

Structure members are accessed using dot (.) operator.

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

struct Point {
    int x, y;
};

int main()
{
    struct Point p1 = { 0, 1 };

    // Accessing members of point p1
    p1.x = 20;
    cout << "x = " << p1.x << ", y = " << p1.y;

    return 0;
}
```

Output:

X=20, y=1

❖ Access specifiers :

In C++, there are three access specifiers:

- public - members are accessible from outside the class
- private - members cannot be accessed (or viewed) from outside the class
- protected - members cannot be accessed from outside the class, however, they can be accessed in inherited classes.

In the following example, we demonstrate the differences between public and private members:

```
class MyClass {
    public: // Public access specifier
    int x; // Public attribute
    private: // Private access specifier
    int y; // Private attribute
};
```

```
int main() {
    MyClass myObj;
```



```

myObj.x = 25; // Allowed (public)
myObj.y = 50; // Not allowed (private)
return 0;
}

```

If you try to access a private member, an error occurs:
error: y is private

❖ Member function inside and outside of class:

Member Function:

A function , which is declare and / or Defined inside the class Boundry is called Member Function.

we have two ways to define or declare member functions in C++

1st way:

declare and define member function inside a class boundry

```

int x;      // Declaration
x=10;      // Definition/ initialization
int x=10;   // Declaration & Definition

```

Example:

```

class Demo
{
    private:
        int x,y;          // Data Members by Default "Private" in side a class.
        void showAddition()
        {
            cout<<"Sum od two Values = "<<(x+y)<<endl;
        }
    public:
        void getData()// function is Declared and Define here (Calling Function)
        {
            cout<<"Enter Value of X= "<<endl;
            cin>>x;
            cout<<"Enter Value of y= "<<endl;
            cin>>y;
            //showAddition();// calling other Member function in getData() function without Object.
        }
};

int main()          //procedural Funtion
{
    Demo d;         // Object Creation of a class Demo
}

```

```

d.getData();      //called Function
d.showAddition(); //error
return 0;

}

```

2nd way:

Declare a member function prototype inside a class boundary and define function outside the class boundary using scope resolution Operator (::)

```

class Demo
{
private:

int x,y;                                // Data Members by Default "Private"

public:                                  // Member Functions // By Default private

void getData();// Member Function Prototype Declaration

void showAddition()
{
    cout<<"Sum of two Values = "<<(x+y)<<endl;
}

}; // end of class

```

::<-----Scope Resolution Operator

// FunctionReturn Type ClassName::FunctionName(Arguments List) ---- Syntax

```

void Demo::getData() // Member Function
{
    cout<<"Enter Value of X= "<<endl;
    cin>>x;
    cout<<"Enter Value of y= "<<endl;
    cin>>y;
}

int main()                // procedural Function
{
    Demo d;                // required a Object
    d.getData();
    d.showAddition();      // member Function

    return 0;
}

```

❖ Array of objects:

- Like array of other user-defined data types, an array of type class can also be created.
- The array of type class contains the objects of the class as its individual elements.
- Thus, an array of a class type is also known as an array of objects.
- An array of objects is declared in the same way as an array of any built-in data type.

Syntax:

class_name array_name [size] ;

Example:

```
#include <iostream>

class MyClass {
    int x;
public:
    void setX(int i) { x = i; }
    int getX() { return x; }
};

void main()
{
    MyClass obs[4];
    int i;

    for(i=0; i < 4; i++)
        obs[i].setX(i);

    for(i=0; i < 4; i++)
        cout << "obs[" << i << "].getX(): " << obs[i].getX() << "\n";

    getch();
}
```

Output:

```
obs[0].getX(): 0
obs[1].getX(): 1
obs[2].getX(): 2
obs[3].getX(): 3
```

➤ *Different methods to initialize the Array of objects with parameterized constructors:-*

1.Using malloc(): To avoid the call of non-parameterised constructor, use malloc() method. “malloc” or “memory allocation” method in C++ is used to dynamically allocate a single large block of memory with the specified size. It returns a pointer of type void which can be cast into a pointer of any form.

Example:-

```
#include <iostream>
#define N 5
```

```

using namespace std;

class Test {
    // private variables
    int x, y;

public:
    // parameterised constructor
    Test(int x, int y)
    {
        this->x = x;
        this->y = y;
    }

    // function to print
    void print()
    {
        cout << x << " " << y << endl;
    }
};

int main()
{
    // allocating dynamic array
    // of Size N using malloc()
    Test* arr = (Test*)malloc(sizeof(Test) * N);

    // calling constructor
    // for each index of array
    for (int i = 0; i < N; i++) {
        arr[i] = Test(i, i + 1);
    }

    // printing contents of array
    for (int i = 0; i < N; i++) {
        arr[i].print();
    }

    return 0;
}

```

Output:

```

0 1
1 2
2 3
3 4
4 5

```

2.Using new keyword: The new operator denotes a request for memory allocation on the Heap. If sufficient memory is available, the new operator initializes the memory and returns the address of the newly allocated and initialized memory to the pointer variable. Here, pointer-variable is the pointer of type data-type. Data-type could be any built-in data type including array or any user-defined data types including structure and class.

For dynamic initialization new keyword require non parameterised constructor if we add a parameterised constructor. So we will use a dummy constructor for it.

Example:-

```
#include <iostream>
#define N 5

using namespace std;

class Test {
    // private variables
    int x, y;

public:
    // dummy constructor
    Test() {}

    // parameterised constructor

    Test(int x, int y)
    {
        this->x = x;
        this->y = y;
    }

    // function to print
    void print()
    {
        cout << x << " " << y << endl;
    }
};

int main()
{
    // allocating dynamic array
    // of Size N using new keyword
    Test* arr = new Test[N];

    // calling constructor
    // for each index of array
    for (int i = 0; i < N; i++) {
        arr[i] = Test(i, i + 1);
    }
}
```

```

        // printing contents of array
        for (int i = 0; i < N; i++) {
            arr[i].print();
        }

        return 0;
    }

```

Output:

```

0 1
1 2
2 3
3 4
4 5

```

If we don't use the dummy constructor compiler would show the error given below

Compiler Error:

error: no matching function for call to 'Test::Test()'
 Test *arr=new Test[N];

3.Using Double pointer (pointer to pointer concept): A pointer to a pointer is a form of multiple indirections, or a chain of pointers. Normally, a pointer contains the address of a variable. When we define a pointer to a pointer, the first pointer contains the address of the second pointer, which points to the location that contains the actual value as shown below.

Here we can assign a number of blocks to be allocated and thus for every index we have to call parameterised constructor using the new keyword to initialise.

Example:-

```

#include <iostream>
#define N 5

using namespace std;

class Test {
    // private variables
    int x, y;

public:
    // parameterised constructor

    Test(int x, int y)
        : x(x), y(y)
    {
    }

    // function to print
    void print()
    {
        cout << x << " " << y << endl;
    }
}

```

```

};

int main()
{
    // allocating array using
    // pointer to pointer concept
    Test** arr = new Test*[N];

    // calling constructor for each index
    // of array using new keyword
    for (int i = 0; i < N; i++) {
        arr[i] = new Test(i, i + 1);
    }

    // printing contents of array
    for (int i = 0; i < N; i++) {
        arr[i]->print();
    }

    return 0;
}

```

Output:

```

0 1
1 2
2 3
3 4
4 5

```

4.Using Vector of type class: Vector is one of the most powerful element of Standard Template Library makes it easy to write any complex codes related to static or dynamic array in an efficient way. It takes one parameter that can be of any type and thus we use our Class as a type of vector and push Objects in every iteration of the loop.

Vectors are same as dynamic arrays with the ability to resize itself automatically when an element is inserted or deleted, with their storage being handled automatically by the container. Vector elements are placed in contiguous storage so that they can be accessed and traversed using iterators. In vectors, data is inserted at the end.

Example:-

```

#include <iostream>
#include <vector>
#define N 5

using namespace std;

class Test {
    // private variables
    int x, y;

public:

```

```

// parameterised constructor

Test(int x, int y)
: x(x), y(y)
{
}

// function to print
void print()
{
    cout << x << " " << y << endl;
}

};

int main()
{
    // vector of type Test class
    vector<Test> v;

    // inserting object at the end of vector
    for (int i = 0; i < N; i++)
        v.push_back(Test(i, i + 1));

    // printing object content
    for (int i = 0; i < N; i++)
        v[i].print();

    return 0;
}

```

Output:

```

0 1
1 2
2 3
3 4
4 5

```


UNIT-2

❖ Concepts of reference

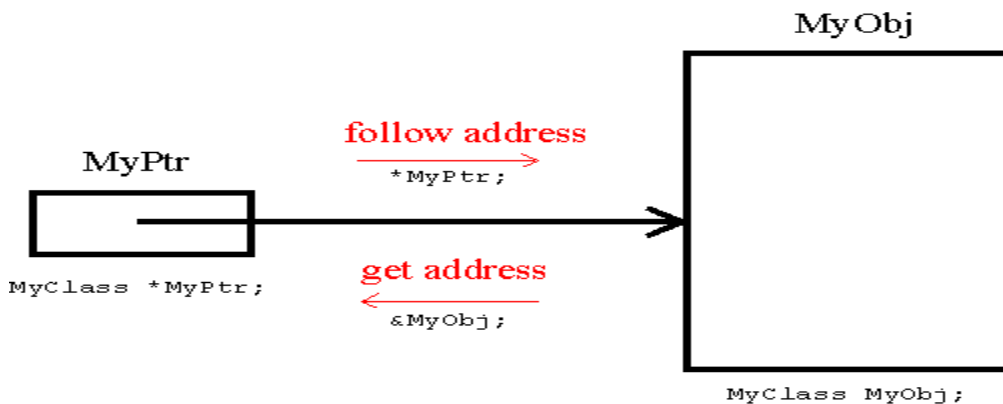
OO Concept: Pointers & References:

Pointers and references are not part of OO, languages like JAVA don't have pointers. However, they are so fundamental to the way C++ supports OO concepts that they deserve a topic.

Object, have data members, and so, like built-in data types, occupy memory. In order to access any data type its necessary to know whereabouts in memory it is. The compiler and linker can decide addresses for anything defined at compile time. However OO programs are dynamic, with objects being created at execution time. Then, in order to access them, its necessary to go via some type of pointer that holds the memory address. C++ has two types: Pointers and References.

➤ Pointers:

A Pointer holds the address of an object



Pointers hold addresses of other data types. The **address operator &** is used to form an address and the **dereference operator *** is used to follow the address. For example:-

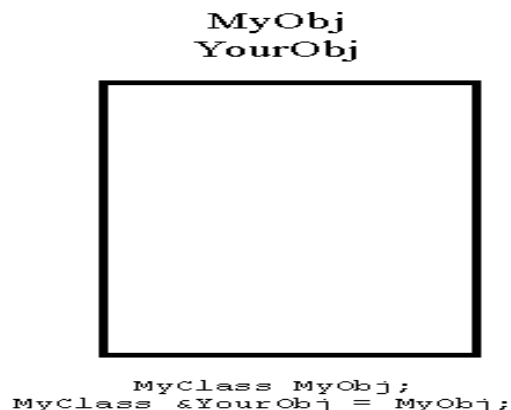
```
Int_t MyInt = 2;  
Int_t *MyIntPtr = &MyInt;  
*MyIntPtr = 4;
```

MyIntPtr is a pointer to MyInt. By dereferencing the pointer MyInt is retrieved. In this example it is then assigned to so MyInt receives the value 4.

Pointers know the size of the data type they point to. Where they point to arrays of the same type, the **increment operator ++** and the **decrement operator --** step the address by this size

➤ References:

References:
another name for
the same object



A reference is simply an alternative identifier for a variable or object. It's rather like the FORTRAN EQUIVALENCE, except that it can be defined at execution time. A reference is a type of pointer and when it is declared it must be initialised with the variable it is a reference to, for example:-

```
Int_t &MyInt = YourInt;
```

This makes MyInt a reference to YourInt. From now on the two identifiers are equivalent. Expressions such as:-

```
YourInt = 7;  
are identical to:-  
MyInt = 7;
```

Behind the scenes, the compiler knows that it only has a pointer and has to do some dereferencing, but this is all automatic - it's better to forget about this trickery and just think of it as an alternative name.

Once defined, a reference cannot be reassigned a new value; any attempt to modify it is interpreted as an attempt to modify the variable it's a reference to, and trying to store a new address:-

```
YourInt = &HerInt;
```

is rejected by the compiler as an attempt to store a pointer in a non-pointer variable.

Using Pointers and References in Function Calls

References cannot do anything that pointers cannot do, however, they can make some code a lot clearer by removing lots of explicit dereferences. This is particularly true when passing modifiable arguments to a function. Consider this function:-

```
void ChangelT(int num) {  
    num = 5;  
}
```

and what happens if we do:-

```
int MyNum = 1;  
ChangelT(MyNum);  
cout << "MyNum = " << MyNum << endl;
```

A FORTRAN programmer could be forgiven for thinking that MyNum is now 5 because FORTRAN copies arguments by reference, but C++ **copies arguments by value - changes within a function are not passed back**. ChangelT can change num as much as it likes, it's just a local copy that changes and it won't be returned. If a function is to change its arguments then one way to do it is:-

```
void ChangelT(int *num) {  
    *num = 5;  
}
```

```
int MyNum = 1;  
ChangelT(&MyNum);  
cout << "MyNum = " << MyNum << endl;
```

Now we pass the address of MyNum. ChangelT has a pointer with which to modify MyNum. It works, but it's ugly, littering the code with & and *, but C++ offers a neater alternative:-

```
void ChangelT(int &num) {  
    num = 5;  
}
```

```
int MyNum = 1;  
ChangelT(MyNum);  
cout << "MyNum = " << MyNum << endl;
```

num is declared to be a reference, so when MyNum is passed its address is used to initialise num, which now just becomes just another name for MyInt. So argument passing acts as in FORTRAN.

Not only does argument passing this way make the code easier to read, it can save a lot of time too. As we have seen, the default is to copy by value, and this applies regardless of whether its just one byte or some enormous object. C++ is quite prepared to call a copy of an object into existence just for the function call. By making functions pass by reference avoids this and leads to another golden rule **Always pass objects to functions as pointers or references never by value.**

Functions can return pointers and references and these can then be used on the left hand side of an assignment statement. Consider:-

```
int& HisNum() {  
    int num;  
    return num;  
}
```

this function returns an address as a reference which can then be assigned to:-

```
HisNum() = 6;
```

The HisNum function is called and returns an address which is used as the initialisation value of an unnamed reference to int, which is now just another name for num that is then assigned to. Actually this example has a dreadful error: num is a local variable that ceases to exist once the function exits so the assignment is writing to a bad memory address. When writing functions that return address and references, be very careful that the address remains valid once the function has returned.

References are less powerful than pointers:

- 1) Once a reference is created, it cannot be later made to reference another object; it cannot be resealed. This is often done with pointers.
 - 2) References cannot be NULL. Pointers are often made NULL to indicate that they are not pointing to any valid thing.
 - 3) A reference must be initialized when declared. There is no such restriction with pointers
- Due to the above limitations, references in C++ cannot be used for implementing data structures like Linked List, Tree, etc. In Java, references don't have the above restrictions and can be used to implement all data structures. References being more powerful in Java is the main reason Java doesn't need pointers.

References are safer and easier to use:

- 1) Safer: Since references must be initialized, wild references like **wild pointers** are unlikely to exist. It is still possible to have references that don't refer to a valid location (See questions 5 and 6 in the below exercise)
- 2) Easier to use: References don't need a dereferencing operator to access the value. They can be used like normal variables. '&' operator is needed only at the time of declaration. Also, members of an object reference can be accessed with dot operator ('.'), unlike pointers where arrow operator (->) is needed to access members.

Together with the above reasons, there are few places like the copy constructor argument where pointer cannot be used. Reference must be used to pass the argument in the copy constructor. Similarly, references must be used for overloading some operators like ++.

❖ dynamic memory allocation using new and delete operators

Dynamic memory allocation in C/C++ refers to performing memory allocation manually by programmer. Dynamically allocated memory is allocated on **Heap** and non-static and local variables get memory allocated on **Stack**.

What are applications?

- One use of dynamically allocated memory is to allocate memory of variable size which is not possible with compiler allocated memory except variable length arrays.
- The most important use is flexibility provided to programmers. We are free to allocate and deallocate memory whenever we need and whenever we don't need anymore. There are many cases where this flexibility helps. Examples of such cases are Linked List, Tree, etc.

How is it different from memory allocated to normal variables?

For normal variables like "int a", "char str[10]", etc, memory is automatically allocated and deallocated. For dynamically allocated memory like "int *p = new int[10]", it is programmers responsibility to deallocate memory when no longer needed. If programmer doesn't deallocate memory, it causes memory leak (memory is not deallocated until program terminates).

How is memory allocated/deallocated in C++?

C uses malloc() and calloc() function to allocate memory dynamically at run time and uses free() function to free dynamically allocated memory. C++ supports these functions and also has two operators **new** and **delete** that perform the task of allocating and freeing the memory in a better and easier way.

➤ new operator

The new operator denotes a request for memory allocation on the Free Store. If sufficient memory is available, new operator initializes the memory and returns the address of the newly allocated and initialized memory to the pointer variable.

- **Syntax to use new operator:** To allocate memory of any data type, the syntax is:
 - pointer-variable = **new** data-type;
- Here, pointer-variable is the pointer of type data-type. Data-type could be any built-in data type including array or any user defined data types including structure and class.

Example:

```
// Pointer initialized with NULL
// Then request memory for the variable
int *p = NULL;
p = new int;

OR

// Combine declaration of pointer
// and their assignment
int *p = new int;
```

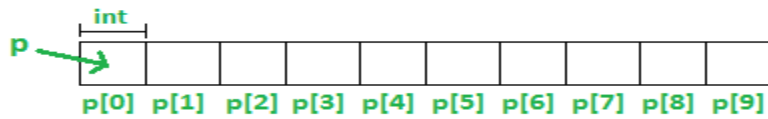
- **Initialize memory:** We can also initialize the memory using new operator:
 - pointer-variable = **new** data-type(value);
 - **Example:**
 - int *p = **new** int(25);
 - float *q = **new** float(75.25);
 - **Allocate block of memory:** new operator is also used to allocate a block(an array) of memory of type data-type.
 - pointer-variable = **new** data-type[size];
- where size(a variable) specifies the number of elements in an array.

Example:

```
int *p = new int[10]
```

Dynamically allocates memory for 10 integers continuously of type int and returns pointer to the first element of the sequence, which is assigned to p(a pointer). p[0] refers to first element, p[1]

refers to second element and so on.



Normal Array Declaration vs Using new

There is a difference between declaring a normal array and allocating a block of memory using new. The most important difference is, normal arrays are deallocated by compiler (If array is local, then deallocated when function returns or completes). However, dynamically allocated arrays always remain there until either they are deallocated by programmer or program terminates.

What if enough memory is not available during runtime?

If enough memory is not available in the heap to allocate, the new request indicates failure by throwing an exception of type **std::bad_alloc**, unless “**nothrow**” is used with the new operator, in which case it returns a NULL. Therefore, it may be good idea to check for the pointer variable produced by new before using it program.

```
int *p = new(nothrow) int;
if (!p)
{
    cout << "Memory allocation failed\n";
}
```

➤ delete operator

Since it is programmer's responsibility to deallocate dynamically allocated memory, programmers are provided delete operator by C++ language.

Syntax:

// Release memory pointed by pointer-variable

delete pointer-variable;

Here, pointer-variable is the pointer that points to the data object created by new.

Examples:

```
delete p;
delete q;
```

To free the dynamically allocated array pointed by pointer-variable, use following form of delete:

```
// Release block of memory
// pointed by pointer-variable
delete[] pointer-variable;
```

Example:

```
// It will free the entire array
// pointed by p.
delete[] p;
```

C++ program to illustrate dynamic allocation and deallocation of memory using new and delete

```
#include <iostream>
using namespace std;
int main ()
{
    // Pointer initialization to null
    int* p = NULL;
```

```

// Request memory for the variable
// using new operator
p = new(nothrow) int;
if (!p)
    cout << "allocation of memory failed\n";
else
{
    // Store value at allocated address
    *p = 29;
    cout << "Value of p: " << *p << endl;
}
// Request block of memory
// using new operator
float *r = new float(75.25);

cout << "Value of r: " << *r << endl;

// Request block of memory of size n
int n = 5;
int *q = new(nothrow) int[n];

if (!q)
    cout << "allocation of memory failed\n";
else
{
    for (int i = 0; i < n; i++)
        q[i] = i+1;

    cout << "Value store in block of memory: ";
    for (int i = 0; i < n; i++)
        cout << q[i] << " ";
}
// freed the allocated memory
delete p;
delete r;
// freed the block of allocated memory
delete[] q;

return 0;
}

```

Output:

Value of p: 29

Value of r: 75.25

Value store in block of memory: 1 2 3 4 5

C++ new and delete Operator for Objects

```

#include <iostream>
using namespace std;

```

```

class Student {
    int age;

public:

    // constructor initializes age to 12
    Student() : age(12) {}
    void getAge() {
        cout << "Age = " << age << endl;
    }
};

int main() {

    // dynamically declare Student object
    Student* ptr = new Student();
    // call getAge() function
    ptr->getAge();
    // ptr memory is released
    delete ptr;

    return 0;
}

```

Output

Age = 12

In this program, we have created a Student class that has a private variable age. We have initialized age to 12 in the default constructor Student() and print its value with the function getAge().

In main(), we have created a Student object using the new operator and use the pointer ptr to point to its address.

The moment the object is created, the Student() constructor initializes age to 12.

We then call the getAge() function using the code:

```
ptr->getAge();
```

Notice the arrow operator ->. This operator is used to access class members using pointers

❖ Inline function

What is an Inline function in C++?

One of the major objectives of using functions in a program is to save memory space, which becomes appreciable when a function is likely to be called many times. However, every time a function is called, it takes a lot of extra time in executing tasks such as jumping to the calling function. When a function is small, a substantial percentage of execution time may be spent in such overheads and sometimes maybe the time taken for jumping to the calling function will be greater than the time taken to execute that function.

One solution to this problem is to use macro definitions, commonly known as macros. Preprocessor macros are popular in C, but the major drawback with macros is that they are not really functions and therefore, the usual error checking process does not occur during compilation.

C++ has a different solution to this problem. **To eliminate the time of calls to small functions, C++ proposes a new function called inline function. An inline function is a function that is expanded in line when it is invoked thus saving time. The compiler replaces the function call with the corresponding function code that reduces the overhead of function calls.**

We should note that inlining is only a request to the compiler, not a command. The compiler can ignore and skip the request for inlining. The compiler may not perform inlining in the following circumstances:

- If a function contains a loop. (for, while, do-while)
- If a function is recursive.
- If a function contains static variables.
- If a function contains a switch command or goto statement.
- For a function not returning values, if a return statement exists.

Syntax:

For an inline function, declaration and definition must be done together.

```
inline return-type function-name(parameters)
{
    // function code
}
```

Example:

```
#include <iostream>
using namespace std;
inline int cube(int s)
{
    return s*s*s;
}
inline int inc(int a)
{
    return ++a;
}
int main()
{
    int a = 11;
    cout << "The cube of 3 is: " << cube(3) << "n";
    cout << "Incrementing a " << inc(a) << "n";
    return 0;
}
```

Output:

The cube of 3 is: 27

Incrementing a 12

Explanation:

It is a simple example which shows two inline functions declared using the inline keyword as a prefix.

When to use Inline function?

We can use Inline function as per our needs. Some useful recommendation are mentioned below-

- We can use the inline function when performance is needed.
- We can use the inline function over macros.
- We prefer to use the inline keyword outside the class with the function definition to hide implementation details of the function.

Points to be remembered while using Inline functions

- We must keep inline functions small, small inline functions have better efficiency and good results.
- Inline functions do increase efficiency, but we should not turn all the functions inline. Because if we make large functions inline, it may lead to code bloat and might end up decreasing the efficiency.
- It is recommended to define large functions outside the definition of a class using scope resolution:: operator, because if we define such functions inside a class definition, then they might become inline automatically and again affecting the efficiency of our code.

Advantages of Inline function

- Function call overhead doesn't occur.
- It saves the overhead of a return call from a function.
- It saves the overhead of push/pop variables on the stack when the function is called.
- When we use the inline function it may enable the compiler to perform context-specific optimization on the function body, such optimizations are not possible for normal function calls.
- It increases the locality of reference by utilizing the instruction cache.
- An inline function may be useful for embedded systems because inline can yield less code than the function call preamble and return.

Moving on with this article on Inline function in C++

Limitations of Inline functions

- Large Inline functions cause Cache misses and affect efficiency negatively.
- Compilation overhead of copying the function body everywhere in the code at the time of compilation which is negligible in the case of small programs, but it may make a big difference in large codebases.
- If we require address of the function in a program, the compiler cannot perform inlining on such functions. Because for providing the address to a function, the compiler will have to allocate storage to it. But inline functions don't get storage, they are kept in Symbol table.
- Inline functions might cause thrashing because it might increase the size of the binary executable file. Thrashing in memory causes the performance of the computer to degrade and it also affects the efficiency of our code.
- The inline function may increase compile time overhead if someone tried to changes the code inside the inline function then all the calling location has to be recompiled again because the compiler would require to replace all the code once again to reflect the changes, otherwise it will continue with old functionality without any change.

Inline function and classes:

It is also possible to define the inline function inside the class. In fact, all the functions defined inside the class are implicitly inline. Thus, all the restrictions of inline functions are also applied here. If you need to explicitly declare inline function in the class then just declare the function inside the class and define it outside the class using inline keyword.

Example:

```
#include <iostream>
using namespace std;
class operation
{
    int a,b,add,sub,mul;
    float div;
public:
    void get();
```

```

        void sum();
        void difference();
        void product();
        void division();
};

inline void operation :: get()
{
    cout << "Enter first value:";
    cin >> a;
    cout << "Enter second value:";
    cin >> b;
}

inline void operation :: sum()
{
    add = a+b;
    cout << "Addition of two numbers: " << a+b << "\n";
}

inline void operation :: difference()
{
    sub = a-b;
    cout << "Difference of two numbers: " << a-b << "\n";
}

inline void operation :: product()
{
    mul = a*b;
    cout << "Product of two numbers: " << a*b << "\n";
}

inline void operation :: division()
{
    div=a/b;
    cout<<"Division of two numbers: "<<a/b<<"\n" ;
}

int main()
{
    cout << "Program using inline function\n";
    operation s;
    s.get();
    s.sum();
    s.difference();
    s.product();
    s.division();
    return 0;
}

```

Output:

Enter first value: 45

Enter second value: 15

Addition of two numbers: 60

Difference of two numbers: 30

Product of two numbers: 675

Division of two numbers: 3

❖ Function overloading

C++ Overloading

When we create two or more members of a class having the same name but different in number or type of parameters, it is known as C++ overloading. In C++, we can overload:

- methods,
- constructors, and
- indexed properties

Types of overloading in C++



What is function overloading in C++?

Function Overloading in C++ can be defined as the process of having two or more member functions of a class with the same name, but different in parameters. **In function overloading, the function can be redefined either by using different types of arguments or a different number of arguments according to the requirement.** It is only through these differences compiler can differentiate between the two overloaded functions.

One of the major advantages of Function overloading is that it increases the readability of the program because we don't need to use different names for the same action again and again.

➤ *By changing the Number of Arguments*

In this way of function overloading, we define two functions with the same names but a different number of parameters of the same type. For example, in the below-mentioned program, we have made two add() functions to return the sum of two and three integers.

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

int add(int a, int b)
{
    cout << a+b << endl;
    return 0;
}

int add(int a, int b, int c)
{
```

```

        cout << a+b+c << endl;
        return 0;
    }

    int main()
    {
        add(20, 40);
        add(40, 20, 30);
    }

```

Output:

60
90

In the above example, we overload add() function by changing its number of arguments. First, we define an add() function with two parameters, then we overload it by again defining the add() function but this time with three parameters.

➤ **By having different types of Arguments**

In this method, we define two or more functions with the same name and the same number of parameters, but the data type used for these parameters are different. For example in this program, we have three add() function, the first one gets two integer arguments, the second one gets two float arguments and the third one gets two double arguments.

```

#include <iostream>
using namespace std;

int add(int x, int y) // first definition
{
    cout << x+y << endl;
    return 0;
}

float add(float a, float b)
{
    cout << a+b << endl;
    return 0;
}

double add(double x, double y)
{
    cout << x+y << endl;
    return 0;
}

int main()
{
    add(20, 40);
    add(23.45f, 34.5f);
    add(40.24, 20.433);
}

```

Output:

60
57.95
60.673

In the above example, we define add() function three times. First using integers as parameters, second using float as parameters and third using double as a parameter.

Thus we override the add() function twice.

Function Overloading and Ambiguity:

When the compiler is unable to decide which function it should invoke first among the overloaded functions, this situation is known as function overloading ambiguity. The compiler does not run the program if it shows ambiguity error. Causes of Function Overloading ambiguity:

- Type Conversion.
- Function with default arguments.
- Function with a pass by reference

1) Type conversion:

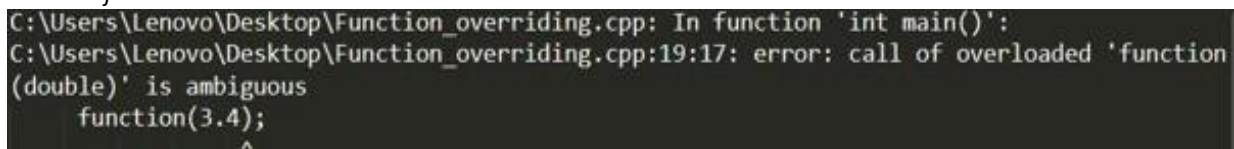
```
#include<iostream>
using namespace std;

void function(float);
void function(int);

void function(float x)
{
    std::cout << "Value of x is : " <<x<< std::endl;
}

void function(int y)
{
    std::cout << "Value of y is : " <<y<< std::endl;
}

int main()
{
    function(3.4);
    function(34);
    return 0;
}
```



```
C:\Users\Lenovo\Desktop\Function_overriding.cpp: In function 'int main()':
C:\Users\Lenovo\Desktop\Function_overriding.cpp:19:17: error: call of overloaded 'function
(double)' is ambiguous
    function(3.4);
               ^
```

The above example throws an error – “call of overloaded ‘function(double)’ is ambiguous”. The function(3.4) will call the first function. The function(34) calls the second function according to our prediction. But this is not what happens because in C++ all the floating-point constants are treated as double not as a float. If we replace the float variable to a double variable, the program will work fine. Therefore we call this a type conversion error from float to double.

2) Function with Default Arguments:

```
#include<iostream>
```

```

using namespace std;

void function(int);
void function(int,int);

void function(int x)
{
    std::cout << "Value of x is : " <<x<< std::endl;
}

void function(int y,int z=12)
{
    std::cout << "Value of y is : " <<y<< std::endl;
    std::cout << "Value of z is : " <<z<< std::endl;
}

int main()
{
    function(12);
    return 0;
}

```

```

C:\Users\Lenovo\Desktop\Function_overriding.cpp: In function 'int main()':
C:\Users\Lenovo\Desktop\Function_overriding.cpp:17:16: error: call of overloaded 'function
(int)' is ambiguous
    function(12);
               ^

```

The above example gives an error saying “call of overloaded ‘fun(int)’ is ambiguous”, this is because function(int y, int z=12) can be called in two ways:

1. By calling the function with one argument (and it will automatically take the value of z = 12)
2. By calling the function with two arguments.

When we call the function: function(12) we full fill the condition of both function(int) and function(int, int) thus the compiler gets into an ambiguity shows an error.

3) Function with pass by reference:

```

#include <iostream>
using namespace std;

void function(int);
void function(int &);

void function(int a)
{
    std::cout << "Value of a is : " <<a<< std::endl;
}

void function(int &b)
{
    std::cout << "Value of b is : " <<b<< std::endl;
}

```

```

int main()
{
    int x=10;
    function(x);
    return 0;
}

```

```

C:\Users\Lenovo\Desktop\Function_overriding.cpp: In function 'int main()':
C:\Users\Lenovo\Desktop\Function_overriding.cpp:18:11: error: call of overloaded 'function
(int&)' is ambiguous
    function(x);
            ^

```

The above program gives an error saying “call of overloaded ‘fun(int&)’ is ambiguous”. As we see the first function takes one integer argument and the second function takes a reference parameter as an argument. In this case, the compiler is not able to understand which function is needed by the user as there is no syntactical difference between the fun(int) and fun(int &) thus it shots an error of ambiguity.

Functions that cannot be overloaded in C++

1) Function declarations that differ only in the return type. For example, the following program fails in compilation.

```

#include<iostream>
int foo() {
    return 10;
}

char foo() {
    return 'a';
}

int main()
{
    char x = foo();
    getchar();
    return 0;
}

```

2) Member function declarations with the same name and the name parameter-type-list cannot be overloaded if any of them is a static member function declaration. For example, following program fails in compilation.

```

#include<iostream>
class Test {
    static void fun(int i) {}
    void fun(int i) {}
};

int main()
{
    Test t;
    getchar();
    return 0;
}

```

3) Parameter declarations that differ only in a pointer * versus an array [] are equivalent. That is, the array declaration is adjusted to become a pointer declaration. Only the second and subsequent array dimensions are significant in parameter types. For example, following two function declarations are equivalent.

```
int fun(int *ptr);  
int fun(int ptr[]); // redeclaration of fun(int *ptr)
```

4) Parameter declarations that differ only in that one is a function type and the other is a pointer to the same function type are equivalent.

```
void h(int ());  
void h(int (*)( )); // redeclaration of h(int())
```

5) Parameter declarations that differ only in the presence or absence of const and/or volatile are equivalent. That is, the const and volatile type-specifiers for each parameter type are ignored when determining which function is being declared, defined, or called. For example, following program fails in compilation with error “redefinition of ‘int f(int)’ “

```
#include<iostream>  
#include<stdio.h>  
  
using namespace std;  
  
int f ( int x) {  
    return x+10;  
}  
  
int f ( const int x) {  
    return x+10;  
}  
  
int main() {  
    getchar();  
    return 0;  
}
```

Only the const and volatile type-specifiers at the outermost level of the parameter type specification are ignored in this fashion; const and volatile type-specifiers buried within a parameter type specification are significant and can be used to distinguish overloaded function declarations. In particular, for any type T,

“pointer to T,” “pointer to const T,” and “pointer to volatile T” are considered distinct parameter types, as are “reference to T,” “reference to const T,” and “reference to volatile T.”

❖ Function with default arguments

Default Arguments in C++

A default argument is a value provided in a function declaration that is automatically assigned by the compiler if the caller of the function doesn’t provide a value for the argument with a default value. Following is a simple C++ example to demonstrate the use of default arguments. We don’t have to write 3 sum functions, only one function works by using default values for 3rd and 4th arguments.

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



```

// A function with default arguments, it can be called with
// 2 arguments or 3 arguments or 4 arguments.
int sum(int x, int y, int z=0, int w=0)
{
    return (x + y + z + w);
}

/* Driver program to test above function */
int main()
{
    cout << sum(10, 15) << endl;
    cout << sum(10, 15, 25) << endl;
    cout << sum(10, 15, 25, 30) << endl;
    return 0;
}

```

Output:

```

25
50
80

```

When Function overloading is done along with default values . Then we need to make sure it will not be ambiguous. The compiler will throw error if ambiguous

Key Points:

- Default arguments are different from constant arguments as constant arguments can't be changed whereas default arguments can be overwritten if required.
- Default arguments are overwritten when calling function provides values for them. For example, calling of function `sum(10, 15, 25, 30)` overwrites the value of `z` and `w` to 25 and 30 respectively.
- During calling of function, arguments from calling function to called function are copied from left to right. Therefore, `sum(10, 15, 25)` will assign 10, 15 and 25 to `x`, `y`, and `z`. Therefore, the default value is used for `w` only.
- Once default value is used for an argument in function definition, all subsequent arguments to it must have default value. It can also be stated as default arguments are assigned from right to left. For example, the following function definition is invalid as subsequent argument of default variable `z` is not default.

```

// Invalid because z has default value, but w after it
// doesn't have default value
int sum(int x, int y, int z=0, int w)

```

❖ Constructors and destructors

➤ Constructor in c++

What is constructor?

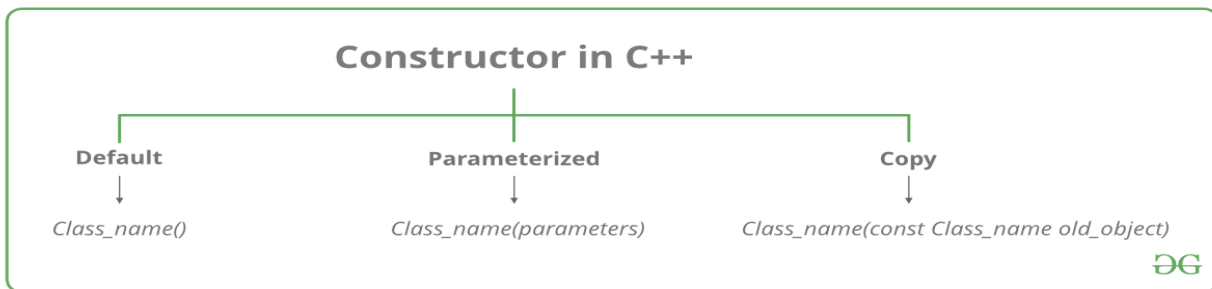
A constructor is a member function of a class which initializes objects of a class. In C++, Constructor is automatically called when object(instance of class) create. It is special member function of the class.

How constructors are different from a normal member function?

A constructor is different from normal functions in following ways:

- Constructor has same name as the class itself
- Constructors don't have return type
- A constructor is automatically called when an object is created.

- If we do not specify a constructor, C++ compiler generates a default constructor for us (expects no parameters and has an empty body).



Types of Constructors

1. Default constructor is the constructor which doesn't take any argument. It has no parameters.

```

// Cpp program to illustrate the
// concept of Constructors
#include <iostream>
using namespace std;

class construct
{
public:
    int a, b;

    // Default Constructor
    construct()
    {
        a = 10;
        b = 20;
    }
};

int main()
{
    // Default constructor called automatically
    // when the object is created
    construct c;
    cout << "a: " << c.a << endl
         << "b: " << c.b;
    return 1;
}
  
```

Output:

```

a: 10
b: 20
  
```

Note: Even if we do not define any constructor explicitly, the compiler will automatically provide a default constructor implicitly.

2. Parameterized Constructors: It is possible to pass arguments to constructors. Typically, these arguments help initialize an object when it is created. To create a parameterized constructor, simply

add parameters to it the way you would to any other function. When you define the constructor's body, use the parameters to initialize the object.

```
// CPP program to illustrate
// parameterized constructors
#include <iostream>
using namespace std;

class Point
{
private:
    int x, y;
public:
    // Parameterized Constructor
    Point(int x1, int y1)
    {
        x = x1;
        y = y1;
    }

    int getX()
    {
        return x;
    }
    int getY()
    {
        return y;
    }
};

int main()
{
    // Constructor called
    Point p1(10, 15);
    // Access values assigned by constructor
    cout << "p1.x = " << p1.getX() << ", p1.y = " << p1.getY();

    return 0;
}
```

Output:

p1.x = 10, p1.y = 15

When an object is declared in a parameterized constructor, the initial values have to be passed as arguments to the constructor function. The normal way of object declaration may not work. The constructors can be called explicitly or implicitly.

Example e = Example(0, 50); // Explicit call

Example e(0, 50); // Implicit call

Uses of Parameterized constructor:

1. It is used to initialize the various data elements of different objects with different values when they are created.

2. It is used to overload constructors (constructor overloading).

3. Copy Constructor: A copy constructor is a member function which initializes an object using another object of the same class.

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

class Point
{
private:
    int x, y;
public:
    Point(int x1, int y1) { x = x1; y = y1; }

    // Copy constructor
    Point(const Point &p2) {x = p2.x; y = p2.y; }

    int getX()          { return x; }
    int getY()          { return y; }
};

int main()
{
    Point p1(10, 15); // Normal constructor is called here
    Point p2 = p1; // Copy constructor is called here

    // Let us access values assigned by constructors
    cout << "p1.x = " << p1.getX() << ", p1.y = " << p1.getY();
    cout << "\np2.x = " << p2.getX() << ", p2.y = " << p2.getY();

    return 0;
}
```

Output:

```
p1.x = 10, p1.y = 15
p2.x = 10, p2.y = 15
```

When is copy constructor called?

In C++, a Copy Constructor may be called in following cases:

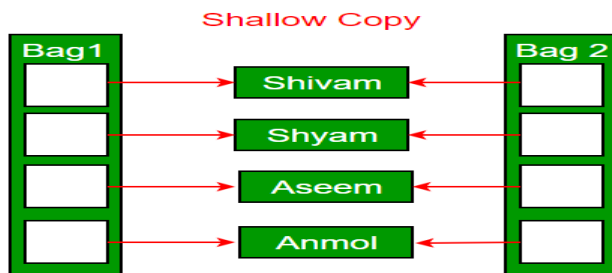
1. When an object of the class is returned by value.
2. When an object of the class is passed (to a function) by value as an argument.
3. When an object is constructed based on another object of the same class.
4. When the compiler generates a temporary object.

It is, however, not guaranteed that a copy constructor will be called in all these cases, because the C++ Standard allows the compiler to optimize the copy away in certain cases, one example is the return value optimization (sometimes referred to as RVO).

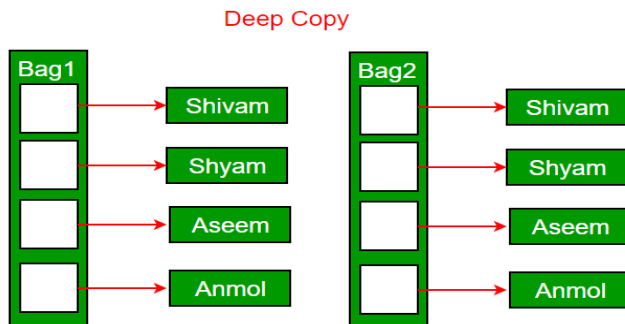
When is user-defined copy constructor needed?

If we don't define our own copy constructor, the C++ compiler creates a default copy constructor for each class which does a member-wise copy between objects. The compiler created copy constructor works fine in general. We need to define our own copy constructor only if an object has pointers or any runtime allocation of the resource like file handle, a network connection..etc.

Default constructor does only shallow copy.



Deep copy is possible only with user defined copy constructor. In user defined copy constructor, we make sure that pointers (or references) of copied object point to new memory locations.



Copy constructor vs Assignment Operator

Which of the following two statements call copy constructor and which one calls assignment operator?

```
MyClass t1, t2;
MyClass t3 = t1; // ----> (1)
t2 = t1;          // ----> (2)
```

Copy constructor is called when a new object is created from an existing object, as a copy of the existing object. Assignment operator is called when an already initialized object is assigned a new value from another existing object. In the above example (1) calls copy constructor and (2) calls assignment operator

Write an example class where copy constructor is needed?

Following is a complete C++ program to demonstrate use of Copy constructor. In the following String class, we must write copy constructor.

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

class String
{
private:
    char *s;
    int size;
public:
    String(const char *str = NULL); // constructor
    ~String() { delete [] s; } // destructor
    String(const String&); // copy constructor
    void print() { cout << s << endl; } // Function to print string
```

```

        void change(const char *); // Function to change
    };

    String::String(const char *str)
    {
        size = strlen(str);
        s = new char[size+1];
        strcpy(s, str);
    }

    void String::change(const char *str)
    {
        delete [] s;
        size = strlen(str);
        s = new char[size+1];
        strcpy(s, str);
    }

    String::String(const String& old_str)
    {
        size = old_str.size;
        s = new char[size+1];
        strcpy(s, old_str.s);
    }

    int main()
    {
        String str1("GeeksQuiz");
        String str2 = str1;

        str1.print(); // what is printed ?
        str2.print();

        str2.change("GeeksforGeeks");

        str1.print(); // what is printed now ?
        str2.print();
        return 0;
    }

```

Output:

```

GeeksQuiz
GeeksQuiz
GeeksQuiz
GeeksforGeeks

```

What would be the problem if we remove copy constructor from above code?

If we remove copy constructor from the above program, we don't get the expected output. The changes made to str2 reflect in str1 as well which is never expected.

```
#include<iostream>
```

```

#include<cstring>
using namespace std;

class String
{
private:
    char *s;
    int size;
public:
    String(const char *str = NULL); // constructor
    ~String() { delete [] s; } // destructor
    void print() { cout << s << endl; }
    void change(const char *); // Function to change
};

String::String(const char *str)
{
    size = strlen(str);
    s = new char[size+1];
    strcpy(s, str);
}

void String::change(const char *str)
{
    delete [] s;
    size = strlen(str);
    s = new char[size+1];
    strcpy(s, str);
}

int main()
{
    String str1("GeeksQuiz");
    String str2 = str1;

    str1.print(); // what is printed ?
    str2.print();

    str2.change("GeeksforGeeks");

    str1.print(); // what is printed now ?
    str2.print();
    return 0;
}

```

Output:

```

GeeksQuiz
GeeksQuiz
GeeksforGeeks

```

Can we make copy constructor private?

Yes, a copy constructor can be made private. When we make a copy constructor private in a class, objects of that class become non-copyable. This is particularly useful when our class has pointers or dynamically allocated resources. In such situations, we can either write our own copy constructor like above String example or make a private copy constructor so that users get compiler errors rather than surprises at runtime.

Why argument to a copy constructor must be passed as a reference?

A copy constructor is called when an object is passed by value. Copy constructor itself is a function. So if we pass an argument by value in a copy constructor, a call to copy constructor would be made to call copy constructor which becomes a non-terminating chain of calls. Therefore compiler doesn't allow parameters to be passed by value

Why copy constructor argument should be const in C++?

When we create our own copy constructor, we pass an object by reference and we generally pass it as a const reference.

One reason for passing const reference is, we should use const in C++ wherever possible so that objects are not accidentally modified. This is one good reason for passing reference as const, but there is more to it. For example, predict the output of following C++ program. Assume that copy elision is not done by compiler.

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

class Test
{
    /* Class data members */
public:
    Test(Test &t) { /* Copy data members from t*/}
    Test() { /* Initialize data members */}
};

Test fun()
{
    cout << "fun() Called\n";
    Test t;
    return t;
}

int main()
{
    Test t1;
    Test t2 = fun();
    return 0;
}
```

Output:

Compiler Error in line "Test t2 = fun();"

The program looks fine at first look, but it has compiler error. If we add const in copy constructor, the program works fine, i.e., we change copy constructor to following.

```
Test(const Test &t) { cout << "Copy Constructor Called\n"; }
```


Or if we change the line "Test t2 = fun();" to following two lines, then also the program works fine.

```
Test t2;  
t2 = fun();
```

The function fun() returns by value. So the compiler creates a temporary object which is copied to t2 using copy constructor in the original program (The temporary object is passed as an argument to copy constructor). The reason for compiler error is, compiler created temporary objects cannot be bound to non-const references and the original program tries to do that. It doesn't make sense to modify compiler created temporary objects as they can die any moment.

➤ **Destructors in C++**

What is destructor?

Destructor is a member function which destructs or deletes an object.

Syntax:

```
~Constructor-name();
```

Properties of Destructor:

- Destructor function is automatically invoked when the objects are destroyed.
- It cannot be declared static or const.
- The destructor does not have arguments.
- It has no return type not even void.
- An object of a class with a Destructor cannot become a member of the union.
- A destructor should be declared in the public section of the class.
- The programmer cannot access the address of destructor.

When is destructor called?

A destructor function is called automatically when the object goes out of scope:

- (1) the function ends
- (2) the program ends
- (3) a block containing local variables ends
- (4) a delete operator is called

How destructors are different from a normal member function?

Destructors have same name as the class preceded by a tilde (~)

Destructors don't take any argument and don't return anything

```
class String {  
private:  
    char* s;  
    int size;  
  
public:  
    String(char*); // constructor  
    ~String(); // destructor  
};  
  
String::String(char* c)  
{  
    size = strlen(c);  
    s = new char[size + 1];  
    strcpy(s, c);  
}  
String::~~String() { delete[] s; }
```

Can there be more than one destructor in a class?

No, there can only one destructor in a class with classname preceded by ~, no parameters and no return type.

When do we need to write a user-defined destructor?

If we do not write our own destructor in class, compiler creates a default destructor for us. The default destructor works fine unless we have dynamically allocated memory or pointer in class. When a class contains a pointer to memory allocated in class, we should write a destructor to release memory before the class instance is destroyed. This must be done to avoid memory leak.

Virtual Destructor

Deleting a derived class object using a pointer of base class type that has a non-virtual destructor results in undefined behavior. To correct this situation, the base class should be defined with a virtual destructor. For example, following program results in undefined behavior.

```
// CPP program without virtual destructor
// causing undefined behavior
#include<iostream>

using namespace std;

class base {
public:
    base()
    { cout<<"Constructing base \n"; }
    ~base()
    { cout<<"Destructing base \n"; }
};

class derived: public base {
public:
    derived()
    { cout<<"Constructing derived \n"; }
    ~derived()
    { cout<<"Destructing derived \n"; }
};

int main(void)
{
    derived *d = new derived();
    base *b = d;
    delete b;
    getchar();
    return 0;
}
```

Although the output of following program may be different on different compilers, when compiled using Dev-CPP, it prints following:

```
Constructing base
Constructing derived
Destructing base
```

Making base class destructor virtual guarantees that the object of derived class is destructed properly, i.e., both base class and derived class destructors are called. For example,

```
// A program with virtual destructor
#include<iostream>

using namespace std;

class base {
public:
    base()
    { cout<<"Constructing base \n"; }
    virtual ~base()
    { cout<<"Destructing base \n"; }
};

class derived: public base {
public:
    derived()
    { cout<<"Constructing derived \n"; }
    ~derived()
    { cout<<"Destructing derived \n"; }
};

int main(void)
{
    derived *d = new derived();
    base *b = d;
    delete b;
    getchar();
    return 0;
}
```

Output:

Constructing base
Constructing derived
Destructing derived
Destructing base

As a guideline, any time you have a virtual function in a class, you should immediately add a virtual destructor (even if it does nothing). This way, you ensure against any surprises later.

Pure virtual destructor in C++

Can a destructor be pure virtual in C++?

Yes, it is possible to have pure virtual destructor. Pure virtual destructors are legal in standard C++ and one of the most important things to remember is that if a class contains a pure virtual destructor, it must provide a function body for the pure virtual destructor. You may be wondering why a pure virtual function requires a function body. The reason is because destructors (unlike other functions) are not actually 'overridden', rather they are always called in the reverse order of the class derivation. This means that a derived class' destructor will be invoked first, then base class destructor will be called. If the definition of the pure virtual destructor is not provided, then what function body will be called during object destruction? Therefore the compiler and linker enforce the existence of a function body

for pure virtual destructors.

Consider the following program:

```
#include <iostream>
class Base
{
public:
    virtual ~Base()=0; // Pure virtual destructor
};

class Derived : public Base
{
public:
    ~Derived()
    {
        std::cout << "~Derived() is executed";
    }
};

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

The linker will produce **following error** in the above program.

```
test.cpp:(.text$_ZN7DerivedD1Ev[__ZN7DerivedD1Ev]+0x4c):
undefined reference to `Base::~~Base()'
```

Now if the definition for the pure virtual destructor is provided, then the program compiles & runs fine.

```
#include <iostream>
class Base
{
public:
    virtual ~Base()=0; // Pure virtual destructor
};
Base::~~Base()
{
    std::cout << "Pure virtual destructor is called";
}

class Derived : public Base
{
public:
    ~Derived()
    {
        std::cout << "~Derived() is executed\n";
    }
};
```

```

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

```

Output:

~Derived() is executed

Pure virtual destructor is called

It is important to note that a class becomes abstract class when it contains a pure virtual destructor.

❖ friend function and classes

Friend Class A friend class can access private and protected members of other class in which it is declared as friend. It is sometimes useful to allow a particular class to access private members of other class

```

#include <iostream>
class A {
private:
    int a;
public:
    A() { a = 0; }
    friend class B; // Friend Class
};

class B {
private:
    int b;
public:
    void showA(A& x)
    {
        // Since B is friend of A, it can access
        // private members of A
        std::cout << "A::a=" << x.a;
    }
};

int main()
{
    A a;
    B b;
    b.showA(a);
    return 0;
}

```

Output:

A::a=0

Friend Function Like friend class, a friend function can be given special grant to access private and protected members. A friend function can be:

a) A method of another class

b) A global function

a) A method of another class

```
#include <iostream>

class B;

class A {
public:
    void showB(B&);
};

class B {
private:
    int b;

public:
    B() { b = 0; }
    friend void A::showB(B& x); // Friend function
};

void A::showB(B& x)
{
    // Since showB() is friend of B, it can
    // access private members of B
    std::cout << "B::b = " << x.b;
}

int main()
{
    A a;
    B x;
    a.showB(x);
    return 0;
}
```

Output:

B::b = 0

b) A global function

```
#include <iostream>

class A {
    int a;
public:
    A() { a = 0; }
    // global friend function
    friend void showA(A&);
};
```

```
};

void showA(A& x)
{
    // Since showA() is a friend, it can access
    // private members of A
    std::cout << "A::a=" << x.a;
}

int main()
{
    A a;
    showA(a);
    return 0;
}
```

Output:

A::a = 0

Following are some important points about friend functions and classes:

- 1) Friends should be used only for limited purpose. too many functions or external classes are declared as friends of a class with protected or private data, it lessens the value of encapsulation of separate classes in object-oriented programming.
- 2) Friendship is not mutual. If class A is a friend of B, then B doesn't become a friend of A automatically.
- 3) Friendship is not inherited.
- 4) The concept of friends is not there in Java.

Charastices of a Friend Function:

1. It is not in the scope of the class to which it has been declared as Friend Function.
2. Since it is not in a scope of class , so it can not be called by the class object.
3. it is invoked or call as normal procedural function, without any object.
4. Unkile a member funtion , it can can not access the data members directly. its has to accessed by the class Object
5. it can be declared in public and private Section.
6. Usually it has the Objects as an Arguments.

❖ Using this pointer

Every object in C++ has access to its own address through an important pointer called this pointer.

The this pointer is an implicit parameter to all member functions. Therefore, inside a member function, this may be used to refer to the invoking object.

The 'this' pointer is passed as a hidden argument to all nonstatic member function calls and is available as a local variable within the body of all nonstatic functions. 'this' pointer is not available in static member functions as static member functions can be called without any object (with class name).

For a class X, the type of this pointer is 'X* '. Also, if a member function of X is declared as const, then the type of this pointer is 'const X *'

Friend functions do not have a **this** pointer, because friends are not members of a class. Only member functions have a **this** pointer.

C++ lets object destroy themselves by calling the following code :

delete this;

Following are the situations where 'this' pointer is used:

1) When local variable's name is same as member's name

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

/* local variable is same as a member's name */
class Test
{
private:
int x;
public:
void setX (int x)
{
    // The 'this' pointer is used to retrieve the object's x
    // hidden by the local variable 'x'
    this->x = x;
}
void print() { cout << "x = " << x << endl; }
};

int main()
{
    Test obj;
    int x = 20;
    obj.setX(x);
    obj.print();
    return 0;
}
```

Output:

x = 20

For constructors, initializer list can also be used when parameter name is same as member's name.

2) To return reference to the calling object

```
/* Reference to the calling object can be returned */
Test& Test::func ()
{
    // Some processing
    return *this;
}
```

When a reference to a local object is returned, the returned reference can be used to **chain function calls** on a single object.

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

```
class Test
{
private:
int x;
```



```
int y;  
public:  
Test(int x = 0, int y = 0) { this->x = x; this->y = y; }  
Test &setX(int a) { x = a; return *this; }  
Test &setY(int b) { y = b; return *this; }  
void print() { cout << "x = " << x << " y = " << y << endl; }  
};
```

```
int main()  
{  
Test obj1(5, 5);
```

```
// Chained function calls. All calls modify the same object  
// as the same object is returned by reference  
obj1.setX(10).setY(20);
```

```
obj1.print();  
return 0;  
}
```

Output:

x = 10 y = 20

UNIT-3

❖ Inheritance

➤ Inheritance in C++

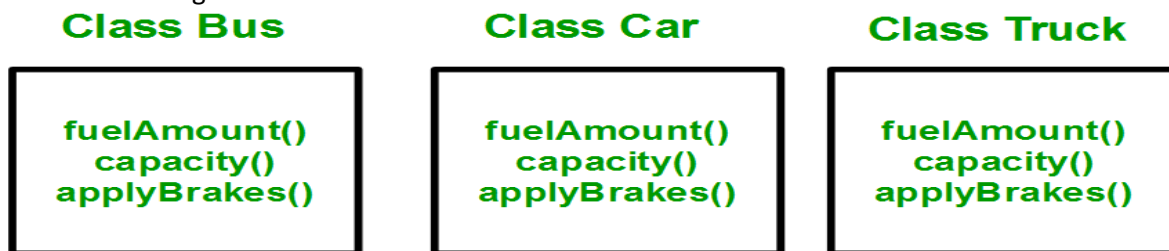
The capability of a class to derive properties and characteristics from another class is called **Inheritance**. Inheritance is one of the most important feature of Object Oriented Programming.

Sub Class: The class that inherits properties from another class is called Sub class or Derived Class.

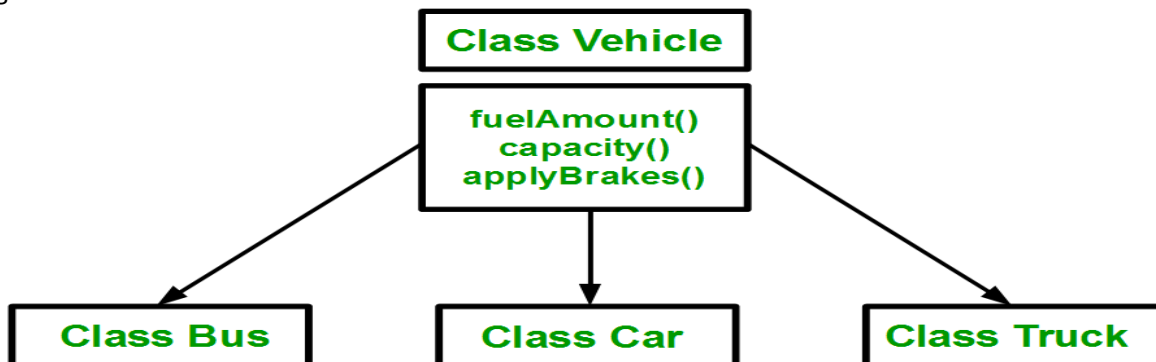
Super Class: The class whose properties are inherited by sub class is called Base Class or Super class.

Why and when to use inheritance?

Consider a group of vehicles. You need to create classes for Bus, Car and Truck. The methods `fuelAmount()`, `capacity()`, `applyBrakes()` will be same for all of the three classes. If we create these classes avoiding inheritance then we have to write all of these functions in each of the three classes as shown in below figure:



You can clearly see that above process results in duplication of same code 3 times. This increases the chances of error and data redundancy. To avoid this type of situation, inheritance is used. If we create a class Vehicle and write these three functions in it and inherit the rest of the classes from the vehicle class, then we can simply avoid the duplication of data and increase re-usability. Look at the below diagram in which the three classes are inherited from vehicle class:



Using inheritance, we have to write the functions only one time instead of three times as we have inherited rest of the three classes from base class(Vehicle).

Implementing inheritance in C++: For creating a sub-class which is inherited from the base class we have to follow the below syntax.

Syntax:

```
class subclass_name : access_mode base_class_name
{
    //body of subclass
};
```

Here, **subclass_name** is the name of the sub class, **access_mode** is the mode in which you want to inherit this sub class for example: public, private etc. and **base_class_name** is the name of the base class from which you want to inherit the sub class.

Note: A derived class **doesn't inherit access to private data members**. However, it does inherit a full parent object, which contains any private members which that class declares.

// C++ program to demonstrate implementation of Inheritance

```
#include <bits/stdc++.h>
using namespace std;

//Base class
class Parent
{
    public:
        int id_p;
};

// Sub class inheriting from Base Class(Parent)
class Child : public Parent
{
    public:
        int id_c;
};

//main function
int main()
{
    Child obj1;

    // An object of class child has all data members
    // and member functions of class parent
    obj1.id_c = 7;
    obj1.id_p = 91;
    cout << "Child id is " << obj1.id_c << endl;
    cout << "Parent id is " << obj1.id_p << endl;

    return 0;
}
```

Output:

```
Child id is 7
Parent id is 91
```

In the above program the 'Child' class is publicly inherited from the 'Parent' class so the public data members of the class 'Parent' will also be inherited by the class 'Child'.

➤ **Modes of Inheritance**

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

Note : The private members in the base class cannot be directly accessed in the derived class, while protected members can be directly accessed. For example, Classes B, C and D all contain the variables x, y and z in below example. It is just question of access.

// C++ Implementation to show that a derived class doesn't inherit access to private data members.

// However, it does inherit a full parent object

```
class A
{
public:
    int x;
protected:
    int y;
private:
    int z;
};

class B : public A
{
    // x is public
    // y is protected
    // z is not accessible from B
};

class C : protected A
{
    // x is protected
    // y is protected
    // z is not accessible from C
};

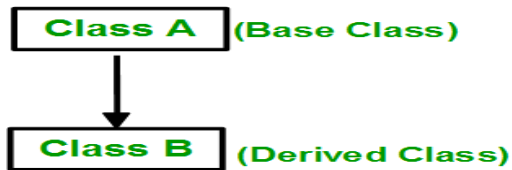
class D : private A // 'private' is default for classes
{
    // x is private
    // y is private
    // z is not accessible from D
};
```

The below table summarizes the above three modes and shows the access specifier of the members of base class in the sub class when derived in public, protected and private modes:

Base class member access specifier	Type of Inheritance		
	Public	Protected	Private
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Not accessible (Hidden)	Not accessible (Hidden)	Not accessible (Hidden)

❖ Types of Inheritance in C++

1. **Single Inheritance:** In single inheritance, a class is allowed to inherit from only one class. i.e. one sub Class is inherited by one base class only.



Syntax:

```
class subclass_name : access_mode base_class
{
    //body of subclass
};
```

// C++ program to explain Single inheritance

```
#include <iostream>
using namespace std;
// base class
class Vehicle {
    public:
    Vehicle()
    {
        cout << "This is a Vehicle" << endl;
    }
};

// sub class derived from one base classe
class Car: public Vehicle{

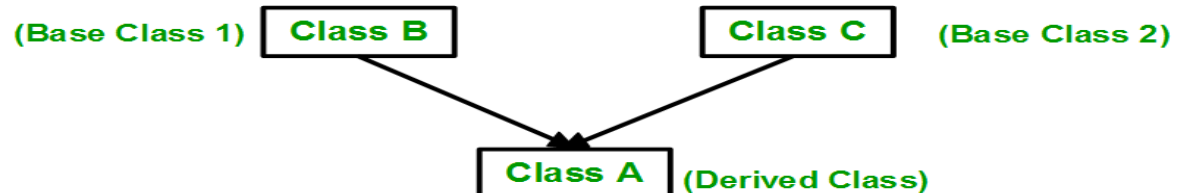
};

// main function
int main()
{
    // creating object of sub class will
    // invoke the constructor of base classes
    Car obj;
    return 0;
}
```

Output:

This is a Vehicle

2. **Multiple Inheritance:** Multiple Inheritance is a feature of C++ where a class can inherit from more than one classes. i.e one **sub class** is inherited from more than one **base classes**.



Syntax:

```
class subclass_name : access_mode base_class1, access_mode base_class2, ....
{
    //body of subclass
};
```

Here, the number of base classes will be separated by a comma (', ') and access mode for every base class must be specified.

// C++ program to explain multiple inheritance

```
#include <iostream>
using namespace std;
// first base class
class Vehicle {
    public:
    Vehicle()
    {
        cout << "This is a Vehicle" << endl;
    }
};
// second base class
class FourWheeler {
    public:
    FourWheeler()
    {
        cout << "This is a 4 wheeler Vehicle" << endl;
    }
};

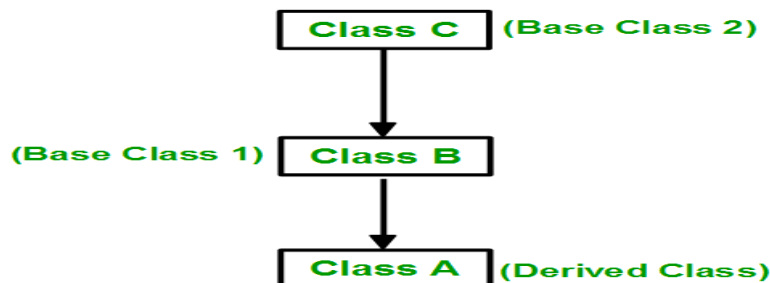
// sub class derived from two base classes
class Car: public Vehicle, public FourWheeler {
};

int main(){
    Car obj;
    Return 0;
}
```

Output:

```
This is a Vehicle
This is a 4 wheeler Vehicle
```

3. **Multilevel Inheritance:** In this type of inheritance, a derived class is created from another derived class.

**// C++ program to implement Multilevel Inheritance**

```

#include <iostream>
using namespace std;
// base class
class Vehicle
{
    public:
    Vehicle()
    {
        cout << "This is a Vehicle" << endl;
    }
};
class fourWheeler: public Vehicle {
    public:
    fourWheeler()
    {
        cout<<"Objects with 4 wheels are vehicles"<<endl;
    }
};
// sub class derived from two base classes
class Car: public fourWheeler{
    public:
    car()
    {
        cout<<"Car has 4 Wheels"<<endl;
    }
};
// main function
int main()
{
    //creating object of sub class will
    //invoke the constructor of base classes
    Car obj;
    return 0;
}

```

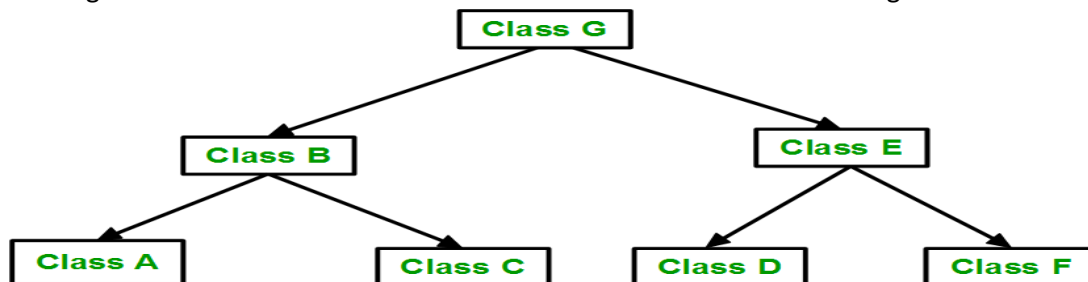
output:

```

This is a Vehicle
Objects with 4 wheels are vehicles
Car has 4 Wheels

```

4. **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.



// C++ program to implement Hierarchical Inheritance

```
#include <iostream>
using namespace std;
// base class
class Vehicle
{
    public:
    Vehicle()
    {
        cout << "This is a Vehicle" << endl;
    }
};
// first sub class
class Car: public Vehicle {

};
// second sub class
class Bus: public Vehicle {

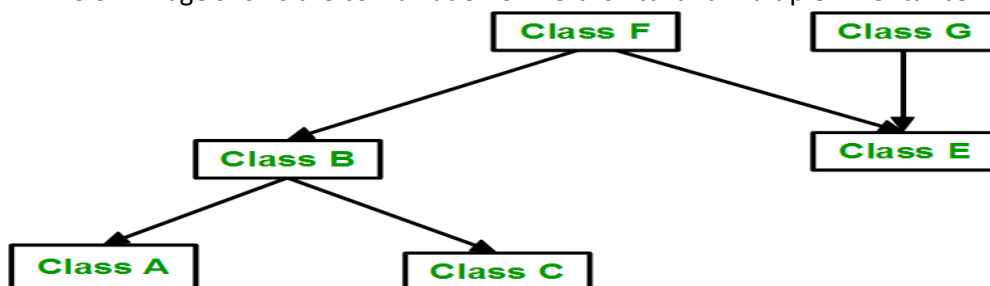
};

// main function
int main()
{
    // creating object of sub class will
    // invoke the constructor of base class
    Car obj1;
    Bus obj2;
    return 0;
}
```

Output:

This is a Vehicle
This is a Vehicle

5. **Hybrid (Virtual) Inheritance**: Hybrid Inheritance is implemented by combining more than one type of inheritance. For example: Combining Hierarchical inheritance and Multiple Inheritance. Below image shows the combination of hierarchical and multiple inheritance:



// C++ program for Hybrid Inheritance

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



```

// base class
class Vehicle
{
    public:
    Vehicle() {
        cout << "This is a Vehicle" << endl;
    }
};
//base class
class Fare
{
    public:
    Fare() {
        cout<<"Fare of Vehicle\n";
    }
};
// first sub class
class Car: public Vehicle
{
};
// second sub class
class Bus: public Vehicle, public Fare
{
};

// main function
int main()
{
    // creating object of sub class will invoke the constructor of base class
    Bus obj2;
    return 0;
}

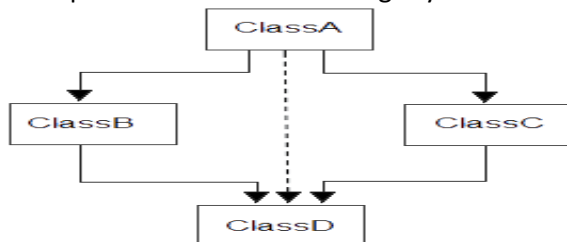
```

Output:

This is a Vehicle
Fare of Vehicle

A special case of hybrid inheritance: Multipath inheritance:

A derived class with two base classes and these two base classes have one common base class is called multipath inheritance. An ambiguity can arise in this type of inheritance.



// C++ program demonstrating ambiguity in Multipath Inheritance

```

#include<iostream.h>
#include<conio.h>
class ClassA
{
    public:
    int a;
};
class ClassB : public ClassA
{
    public:
    int b;
};
class ClassC : public ClassA
{
    public:
    int c;
};
class ClassD : public ClassB, public ClassC
{
    public:
    int d;
};

void main()
{
    ClassD obj;

    //obj.a = 10;           //Statement 1, Error
    //obj.a = 100;          //Statement 2, Error

    obj.ClassB::a = 10;     //Statement 3
    obj.ClassC::a = 100;    //Statement 4

    obj.b = 20;
    obj.c = 30;
    obj.d = 40;

    cout<< "\n A from ClassB : "<< obj.ClassB::a;
    cout<< "\n A from ClassC : "<< obj.ClassC::a;

    cout<< "\n B : "<< obj.b;
    cout<< "\n C : "<< obj.c;
    cout<< "\n D : "<< obj.d;
}

```

Output:

```

A from ClassB : 10
A from ClassC : 100
B : 20

```

C : 30

D : 40

In the above example, both ClassB & ClassC inherit ClassA, they both have single copy of ClassA. However ClassD inherit both ClassB & ClassC, therefore ClassD have two copies of ClassA, one from ClassB and another from ClassC.

If we need to access the data member a of ClassA through the object of ClassD, we must specify the path from which a will be accessed, whether it is from ClassB or ClassC, bco'z compiler can't differentiate between two copies of ClassA in ClassD.

There are 2 ways to avoid this ambiguity:

1. Use scope resolution operator

2. Use virtual base class

Avoiding ambiguity using scope resolution operator:

Using scope resolution operator we can manually specify the path from which data member a will be accessed, as shown in statement 3 and 4, in the above example.

obj.ClassB::a = 10; //Statement 3

obj.ClassC::a = 100; //Statement 4

Note : Still, there are two copies of ClassA in ClassD.

Avoiding ambiguity using virtual base class:

```
#include<iostream.h>
#include<conio.h>
class ClassA
{
    public:
    int a;
};
class ClassB : virtual public ClassA
{
    public:
    int b;
};
class ClassC : virtual public ClassA
{
    public:
    int c;
};
class ClassD : public ClassB, public ClassC
{
    public:
    int d;
};

void main()
{
    ClassD obj;

    obj.a = 10;     //Statement 3
    obj.a = 100;    //Statement 4
```

```

        obj.b = 20;
        obj.c = 30;
        obj.d = 40;

        cout<< "\n A : "<< obj.a;
        cout<< "\n B : "<< obj.b;
        cout<< "\n C : "<< obj.c;
        cout<< "\n D : "<< obj.d;
    }

```

Output:

```

A : 100
B : 20
C : 30
D : 40

```

According to the above example, ClassD has only one copy of ClassA, therefore, statement 4 will overwrite the value of a, given at statement 3.

❖ Multiple Inheritance

Multiple Inheritance is a feature of C++ where a class can inherit from more than one classes. The constructors of inherited classes are called in the same order in which they are inherited. For example, in the following program, B's constructor is called before A's constructor.

```

#include<iostream>
using namespace std;
class A
{
    public:
        A() { cout << "A's constructor called" << endl; }
};
class B
{
    public:
        B() { cout << "B's constructor called" << endl; }
};
class C: public B, public A // Note the order
{
    public:
        C() { cout << "C's constructor called" << endl; }
};

int main()
{
    C c;
    return 0;
}

```

Output:

```

B's constructor called
A's constructor called

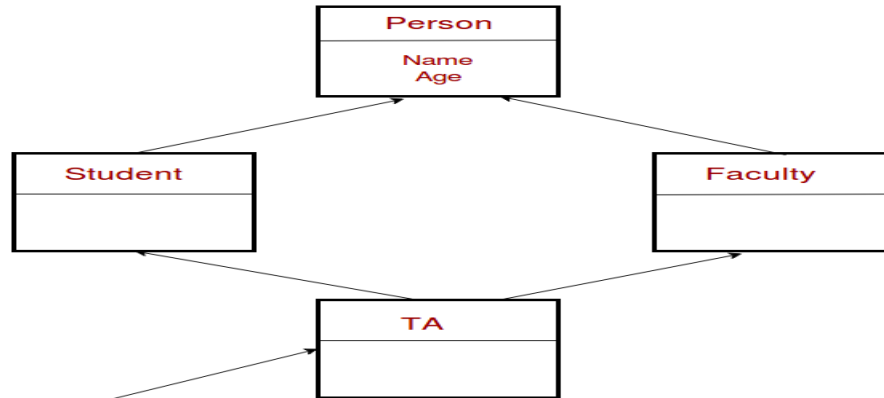
```

C's constructor called

The destructors are called in reverse order of constructors.

➤ **The diamond problem**

The diamond problem occurs when two superclasses of a class have a common base class. For example, in the following diagram, the TA class gets two copies of all attributes of Person class, this causes ambiguities.



Name and Age needed only once

For example, consider the following program.

```
#include<iostream>
using namespace std;
class Person {
    // Data members of person
public:
    Person(int x) { cout << "Person::Person(int ) called" << endl; }
};
class Faculty : public Person {
    // data members of Faculty
public:
    Faculty(int x):Person(x) {
        cout<<"Faculty::Faculty(int ) called"<< endl;
    }
};
class Student : public Person {
    // data members of Student
public:
    Student(int x):Person(x) {
        cout<<"Student::Student(int ) called"<< endl;
    }
};
class TA : public Faculty, public Student {
public:
    TA(int x):Student(x), Faculty(x) {
        cout<<"TA::TA(int ) called"<< endl;
    }
};
int main() {
```

```

    TA ta1(30);
}

```

Output:

```

Person::Person(int ) called
Faculty::Faculty(int ) called
Person::Person(int ) called
Student::Student(int ) called
TA::TA(int ) called

```

In the above program, constructor of 'Person' is called two times. Destructor of 'Person' will also be called two times when object 'ta1' is destructed. So object 'ta1' has two copies of all members of 'Person', this causes ambiguities. The solution to this problem is 'virtual' keyword. We make the classes 'Faculty' and 'Student' as virtual base classes to avoid two copies of 'Person' in 'TA' class. For example,

consider the following program

```

#include<iostream>
using namespace std;
class Person {
public:
    Person(int x) { cout << "Person::Person(int ) called" << endl; }
    Person() { cout << "Person::Person() called" << endl; }
};

class Faculty : virtual public Person {
public:
    Faculty(int x):Person(x) {
        cout<<"Faculty::Faculty(int ) called"<< endl;
    }
};

class Student : virtual public Person {
public:
    Student(int x):Person(x) {
        cout<<"Student::Student(int ) called"<< endl;
    }
};

class TA : public Faculty, public Student {
public:
    TA(int x):Student(x), Faculty(x) {
        cout<<"TA::TA(int ) called"<< endl;
    }
};

int main() {
    TA ta1(30);
}

```

Output:

```

Person::Person() called
Faculty::Faculty(int ) called

```

Student::Student(int) called

TA::TA(int) called

In the above program, constructor of 'Person' is called once. One important thing to note in the above output is, the default constructor of 'Person' is called. When we use 'virtual' keyword, the default constructor of grandparent class is called by default even if the parent classes explicitly call parameterized constructor.

How to call the parameterized constructor of the 'Person' class? The constructor has to be called in 'TA' class. For example, see the following program.

```
#include<iostream>
using namespace std;
class Person {
public:
    Person(int x) { cout << "Person::Person(int ) called" << endl; }
    Person() { cout << "Person::Person() called" << endl; }
};

class Faculty : virtual public Person {
public:
    Faculty(int x):Person(x) {
        cout<<"Faculty::Faculty(int ) called"<< endl;
    }
};

class Student : virtual public Person {
public:
    Student(int x):Person(x) {
        cout<<"Student::Student(int ) called"<< endl;
    }
};

class TA : public Faculty, public Student {
public:
    TA(int x):Student(x), Faculty(x), Person(x) {
        cout<<"TA::TA(int ) called"<< endl;
    }
};

int main() {
    TA ta1(30);
}
```

Output:

Person::Person(int) called

Faculty::Faculty(int) called

Student::Student(int) called

TA::TA(int) called

In general, it is not allowed to call the grandparent's constructor directly, it has to be called through parent class. It is allowed only when 'virtual' keyword is used.

As an exercise, predict the output of following programs.

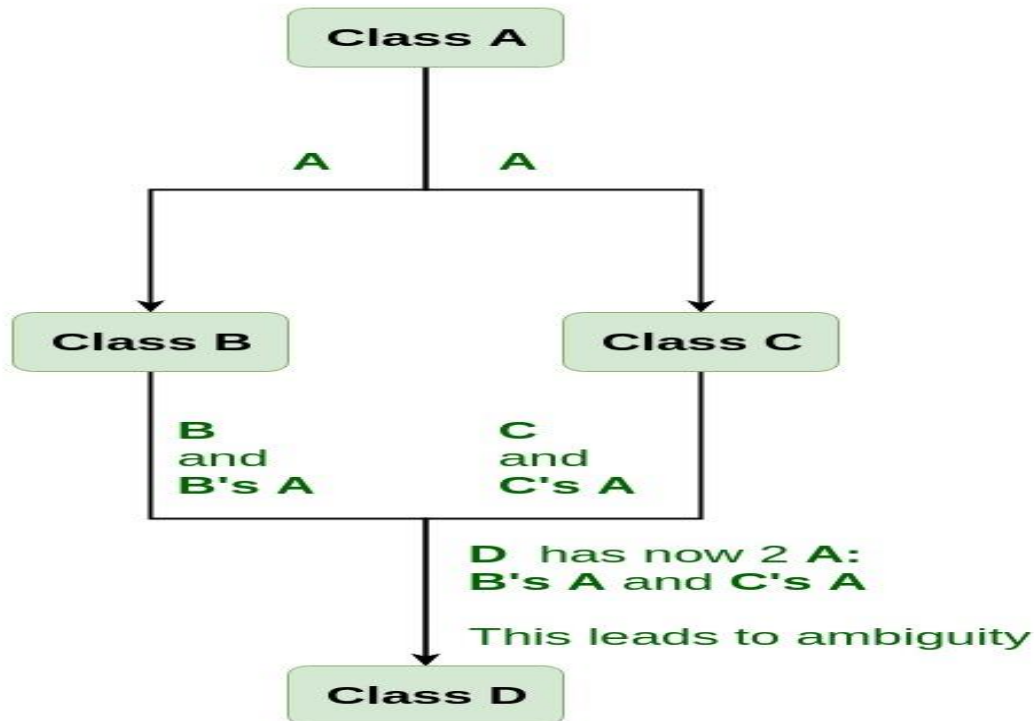
❖ Virtual base class

➤ Virtual base class in C++

Virtual base classes are used in virtual inheritance in a way of preventing multiple “instances” of a given class appearing in an inheritance hierarchy when using multiple inheritances.

Need for Virtual Base Classes:

Consider the situation where we have one class **A**. This class is **A** is inherited by two other classes **B** and **C**. Both these class are inherited into another in a new class **D** as shown in figure below.



As we can see from the figure that data members/function of class **A** are inherited twice to class **D**. One through class **B** and second through class **C**. When any data / function member of class **A** is accessed by an object of class **D**, ambiguity arises as to which data/function member would be called? One inherited through **B** or the other inherited through **C**. This confuses compiler and it displays error.

Example: To show the need of Virtual Base Class in C++

```
#include <iostream>
using namespace std;
class A {
public:
    void show()
    {
        cout << "Hello form A \n";
    }
};
class B : public A {
};
class C : public A {
};
class D : public B, public C {
```



```
};

int main()
{
    D object;
    object.show();
}
```

Compile Errors:

```
prog.cpp: In function 'int main()':
prog.cpp:29:9: error: request for member 'show' is ambiguous
    object.show();
        ^
prog.cpp:8:8: note: candidates are: void A::show()
    void show()
        ^
prog.cpp:8:8: note:         void A::show()
```

How to resolve this issue?

To resolve this ambiguity when class **A** is inherited in both class **B** and class **C**, it is declared as **virtual base class** by placing a keyword **virtual** as :

Syntax for Virtual Base Classes:

Syntax 1:

```
class B : virtual public A
{
};
```

Syntax 2:

```
class C : public virtual A
{
};
```

Note: **virtual** can be written before or after the **public**. Now only one copy of data/function member will be copied to class **C** and class **B** and class **A** becomes the virtual base class.

Virtual base classes offer a way to save space and avoid ambiguities in class hierarchies that use multiple inheritances. When a base class is specified as a virtual base, it can act as an indirect base more than once without duplication of its data members. A single copy of its data members is shared by all the base classes that use virtual base.

Example 1

```
#include <iostream>
using namespace std;
class A {
    public:
        int a;
        A() // constructor
        {
            a = 10;
        }
};
class B : public virtual A {
};
class C : public virtual A {
```

```

};
class D : public B, public C {
};

int main()
{
    D object; // object creation of class d
    cout << "a = " << object.a << endl;

    return 0;
}

```

Output:

a = 10

Explanation : The class **A** has just one data member **a** which is **public**. This class is virtually inherited in class **B** and class **C**. Now class **B** and class **C** becomes virtual base class and no duplication of data member **a** is done.

Example 2:

```

#include <iostream>
using namespace std;
class A {
    public:
        void show()
        {
            cout << "Hello from A \n";
        }
};

class B : public virtual A {
};

class C : public virtual A {
};

class D : public B, public C {
};

int main()
{
    D object;
    object.show();
}

```

Output:

Hello from A

❖ Function overriding

➤ C++ Function Overriding

As we know, inheritance is a feature of OOP that allows us to create derived classes from a base class. The derived classes inherit features of the base class.

Suppose, the same function is defined in both the derived class and the based class. Now if we call this function using the object of the derived class, the function of the derived class is executed.

This is known as function overriding in C++. The function in derived class overrides the function in base class.

It is the redefinition of base class function in its derived class **with same signature i.e return type and parameters.**

It can only be done in derived class.

Example

// C++ program to demonstrate function overriding

```
#include <iostream>
using namespace std;
class Base {
    public:
    void print() {
        cout << "Base Function" << endl;
    }
};
class Derived : public Base {
    public:
    void print() {
        cout << "Derived Function" << endl;
    }
};

int main() {
    Derived derived1;
    derived1.print();
    return 0;
}
```

Output

Derived Function

Here, the same function print() is defined in both Base and Derived classes.

So, when we call print() from the Derived object derived1, the print() from Derived is executed by overriding the function in Base.

➤ Access Overridden Function(Base class function)

To access the overridden function of the base class, we use the **scope resolution operator ::**

We can also access the overridden function by using a pointer of the base class to point to an object of the derived class and then calling the function from that pointer.

Example 1:

//C++ program to Access Overridden Function to the Base Class

//in main() using the scope resolution operator ::

```
#include <iostream>
using namespace std;
class Base {
    public:
    void print() {
        cout << "Base Function" << endl;
    }
};
class Derived : public Base {
```

```

        public:
        void print() {
            cout << "Derived Function" << endl;
        }
};

int main() {
    Derived derived1, derived2;
    derived1.print();
    // access print() function of the Base class
    derived2.Base::print();

    return 0;
}

```

Output

```

    Derived Function
    Base Function

```

Here, this statement
derived2.Base::print();

Example 2:

**//C++ program to call the overridden function
//from a member function of the derived class**

```

#include <iostream>
using namespace std;
class Base {
    public:
    void print() {
        cout << "Base Function" << endl;
    }
};

class Derived : public Base {
    public:
    void print() {
        cout << "Derived Function" << endl;

        // call overridden function
        Base::print();
    }
};

int main() {
    Derived derived1;
    derived1.print();
    return 0;
}

```

Output

```

    Derived Function
    Base Function

```

In this program, we have called the overridden function inside the Derived class itself.

Note: the code **Base::print();** which calls the overridden function inside the Derived class.

Example 3:

//C++ program to access overridden function using pointer

//of Base type that points to an object of Derived class

```
#include <iostream>
using namespace std;
class Base {
    public:
    void print() {
        cout << "Base Function" << endl;
    }
};
class Derived : public Base {
    public:
    void print() {
        cout << "Derived Function" << endl;
    }
};

int main() {
    Derived derived1;
    // pointer of Base type that points to derived1
    Base* ptr = &derived1;
    // call function of Base class using ptr
    ptr->print();

    return 0;
}
```

Output

Base Function

In this program, we have created a pointer of Base type named ptr. This pointer points to the Derived object derived1

//pointer of Base type that points to derived1

Base* ptr = &derived1;

When we call the print() function using ptr, it calls the overridden function from Base.

// call function of Base class using ptr

ptr->print();

This is because even though ptr points to a Derived object, it is actually of Base type. So, it calls the member function of Base.

In order to override the Base function instead of accessing it, we need to use virtual functions in the Base class.

➤ **Function Overloading VS Function Overriding**

1. **Inheritance:** Overriding of functions occurs when one class is inherited from another class. Overloading can occur without inheritance.
2. **Function Signature:** Overloaded functions must differ in function signature ie either number of parameters or type of parameters should differ. In overriding, function signatures must be same.

3. **Scope of functions:** Overridden functions are in different scopes; whereas overloaded functions are in same scope.
4. **Behavior of functions:** Overriding is needed when derived class function has to do some added or different job than the base class function. Overloading is used to have same name functions which behave differently depending upon parameters passed to them.

❖ Abstract class and pure virtual function

Pure Virtual Functions

Pure virtual functions are used

- if a function doesn't have any use in the base class
- but the function must be implemented by all its derived classes

Let's take an example,

Suppose, we have derived Triangle, Square and Circle classes from the Shape class, and we want to calculate the area of all these shapes.

In this case, we can create a pure virtual function named `calculateArea()` in the Shape. Since it's a pure virtual function, all derived classes Triangle, Square and Circle must include the `calculateArea()` function with implementation.

NOTE: A pure virtual function doesn't have the function body and it **must end with = 0**.

Example:

```
class Shape {
    public:
        // creating a pure virtual function
        virtual void calculateArea() = 0;
};
```

Note: The `= 0` syntax doesn't mean we are assigning 0 to the function. It's just the way we define pure virtual functions.

Abstract Class

A class that contains a pure virtual function is known as an abstract class. In the above example, the class Shape is an abstract class.

We cannot create objects of an abstract class. However, we can derive classes from them, and use their data members and member functions (except pure virtual functions).

Example:

//C++ Abstract Class and Pure Virtual Function

// C++ program to calculate the area of a square and a circle

```
#include <iostream>
using namespace std;
// Abstract class
class Shape {
    protected:
        float dimension;

    public:
        void getDimension() {
            cin >> dimension;
        }
};
```

```

        // pure virtual Function
        virtual float calculateArea() = 0;
};
// Derived class
class Square : public Shape {
    public:
        float calculateArea() {
            return dimension * dimension;
        }
};
// Derived class
class Circle : public Shape {
    public:
        float calculateArea() {
            return 3.14 * dimension * dimension;
        }
};

int main() {
    Square square;
    Circle circle;

    cout << "Enter the length of the square: ";
    square.getDimension();
    cout << "Area of square: " << square.calculateArea() << endl;

    cout << "\nEnter radius of the circle: ";
    circle.getDimension();
    cout << "Area of circle: " << circle.calculateArea() << endl;

    return 0;
}

```

Output

```

Enter length to calculate the area of a square: 4
Area of square: 16

```

```

Enter radius to calculate the area of a circle: 5
Area of circle: 78.5

```

In this program, virtual float calculateArea() = 0; inside the Shape class is a pure virtual function. That's why we must provide the implementation of calculateArea() in both of our derived classes, or else we will get an error.

➤ **Some Interesting Facts:**

1) A class is abstract if it has at least one pure virtual function.

In the following example, Test is an abstract class because it has a pure virtual function show().

// pure virtual functions make a class abstract

```

#include<iostream>
using namespace std;
class Test {

```

```

        int x;
    public:
        virtual void show() = 0;
        int getX() { return x; }
};

int main(void)
{
    Test t;
    return 0;
}

```

Output:

Compiler Error: cannot declare variable 't' to be of abstract type 'Test' because the following virtual functions are pure within 'Test': note: virtual void Test::show()

2) We can have pointers and references of abstract class type.

For example the following program works fine.

```

#include<iostream>
using namespace std;
class Base {
    public:
        virtual void show() = 0;
};
class Derived: public Base {
    public:
        void show() { cout << "In Derived \n"; }
};

int main(void)
{
    Base *bp = new Derived();
    bp->show();
    return 0;
}

```

Output:

In Derived

3) If we do not override the pure virtual function in derived class, then derived class also becomes abstract class.

The following example demonstrates the same.

```

#include<iostream>
using namespace std;
class Base {
    public:
        virtual void show() = 0;
};
class Derived : public Base { };

int main(void)

```



```

{
    Derived d;
    return 0;
}

```

Compiler Error:

*cannot declare variable 'd' to be of abstract type
'Derived' because the following virtual functions are pure within
'Derived': virtual void Base::show()*

4) An abstract class can have constructors.

For example, the following program compiles and runs fine.

```

#include<iostream>
using namespace std;
// An abstract class with constructor
class Base {
    protected:
        int x;
    public:
        virtual void fun() = 0;
        Base(int i) { x = i; }
};
class Derived: public Base {
    int y;
    public:
        Derived(int i, int j):Base(i) { y = j; }
        void fun() { cout << "x = " << x << ", y = " << y; }
};

int main(void)
{
    Derived d(4, 5);
    d.fun();
    return 0;
}

```

Output:

x = 4, y = 5

Comparison with Java

In Java, a class can be made abstract by using abstract keyword. Similarly a function can be made pure virtual or abstract by using abstract keyword.

Interface vs Abstract Classes:

An interface does not have implementation of any of its methods, it can be considered as a collection of method declarations. In C++, an interface can be simulated by making all methods as pure virtual. In Java, there is a separate keyword for interface.

UNIT – 4

❖ Constant data member and member function

➤ Introduction to Const Keyword in C++

Constant is something that doesn't change. In C language and C++ we use the keyword `const` to make program elements constant. `const` keyword can be used in many contexts in a C++ program. It can be used with:

1. Variables
2. Pointers
3. Function arguments and return types
4. **Class Data members**
5. **Class Member functions**
6. **Objects**

1) Constant Variables in C++

If you make any variable as constant, using `const` keyword, you cannot change its value. Also, the constant variables must be initialized while they are declared.

```
int main
{
    const int i = 10;
    const int j = i + 10; // works fine
    i++; // this leads to Compile time error
}
```

In the above code we have made `i` as constant, hence if we try to change its value, we will get compile time error. Though we can use it for substitution for other variables.

2) Pointers with const keyword in C++

Pointers can be declared using `const` keyword too. When we use `const` with pointers, we can do it in two ways, either we can apply `const` to what the pointer is pointing to, or we can make the pointer itself a constant.

- **Pointer to a const variable**

This means that the pointer is pointing to a `const` variable.

```
const int* u;
```

Here, `u` is a pointer that can point to a `const int` type variable. We can also write it like,

```
char const* v;
```

still it has the same meaning. In this case also, `v` is a pointer to an `char` which is of `const` type.

Pointers to a `const` variable is very useful, as this can be used to make any string or array immutable(i.e they cannot be changed).

- **const Pointer**

To make a pointer constant, we have to put the `const` keyword to the right of the `*`.

```
int x = 1;
int* const w = &x;
```

Here, `w` is a pointer, which is `const`, that points to an `int`. Now we can't change the pointer, which means it will always point to the variable `x` but can change the value that it points to, by changing the value of `x`.

The constant pointer to a variable is useful where you want a storage that can be changed in value but not moved in memory. Because the pointer will always point to the same memory location, because it is defined with `const` keyword, but the value at that memory location can be changed.

NOTE: We can also have a `const` pointer pointing to a `const` variable.

```
const int* const x;
```

3) const Function Arguments and Return types

We can make the return type or arguments of a function as const. Then we cannot change any of them.

```
void f(const int i)
{
    i++; // error
}
const int g()
{
    return 1;
}
```

Some Important points to Remember

1. For built in datatypes, returning a const or non-const value, doesn't make any difference.

```
const int h()
{
    return 1;
}

int main()
{
    const int j = h();
    int k = h();
}
```

Both j and k will be assigned the value 1. No error will occur.

- 2 For user defined datatypes, returning const, will prevent its modification.
- 3 Temporary objects created while program execution are always of const type.
- 4 If a function has a non-const parameter, it cannot be passed a const argument while making a call.

```
void t(int*)
{
    // function logic
}
```

If we pass a const int* argument to the function t, it will give error.

- 5 But, a function which has a const type parameter, can be passed a const type argument as well as a non-const argument.

```
void g(const int*)
{
    // function logic
}
```

This function can have a int* as well as const int* type argument.

➤ Constant data member and member function

4) Defining Class Data members as const

These are data variables in class which are defined using const keyword. They are not initialized during declaration. Their initialization is done in the constructor.

```
class Test
{
    const int i;
public:
```

```

    Test(int x):i(x)
    {
        cout << "\ni value set: " << i;
    }
};

int main()
{
    Test t(10);
    Test s(20);
}

```

In this program, `i` is a constant data member, in every object its independent copy will be present, hence it is initialized with each object using the constructor. And once initialized, its value cannot be changed.

The above way of initializing a class member is known as **Initializer List in C++**.

Data members that are both static and const have their own rules for initialization.

5) Defining Class Object as const

When an object is declared or created using the `const` keyword, its data members can never be changed, during the object's lifetime.

Syntax:

```
const class_name object;
```

For example, if in the class `Test` defined above, we want to define a constant object, we can do it like:

```
const Test r(30);
```

Only const methods can be called for a const object

The const property of an object goes into effect after the constructor finishes executing and ends before the class's destructor executes. So the constructor and destructor can modify the object, but other methods of the class can't.

6) Defining Class's Member function as const

A const member function never modifies data members in an object.

Syntax:

```
return_type function_name() const;
```

- **Example for const Object and const Member function**

```

class StarWars
{
    public:
    int i;
    StarWars(int x) // constructor
    {
        i = x;
    }

    int falcon() const // constant function
    {
        /*
            can do anything but will not
            modify any data members
        */
        cout << "Falcon has left the Base";
    }
}

```

```

        int gamma()
        {
            i++;
        }
};

int main()
{
    StarWars objOne(10);    // non const object
    const StarWars objTwo(20); // const object

    objOne.falcon(); // No error
    objTwo.falcon(); // No error

    cout << objOne.i << objTwo.i;

    objOne.gamma(); // No error
    objTwo.gamma(); // Compile time error
}

```

Object

```

Falcon has left the Base
Falcon has left the Base
10 20

```

Here, we can see, that **const member function never changes data members of class, and it can be used with both const and non-const objects. But a const object cannot be used with a member function which tries to change its data members**(The this pointer passed to a const method is a pointer to a const object. That means the pointer can not be used to modify the object's data members. Any attempt to change a data member of the object that called a const method will result in a syntax error, as will attempting to call a non-const method for that object).

A const method can be overloaded with a non-const version. The choice of which version to use is made by the compiler based on the context in which the method is called.

Constructors and destructors can never be declared as const. They are always allowed to modify an object even if the object is const.

mutable Keyword

mutable keyword is used with member variables of class, which we want to change even if the object is of const type. Hence, mutable data members of a const objects can be modified.

```

class Zee
{
    int i;
    mutable int j;
public:
    Zee()
    {
        i = 0;
        j = 0;
    }
}

```

```

void fool() const
{
    i++; // will give error
    j++; // works, because j is mutable
}
};

int main()
{
    const Zee obj;
    obj.fool();
}

```

❖ Static data member and member function

➤ Static Data Members

In normal situations when we instantiate objects of a class each object gets its own copy of all normal member variables. When we declare a member of a class as **static** it means no matter how many objects of the class are created, there is **only one copy of the static member**. This means **one single copy** of that data member is **shared** between **all objects of that class**. All static data is **initialized to zero** when the first object is created, if no other initialization is present. Static data members can be initialized outside the class using the **scope resolution operator (::)** to identify which class it belongs to. Since Static data members are **class level variables**, we **do not require the object** to access these variables and they can be accessed simply by using the class name and **scope resolution(::) operator** with the variable name to access its value. Also a static data member cannot be private. A very common use of static data members is to keep a count of total objects of a particular class (program counter kind of application)

```

#include <iostream>
using namespace std;
class Cube {
    private:
        int side; // normal data member
    public:
        static int objectCount; // static data member
    // Constructor definition
    Cube()
    {
        // Increase every time object is created
        objectCount++;
    }
};
// Initialize static member of class Box
int Cube::objectCount = 0;

int main(void) {
    Cube c1;
    // Object Count.
    cout << "Total objects: " << Cube::objectCount << endl;
    Cube c2;
}

```

```

        // Object Count.
        cout << "Total objects: " << Cube::objectCount << endl;
        return 0;
    }

```

Output

```

    Total Objects: 1
    Total Objects: 2

```

Program Explanation

In this program, we have created a class Cube with 1 normal variable and 1 static variable. In the default constructor of this class we are incrementing the static variable objectCount value by 1. So everytime an object is created this value will be incremented. In the main() function we create 1 object and print the objectCount variable using the classname and (::) scope resolution operator. Since we created 1 object, its default constructor was called which incremented the value of objectCount variable by one and hence we get the output 1. Then we again create one more new object c2 and again print the objectCount value. This time however it prints 2 since the default constructor for object c2 was called again and it incremented the objectCount value by 1 again and since this is a static member it is shared between both the objects c1 and c2 so they will have same values that is 2.

➤ Static Member Functions

By declaring a function member as **static**, you make it **independent** of any particular object of the class. A **static member function** can be called even if no objects of the class exist and the static functions are accessed using only the class name and the **scope resolution operator (::)**

A static member function can **only access static data member**, other **static member functions** and any **other functions** from **outside** the class.

Static member functions have a **class scope** and they do not have access to the **this** pointer of the class.

```

#include <iostream>
using namespace std;
class Cube {
    private:
        int side; // normal data member
    public:
        static int objectCount; // static data member
        // Constructor definition
        Cube()
        {
            // Increase every time object is created
            objectCount++;
        }
        // creating a static function that returns static data member
        static int getCount() {
            return objectCount;
        }
};
// Initialize static member of class Box
int Cube::objectCount = 0;

int main(void) {
    Cube c1;
    // Object Count.

```

```

    cout << "Total objects: " << Cube::getCount() << endl;
    Cube c2;
    // Object Count.
    cout << "Total objects: " << Cube::getCount() << endl;
    return 0;
}

```

Output

Total Objects: 1
Total Objects: 2

Program Explanation

If you compare the first program and this program, the only difference is that in this program we have created a static member function named `getCount()` which returns the static data member `objectCount` value. Since `getCount` is a static member function, it can access only static data and can be directly called by using the scope resolution operator (`::`)

Some interesting facts about static member functions in C++

- static member functions do not have this pointer.
- A static member function cannot be virtual.
- Member function declarations with the same name and the name parameter-type-list cannot be overloaded if any of them is a static member function declaration.
- A static member function can not be declared `const`, `volatile`, or `const volatile`.

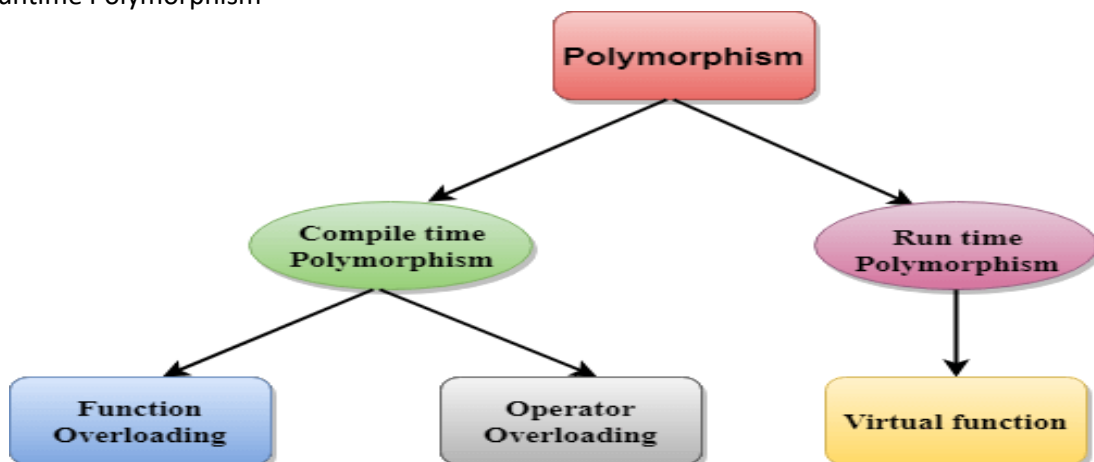
❖ Polymorphism

➤ Polymorphism in C++

The word polymorphism means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form. A real-life example of polymorphism, a person at the same time can have different characteristics. Like a man at the same time is a father, a husband, an employee. So the same person possesses different behavior in different situations. This is called polymorphism. Polymorphism is considered as one of the important features of Object Oriented Programming.

In C++ polymorphism is mainly divided into two types:

- Compile time Polymorphism
- Runtime Polymorphism



1. **Compile time polymorphism:** This type of polymorphism is achieved by **function overloading** or **operator overloading**.

Function Overloading: When there are multiple functions with same name but different parameters then these functions are said to be **overloaded**. Functions can be overloaded by **change in number of arguments** or/and **change in type of arguments**.

// C++ program for function overloading

```
#include <iostream>
using namespace std;
class Geeks
{
    public:
        // function with 1 int parameter
        void func(int x)
        {
            cout << "value of x is " << x << endl;
        }

        // function with same name but 1 double parameter
        void func(double x)
        {
            cout << "value of x is " << x << endl;
        }

        // function with same name and 2 int parameters
        void func(int x, int y)
        {
            cout << "value of x and y is " << x << ", " << y << endl;
        }
};

int main() {

    Geeks obj1;

    // Which function is called will depend on the parameters passed
    // The first 'func' is called
    obj1.func(7);

    // The second 'func' is called
    obj1.func(9.132);

    // The third 'func' is called
    obj1.func(85,64);
    return 0;
}
```

Output:

```
value of x is 7
value of x is 9.132
value of x and y is 85, 64
```

In the above example, a single function named func acts differently in three different situations which is the property of polymorphism.

Operator Overloading: C++ also provide option to overload operators. For example, we can make the operator '+' for string class to concatenate two strings. We know that this is the addition operator whose task is to add two operands. So a single operator '+' when placed between integer operands , adds them and when placed between string operands, concatenates them.

Example: adding two complex number using operator(+) overloading

```
// CPP program to illustrate
// Operator Overloading
#include<iostream>
using namespace std;

class Complex {
private:
    int real, imag;
public:
    Complex(int r = 0, int i = 0) {real = r;  imag = i;}

    // This is automatically called when '+' is used with
    // between two Complex objects
    Complex operator + (Complex const &obj) {
        Complex res;
        res.real = real + obj.real;
        res.imag = imag + obj.imag;
        return res;
    }
    void print() { cout << real << " + i" << imag << endl; }
};

int main()
{
    Complex c1(10, 5), c2(2, 4);
    Complex c3 = c1 + c2; // An example call to "operator+"
    c3.print();
}
```

Output:

12 + i9

In the above example the operator '+' is overloaded. The operator '+' is an addition operator and can add two numbers(integers or floating point) but here the operator is made to perform addition of two imaginary or complex numbers.

2. Runtime polymorphism: This type of polymorphism is achieved by Function Overriding.

Function overriding on the other hand occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be **overridden**.

// C++ program for function overriding

```
#include <bits/stdc++.h>
using namespace std;
class base
{
```

```

    public:
        virtual void print ()
        { cout<< "print base class" <<endl; }

        void show ()
        { cout<< "show base class" <<endl; }
};

class derived:public base
{
    public:
        void print () //print () is already virtual function in derived class, we could also declared
        //as virtual void print () explicitly
        { cout<< "print derived class" <<endl; }

        void show ()
        { cout<< "show derived class" <<endl; }
};

//main function
int main()
{
    base *bptr;
    derived d;
    bptr = &d;

    //virtual function, binded at runtime (Runtime polymorphism)
    bptr->print();

    // Non-virtual function, binded at compile time
    bptr->show();

    return 0;
}

```

Output:

```

print derived class
show base class

```

Virtual Function

A virtual function is another way of implementing run-time polymorphism in C++. It is a special function defined in a base class and redefined in the derived class. To declare a virtual function, you should use the virtual keyword. The keyword should precede the declaration of the function in the base class. If a virtual function class is inherited, the virtual class redefines the virtual function to suit its needs. Above example is used virtual function concept.

➤ **Compile-Time Polymorphism Vs. Run-Time Polymorphism**

Here are the major differences between the two:

Compile-time polymorphism	Run-time polymorphism
It's also called early binding or static polymorphism	It's also called late/dynamic binding or dynamic polymorphism
The method is called/invoked during compile time	The method is called/invoked during run time
Implemented via function overloading and operator overloading	Implemented via method overriding and virtual functions
Example, method overloading. Many methods may have similar names but different number or types of arguments	Example, method overriding. Many methods may have a similar name and the same prototype.
Faster execution since the methods discovery is done during compile time	Slower execution since method discoverer is done during runtime.
Less flexibility for problem-solving is provided since everything is known during compile time.	Much flexibility is provided for solving complex problems since methods are discovered during runtime.

❖ Operator overloading

➤ C++ Operators Overloading

Operator overloading is a compile-time polymorphism in which the operator is overloaded to provide the special meaning to the user-defined data type. Operator overloading is used to overload or redefines most of the operators available in C++. It is used to perform the operation on the user-defined data type. For example, C++ provides the ability to add the variables of the user-defined data type that is applied to the built-in data types.

The advantage of Operators overloading is to perform different operations on the same operand.

Operator that cannot be overloaded are as follows:

Scope operator (::)

Sizeof

member selector (.)

member pointer selector ()*

ternary operator (?:)

Syntax of Operator Overloading

```
return_type class_name :: operator op(argument_list)
{
    // body of the function.
}
```

Where the **return type** is the type of value returned by the function.

class_name is the name of the class.

operator op is an operator function where op is the operator being overloaded, and the operator is the keyword.

Operator Overloading can be done by using three approaches, they are

1. Overloading unary operator.
2. Overloading binary operator.
3. Overloading binary operator using a friend function.

Rules for Operator Overloading

- Existing operators can only be overloaded, but the new operators cannot be overloaded.
- The overloaded operator contains atleast one operand of the user-defined data type.
- We cannot use friend function to overload certain operators. However, the member function can be used to overload those operators = () [] ->
- When unary operators are overloaded through a member function take no explicit arguments, but, if they are overloaded by a friend function, takes one argument.
- When binary operators are overloaded through a member function takes one explicit argument, and if they are overloaded through a friend function takes two explicit arguments.

1. Unary operator overloading

The unary operators operate on a single operand and following are the examples of Unary operators –

- The increment (++) and decrement (--) operators.
- The unary minus (-) operator.
- The logical not (!) operator.

The unary operators operate on the object for which they were called and normally, this operator appears on the left side of the object, as in !obj, -obj, and ++obj but sometime they can be used as postfix as well like obj++ or obj--.

//++ operator overloading

```
#include <iostream>
using namespace std;
class Test
{
    private:
        int num;
    public:
        Test(): num(8){}
        void operator ++()    {
            num = num+2;
        }
        void Print() {
            cout<<"The Count is: "<<num;
        }
};
int main()
{
    Test tt;
    ++tt; // calling of a function "void operator ++()"
    tt.Print();
    return 0;
}
```

Output:

The Count is: 10

2. Binary operator overloading

The binary operators take two arguments and following are the examples of Binary operators. You use binary operators very frequently like addition (+) operator, subtraction (-) operator and division (/) operator

// + operator overloading

```
#include <iostream>
using namespace std;
class A
{
    int x;
    public:
    A(){}
    A(int i)
    {
        x=i;
    }
    void operator+(A);
    void display();
};

void A :: operator+(A a)
{
    int m = x+a.x;
    cout<<"The result of the addition of two objects is : "<<m;

}
int main()
{
    A a1(5);
    A a2(4);
    a1+a2;
    return 0;
}
```

Output:

The result of the addition of two objects is : 9

// < operator overloading

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

class Distance {
    private:
        int feet;        // 0 to infinite
        int inches;      // 0 to 12

    public:
        // required constructors
        Distance() {
```

```

    feet = 0;
    inches = 0;
}
Distance(int f, int i) {
    feet = f;
    inches = i;
}

// method to display distance
void displayDistance() {
    cout << "F: " << feet << "I:" << inches << endl;
}

// overloaded minus (-) operator
Distance operator- () {
    feet = -feet;
    inches = -inches;
    return Distance(feet, inches);
}

// overloaded < operator
bool operator <(const Distance& d) {
    if(feet < d.feet) {
        return true;
    }
    if(feet == d.feet && inches < d.inches) {
        return true;
    }

    return false;
}
};

int main() {
    Distance D1(11, 10), D2(5, 11);

    if( D1 < D2 ) {
        cout << "D1 is less than D2 " << endl;
    } else {
        cout << "D2 is less than D1 " << endl;
    }

    return 0;
}

```

Output

D2 is less than D1

// < operator overloading

You can overload the assignment operator (=) just as you can other operators and it can be used to

create an object just like the copy constructor.

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

class Distance {
private:
    int feet;      // 0 to infinite
    int inches;    // 0 to 12

public:
    // required constructors
    Distance() {
        feet = 0;
        inches = 0;
    }
    Distance(int f, int i) {
        feet = f;
        inches = i;
    }
    void operator = (const Distance &D ) {
        feet = D.feet;
        inches = D.inches;
    }

    // method to display distance
    void displayDistance() {
        cout << "F: " << feet << " I:" << inches << endl;
    }
};

int main() {
    Distance D1(11, 10), D2(5, 11);

    cout << "First Distance : ";
    D1.displayDistance();
    cout << "Second Distance :";
    D2.displayDistance();

    // use assignment operator
    D1 = D2;
    cout << "First Distance :";
    D1.displayDistance();

    return 0;
}
```

Output

```
First Distance : F: 11 I:10
Second Distance :F: 5 I:11
```


First Distance :F: 5 I:11

3. Overloading operator using friend function

// + operator overloading

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

```
class Distance {
public:
```

```
    // Member Object
    int feet, inch;
```

```
    // No Parameter Constructor
    Distance()
    {
        this->feet = 0;
        this->inch = 0;
    }
```

```
    // Constructor to initialize the object's value
    // Parametrized Constructor
    Distance(int f, int i)
    {
        this->feet = f;
        this->inch = i;
    }
```

```
    // Declaring friend function using friend keyword
    friend Distance operator+(Distance&, Distance&);
};
```

```
// Implementing friend function with two parameters
Distance operator+(Distance& d1, Distance& d2) // Call by reference
{
    // Create an object to return
    Distance d3;

    // Perform addition of feet and inches
    d3.feet = d1.feet + d2.feet;
    d3.inch = d1.inch + d2.inch;

    // Return the resulting object
    return d3;
}
```

```
// Driver Code
int main()
{
```

```

// Declaring and Initializing first object
Distance d1(8, 9);

// Declaring and Initializing second object
Distance d2(10, 2);

// Declaring third object
Distance d3;

// Use overloaded operator
d3 = d1 + d2;

// Display the result
cout << "\nTotal Feet & Inches: " << d3.feet << " " << d3.inch;
return 0;
}

```

Output:

Total Feet & Inches: 18'11

// extraction operator >> and insertion operator << overloading

The stream insertion and stream extraction operators also can be overloaded to perform input and output for user-defined types like an object.

Here, it is important to make operator overloading function a friend of the class because it would be called without creating an object

```

#include <iostream>
using namespace std;

class Distance {
private:
    int feet;      // 0 to infinite
    int inches;    // 0 to 12

public:
    // required constructors
    Distance() {
        feet = 0;
        inches = 0;
    }
    Distance(int f, int i) {
        feet = f;
        inches = i;
    }
    friend ostream &operator<<( ostream &output, const Distance &D ) {
        output << "F : " << D.feet << " I : " << D.inches;
        return output;
    }

    friend istream &operator>>( istream &input, Distance &D ) {
        input >> D.feet >> D.inches;
    }
}

```

```

        return input;
    }
};

int main() {
    Distance D1(11, 10), D2(5, 11), D3;

    cout << "Enter the value of object : " << endl;
    cin >> D3;
    cout << "First Distance : " << D1 << endl;
    cout << "Second Distance : " << D2 << endl;
    cout << "Third Distance : " << D3 << endl;

    return 0;
}

```

Output

```

Enter the value of object :
6
8
First Distance : F : 11 I : 10
Second Distance : F : 5 I : 11
Third Distance : F : 6 I : 8

```

❖ Dynamic binding and virtual function

➤ Early binding and Late binding in C++

Binding refers to the process of converting identifiers (such as variable and performance names) into addresses. Binding is done for each variable and functions. For functions, it means that matching the call with the right function definition by the compiler. It takes place either at compile time or at runtime.

Early Binding or Static Binding or Compile-time Binding

In early binding, the compiler matches the function call with the correct function definition at compile time. It is also known as **Static Binding** or **Compile-time Binding**. By default, the compiler goes to the function definition which has been called during compile time. So, all the function calls you have studied till now are due to early binding.

We have learned about function overriding in which the base and derived classes have functions with the same name, parameters and return type. In that case also, early binding takes place.

Late Binding or Dynamic Binding or Runtime Binding

In the case of late binding, the compiler matches the function call with the correct function definition at runtime. It is also known as **Dynamic Binding** or **Runtime Binding**.

In late binding, the compiler identifies the type of object at runtime and then matches the function call with the correct function definition.

By default, early binding takes place. So if by any means we tell the compiler to perform late binding, This can be achieved by declaring a **virtual function**.

```

#include <iostream>
using namespace std;
class base {
public:
    virtual void print()

```

```

        {
            cout << "print base class" << endl;
        }

        void show()
        {
            cout << "show base class" << endl;
        }
};

class derived : public base {
public:
    void print()
    {
        cout << "print derived class" << endl;
    }

    void show()
    {
        cout << "show derived class" << endl;
    }
};

int main()
{
    base* bptr;
    derived d;
    bptr = &d;

    // virtual function, binded at runtime
    bptr->print();

    // Non-virtual function, binded at compile time
    bptr->show();
}

```

Output

```

print derived class
show base class

```

Explanation: Runtime polymorphism is achieved only through a pointer (or reference) of base class type. Also, a base class pointer can point to the objects of base class as well as to the objects of derived class. In above code, base class pointer 'bptr' contains the address of object 'd' of derived class.

Late binding(Runtime) is done in accordance with the content of pointer (i.e. location pointed to by pointer) and Early binding(Compile time) is done according to the type of pointer, since print() function is declared with virtual keyword so it will be bound at run-time (output is *print derived class* as pointer is pointing to object of derived class) and show() is non-virtual so it will be bound during compile time(output is *show base class* as pointer is of base type).

➤ *Virtual function*

A virtual function is a member function which is declared within a base class and is re-defined(Overriden) by a derived class. When you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class's version of the function.

- Virtual functions ensure that the correct function is called for an object, regardless of the type of reference (or pointer) used for function call.
- They are mainly used to achieve Runtime polymorphism
- Functions are declared with a **virtual** keyword in base class.
- The resolving of function call is done at Run-time.

Rules for Virtual Functions

1. Virtual functions cannot be static and also cannot be a friend function of another class.
2. Virtual functions should be accessed using pointer or reference of base class type to achieve run time polymorphism.
3. The prototype of virtual functions should be same in base as well as derived class.
4. They are always defined in base class and overridden in derived class. It is not mandatory for derived class to override (or re-define the virtual function), in that case base class version of function is used.
5. A class may have virtual destructor but it cannot have a virtual constructor.

Problem without Virtual Keyword

Let's try to understand what is the issue that virtual keyword fixes,

```
#include<iostream>
using namespace std;
class Base
{
    public:
    void show()
    {
        cout << "Base class";
    }
};

class Derived:public Base
{
    public:
    void show()
    {
        cout << "Derived Class";
    }
};

int main()
{
    Base* b;    //Base class pointer
    Derived d;  //Derived class object
    b = &d;
    b->show();  //Early Binding Occurs
}
```

Output

Base class

When we use Base class's pointer to hold Derived class's object, base class pointer or reference will always call the base version of the function.

Using Virtual Keyword in C++

We can make base class's methods virtual by using virtual keyword while declaring them. Virtual keyword will lead to Late Binding of that method.

```
#include<iostream>
using namespace std;
class Base
{
    public:
    virtual void show()
    {
        cout << "Base class\n";
    }
};

class Derived:public Base
{
    public:
    void show()
    {
        cout << "Derived Class";
    }
};

int main()
{
    Base* b;    //Base class pointer
    Derived d;  //Derived class object
    b = &d;
    b->show();  //Late Binding Occurs
}
```

Output

Derived class

On using Virtual keyword with Base class's function, Late Binding takes place and the derived version of function will be called, because base class pointer points to Derived class object.

Using Virtual Keyword and Accessing Private Method of Derived class

We can call private function of derived class from the base class pointer with the help of virtual keyword. Compiler checks for access specifier only at compile time. So at run time when late binding occurs it does not check whether we are calling the private function or public function.

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

class A
{
    public:
```

```

virtual void show()
{
    cout << "Base class\n";
}

};

class B: public A
{
private:
    virtual void show()
    {
        cout << "Derived class\n";
    }
};

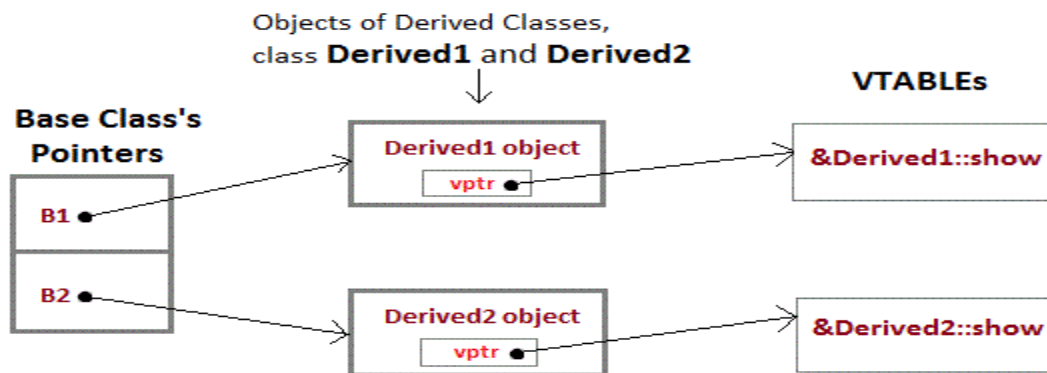
int main()
{
    A *a;
    B b;
    a = &b;
    a->show();
}

```

Output

Derived class

Mechanism of Late Binding in C++



vptr, is the vpointer, which points to the Virtual Function for that object.

VTABLE, is the table containing address of Virtual Functions of each class.

To accomplish late binding, Compiler creates **VTABLEs**, for each class with virtual function. The address of virtual functions is inserted into these tables. Whenever an object of such class is created the compiler secretly inserts a pointer called **vpointer**, pointing to VTABLE for that object. Hence when function is called, compiler is able to resolve the call by binding the correct function using the vpointer.

Important Points to Remember

1. Only the Base class Method's declaration needs the **Virtual** Keyword, not the definition.
2. If a function is declared as **virtual** in the base class, it will be virtual in all its derived classes.
3. The address of the virtual Function is placed in the **VTABLE** and the copiler uses **VPTR**(vpointer) to point to the Virtual Function.

UNIT – 5

❖ Exception handling

➤ *Exception Handling in C++*

Errors can be broadly categorized into two types.

1. Compile Time Errors
2. Run Time Errors

Compile Time Errors – Errors caught during compiled time is called Compile time errors. Compile time errors include library reference, syntax error or incorrect class import.

Run Time Errors - They are also known as **exceptions**. An exception caught during run time creates serious issues.

Errors hinder normal execution of program. Exception handling is the process of handling errors and exceptions in such a way that they do not hinder normal execution of the system. For example, User divides a number by zero, this will compile successfully but an exception or run time error will occur due to which our applications will be crashed. In order to avoid this we'll introduce exception handling technics in our code.

In C++, Error handling is done using three keywords:

- try
- catch
- throw

Syntax:

```
try
{
    //code
    throw parameter;
}
catch(exceptionname ex)
{
    //code to handle exception
}
```

try block

The code which can throw any exception is kept inside(or enclosed in) a **try** block. Then, when the code will lead to any error, that error/exception will get caught inside the **catch** block.

catch block

catch block is intended to catch the error and handle the exception condition. We can have multiple catch blocks to handle different types of exception and perform different actions when the exceptions occur. For example, we can display descriptive messages to explain why any particular excpetion occurred.

throw statement

It is used to throw exceptions to exception handler i.e. it is used to communicate information about error. A **throw** expression accepts one parameter and that parameter is passed to handler.

throw statement is used when we explicitly want an exception to occur, then we can use **throw** statement to throw or generate that exception.

Understanding Need of Exception Handling

Below program compiles successfully but the program fails at runtime, leading to an exception.

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

```

using namespace std;
int main()
{
    int a=10,b=0,c;
    c=a/b;
    cout<<"last line of program"<<endl;
    return 0;
}

```

The above program will not run, and will show runtime error on screen, because we are trying to divide a number with 0, which is not possible.

How to handle this situation? We can handle such situations using exception handling and can inform the user that you cannot divide a number by zero, by displaying a message.

1) Using try, catch and throw Statement

Now we will update the above program and include exception handling in it.

```

#include <iostream>
#include<conio.h>
using namespace std;
int main()
{
    int a=10, b=0, c;
    // try block activates exception handling
    try
    {
        if(b == 0)
        {
            // throw custom exception
            throw "Division by zero not possible";
            c = a/b;
        }
    }
    catch(char const* ex)// catches exception
    {
        cout<<ex;
    }
    cout<<"\nlast line of program"<<endl;
    return 0;
}

```

Output:

```

Division by zero not possible
Last line of program

```

In the code above, we are checking the divisor, if it is zero, we are throwing an exception message, then the catch block catches that exception and prints the message.

The catch statement takes a **parameter**: in our example we use a char const* variable (ex) (because we are throwing an exception of string type in the try block).

Doing so, the user will never know that our program failed at runtime, he/she will only see the message "Division by zero not possible".

This is gracefully handling the exception condition which is why exception handling is used.

2) Handle Any Type of Exceptions (...) or Generalized catch block

If you do not know the throw type used in the try block, you can use the "three dots" syntax (...) inside the catch block, which will handle any type of exception:

Example

```
try {
    int age = 15;
    if (age >= 18) {
        cout << "Access granted - you are old enough.";
    } else {
        throw 505;
    }
}
catch (...) {
    cout << "Access denied - You must be at least 18 years old.\n";
}
```

Output:

Access denied - You must be at least 18 years old.

3) Using Multiple catch blocks

Below program contains multiple catch blocks to handle different types of exception in different way.

```
#include <iostream>
#include <conio.h>
using namespace std;

int main()
{
    int x[3] = {-1,2};
    for(int i=0; i<2; i++)
    {
        int ex = x[i];
        try
        {
            if (ex > 0)
                // throwing numeric value as exception
                throw ex;
            else
                // throwing a character as exception
                throw 'ex';
        }
        catch (int ex) // to catch numeric exceptions
        {
            cout << "Integer exception\n";
        }
        catch (char ex) // to catch character/string exceptions
        {
            cout << "Character exception\n";
        }
    }
}
```

Output:

Character exception

Integer exception

The above program is self-explanatory, if the value of integer in the array x is less than 0, we are throwing a numeric value as exception and if the value is greater than 0, then we are throwing a character value as exception. And we have two different catch blocks to catch those exceptions.

4) Implicit type conversion doesn't happen for primitive types. For example, in the following program 'a' is not implicitly converted to int

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

int main()
{
    try {
        throw 'a';
    }
    catch (int x) {
        cout << "Caught " << x;
    }
    catch (...) {
        cout << "Default Exception\n";
    }
    return 0;
}
```

Output:

Default Exception

5) If an exception is thrown and not caught anywhere, the program terminates abnormally. For example, in the following program, a char is thrown, but there is no catch block to catch a char.

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

int main()
{
    try {
        throw 'a';
    }
    catch (int x) {
        cout << "Caught ";
    }
    return 0;
}
```

Output:

terminate called after throwing an instance of 'char'

We can change this abnormal termination behavior by writing our own unexpected function.

Customizing termination behavior for uncaught exception

Whenever an exception arises in C++, it is handled as per the behavior defined using the try-catch block. However, there is often the case when an exception is thrown but isn't caught because the exception handling subsystem fails to find a matching catch block for that particular exception. In that case, the following set of actions takes place:

The exception handling subsystem calls the function: **unexpected()**. This function, provided by the default C++ library, defines the behavior when an uncaught exception arises. By default, **unexpected** calls **terminate()**.

The terminate function defines the actions that are to be performed during process termination. This, **by default, calls abort()**.

The process is aborted.

The terminate() and unexpected() simply call other functions to actually handle an error. As explained above, terminate calls abort(), and unexpected() calls terminate(). Thus, both functions halt the program execution when an exception handling error occurs. However, you can change the way termination occurs.

To change the terminate handler, the function used is set_terminate(terminate_handler newhandler), which is defined in the header <exception>.

The following program demonstrates how to set a custom termination handler:

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

// definition of custom termination function
void myhandler()
{
    cout << "Inside new terminate handler\n";
    abort();
}

int main()
{
    // set new terminate handler
    set_terminate(myhandler);
    try {
        cout << "Inside try block\n";
        throw 'a';
    }
    catch (int a) // won't catch an int exception
    {
        cout << "Inside catch block\n";
    }
    return 0;
}
```

Output:

```
Inside try block
Inside new terminate handler
```

6) A derived class exception should be caught before a base class exception.

Exception Handling – catching base and derived classes as exceptions:

If both base and derived classes are caught as exceptions then catch block of derived class must appear

before the base class.

If we put base class first then the derived class catch block will never be reached. For example, following C++ code prints “Caught Base Exception”

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

class Base {};
class Derived: public Base {};
int main()
{
    Derived d;
    // some other stuff
    try {
        // Some monitored code
        throw d;
    }
    catch(Base b) {
        cout<<"Caught Base Exception";
    }
    catch(Derived d) { //This catch block is NEVER executed
        cout<<"Caught Derived Exception";
    }
    return 0;
}
```

In the above C++ code, if we change the order of catch statements then both catch statements become reachable. Following is the modified program and it prints “Caught Derived Exception”

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

class Base {};
class Derived: public Base {};
int main()
{
    Derived d;
    // some other stuff
    try {
        // Some monitored code
        throw d;
    }
    catch(Derived d) {
        cout<<"Caught Derived Exception";
    }
    catch(Base b) {
        cout<<"Caught Base Exception";
    }
    return 0;
}
```

7) Like Java, C++ library has a standard exception class which is base class for all standard exceptions. All objects thrown by components of the standard library are derived from this class. Therefore, all standard exceptions can be caught by catching this type

8) Unlike Java, in C++, all exceptions are unchecked. Compiler doesn't check whether an exception is caught or not. For example, in C++, it is not necessary to specify all uncaught exceptions in a function declaration. Although it's a recommended practice to do so. For example, the following program compiles fine, but ideally signature of fun() should list unchecked exceptions.

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

// This function signature is fine by the compiler, but not recommended.
// Ideally, the function should specify all uncaught exceptions and function
// signature should be "void fun(int *ptr, int x) throw (int *, int)"
void fun(int *ptr, int x)
{
    if (ptr == NULL)
        throw ptr;
    if (x == 0)
        throw x;
    /* Some functionality */
}

int main()
{
    try {
        fun(NULL, 0);
    }
    catch(...) {
        cout << "Caught exception from fun()";
    }
    return 0;
}
```

Output:

Caught exception from fun()

A better way to write above code

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

// Here we specify the exceptions that this function
// throws.
void fun(int *ptr, int x) throw (int *, int)
{
    if (ptr == NULL)
        throw ptr;
    if (x == 0)
        throw x;
    /* Some functionality */
}
```

```

int main()
{
    try {
        fun(NULL, 0);
    }
    catch(...) {
        cout << "Caught exception from fun()";
    }
    return 0;
}

```

Output:

Caught exception from fun()

9) In C++, try-catch blocks can be nested. Also, an exception can be re-thrown using “throw;”

```

#include <iostream>
using namespace std;

int main()
{
    try {
        try {
            throw 20;
        }
        catch (int n) {
            cout << "Handle Partially ";
            throw; // Re-throwing an exception
        }
    }
    catch (int n) {
        cout << "Handle remaining ";
    }
    return 0;
}

```

Output:

Handle Partially Handle remaining

A function can also re-throw a function using same “throw;”. A function can handle a part and can ask the caller to handle remaining.

10) When an exception is thrown, all objects created inside the enclosing try block are destructed before the control is transferred to catch block.

```

#include <iostream>
using namespace std;

class Test {
public:
    Test() { cout << "Constructor of Test " << endl; }
    ~Test() { cout << "Destructor of Test " << endl; }
};

```



```

int main()
{
    try {
        Test t1;
        throw 10;
    }
    catch (int i) {
        cout << "Caught " << i << endl;
    }
}

```

Output:

```

Constructor of Test
Destructor of Test
Caught 10

```

❖ Template

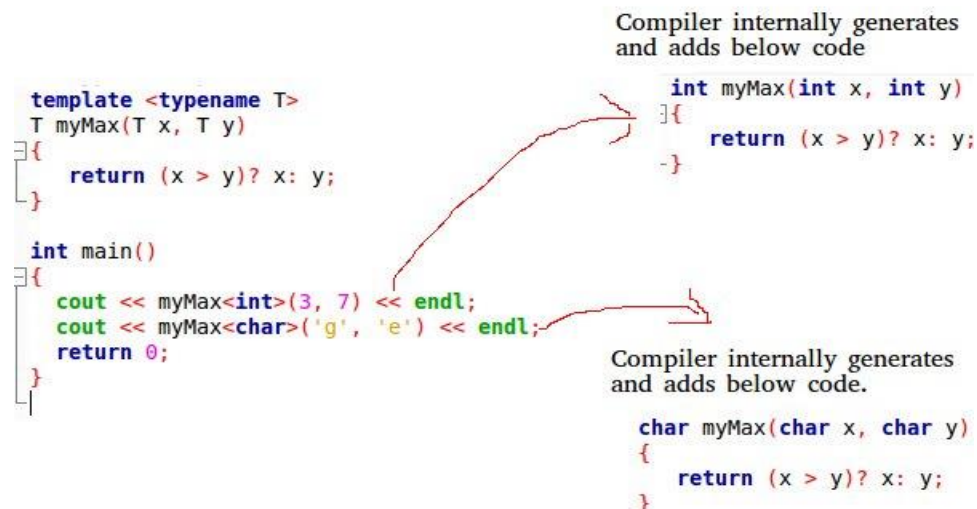
➤ Templates in C++

A template is a simple and yet very powerful tool in C++. The simple idea is to pass data type as a parameter so that we don't need to write the same code for different data types. For example, a software company may need sort() for different data types. Rather than writing and maintaining the multiple codes, we can write one sort() and pass data type as a parameter.

C++ adds two new keywords to support templates: **'template'** and **'typename'**. The second keyword can always be replaced by keyword **'class'**.

How templates work?

Templates are expanded at compiler time. This is like macros. The difference is, compiler does type checking before template expansion. The idea is simple, source code contains only function/class, but compiled code may contain multiple copies of same function/class.



Function Templates:

We write a generic function that can be used for different data types. Examples of function templates are sort(), max(), min(), printArray().

Syntax:

template <class identifier> function_declaration;

template <typename identifier> function_declaration;

Example:

```
#include <iostream>
using namespace std;
// One function works for all data types. This would work
// even for user defined types if operator '>' is overloaded
template <typename T>
T myMax(T x, T y)
{
    return (x > y)? x: y;
}

int main()
{
    cout << myMax<int>(3, 7) << endl; // Call myMax for int
    cout << myMax<double>(3.0, 7.0) << endl; // call myMax for double
    cout << myMax<char>('g', 'e') << endl; // call myMax for char

    return 0;
}
```

Output:

```
7
7
g
```

Below is the program to implement Bubble Sort using templates in C++:

```
#include <iostream>
using namespace std;
// We can use this for any data type that supports
// comparison operator < and swap works for it.
template <class T>
void bubbleSort(T a[], int n) {
    for (int i = 0; i < n - 1; i++)
        for (int j = n - 1; i < j; j--)
            if (a[j] < a[j - 1])
                swap(a[j], a[j - 1]);
}

// Driver Code
int main() {
    int a[5] = {10, 50, 30, 40, 20};
    int n = sizeof(a) / sizeof(a[0]);

    // calls template function
    bubbleSort<int>(a, 5);

    cout << " Sorted array : ";
```

```

        for (int i = 0; i < n; i++)
            cout << a[i] << " ";
        cout << endl;

    return 0;
}

```

Output:

Sorted array : 10 20 30 40 50

Class Templates :

class templates are useful when a class defines something that is independent of the data type. Can be useful for classes like LinkedList, BinaryTree, Stack, Queue, Array, etc.

Example:

```

#include <iostream>
using namespace std;

template <typename T>
class Array {
private:
    T *ptr;
    int size;
public:
    Array(T arr[], int s);
    void print();
};

template <typename T>
Array<T>::Array(T arr[], int s) {
    ptr = new T[s];
    size = s;
    for(int i = 0; i < size; i++)
        ptr[i] = arr[i];
}

template <typename T>
void Array<T>::print() {
    for (int i = 0; i < size; i++)
        cout << " "<<*(ptr + i);
    cout << endl;
}

int main() {
    int arr[5] = {1, 2, 3, 4, 5};
    Array<int> a(arr, 5);
    a.print();
    return 0;
}

```

Output:

1 2 3 4 5

➤ *Some important point of template*

Can there be more than one arguments to templates?

Yes, like normal parameters, we can pass more than one data types as arguments to templates. The following example demonstrates the same.

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

template<class T, class U>
class A {
    T x;
    U y;
public:
    A() { cout<<"Constructor Called"<<endl; }
};

int main() {
    A<char, char> a;
    A<int, double> b;
    return 0;
}
```

Output:

```
Constructor Called
Constructor Called
```

Can we specify default value for template arguments?

Yes, like normal parameters, we can specify default arguments to templates. The following example demonstrates the same.

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

template<class T, class U = char>
class A {
public:
    T x;
    U y;
    A() { cout<<"Constructor Called"<<endl; }
};

int main() {
    A<char> a; // This will call A<char, char>
    return 0;
}
```

Output:

```
Constructor Called
```

What is the difference between function overloading and templates?

Both function overloading and templates are examples of polymorphism feature of OOP. Function overloading is used when multiple functions do similar operations, templates are used when multiple functions do identical operations.

Can we pass nontype parameters to templates?

We can pass non-type arguments to templates. Non-type parameters are mainly used for specifying max or min values or any other constant value for a particular instance of a template. The important thing to note about non-type parameters is, they must be const. The compiler must know the value of non-type parameters at compile time. Because compiler needs to create functions/classes for a specified non-type value at compile time. In below program, if we replace 10000 or 256 with a variable, we get a compiler error.

// A C++ program to demonstrate working of non-type parameters to templates in C++.

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

template <class T, int max>
int arrMin(T arr[], int n)
{
    int m = max;
    for (int i = 0; i < n; i++)
        if (arr[i] < m)
            m = arr[i];
    return m;
}

int main()
{
    int arr1[] = {10, 20, 15, 12};
    int n1 = sizeof(arr1)/sizeof(arr1[0]);

    char arr2[] = {1, 2, 3};
    int n2 = sizeof(arr2)/sizeof(arr2[0]);

    // Second template parameter to arrMin must be a constant
    cout << arrMin<int, 10000>(arr1, n1) << endl;
    cout << arrMin<char, 256>(arr2, n2);
    return 0;
}
```

Output:

```
10
1
```

➤ Generics in C++

Generics is the idea to allow type (Integer, String, ... etc and user-defined types) to be a parameter to methods, classes and interfaces. For example, classes like an array, map, etc, which can be used using generics very efficiently. We can use them for any type.

The method of Generic Programming is implemented to increase the efficiency of the code. Generic Programming enables the programmer to write a general algorithm which will work with all data types. It eliminates the need to create different algorithms if the data type is an integer, string or a character.

The advantages of Generic Programming are

- Code Reusability
- Avoid Function Overloading
- Once written it can be used for multiple times and cases.

Generics can be implemented in C++ using Templates

Generic Functions using Template:

We write a generic function that can be used for different data types.

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

// One function works for all data types.
// This would work even for user defined types
// if operator '>' is overloaded
template <typename T>

T myMax(T x, T y)
{
    return (x > y) ? x : y;
}

int main()
{

    // Call myMax for int
    cout << myMax<int>(3, 7) << endl;

    // call myMax for double
    cout << myMax<double>(3.0, 7.0) << endl;

    // call myMax for char
    cout << myMax<char>('g', 'e') << endl;

    return 0;
}
```

Output:

```
7
7
g
```

Generic Class using Template:

Like function templates, class templates are useful when a class defines something that is independent of data type. Can be useful for classes like LinkedList, binary tree, Stack, Queue, Array, etc.

Following is a simple example of template Array class.

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

template <typename T>
class Array {
private:
    T* ptr;
    int size;

public:
```

```

    Array(T arr[], int s);
    void print();
};

template <typename T>
Array<T>::Array(T arr[], int s)
{
    ptr = new T[s];
    size = s;
    for (int i = 0; i < size; i++)
        ptr[i] = arr[i];
}

template <typename T>
void Array<T>::print()
{
    for (int i = 0; i < size; i++)
        cout << " " << *(ptr + i);
    cout << endl;
}

int main()
{
    int arr[5] = { 1, 2, 3, 4, 5 };
    Array<int> a(arr, 5);
    a.print();
    return 0;
}

```

Output:

```
1 2 3 4 5
```

Working with multi-type Generics:

```

#include <iostream>
using namespace std;

template <class T, class U>
class A {
    T x;
    U y;

public:
    A()
    {
        cout << "Constructor Called" << endl;
    }
};

int main()
{

```

```

    A<char, char> a;
    A<int, double> b;
    return 0;
}

```

Output:

```

    Constructor Called
    Constructor Called

```

➤ *template with static keyword*

Each instance of a template contains its own static variable.

Function templates and static variables:

Each instantiation of function template has its own copy of local static variables. For example, in the following program there are two instances: void fun(int) and void fun(double). So two copies of static variable i exist.

```

#include <iostream>
using namespace std;

template <typename T>
void fun(const T& x)
{
    static int i = 10;
    cout << ++i;
    return;
}

int main()
{
    fun<int>(1); // prints 11
    cout << endl;
    fun<int>(2); // prints 12
    cout << endl;
    fun<double>(1.1); // prints 11
    cout << endl;
    return 0;
}

```

Output

```

11
12
11

```

Class templates and static variables:

The rule for class templates is same as function templates

Each instantiation of class template has its own copy of member static variables. For example, in the following program there are two instances Test. So two copies of static variable count exist.

```

#include <iostream>
using namespace std;

template <class T>
class Test {
    private:

```



```

        T val;
    public:
        static int count;
        Test()
        {
            count++;
        }
        // some other stuff in class
};

template<class T>
int Test<T>::count = 0;

int main()
{
    Test<int> a;           // value of count for Test<int> is 1 now
    Test<int> b;           // value of count for Test<int> is 2 now
    Test<double> c;        // value of count for Test<double> is 1 now
    cout << Test<int>::count << endl;    // prints 2
    cout << Test<double>::count << endl; // prints 1
    return 0;
}

```

Output:

```

2
1

```

➤ **template specialization**

Template specialization allows us to have different code for a particular data type.

Template in C++ is a feature. We write code once and use it for any data type including user defined data types. For example, `sort()` can be written and used to sort any data type items. A class stack can be created that can be used as a stack of any data type.

What if we want a different code for a particular data type? Consider a big project that needs a function `sort()` for arrays of many different data types. Let Quick Sort be used for all datatypes except `char`. In case of `char`, total possible values are 256 and counting sort may be a better option. Is it possible to use different code only when `sort()` is called for `char` data type?

It is possible in C++ to get a special behavior for a particular data type. This is called template specialization.

// A generic sort function

```

template <class T>
void sort(T arr[], int size)
{
    // code to implement Quick Sort
}

```

// Template Specialization: A function specialized for char data type

```

template <>
void sort<char>(char arr[], int size)
{
    // code to implement counting sort
}

```

```
}
```

Another example could be a class Set that represents a set of elements and supports operations like union, intersection, etc. When the type of elements is char, we may want to use a simple boolean array of size 256 to make a set. For other data types, we have to use some other complex technique.

An Example Program for function template specialization

For example, consider the following simple code where we have general template fun() for all data types except int. For int, there is a specialized version of fun().

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

template <class T>
void fun(T a)
{
    cout << "The main template fun(): "
          << a << endl;
}

template<>
void fun(int a)
{
    cout << "Specialized Template for int type: "
          << a << endl;
}

int main()
{
    fun<char>('a');
    fun<int>(10);
    fun<float>(10.14);
}
```

Output:

```
The main template fun(): a
Specialized Template for int type: 10
The main template fun(): 10.14
```

An Example Program for class template specialization

In the following program, a specialized version of class Test is written for int data type.

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

template <class T>
class Test
{
    // Data members of test
public:
    Test()
    {
        // Initialization of data members
        cout << "General template object \n";
    }
}
```

```

    }
    // Other methods of Test
};

template <>
class Test <int>
{
public:
    Test()
    {
        // Initialization of data members
        cout << "Specialized template object\n";
    }
};

int main()
{
    Test<int> a;
    Test<char> b;
    Test<float> c;
    return 0;
}

```

Output:

```

Specialized template object
General template object
General template object

```

How does template specialization work?

When we write any template based function or class, compiler creates a copy of that function/class whenever compiler sees that being used for a new data type or new set of data types(in case of multiple template arguments).

If a specialized version is present, compiler first checks with the specialized version and then the main template. Compiler first checks with the most specialized version by matching the passed parameter with the data type(s) specified in a specialized version.

➤ **What is template metaprogramming?**

Predict the output of following C++ program.

```

#include <iostream>
using namespace std;
template<int n> struct funStruct
{
    enum { val = 2*funStruct<n-1>::val };
};

template<> struct funStruct<0>
{
    enum { val = 1 };
};

int main()
{
    cout << funStruct<8>::val << endl;
}

```

```

    return 0;
}

```

Output:

256

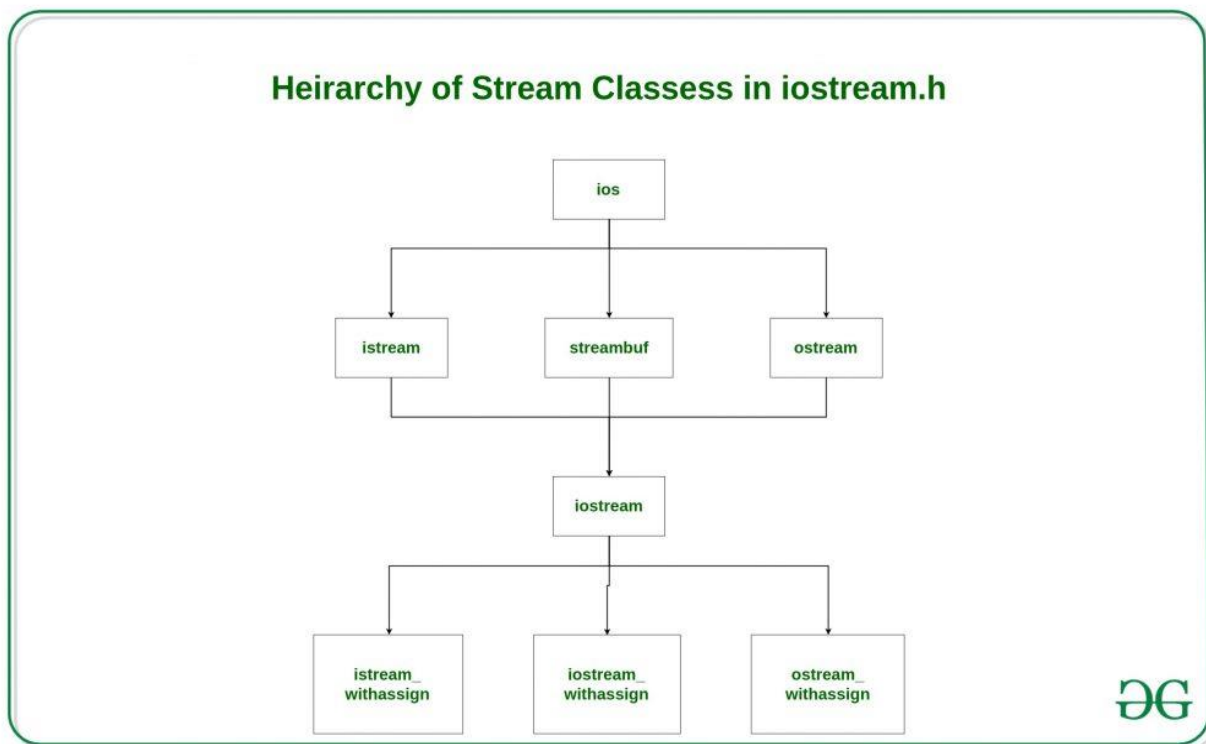
The program calculates “2 raise to the power 8 (or 2^8)”. In fact, the structure funStruct can be used to calculate 2^n for any known n (or constant n). The special thing about above program is: calculation is done at compile time. So, it is compiler that calculates 2^8 . To understand how compiler does this, let us **consider the following facts about templates and enums:**

- 1) We can pass nontype parameters (parameters that are not data types) to class/function templates.
- 2) Like other const expressions, values of enumeration constants are evaluated at compile time.
- 3) When compiler sees a new argument to a template, compiler creates a new instance of the template. Let us take a closer look at the original program. When compiler sees funStruct<8>::val, it tries to create an instance of funStruct with parameter as 8, it turns out that funStruct<7> must also be created as enumeration constant val must be evaluated at compile time. For funStruct<7>, compiler need funStruct<6> and so on. Finally, compiler uses funStruct<1>::val and compile time recursion terminates. So, using templates, we can write programs that do computation at compile time, such programs are called template metaprograms. Template metaprogramming is in fact Turing-complete, meaning that any computation expressible by a computer program can be computed, in some form, by a template metaprogram. Template Metaprogramming is generally not used in practical programs, it is an interesting concept though.

❖ Stream class

➤ C++ Stream Classes Structure

In C++ there are number of stream classes for defining various streams related with files and for doing input-output operations. All these classes are defined in the file **iostream.h**. Figure given below shows the hierarchy of these classes.



1. **ios class** is topmost class in the stream classes hierarchy. It is the base class for **istream**, **ostream**, and **streambuf** class.
2. **istream** and **ostream** serves the base classes for **iostream** class. The class **istream** is used for input and **ostream** for the output.
3. Class **ios** is indirectly inherited to **iostream** class using **istream** and **ostream**. To avoid the duplicity of data and member functions of **ios** class, it is declared as virtual base class when inheriting in **istream** and **ostream** as

```
class istream: virtual public ios
{
};
class ostream: virtual public ios
{
};
```

The **_withassign** classes are provided with extra functionality for the assignment operations that's why **_withassign** classes.

Facilities provided by these stream classes.

➤ **The ios class:**

The **ios** class is responsible for providing all input and output facilities to all other stream classes.

1. **The istream class:** This class is responsible for handling input stream. It provides number of function for handling chars, strings and objects such as **get**, **getline**, **read**, **ignore**, **putback** etc..

Example:

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

int main()
{
    char x;

    // used to scan a single char
    cin.get(x);

    cout << x;
}
```

Input:

g

Output:

g

2. **The ostream class:** This class is responsible for handling output stream. It provides number of function for handling chars, strings and objects such as **write**, **put** etc..

Example:

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

int main()
{
    char x;

    // used to scan a single char
```

```

        cin.get(x);

        // used to put a single char onto the screen.
        cout.put(x);
    }

```

Input:

g

Output:

g

➤ **The iostream:**

This class is responsible for handling both input and output stream as **both istream class and ostream class** is inherited into it. It provides function of both istream class and ostream class for handling chars, strings and objects such as get, getline, read, ignore, putback, put, write etc..

Example:

```

#include <iostream>
using namespace std;

int main()
{

    // this function display
    // ncount character from array
    cout.write("geeksforgeeks", 5);
}

```

Output:

geeks

1. istream_withassign class: This class is variant of istream that allows object assignment. The predefined object cin is an object of this class and thus may be reassigned at run time to a different istream object.

Example: To show that cin is object of istream class.

```

#include <iostream>
using namespace std;

class demo {
public:
    int dx, dy;

    // operator overloading using friend function
    friend void operator>>(demo& d, istream& mycin)
    {
        // cin assigned to another object mycin
        mycin >> d.dx >> d.dy;
    }
};

int main()
{
    demo d;
    cout << "Enter two numbers dx and dy\n";
}

```

```

// calls operator >> function and
// pass d and cin as reference
d >> cin; // can also be written as operator >> (d, cin) ;

```

```

    cout << "dx = " << d.dx << "\tdy = " << d.dy;
}

```

Input:

4 5

Output:

Enter two numbers dx and dy

4 5

dx = 4 dy = 5

2. ostream_withassign class: This class is variant of ostream that allows object assignment. The predefined objects cout, cerr, clog are objects of this class and thus may be reassigned at run time to a different ostream object.

Example: To show that cout is object of ostream class.

```

#include <iostream>
using namespace std;

```

```

class demo {
public:
    int dx, dy;

```

```

    demo()
    {
        dx = 4;
        dy = 5;
    }

```

```

// operator overloading using friend function
friend void operator<<(demo& d, ostream& mycout)
{
    // cout assigned to another object mycout
    mycout << "Value of dx and dy are \n";
    mycout << d.dx << " " << d.dy;
}
};

```

```

int main()
{
    demo d; // default constructor is called

    // calls operator << function and
    // pass d and cout as reference
    d << cout; // can also be written as operator << (d, cout) ;
}

```

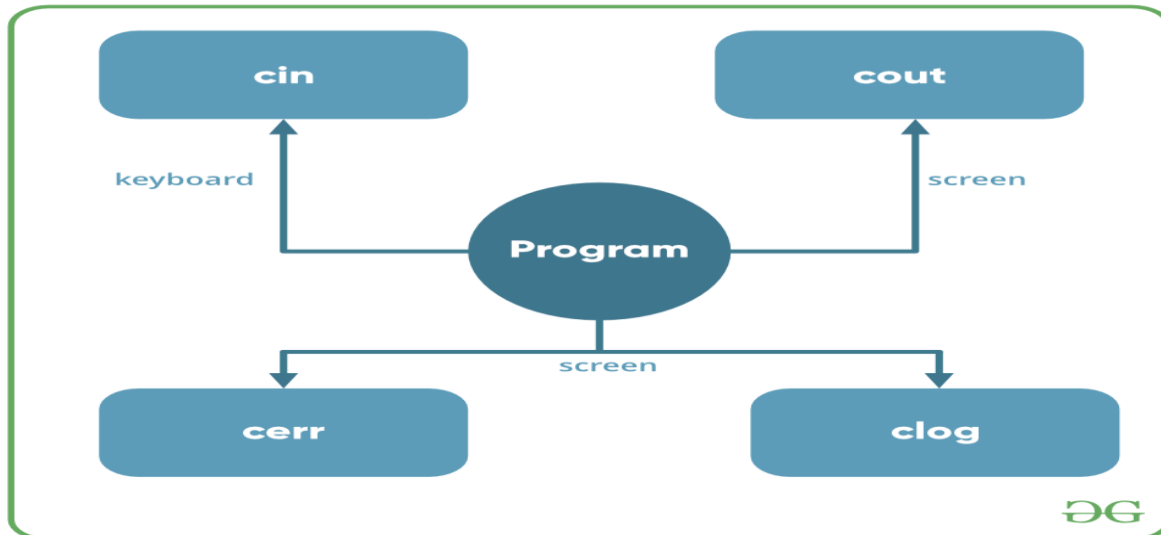
Output:

Value of dx and dy are
4 5

➤ Basic Input / Output in C++

C++ comes with libraries that provide us with many ways for performing input and output. In C++ input and output are performed in the form of a sequence of bytes or more commonly known as streams.

- **Input Stream:** If the direction of flow of bytes is from the device (for example, Keyboard) to the main memory then this process is called input.
- **Output Stream:** If the direction of flow of bytes is opposite, i.e. from main memory to device (display screen) then this process is called output.



Header files available in C++ for Input/Output operations are:

1. **iostream:** iostream stands for standard input-output stream. This header file contains definitions to objects like **cin**, **cout**, **cerr** etc.

2. **iomanip:** iomanip stands for input output manipulators. The methods declared in this file are used for manipulating streams. This file contains definitions of **setw**, **setprecision**, etc.

3. **fstream:** This header file mainly describes the file stream. This header file is used to handle the data being read from a file as input or data being written into the file as output.

The two keywords **cout in C++** and **cin in C++** are used very often for printing outputs and taking inputs respectively. These two are the most basic methods of taking input and printing output in C++. To use cin and cout in C++ one must include the header file iostream in the program.

1. Standard output stream (cout):

Usually the standard output device is the display screen. The C++ cout statement is the instance of the ostream class. It is used to produce output on the standard output device which is usually the display screen. The data needed to be displayed on the screen is inserted in the standard output stream (cout) using the insertion operator(<<).

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

```
int main()
{
    char sample[] = "GeeksforGeeks";
```



```

        cout << sample << " - A computer science portal for geeks";

    return 0;
}

```

Output:

GeeksforGeeks - A computer science portal for geeks

In the above program the insertion operator(<<) inserts the value of the string variable sample followed by the string "A computer science portal for geeks" in the standard output stream cout which is then displayed on screen.

2. standard input stream (cin):

Usually the input device in a computer is the keyboard. C++ cin statement is the instance of the class istream and is used to read input from the standard input device which is usually a keyboard.

The extraction operator(>>) is used along with the object cin for reading inputs. The extraction operator extracts the data from the object cin which is entered using the keyboard.

```

#include <iostream>
using namespace std;

int main()
{
    int age;

    cout << "Enter your age:";
    cin >> age;
    cout << "\nYour age is: " << age;

    return 0;
}
Input :

```

18

Output:

*Enter your age:
Your age is: 18*

The above program asks the user to input the age. The object cin is connected to the input device. The age entered by the user is extracted from cin using the extraction operator(>>) and the extracted data is then stored in the variable age present on the right side of the extraction operator.

3. Un-buffered standard error stream (cerr):

The C++ cerr is the standard error stream which is used to output the errors. This is also an instance of the istream class. As cerr in C++ is un-buffered so it is used when one needs to display the error message immediately. It does not have any buffer to store the error message and display later.

```

#include <iostream>
using namespace std;

int main()
{
    cerr << "An error occurred";
    return 0;
}

```

Output:

An error occurred

4. buffered standard error stream (clog):

This is also an instance of `ostream` class and used to display errors but unlike `cerr` the error is first inserted into a buffer and is stored in the buffer until it is not fully filled. The error message will be displayed on the screen too.

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

int main()
{
    clog << "An error occurred";

    return 0;
}
```

Output:

An error occurred

❖ File handling

File Handling In C++

Files are used to store data in a storage device permanently. File handling provides a mechanism to store the output of a program in a file and to perform various operations on it.

A stream is an abstraction that represents a device on which operations of input and output are performed. A stream can be represented as a source or destination of characters of indefinite length depending on its usage.

In C++ we have a set of file handling methods. These include `ifstream`, `ofstream`, and `fstream`. These classes are derived from `fstreambase` and from the corresponding `ostream` class. These classes, designed to manage the disk files, are declared in `fstream` and therefore we must include `fstream` and therefore we must include this file in any program that uses files.

ofstream: Stream class to write on files

ifstream: Stream class to read from files

fstream: Stream class to both read and write from/to files.

Now the first step to open the particular file for read or write operation. We can open file by

1. passing file name in constructor at the time of object creation
2. using the open method

For e.g.

1. Open File by using constructor

```
ifstream (const char* filename, ios_base::openmode mode = ios_base::in);
ifstream fin(filename, openmode) by default openmode = ios::in
ifstream fin("filename");
```

2. Open File by using open method

Calling of default constructor

```
ifstream fin;
fin.open(filename, openmode)
fin.open("filename");
```

Modes :

Member Constant	Stands For	Access
in *	input	File open for reading: the internal stream buffer supports input operations.
out	output	File open for writing: the internal stream buffer supports output operations.
binary	binary	Operations are performed in binary mode rather than text.
ate	at end	The output position starts at the end of the file.
app	append	All output operations happen at the end of the file, appending to its existing contents.
trunc	truncate	Any contents that existed in the file before it is open are discarded.

Default Open Modes :

ifstream **ios::in**
ofstream **ios::out**
fstream **ios::in | ios::out**

Problem Statement : To read and write a File in C++.

Examples:

```

/* File Handling with C++ using ifstream & ofstream class object*/
/* To write the Content in File*/
/* Then to read the content of file*/
#include <iostream>
/* fstream header file for ifstream, ofstream, fstream classes */
#include <fstream>
using namespace std;

// Driver Code
int main()
{
    // Creation of ofstream class object
    ofstream fout;

    string line;
    // by default ios::out mode, automatically deletes
    // the content of file. To append the content, open in ios::app
    // fout.open("sample.txt", ios::app)
    fout.open("sample.txt");

    // Execute a loop If file successfully opened
    while (fout) {
        // Read a Line from standard input
        getline(cin, line);

        // Press -1 to exit

```

```

        if (line == "-1")
            break;

        // Write line in file
        fout << line << endl;
    }

    // Close the File
    fout.close();

    // Creation of ifstream class object to read the file
    ifstream fin;

    // by default open mode = ios::in mode
    fin.open("sample.txt");

    // Execute a loop until EOF (End of File)
    while (fin) {
        // Read a Line from File
        getline(fin, line);

        // Print line in Console
        cout << line << endl;
    }

    // Close the file
    fin.close();

    return 0;
}

```

Input :

Welcome in GeeksforGeeks. Best way to learn things.
-1

Output :

Welcome in GeeksforGeeks. Best way to learn things.

Below is the implementation by using fstream class.

```

/* File Handling with C++ using fstream class object */
/* To write the Content in File */
/* Then to read the content of file*/
#include <iostream>
    /* fstream header file for ifstream, ofstream, fstream classes */
#include <fstream>
using namespace std;

    // Driver Code
int main()
{
    // Creation of fstream class object
    fstream fio;

```

```

string line;

// by default openmode = ios::in|ios::out mode
// Automatically overwrites the content of file, To append
// the content, open in ios::app
// fio.open("sample.txt", ios::in|ios::out|ios::app)
// ios::trunc mode delete all content before open
fio.open("sample.txt", ios::trunc | ios::out | ios::in);
// Execute a loop If file successfully Opened
while (fio) {
    // Read a Line from standard input
    getline(cin, line);

    // Press -1 to exit
    if (line == "-1")
        break;
    // Write line in file
    fio << line << endl;
}
// Execute a loop untill EOF (End of File)
// point read pointer at beginning of file
fio.seekg(0, ios::beg);

while (fio) {
    // Read a Line from File
    getline(fio, line);

    // Print line in Console
    cout << line << endl;
}
// Close the file
fio.close();

return 0;
}

```

Input :

Welcome in GeeksforGeeks. Best way to learn things.

-1

Output :

Welcome in GeeksforGeeks. Best way to learn things.