Object Oriented Programming

OOPs = Object oriented programming structure

OOPs is a programming model or paradigm which revolves around the concept of "OBJECTS".

An **object** refers to the instance of the class, which contains the instance of the members and behaviors defined in the class template. In the real world, an object is an actual entity to which a user interacts, whereas class is just the blueprint for that object. So the objects consume space and have some characteristic behavior.

A **class** can be understood as a template or a blueprint, which contains some values, known as member data or member, and some set of rules, known as behaviors or functions. So when an object is created, it automatically takes the data and functions that are defined in the class.

It deals with internal designing not external. So it is just to help the programmer to prevent data from mishandling.

Simula is considered the first object-oriented programming language. The programming paradigm where everything is represented as an object is known as a truly object-oriented programming language.

There are several benefits to using object-oriented programming (OOP) in software development:

Modularity: OOP allows you to break a large and complex problem down into smaller and more manageable pieces, or "objects." This makes it easier to write, test, and maintain your code.

Reusability: OOP allows you to reuse code by creating new objects that are similar to existing ones. This can save time and effort when developing software.

Extensibility: OOP allows you to easily add new features to an existing codebase by creating new objects or modifying existing ones.

Maintainability: OOP makes it easier to track and fix errors in your code, as objects are self-contained and can be isolated and tested more easily than procedural code.

Scalability: OOP allows you to build software that can grow and evolve over time, as new objects can be created or added to the existing codebase as needed.

Oops, are based on a bottom-up approach, unlike the structural programming paradigm, which uses a top-down approach.

What are some other programming paradigms other than OOPs?

Programming paradigms refers to the method of classification of programming languages based on their features. There are mainly two types of Programming Paradigms:

Imperative Programming Paradigm
Declarative Programming Paradigm

Now, these paradigms can be further classified based:

1. **Imperative Programming Paradigm:** Imperative programming focuses on HOW to execute program logic and defines control flow as statements that change a program state. This can be further classified as:

a) Procedural Programming Paradigm: Procedural programming specifies the steps a program must take to reach the desired state, usually read in order from top to bottom.

b) Object-Oriented Programming or OOP: Object-oriented programming (OOP) organizes programs as objects that contain some data and have some behavior.

c) Parallel Programming: Parallel programming paradigm breaks a task into subtasks and focuses on executing them simultaneously at the same time.

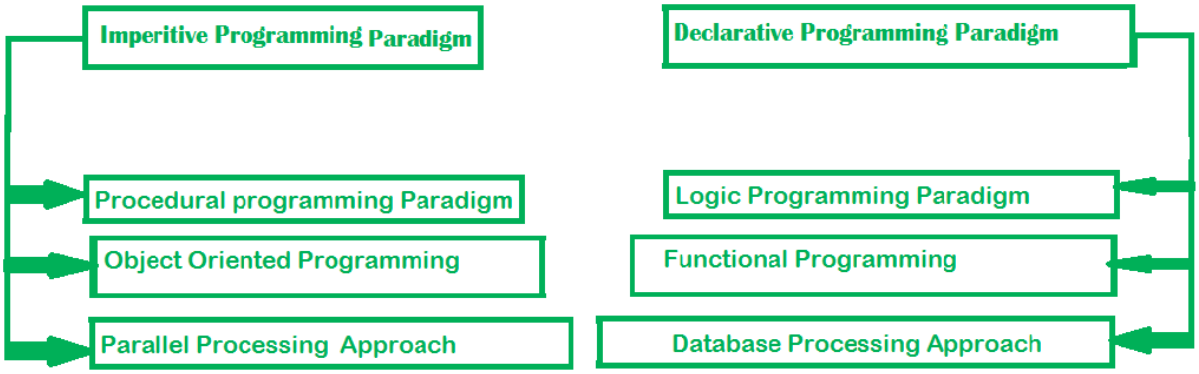
2. Declarative Programming Paradigm: Declarative programming focuses on WHAT to execute and defines program logic, but not a detailed control flow. Declarative paradigm can be further classified into:

a) Logical Programming Paradigm: Logical programming paradigm is based on formal logic, which refers to a set of sentences expressing facts and rules about how to solve a problem

b) Functional Programming Paradigm: Functional programming is a programming paradigm where programs are constructed by applying and composing functions.

c) Database Programming Paradigm: Database programming model is used to manage data and information structured as fields, records, and files. be created or added to the existing codebase as needed.

Programming Paradigms



Why do we need a Constructor in Oops?

Creating an object without a constructor is like buying a car with no color or we will color it after buying. Using Accessor and Mutators functions is philosophically wrong. Therefore the constructor must be there for initializing the object. At the time of placing the order I wanted a white color Scorpio or fortuner etc.

So the purpose of a constructor is to create a new object and set values for any existing object properties.

A constructor in C++ is a special method that is automatically called when an object of a class is created. To create a constructor, use the same name as the class, followed by parentheses ().

There should not be a return type of constructor.

```cpp
//Accessor and Mutator
#include<bits/stdc++.h>
using namespace std;
class Marker{
private:
    string color;
    string nature;
public:
    //Default Constructor or Non - parameterized Constructor
    Marker(){
        color = "Red";
        nature = "Temporary";
    }
    //Parameterized Constructor
    Marker(string color = "Red", string nature = "Temporary"){
        this->color = color;
        this->nature = nature;
    }
    //Copy Constructor
    Marker(Marker &m){
        this->color = m.color;
        this->nature = m.nature;
    }
    pair<string ,string> getter(){
        return {color,nature};
    }
};
int main(){
    Marker m1("Blue","Temporary"), m2("Black","Permanent");
    //Marker m3;
    //When we calling m3 without passing any thing then
    //Error - call of overloaded 'Marker()' is ambiguous, so either comment default constructor
    //Or Pass Atleast One Arguments
    Marker m3(m2);
    cout<<"Marker M1 - Color "<<m1.getter().first<<" Nature "<<m1.getter().second<<endl;
    cout<<"Marker M2 - Color "<<m2.getter().first<<" Nature "<<m2.getter().second<<endl;
    cout<<"Marker M3 - Color "<<m3.getter().first<<" Nature "<<m3.getter().second;
    return 0;
}
```

output.txt

```
Marker M1 - Color Blue Nature Temporary
Marker M2 - Color Black Nature Permanent
Marker M3 - Color Black Nature Permanent
```

Deep Copy Constructor :-

```cpp
//Deep Copy Constructor
#include<bits/stdc++.h>
using namespace std;
class Test{
private:
    int a;
    int *p;
public:
    Test(int x){
        a = x;
        p = new int[5];
    }
    Test(Test &t){
        this->a = t.a;
        //this->p = t.p; // This is wrong.
        this->p = new int[5];//here we called deep copy constructor.
    }
};
```

There are three types of constructors:

- Default / Non Parameterized Constructor – With no parameters. C++ says that a default constructor is automatically provided by the compiler.
- Parameterized Constructor – With Parameters, Create a new instance of a class and also pass arguments simultaneously.
- Copy Constructor – Which creates a new object as a copy of an existing object.

Type of Function inside Class :-

- Constructor
- Accessor
- Mutator
- Facilitators
- Enquiry/Inspector
- Destructor

The using Directive :-

A C++ program can be divided into different namespaces. A namespace is a part of the program in which certain names are recognized; outside of the namespace they're unknown.

The directive using namespace std;

says that all the program statements that follow are within the std namespace. Various program components such as cout are declared within this namespace.

If we didn't use the using directive, we would need to add the std name to many program elements.

For example, in the FIRST program we'd need to say

std::cout << "Every age has a language of its own.";

Scope Resolution Operators (::) - it is used for the following purposes.

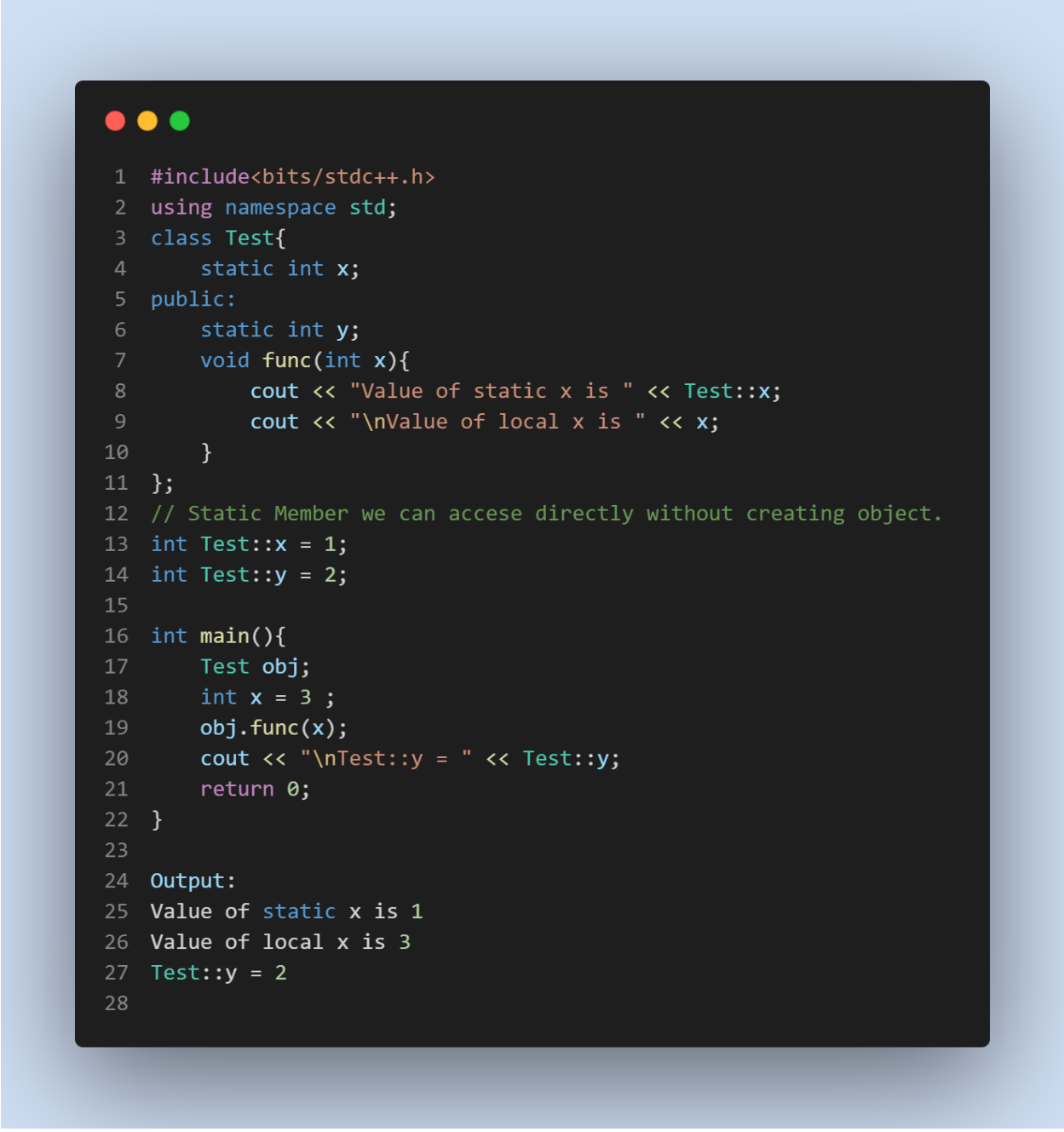
1.To access a global variable when there is a local variable with same name:

```cpp
#include <bits/stdc++.h>
using namespace std;
int x = 4;
int main(){
    int x = 10;
    cout<<"Value of global x is "<<::x;
    cout<<"\nValue of local x is "<<x;
    return 0;
}

Output :
Value of global x is 4
Value of local x is 10
```

2.To Define a Function Outside the class.

```cpp
#include <bits/stdc++.h>
using namespace std;
class A{
private:
    int a,b;
public:
    A(int x, int y){
        a=x,b=y;
    }
    int Multiplication();
};
int A::Multiplication(){
    return a*b;
}
int main(){
    A obj(5,6);
    cout<<"The Multiplication is : "<<obj.Multiplication();
    return 0;
}

Output :
The Multiplication is : 30
```

3.To access a class's static variables.



```cpp
#include<bits/stdc++.h>
using namespace std;
class Test{
    static int x;
public:
    static int y;
    void func(int x){
        cout << "Value of static x is " << Test::x;
        cout << "\nValue of local x is " << x;
    }
};
// Static Member we can accese directly without creating object.
int Test::x = 1;
int Test::y = 2;

int main(){
    Test obj;
    int x = 3 ;
    obj.func(x);
    cout << "\nTest::y = " << Test::y;
    return 0;
}

Output:
Value of static x is 1
Value of local x is 3
Test::y = 2

```

4. In Case of multiple Inheritance: if the same variable name exists in two ancestor classes, we can use scope resolution operator to distinguish.

```cpp
#include<bits/stdc++.h>
using namespace std;
class A{
protected:
    int x;
public:
    A() { x = 10; }
};
class B{
protected:
    int x;
public:
    B() { x = 20; }
};

class C: public A, public B{
public:
    void fun(){
        cout << "A's x is " << A::x;
        cout << "\nB's x is " << B::x;
    }
};
int main(){
    C c;
    c.fun();
    return 0;
}

Output:
A's x is 10
B's x is 20
```

5.Refer to a class inside another class :- refer to a nested class using the scope resolution operator.

```cpp
#include<bits/stdc++.h>
using namespace std;
class outside{
public:
    int x;
    class inside {
    public:
        int x;
        static int y;
        int foo();
    };
};
int outside::inside::y = 5;
int main(){
    outside A;
    outside::inside B;
}

```

Inline Function :-

When a function is declared inline, the compiler places a copy of that specific function's code at each point where the function is called at compile time.

When the function is declared outside the class with inline references then this can become overhead if the execution time of function is less than the switching time from the caller function to called function (callee).

Note :- **Inlining is only a request to the compiler, not a command. Compiler can ignore the request for inlining.**

Compiler may not perform inlining in such circumstances:

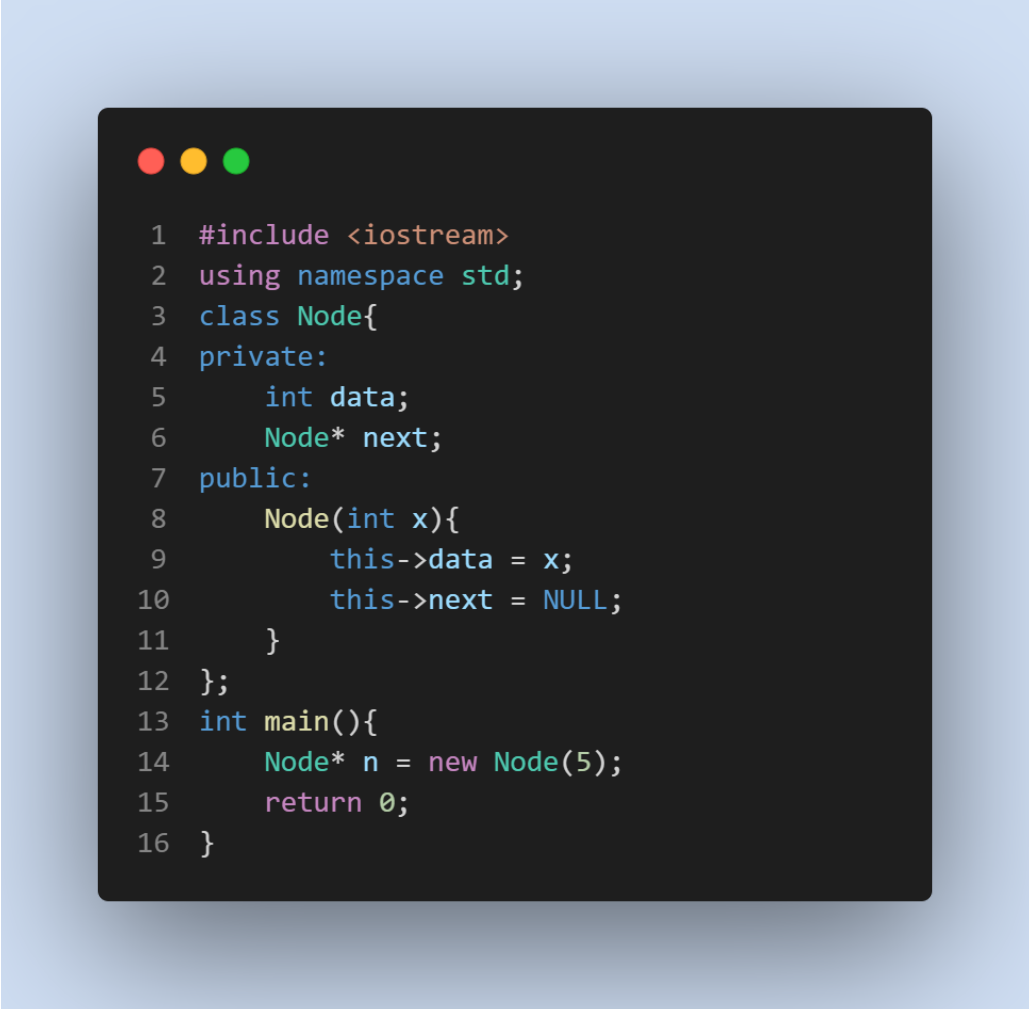
- If the function contains a loop.
- If a function contains static Variables
- If a function is recursive in nature.
- If a function return type is other than void
- If a function contains switch or goto statements.

```cpp
#include <iostream>
using namespace std;
class operation
{
    int a,b,add,sub,mul;
    float div;
public:
    void get();
    void sum();
    void difference();
};
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";
}
int main(){
    cout << "Program using inline function\n";
    operation s;
    s.get();
    s.sum();
    s.difference();
    return 0;
}

```

This Pointer :- **this is a keyword that refers to the current instance of the class.**

There can be 3 main uses of "this" keyword :

- It can be used to declare indexers.
- It can be used to refer to the current class instance variable.
- It can be used to pass the current object as a parameter to another method.



```cpp
#include <iostream>
using namespace std;
class Node{
private:
    int data;
    Node* next;
public:
    Node(int x){
        this->data = x;
        this->next = NULL;
    }
};
int main(){
    Node* n = new Node(5);
    return 0;
}
```

Structure	Class
A Structure is defined with the struct keyword	A class is defined with the class keyword
All the member of a structure are public by default	All the members of a class are private by default
Structure can't be inherit	Class can be inherit
A structure contains only data member	A class contain both data member and member function
There is no data hiding features	Classes having data hiding features by using access specifiers(public, private, protected)

Operator Overloading :- we can make operators work for user-defined classes. This means C++ has the ability to provide the operators with a special meaning for a data type, this ability is known as operator overloading.

For example, we can overload an operator '+' in a class like String so that we can concatenate two strings by just using +. Other example classes where arithmetic operators may be overloaded are Complex Numbers, Fractional Numbers, Big Integer, etc.

```cpp
#include <bits/stdc++.h>
using namespace std;
class Complex{
private:
    int real;
    int img;
public:
    Complex(int r = 0, int i = 0){
        real = r;
        img = i;
    }
    void display(){
        cout << real <<" + i" << img << endl;
    }
    Complex add(Complex c){
        Complex temp;
        temp.real = real + c.real;
        temp.img = img + c.img;
        return temp;
    }
};
int main(){
    Complex c1(5, 3), c2(10, 5), c3;
    c3 = c1.add(c2);
    c3.display();
}

//Output
15 + i8
```

```cpp
#include <bits/stdc++.h>
using namespace std;
class Complex{
private:
    int real;
    int img;
public:
    Complex(int r = 0, int i = 0){
        real = r;
        img = i;
    }
    void display(){
        cout << real <<" + i" << img << endl;
    }
    Complex operator +(Complex c){
        Complex temp;
        temp.real = real + c.real;
        temp.img = img + c.img;
        return temp;
    }
    Complex operator -(Complex c){
        Complex temp;
        temp.real = real - c.real;
        temp.img = img - c.img;
        return temp;
    }
};
int main(){
    Complex c1(5, 3), c2(10, 5), c3, c4;
    c3 = c2 + c1;
    c4 = c2 - c1;
    c4.display();
    c3.display();
}

//Output
5 + i2
15 + i8
```

Operator Overloading using friend function :-

```cpp
// Operator Overloading using Friend Functions
#include <iostream>
using namespace std;
class Complex{
private:
    int real;
    int img;
public:
    Complex(int r = 0, int i = 0){
        real = r;
        img = i;
    }
    void display(){
        cout << real <<" + i"<< img << endl;
    }
    friend Complex operator+(Complex c1, Complex c2);
};
Complex operator+(Complex c1, Complex c2){
    Complex temp;
    temp.real = c1.real + c2.real;
    temp.img = c1.img + c2.img;
    return temp;
}
int main(){
    Complex c1(5, 3), c2(10, 6), c3;
    c3 = c1 + c2;
    // also written like this
    // c3 = operator+(c1,c2);
    c3.display();
}

// Output
15 + i9;
```

Ostream Operators Overloading :-

```cpp
#include <bits/stdc++.h>
using namespace std;
class Complex{
private:
    int real;
    int img;
public:
    Complex(int r, int i){
        real = r;
        img = i;
    }
    friend ostream & operator<<(ostream &o, Complex &c);
};
ostream & operator<<(ostream &out, Complex &c){
    out<<c.real<<" + i"<<c.img<<endl;
    return out;
}
int main(){
    Complex c1(4,6);
    cout<<c1;
    operator<<(cout,c1);
    return 0;
}

// Output
4 + i6
4 + i6
```

Eg - Addition of Rational Numbers

```cpp
// Program with a Class Rational Number
#include <iostream>
using namespace std;
class Rational{
private:
    int p;
    int q;
public:
    Rational(){
        p = 1;
        q = 1;
    }
    Rational(int p, int q){
        this->p = p;
        this->q = q;
    }
    Rational(Rational &r){
        this->p = r.p;
        this->q = r.q;
    }
    int getP() { return p; }
    int getQ() { return q; }
    void setP(int p){
        this->p = p;
    }
    void setQ(int q){
        this->q = q;
    }
    Rational operator+(Rational r){
        Rational t;
        t.p = this->p * r.q + this->q * r.p;
        t.q = this->q * r.q;
        return t;
    }
    friend ostream &operator<<(ostream &os, Rational &r);
};
ostream &operator<<(ostream &os, Rational &r){
    os << r.p << "/" << r.q;
    return os;
}
int main(){
    Rational r1(3, 4), r2(2, 5), r3;
    r3 = r1 + r2;
    cout << "Sum of " << r1 << " and " << r2 << " is " << r3 << endl;
}

// Output
Sum of 3/4 and 2/5 is 23/20
```

Inheritance

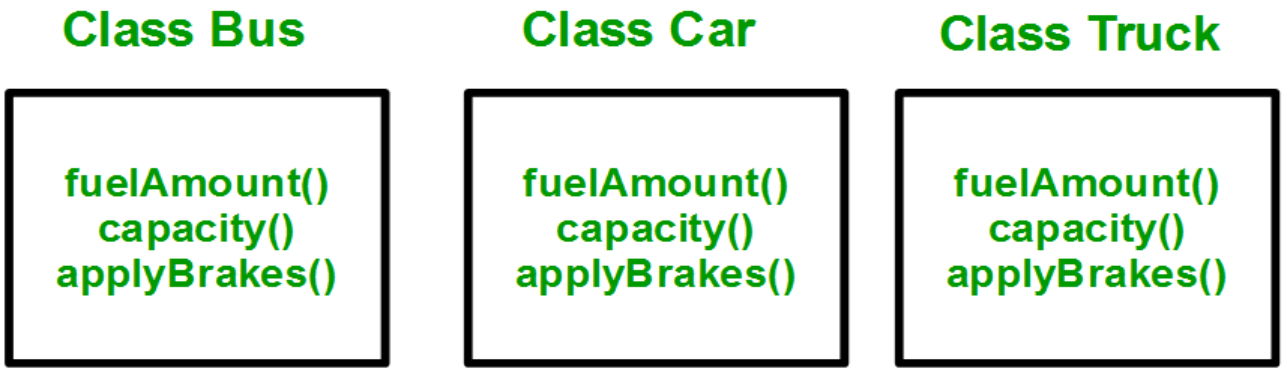
Inheritance :- **It is one of the most important features of Object-Oriented Programming. The Capability of a class to derive properties and characteristics from another class is called inheritance.**

Inheritance is a feature or a process in which, new classes are created from the existing classes. The new class created is called "Derived Class" or "Child Class" or "Sub Class" and the existing class is known as the "Base Class" or "Parent Class" or "Super Class".

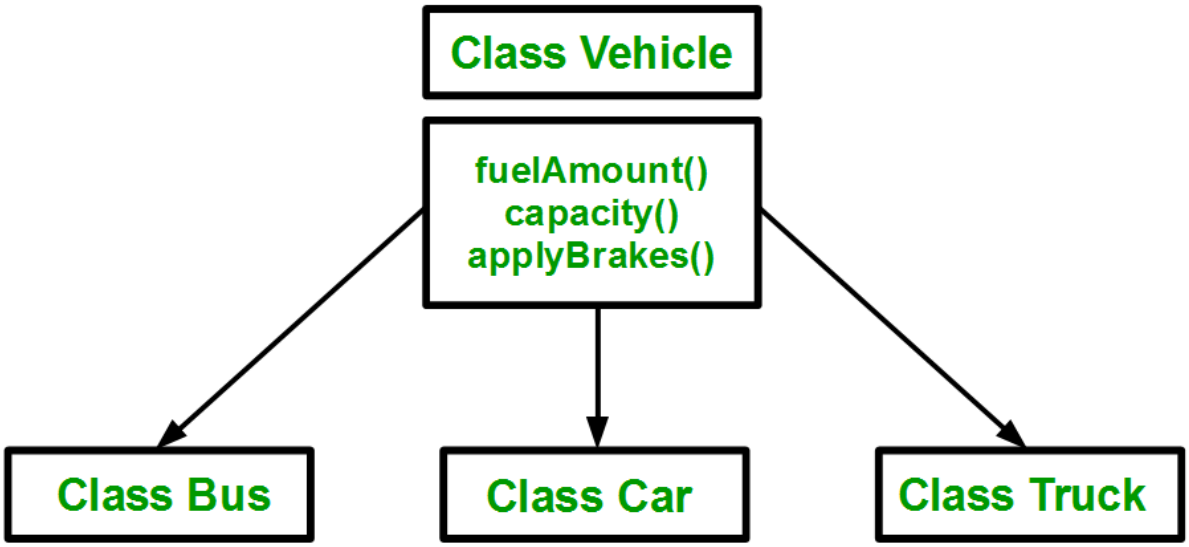
When we say derived class inherits the base class, it means the derived class inherits all the properties of base class without changing the properties of base class and may add new features to its own derived class.

Why and when to use Inheritance.

- Inheritance is a term for reusing code by a mechanism of passing down information and behavior from a parent class to a child or subclass.
- Consider a group of vehicles. You need to create classes for Bus, Car, and Truck. The methods fuelAmount(), capacity(), applyBrakes(), will be the same for all 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 below figure:-



- We can clearly see that the above process results in duplication of the same code 3 times. This increases the chances of error and data redundancy. To avoid this type of situation, inheritance is used.



- If you want the object to use all the behavior of the base class unless explicitly overridden, then inheritance is the simplest, least verbose, most straightforward way to express it.

- The main reason for using inheritance is not as a form of composition - it is so you can get polymorphic behavior. If you don't need polymorphism, you probably should not be using inheritance, at least in C++.

Available vs Accessible

Available : If a class Derived is inheriting from Base class then everything from Base class will be available/present in Derived class.
Accessible: everything is available in Derived class but private members are not accessible.
Example: bank balance of Father is available for Son by not accessible. Son has to ask Father for money. Only Father can access his account.

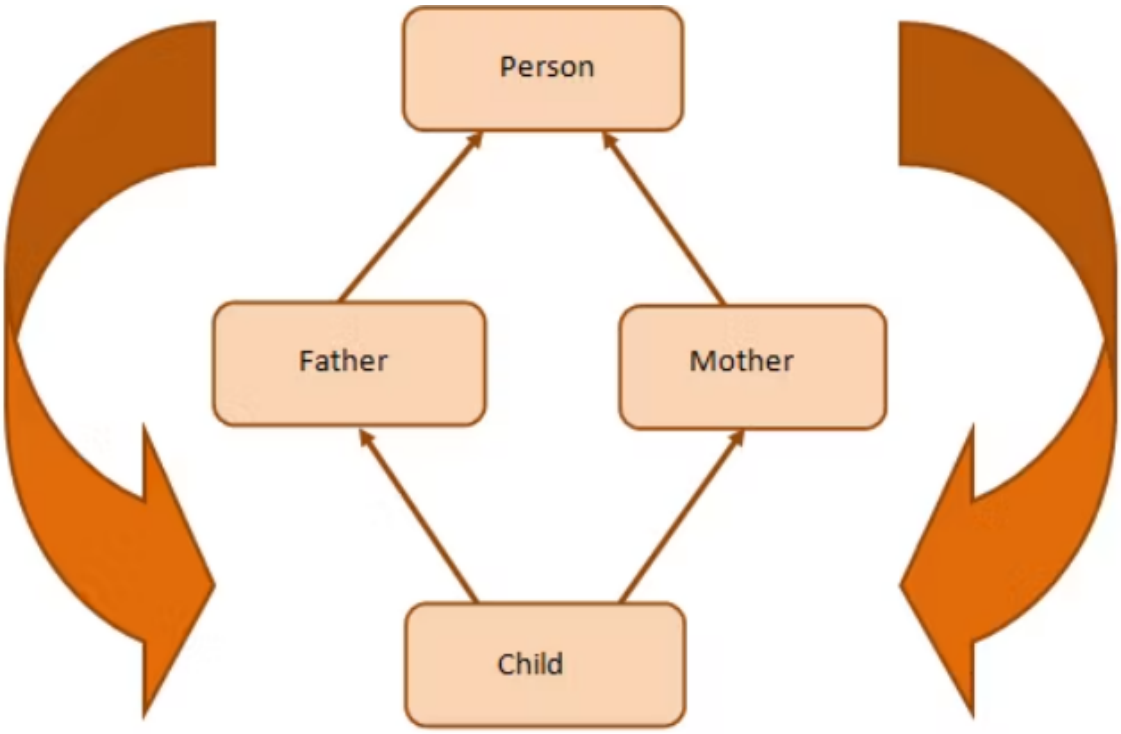
```cpp
#include<bits/stdc++.h>
using namespace std;
// Example of inheritance
class Rectangle{
private:
    int length;
    int breadth;
public:
    Rectangle(){length = 1,breadth=1;};
    Rectangle(int l, int b){length = l,breadth=b;}
    int getLength(){return length;}
    int getBreadth(){return breadth;}
    void setLength(int l){length = l;}
    void setBreadth(int b){breadth = b;}
    int area(){return length * breadth;};
    int perimeter(){return 2 * (length + breadth);};
    bool isSquare(){return length == breadth;};
    ~Rectangle(){cout<<"Rectangle Destroyed";};
};
class Cuboid : public Rectangle{
private:
    int height;
public:
    Cuboid(int h){
        height = h;
    }
    int getHeight() { return height; }
    void setHeight(int h) { height = h; }
    int volume() { return getLength() * getBreadth() * height; }
};
int main(){
    Cuboid c(5);
    c.setLength(10);
    c.setBreadth(7);
    cout << "Volume is " << c.volume() << endl;
}
//Output
Volume is 350
Rectangle Destroyed

```

Constructors in inheritance :-

- Constructors of base class are executed first then the constructor of derived class is executed.
- By default, a non-parameterised constructor of base class is executed.
- A parameterised constructor of base class must be called from derived class constructor.

```cpp
// Example of calling parameterised constructor
#include <iostream>
using namespace std;
class Base{
public:
    Base(){cout<<"Non-param Base"<<endl;}
    Base(int x){cout<<"Param of Base "<<x<<endl;}
};
class Derived:public Base{
public:
    Derived(){cout<<"Non-Param Derived"<<endl;}
    Derived(int y){cout<<"Param of Derived "<<y<<endl;}
    Derived(int x,int y):Base(x){
        cout<<"Param of Derived "<<y<<endl;
    }
};
int main(){
    Derived d(5,10);
    return 0;
}
//Output
Param of Base 5
Param of Derived 10
```

The Diamond Problem (Multiple-path inheritance) :- The Diamond problem occurs when a child class inherits from two parent classes who both share a common grandparent class.



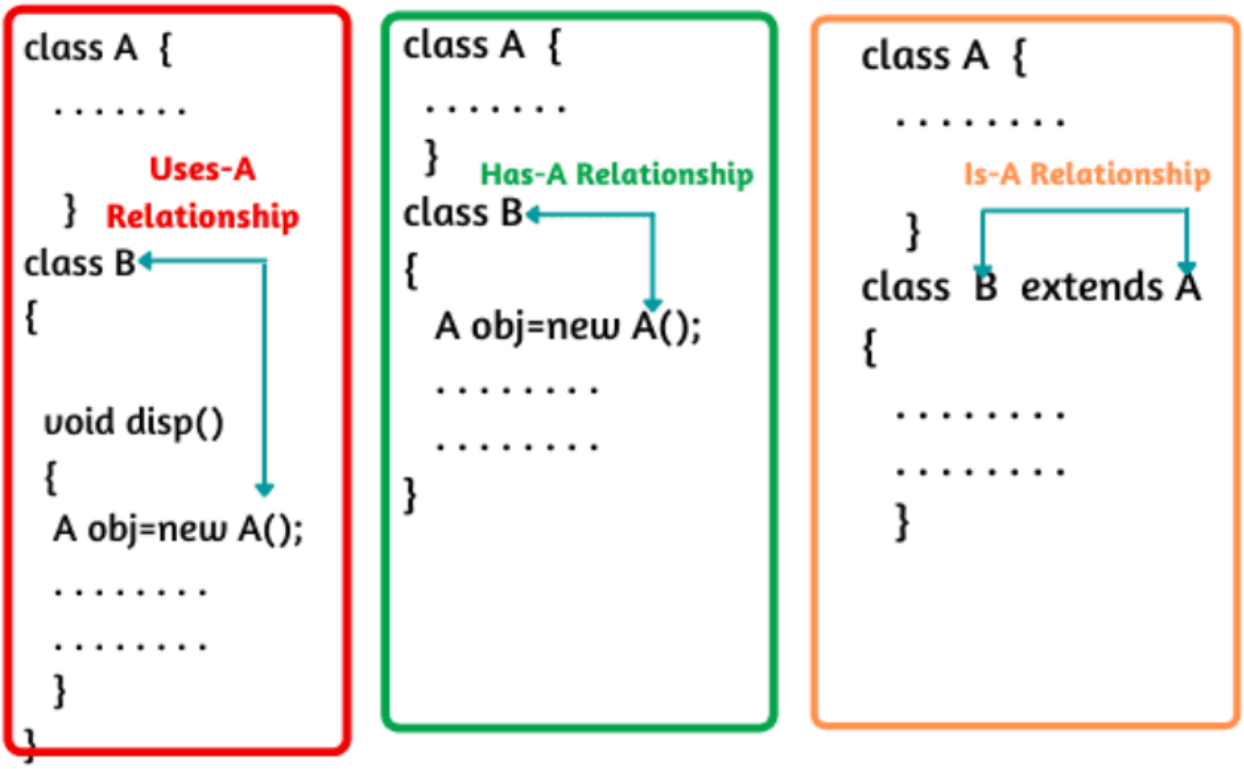
As shown in the figure, class Child inherits the traits of class Person twice—once from Father and again from Mother. This gives rise to ambiguity since the compiler fails to understand which way to go.

```cpp
// Example of diamond problem
#include <iostream>
using namespace std;
class Person { //class Person
public:
    Person(int x){cout << "Person::Person(int) called" << endl;}
};
class Father : public Person { //class Father inherits Person
public:
    Father(int x):Person(x){
        cout << "Father::Father(int) called" << endl;
    }
};
class Mother : public Person{ //class Mother inherits Person
public:
    Mother(int x):Person(x){
        cout << "Mother::Mother(int) called" << endl;
    }
};
class Child : public Father, public Mother{ //Child inherits Father and Mother
public:
    Child(int x):Mother(x), Father(x) {
        cout << "Child::Child(int) called" << endl;
    }
};
int main(){
    Child child(30);
    return 0;
}

//Output
Person::Person(int) called
Father::Father(int) called
Person::Person(int) called
Mother::Mother(int) called
Child::Child(int) called
```

The solution to the diamond problem is to use the virtual keyword. We make the two parent classes (who inherit from the same grandparent class) into virtual classes in order to avoid two copies of the grandparent class in the child class

```cpp
// Example of diamond problem
#include <iostream>
using namespace std;
class Person { //class Person
public:
    Person(){cout << "Person::Person() called" << endl;}
    Person(int x){cout << "Person::Person(int) called" << endl;}
};
class Father : virtual public Person { //class Father inherits Person
public:
    Father(int x):Person(x){
        cout << "Father::Father(int) called" << endl;
    }
};
class Mother : virtual public Person{ //class Mother inherits Person
public:
    Mother(int x):Person(x){
        cout << "Mother::Mother(int) called" << endl;
    }
};
class Child : public Father, public Mother{ //Child inherits Father and Mother
public:
    Child(int x):Mother(x), Father(x){
        cout << "Child::Child(int) called" << endl;
    }
};
int main(){
    Child child(30);
    return 0;
}

//Output
Person::Person() called
Father::Father(int) called
Mother::Mother(int) called
Child::Child(int) called
```

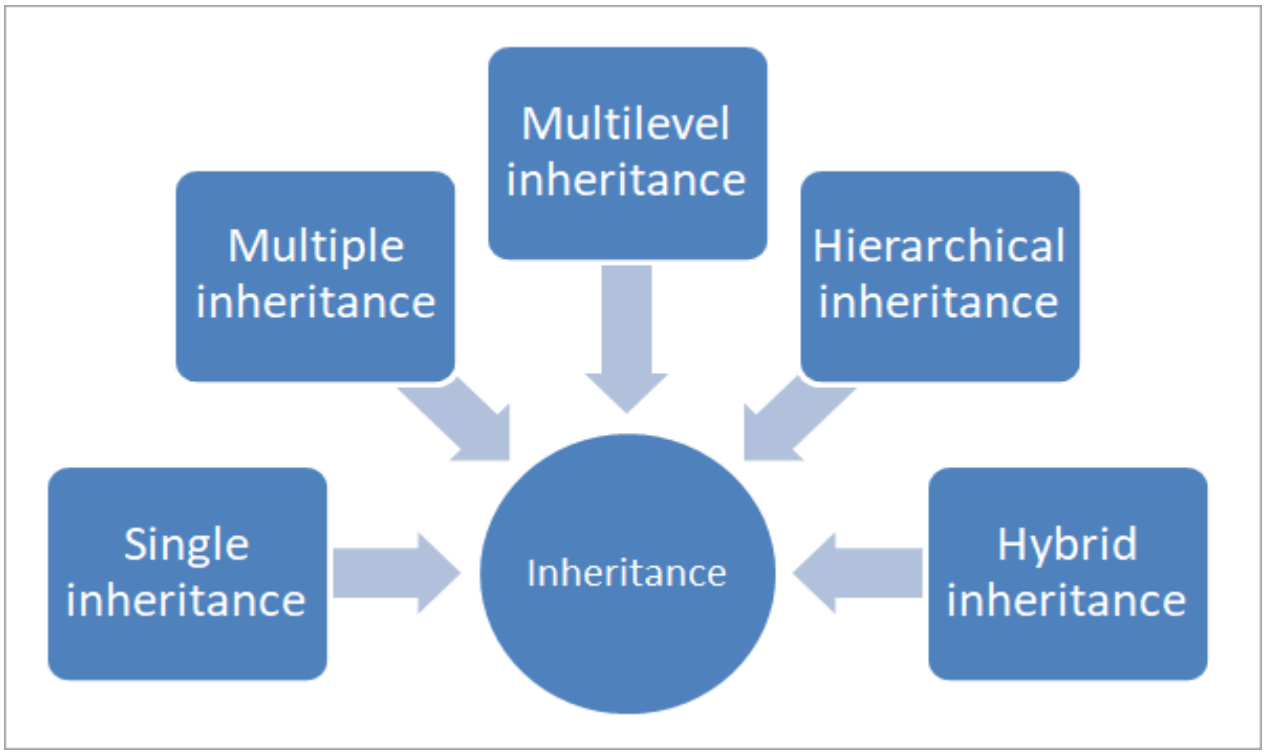
Uses-A, IsA and HasA Relationship :-



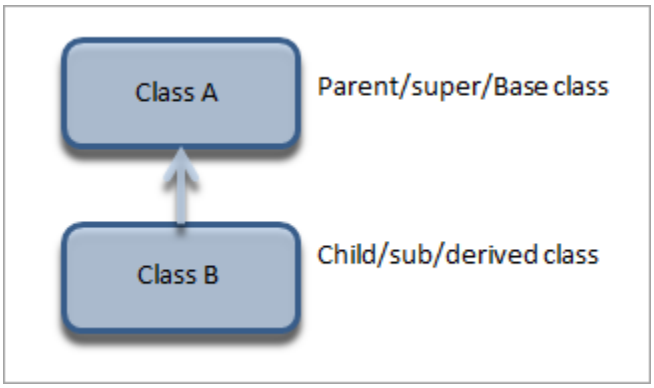
Access Specifiers in C++

Specifier	Within Same Class	In Derived Class	Outside the Class
Private	Yes	No	No
Protected	Yes	Yes	No
Public	Yes	Yes	Yes

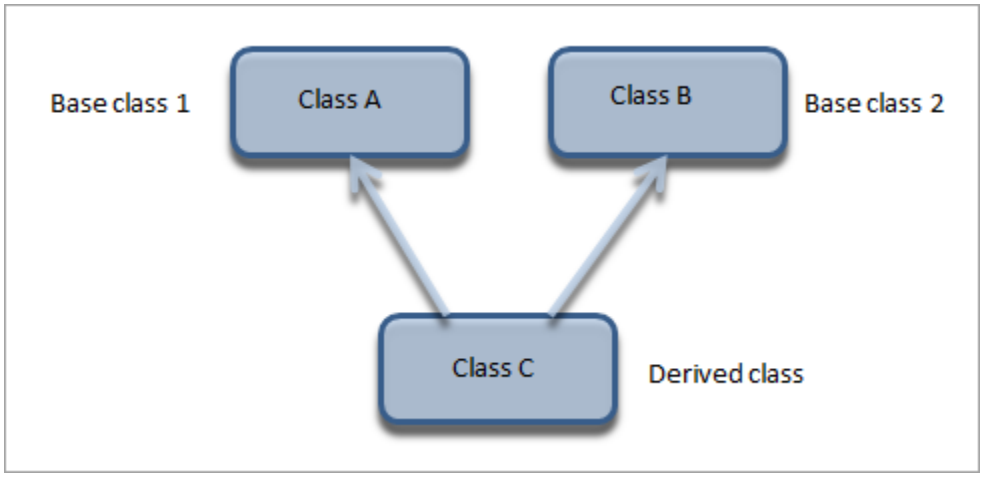
Types Of Inheritance :-



1) Single Inheritance :- In single inheritance, a class derives from one base class only. This means that there is only one subclass that is derived from one superclass.

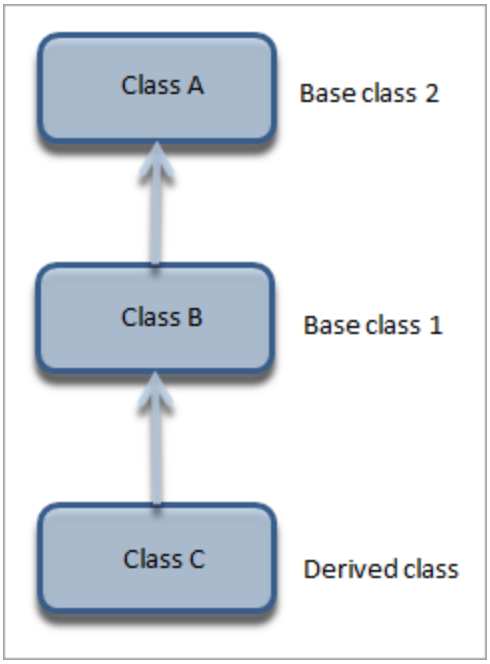


2) Multiple Inheritance :- Multiple inheritance is a type of inheritance in which a class derives from more than one class. As shown in the above diagram, class C is a subclass that has class A and class B as its parent.

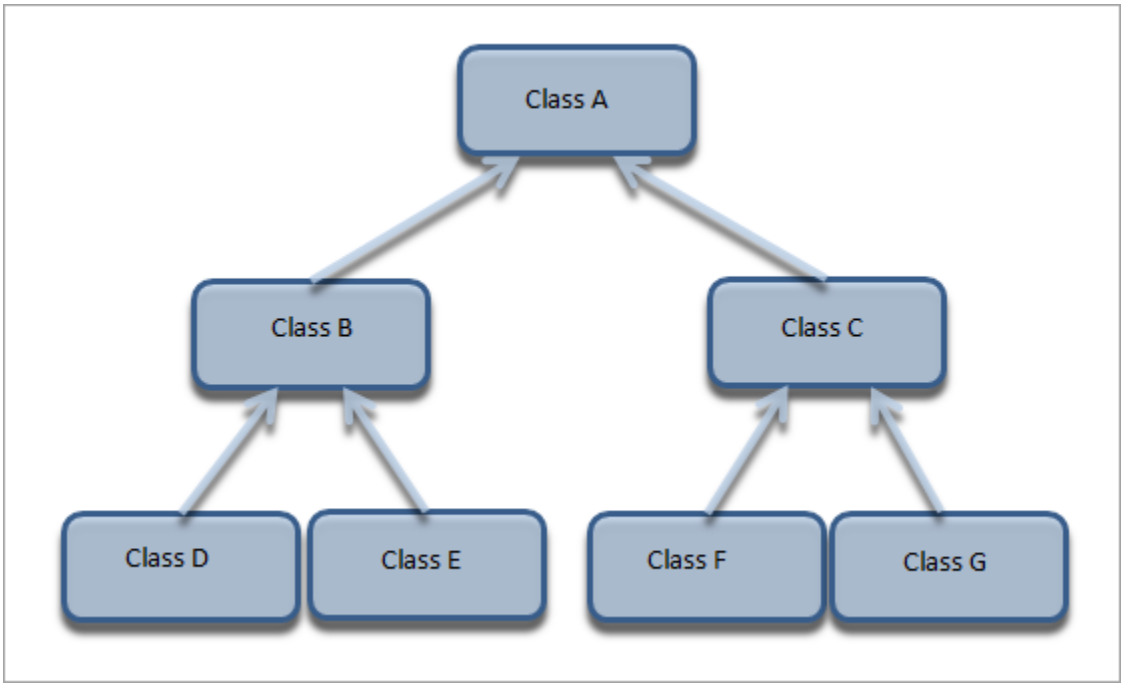


Example :- In a real-life scenario, a child inherits from their father and mother. This can be considered an example of multiple inheritance.

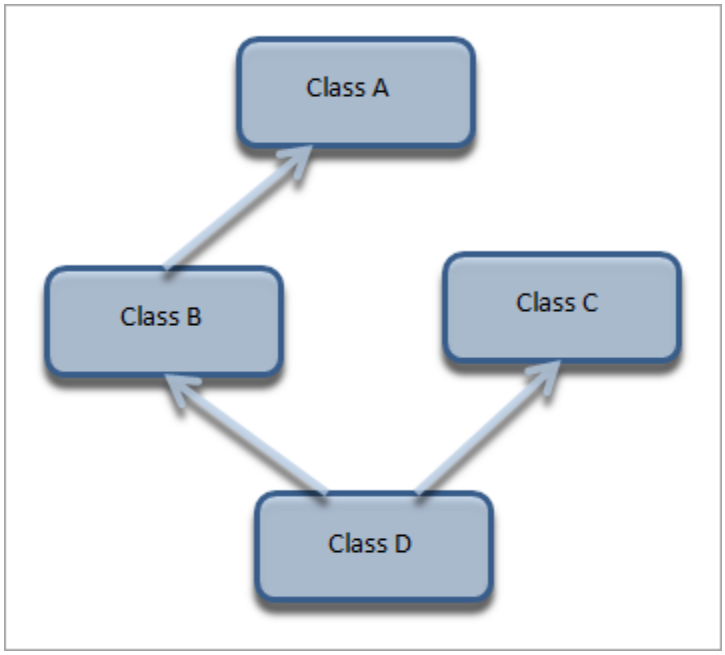
3) Multilevel Inheritance :- In multilevel inheritance, a class is derived from another derived class. This inheritance can have as many levels as long as our implementation doesn't go wayward. In the above diagram, class C is derived from Class B. Class B is in turn derived from class A.



4) Hierarchical Inheritance :- In hierarchical inheritance, more than one class inherits from a single base class as shown in the representation above. This gives it a structure of a hierarchy.

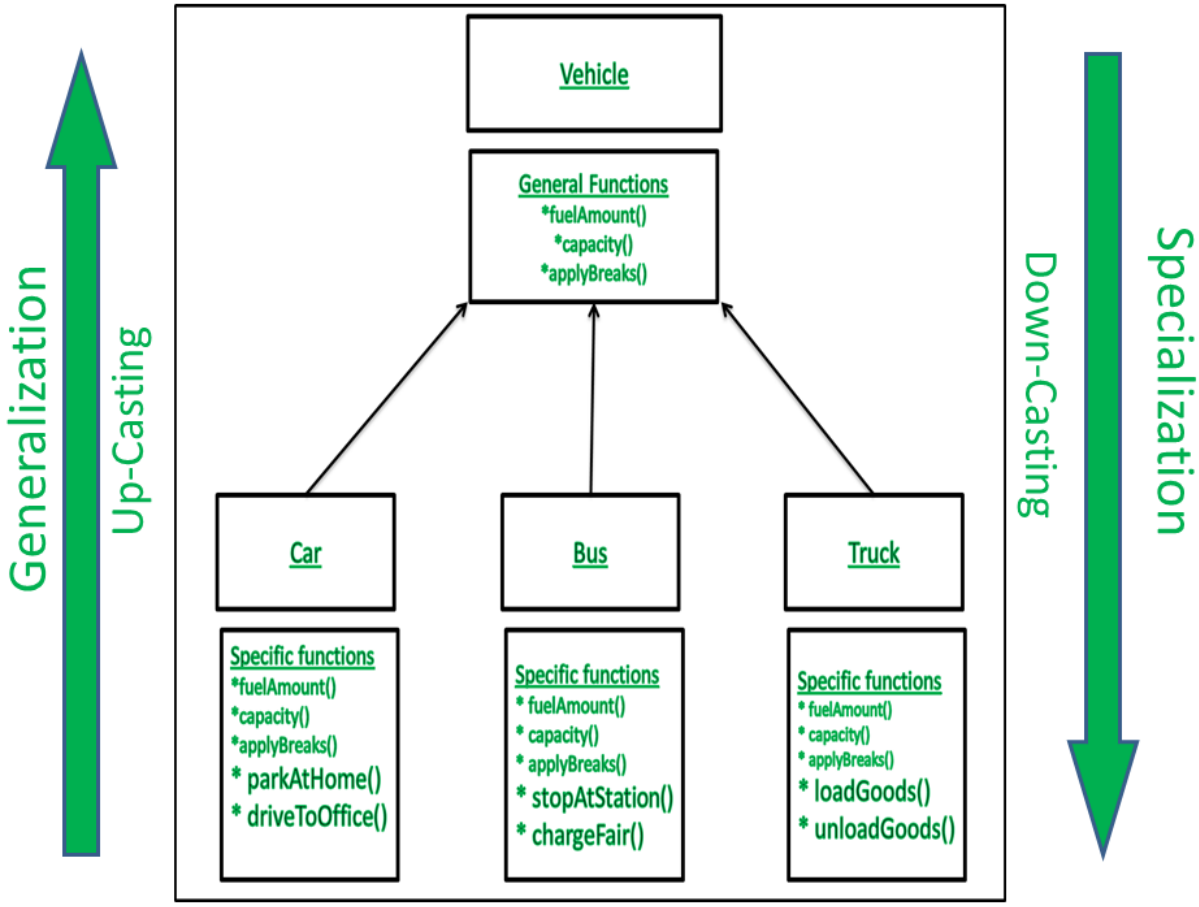


5) Hybrid Inheritance :- Hybrid inheritance is usually a combination of more than one type of inheritance. In the above representation, we have multiple inheritance (B, C, and D) and multilevel inheritance (A, B, and D) to get a hybrid inheritance.



Access Specifier in Inheritance

```cpp
#include<iostream>
using namespace std;
class Base {
public:
    int x;
protected:
    int y;
private:
    int z;
};
class PublicDerived: public Base {
  // x is public
  // y is protected
  // z is not accessible from PublicDerived
};
class ProtectedDerived: protected Base {
  // x is protected
  // y is protected
  // z is not accessible from ProtectedDerived
};
class PrivateDerived: private Base {
  // x is private
  // y is private
  // z is not accessible from PrivateDerived
};
```


Generalization
Up-Casting
Vehicle
General Functions
*fuelAmount()
*capacity()
*applyBreaks()
Car
Bus
Truck
Specific functions
*fuelAmount()
*capacity()
*applyBreaks()
* parkAtHome()
* driveToOffice()
Specific functions
* fuelAmount()
* capacity()
* applyBreaks()
* stopAtStation()
* chargeFair()
Specific functions
* fuelAmount()
* capacity()
* applyBreaks()
* loadGoods()
* unloadGoods()
Down-Casting
Specialization

Program to Demonstrate inheritance

```cpp
// Program to Demonstrate Inheritance
#include <iostream>
using namespace std;
class Employee{
private:
    int eid;
    string name;
public:
    Employee(int e, string n){
        eid = e;
        name = n;
    }
    int getEmployeeID(){return eid;}
    string getName(){return name;}
};
class FulltimeEmployee : public Employee{
private:
    int salary;
public:
    FulltimeEmployee(int e, string n, int sal):Employee(e, n){
        salary = sal;
    }
    int getSalary() { return salary; }
};
class ParttimeEmployee : public Employee{
private:
    int wage;
public:
    ParttimeEmployee(int e, string n, int w) : Employee(e, n){
        wage = w;
    }
    int getWage() { return wage; }
};
int main(){
    ParttimeEmployee p1(1, "John", 300);
    FulltimeEmployee p2(2, "Raj", 5000);
    cout<<"Salary of "<<p2.getName()<<" is "<<p2.getSalary()<<endl;
    cout<<"Daily wage of "<<p1.getName()<<" is "<<p1.getWage()<<endl;
}

//Output
Salary of Raj is 5000
Daily wage of John is 300
```

Base Class Pointer and Derived Class Object in C++:-

Base * ptr;
Ptr = new Derived();

```cpp
//Base Class Pointer and Derived Class Object.
#include <iostream>
using namespace std;
class Base{
public:
    void fun1(){cout<<"fun1 of Base Class" << endl;}
};
class Derived:public Base{
public:
    void fun2(){cout<<"fun2 of Derived Class" << endl;}
};
int main(){
    Base *p;
    p = new Derived();
    p->fun1();
    //The following statement will give compilation error
    p->fun2();
    return 0;
}
// Output
error: ‘class Base’ has no member named ‘fun2’; did you
 mean ‘fun1’?
```

Eg. Suppose there is a basic Car. A basic car means having the basic features of a car. Nothing is automated, no extra features, except for driving a car there are no extra things like there is no air conditioner, no media player, and no keyless entry. Nothing is there.

Then you have an advanced car. And an advanced car is inheriting from a basic car. So, can we have a pointer of a basic car, and to that pointer, can we assign an object of an advanced car? Yes.

Then using the basic class pointer which functions you can call? You can call only the functions which are present in the basic car. It is just like a basic car but there is an advanced car here with all the extra features in the car which you have no idea about.

```cpp
//Base Class Pointer and Derived Class Object.
#include<iostream>
using namespace std;
class BasicCar{
public:
    void Start(){cout<<"Car Started"<<endl;}
};
class AdvanceCar:public BasicCar{
public:
    void PlayMusic(){cout<<"Playing Music"<<endl;}
};

int main()
{
    AdvanceCar a;
    BasicCar *ptr = &a;
    ptr->Start();
    //The following statement will throw compilation error
    ptr->PlayMusic();
    return 0;
}

//Output
error: ‘class BasicCar’ has no member named ‘PlayMusic’
```

Can we create a Derived pointer assigned to the Base class object in C++?

```
Derived *p;
p = new Base();
```

Is it possible that we have a derived class pointer and I have assigned the object of the base class? **No, not possible**. Why? Let us see the reason.

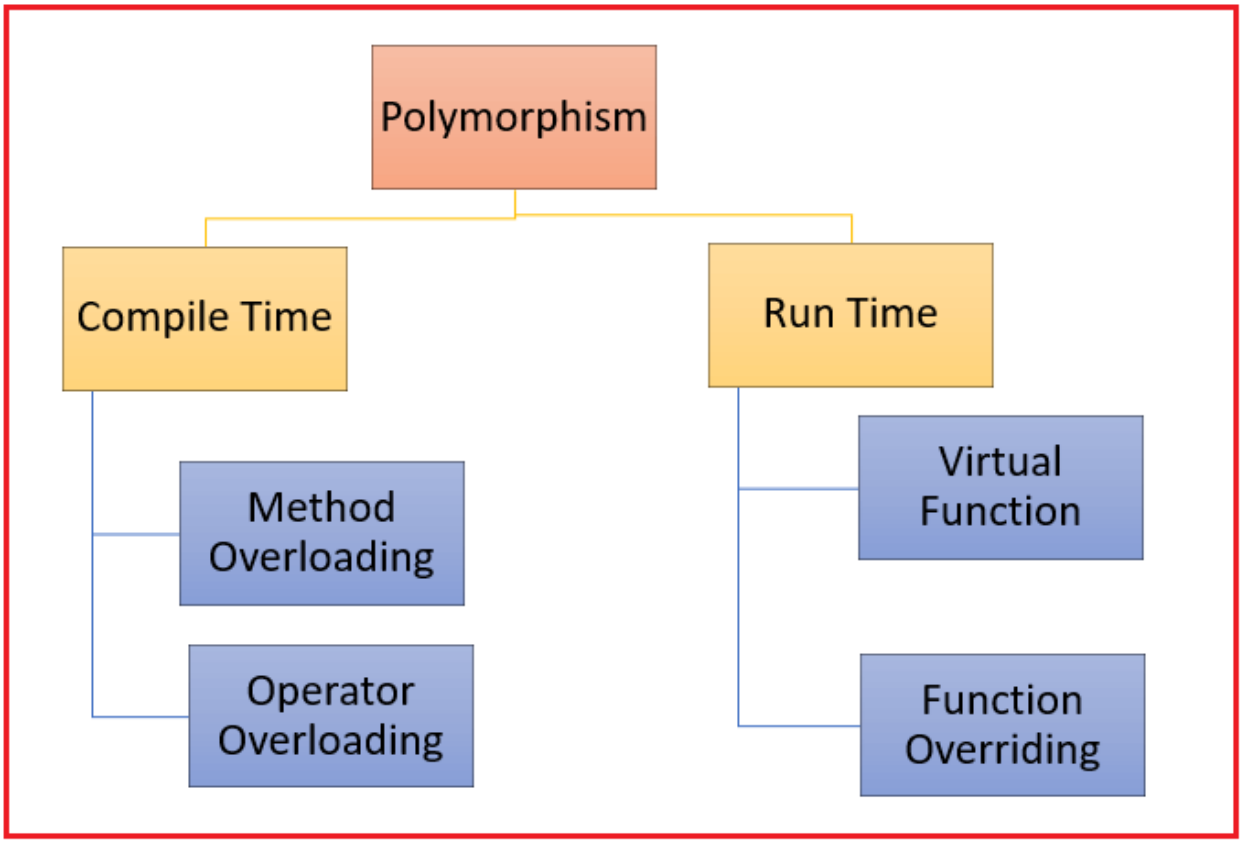
See we have a basic car. **Now if we are calling the basic car as an advanced car.** Then can we get the advanced car features in the basic car? The answer is No.

So, many of the features of our advanced car are not available in the basic car. If I think that it is an advanced car, I cannot use the features. Suppose if I tried keyless entry then it is not there, if I try to open moon roof then it's not there, if I try to start AC then it's not there.

Polymorphism

Polymorphism is considered one of the important features of Object-Oriented Programming. 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.

One real-life example of polymorphism is, that a person at the same time can have different characteristics. A man at the same time is a father, a husband, and an employee. So, the same person possesses different behavior in different situations. This is called polymorphism.

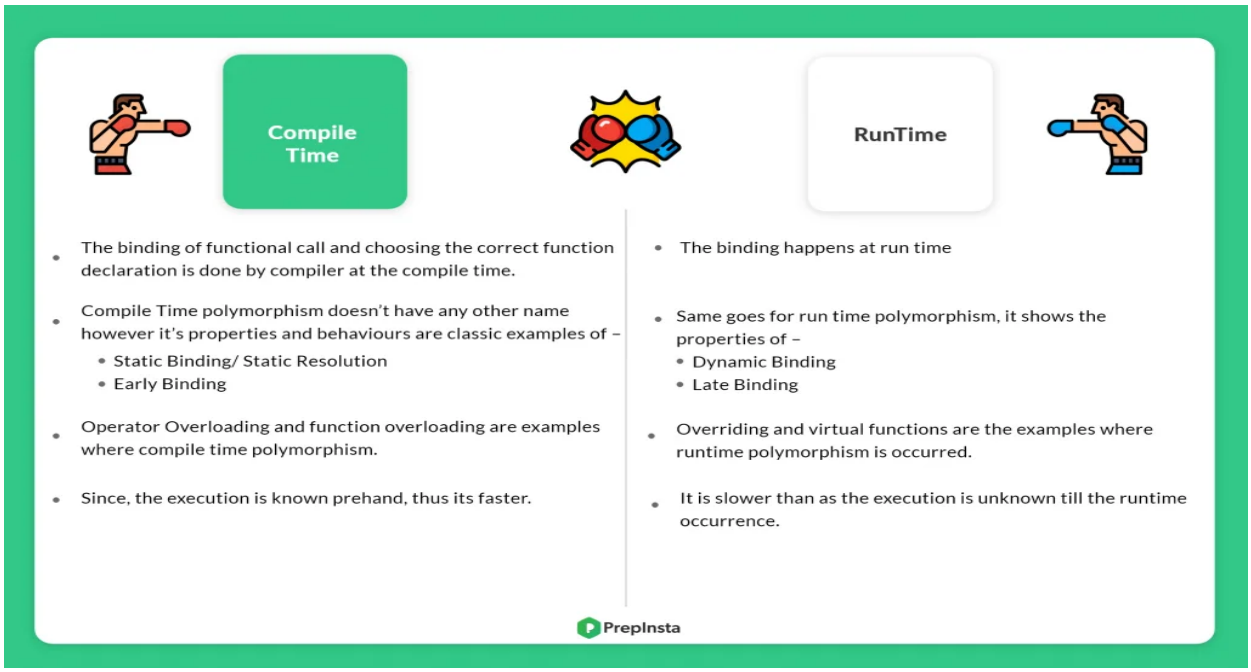


Compile Time Polymorphism in C++ :- This type of polymorphism is achieved by **function overloading** or **operator overloading**. The overloaded functions are invoked by matching the type and number of arguments.

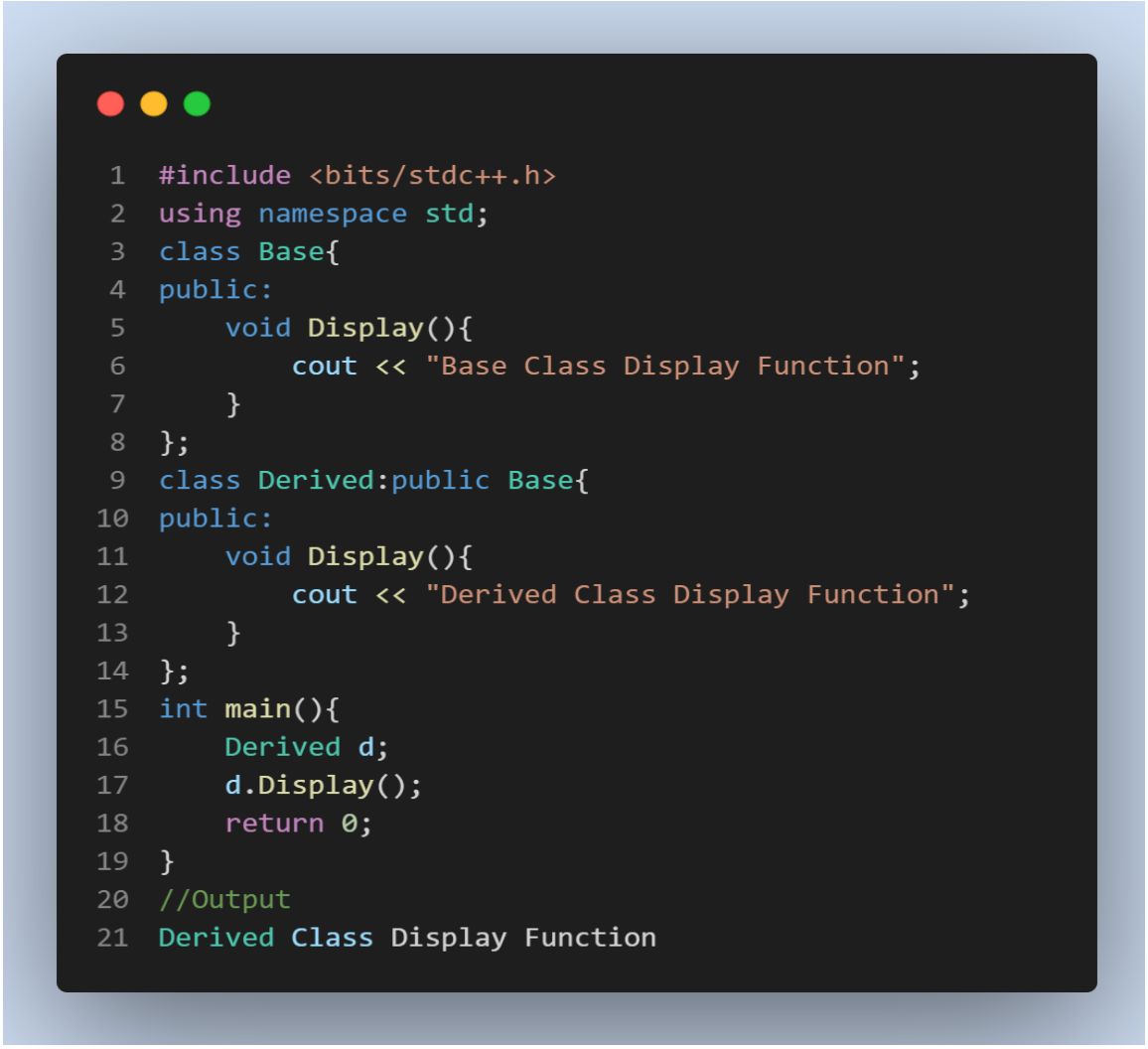
The information is present during compile-time. This means the C++ compiler will select the right function at compile time. Which is also known as static binding or early binding.

Runtime Polymorphism in C++ :- Run time polymorphism is achieved when the object's method is invoked at the run time instead of compile time. It is achieved by method overriding which is also known as dynamic binding or late binding.

This type of polymorphism is achieved by Function Overriding. This happens when an object's method is invoked/called during runtime rather than during compile time.



Function overriding -



```cpp
#include <bits/stdc++.h>
using namespace std;
class Base{
public:
    void Display(){
        cout << "Base Class Display Function";
    }
};
class Derived:public Base{
public:
    void Display(){
        cout << "Derived Class Display Function";
    }
};
int main(){
    Derived d;
    d.Display();
    return 0;
}
//Output
Derived Class Display Function
```

Key Points about Function overriding :-

- Redefining a function of a base class in the derived class called function overriding in c++.
- Function overriding is used for achieving runtime polymorphism.
- The Signature and Prototype of an overrides function must be exactly the same as the base function.

When do we need to override a function in C++?

If the superclass function logic is not fulfilling the sub-class business requirements, then the subclass needs to override that function with the required business logic. Usually, in most real-time applications, the superclass functions are implemented with generic logic which is common for all the next-level sub-classes.

Note :- If a function in the sub-class contains the same signature as the superclass non-private function, then the subclass function is treated as the **overriding function** and the superclass function is treated as the **overridden function.**

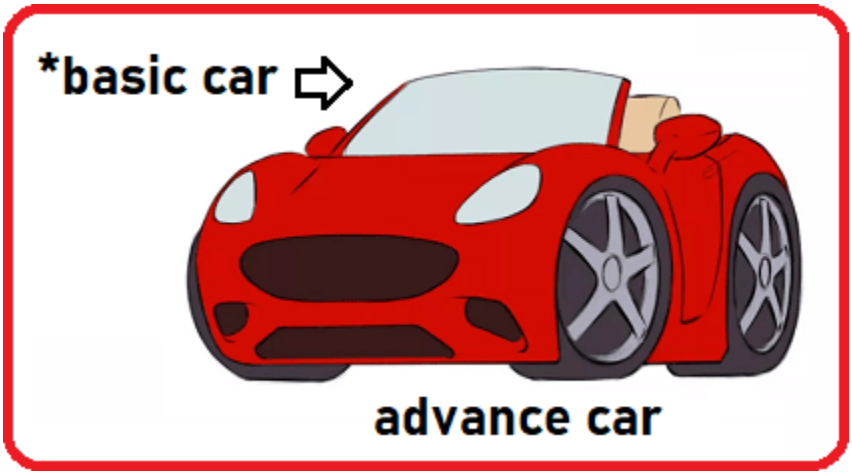
Virtual Function in C++ :- A virtual function in C++ is a member function that is declared within a base class using the virtual keyword and is re-defined by a derived class. When we refer to a **derived class object using the base class pointer** or base class reference variable, and when we can call a virtual function, then it will execute the function from the derived class.

So, Virtual Functions in C++ ensure that the correct function is called for an object, regardless of the expression used to make the function call.

```cpp
#include<bits/stdc++.h>
using namespace std;
class Base{
public:
    void Display(){
        cout << "Display of Base" << endl;
    }
};
class Derived:public Base{
    public:void Display(){
        cout << "Display of Derived " << endl;
    }
};
int main(){
    Base *p = new Derived ();
    p->Display ();
}
//Output
Display of Base
```

```cpp
#include<bits/stdc++.h>
using namespace std;
class Base{
public:
    virtual void Display(){
        cout << "Display of Base" << endl;
    }
};
class Derived:public Base{
    public:void Display(){
        cout << "Display of Derived " << endl;
    }
};
int main(){
    Base *p = new Derived ();
    p->Display ();
}
//Output
Display of Derived
```

Real-Time Example to Understand the need for Virtual Function :



Pointer is the basic car but the actual object is the advanced car. So, I am thinking of it as a basic car. How can I drive? How it will run? Is it will run like what I am thinking or run like an advanced car? It will run like an advanced car because I am pointing at the object of an advanced car.

So, when you call a function that is present in the base case as well as the derived class, whose function must be called? Based on the object the function must be called. Not based on the pointer.

See it is just like if you saw a donkey and say it's a horse, will it run like a horse? You are thinking of it like a horse but that's a donkey. It cannot run like a horse. In the same way, here the basic car is the reference and you are pointing at the advanced car object. So, the object is of the derived class and the pointer is of the base class.

When we call a function then the base class function is called. But logically it is wrong. Whose function must be called? The function must be called based on the object.

Key points about virtual functions :-

1. Virtual functions are used for achieving polymorphism
2. The base class can have virtual functions
3. Virtual functions **can be** overrides in the derived class
4. Pure virtual functions **must be** overrides by the derived class

Rules for virtual functions :-

1. Virtual functions in C++ cannot be static.
2. A virtual function in C++ can also be a friend function of another class.
3. The prototype/Signature of virtual functions should be the same in the base as well as the derived class.
4. In C++, a class may have a virtual destructor but it cannot have a virtual constructor.

Limitation of virtual functions :-

- Slower
- Difficult to debug

Runtime Polymorphism :-

```cpp
#include<bits/stdc++.h>
using namespace std;
class Car{
public:
    void Start(){cout<<"Car Started"<<endl;}
    void Stop(){cout<<"Car Stopped"<<endl;}
};
class Innova:public Car{
public:
    void Start(){cout<<"Innova Started"<<endl;}
    void Stop(){cout<<"Innova Stopped"<<endl;}
};

class Swift:public Car{
public:
    void Start(){cout<<"Swift Started"<<endl;}
    void Stop(){cout<<"Swift Stopped"<<endl;}
};
int main(){
    Car *c = new Innova();
    c->Start();
    c->Stop();
    c = new Swift();
    c->Start();
    c->Stop();
    return 0;
}
//Output
Car Started
Car Stopped
Car Started
Car Stopped
```

Now, suppose we want the derived class function should be called then we need to make the base class functions as virtual functions. This means if we want the Innova class Start function should be called then we have to make the Car class Start function virtual.

```cpp
#include<bits/stdc++.h>
using namespace std;
class Car{
public:
    virtual void Start(){cout<<"Car Started"<<endl;}
    virtual void Stop(){cout<<"Car Stopped"<<endl;}
};
class Innova:public Car{
public:
    void Start(){cout<<"Innova Started"<<endl;}
    void Stop(){cout<<"Innova Stopped"<<endl;}
};

class Swift:public Car{
public:
    void Start(){cout<<"Swift Started"<<endl;}
    void Stop(){cout<<"Swift Stopped"<<endl;}
};
int main(){
    Car *c = new Innova();
    c->Start();
    c->Stop();
    c = new Swift();
    c->Start();
    c->Stop();
    return 0;
}
//Output
Innova Started
Innova Stopped
Swift Started
Swift Stopped
```

Key Points of Runtime Polymorphism in C++:

- Same name different actions
- Runtime Polymorphism is achieved using function overriding
- Virtual functions are abstract functions of the base class
- The derived class must override the virtual functions
- A base class pointer pointing to a derived class object and an override function is called

Abstract class in C++

An abstract class in C++ has at least one pure virtual function by definition. In other words, a function that has no definition and these classes cannot be instantiated. The abstract class's child classes must provide body to the pure virtual function; otherwise, the child class would become an abstract class in its own right.

What is the purpose of the Pure Virtual function?

The purpose of a pure virtual function is to achieve polymorphism. We want the derived classes to override this function. So, it becomes mandatory for derived classes to override the pure virtual function. It means the base class is governing or giving a command to the child classes that you must override the pure virtual functions. So, it is defining an interface.

If a class is having a pure virtual function, then that class is called an **Abstract Class in C++.**

Can we create the object of Abstract Class in C++? No.

Can we create the pointer of the abstract class? Yes.

What is the purpose of inheritance?

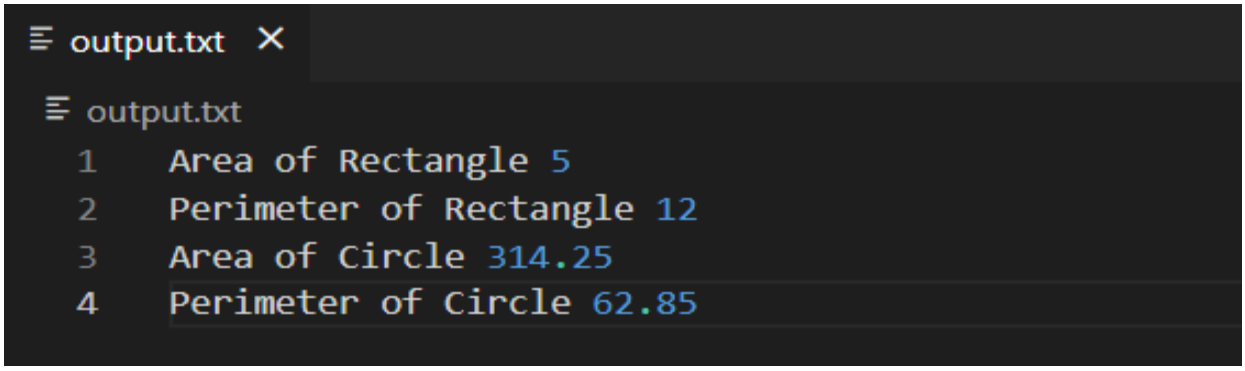
- Two things, one is reusability. The derived class gets the function from the base class.
- The second thing to achieve is polymorphism.

Base (All Concrete)	Base (Some Concrete) (Some Pure Virtual Func)	Base (All Pure Virtual Func)
|	|	|
Reusability	Reusability + Polymorphism	Polymorphism

Key Points of Abstract class in C++:

- The class having a pure virtual function is an abstract class.
- An **Abstract class** can have concrete also.
- The object of abstract class cannot be created
- A derived class must override pure virtual function, otherwise, it will also become an abstract class.
- The pointer of the abstract class can be created.
- The pointer of the abstract class can hold the object of the derived class.
- Abstract classes are used for achieving polymorphism
- Abstract with some concrete and some pure virtual functions.

```cpp
#include<bits/stdc++.h>
using namespace std;
class Shape{
public:
    virtual float Area() = 0;
    virtual float Perimeter() = 0;
};
class Rectangle:public Shape{
private:
    float length;
    float breadth;
public:
    Rectangle(int l = 1, int b = 1){
        length = 1;
        breadth = b;
    }
    float Area(){return length * breadth;}
    float Perimeter(){return 2 * (length + breadth);}
};

class Circle:public Shape{
private:
    float radius;
public:
    Circle(float r){radius = r;}
    float Area(){return 3.1425 * radius * radius;}
    float Perimeter(){return 2 * 3.1425 * radius;}
};

int main(){
    Shape *s = new Rectangle(10, 5);
    cout<<"Area of Rectangle "<<s->Area()<<endl;
    cout<<"Perimeter of Rectangle "<<s->Perimeter()<<endl;
    s = new Circle(10);
    cout<<"Area of Circle "<<s->Area()<<endl;
    cout<<"Perimeter of Circle "<<s->Perimeter()<<endl;
}


```



Friend Function and Friend Classes in C++

A friend function in C++ is defined as a function that can access private, protected, and public members of a class. A friend function can be a member of another class or can be a global function in C++.

Note :- **A friend function of a class is defined outside that class scope but it has the right to access all private, protected, and public members of the class. Even though the prototypes for friend functions appear in the class definition, friends are not member functions.**

Characteristics of a Friend Function in C++:

- It cannot be called using the object as it is not in the scope of that class.
- It can be invoked like a normal function without using the object.
- It cannot access the member names directly and has to use an object name and dot membership operator with the member's name.
- It can be declared either in the private or the public part.
- The function is not in the scope of the class of which it has been declared as a friend

```cpp
#include<bits/stdc++.h>
using namespace std;
class Test{
private:
    int x;
protected:
    int y;
public:
    int z;
    friend void Fun();
};
void Fun(){
    Test t;
    t.x = 10, t.y = 20, t.z = 30;
    cout << " X : " << t.x << endl;
    cout << " Y : " << t.y << endl;
    cout << " Z : " << t.z << endl;
}
int main(){
    Fun();
    return 0;
}
//Output
 X : 10
 Y : 20
 Z : 30
```

Friend Classes in C++:

In C++, a friend class can access private, protected, and public members of another class in which it is declared a friend. It is sometimes useful to allow a particular class to access private members of other classes.

Instead of writing the name before the My Class definition, we can also write friend class Your;

```cpp
#include<bits/stdc++.h>
using namespace std;
class My{
private:
    int x;
protected:
    int y;
public:
    int z;
    friend class Your;
};
class Your{
public:
    My m;
    void Fun(){
        m.x = 10;
        m.y = 20;
        m.z = 30;
        cout << "X = " << m.x << endl;
        cout << "Y = " << m.y << endl;
        cout << "Z = " << m.z << endl;
    }
};
int main(){
    Your obj;
    obj.Fun ();
    return 0;
}
```

Key Points of Friend Functions and Classes in C++ :-

- Friend functions are global functions
- They can access private, protected, and public members of a class upon their objects
- A class can be declared as a friend of another class
- All the functions of the friend class can access private and protected members of other classes.
- **Friendship is not mutual.** If class A is a friend of B, then B doesn't become a friend of A automatically.
- **Friendship is not inherited.**
- Friends should be used only for a limited purpose.
- Too many functions or external classes are declared as friends of a class with protected or private data, which is against the principle of encapsulation in object-oriented programming.

Static Data Members in C++ :-

We can define class members' static using the static keyword in C++. 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 available throughout the program. So static variables or static data members of a class belong to a class. That doesn't belong to an object.**

It means when we make a variable as static, that variable will occupy memory only once. And all the objects can share it.

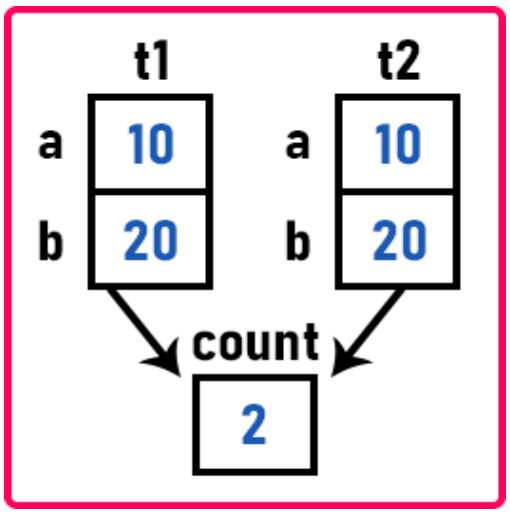
```
class Test{
        private:

                int a;
                int b;

        public:

           static int count;

                Test()
                {
                        a = 10;
                        b = 20;

                        count++;

                }
};
```

Non Static Data Members

Static Data Member

Incrementing Static Data Member

One more thing we have to address syntactically. When we have a static variable inside the class, **you must have declared it outside again.** So, when we declare the class variable outside the class, then it will become a global variable but we want it to be accessible only inside the Test class. So, we can use the scope resolution operator to make it accessible only inside the Test class.



```cpp
#include<bits/stdc++.h>
using namespace std;
class Test{
private:
    int a;
    int b;
public:
    static int count;
    Test(){
        a = 10, b = 20;
        count++;
    }
};
int Test::count = 0;
int main(){
    Test t1, t2;
    cout<<t1.count<<endl;
    cout<<t2.count<<endl;
    cout<<Test::count<<endl;
    return 0;
}
//Output
2
2
2
```

Static Member Functions in C++ :- By declaring a member function as static in C++, we 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 the class name and the scope resolution operator (::) and even we can also access them using objects of the class.

Note :- the important thing is Static Member Functions in C++ can access only static data members of a class. They cannot access non-static data members. So static member functions also belong to a class.

```cpp
#include<bits/stdc++.h>
using namespace std;
class Test{
public:
    int a;
    static int count;
    Test(){
        a = 10;
        count++;
    }
    static int getCount(){
        return count;
    }
};
int Test::count = 0;
int main(){
    cout<<"Calling count without object : "<<Test::count<<endl;
    cout<<"Calling getCount without object : "<<Test::getCount()<<endl;
    Test t1;
    cout<<"Calling count with object t1 : "<<t1.count<<endl;
    cout<<"Calling getCount with object t1 : "<<t1.getCount()<<endl;
}
//Output
Calling count without object : 0
Calling getCount without object : 0
Calling count with object t1 : 1
Calling getCount with object t1 : 1
```

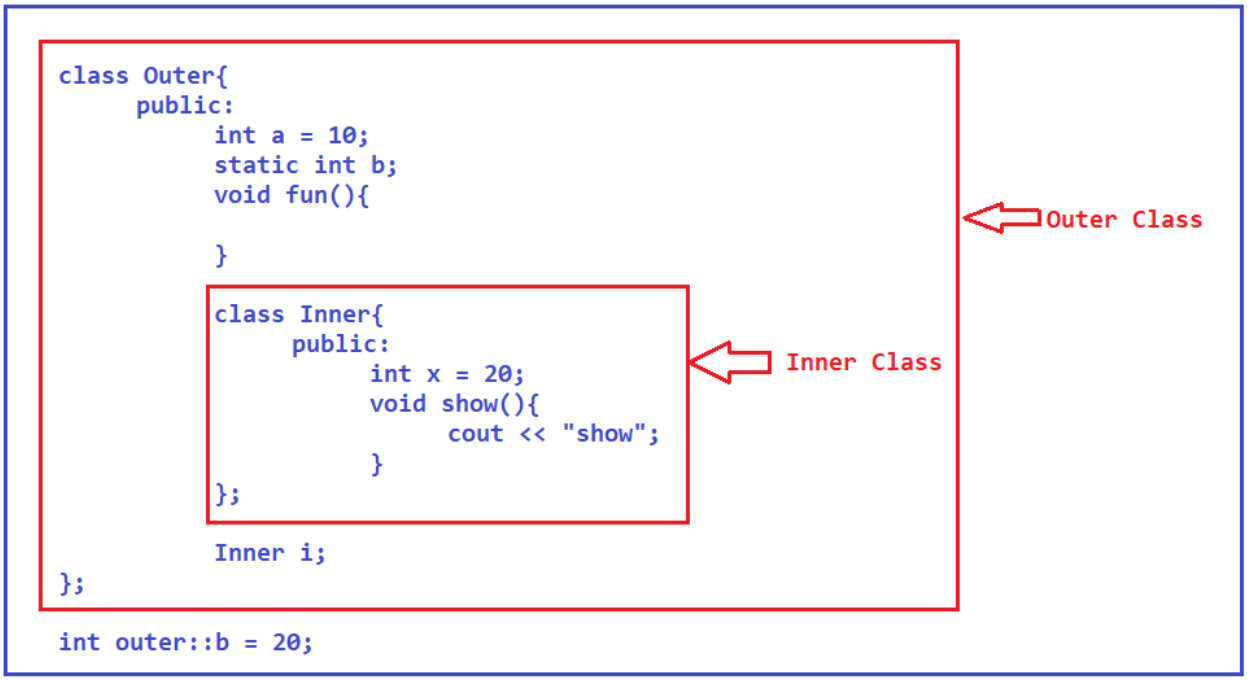
Key Points on Static Members in C++:

- Static data members are members of a class.
- Only one instance of static members is created for the entire class and shared by all objects of that class, no matter how many objects are created.
- It is initialized before any object of this class is created, even before the main starts.
- They can be accessed directly using the class name.
- Static member functions are functions of a class, they can be called using the class name, without creating the object of a class.
- They can access only static data members of a class; they cannot access non-static members of a class.
- **It is visible only within the class, but its lifetime is the entire program.**

Inner Class in C++ :-

Writing a class inside another class is called an **inner class.** It is useful only within that class. **If you are writing a class that is not useful everywhere or limited to one class.** Then you can write a class inside of another one.

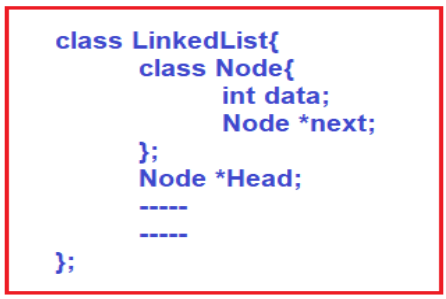
So, it is just like reducing the complexity of a bigger class, we can write smaller classes inside.



Note :- Can we create this object before the declaration of the Inner class? No, it must be done after the definition.

- Inner class can access the members of the Outer class if they are static.
- Can the Outer class create the object of the Inner class? Yes, it can. Now using that object can it access all the members of a class
- We can access only those members which are public. We cannot access private and protected members of the Inner class.

When to use Inner Classes in C++?



```cpp
#include<bits/stdc++.h>
using namespace std;
class Outer{
public:
    void Fun (){
        i.Display();
    }
    class Inner{
    public:
        void Display(){
            cout<<"Display of Inner"<<endl;
        }
    };
    Inner i;
};
int main(){
    Outer::Inner i;
    i.Display();
    return 0;
}
//Output
Display of Inner
```

Exception Handling in C++ :-

There are 3 types of errors that are occurred in programming:-

- **Syntax Error**
- **Logical Error**
- **Runtime Error**

Syntax Error :- What is a Syntax error? While typing the program, if the programmer mistypes something or did not write something properly then there is an error known as Syntax Error. So, who will identify this error? The compiler will identify this error.

Logical Error :- Suppose you wanted to do something and so wrote the procedure or function or some code but when you run the program the results are different i.e. not as expected.

Then is there any tool for solving this type of problem? Yes, that is a debugger. The debugger will help you to run the program line by line or statement by statement, so in each statement, you can see what is happening and wherever it is going wrong and you can catch it and you can solve that problem.

Note :- The most important thing is that these two types of errors (Syntax Error And Logical Error) are faced by programmers.

Runtime Error :- Suppose you are running some business and you wanted me to develop an application for your business. So, I developed an application and that application is perfect. It is neither having any syntax errors nor having any logical errors. Then I delivered the software to you and you are a user and using that application for your business.

So, the user is going to use the application and while using the application he/she must use it properly. **If he/she mishandled the application then it may cause errors during the runtime of a program.** It's not development time. **At the development time, as a programmer, I have faced logical and syntax errors that I have already removed.** Now it is perfect. There are no errors. And you are using the program. So, at runtime, you are facing errors.

What Could Be the Cause of Error During Runtime?

- First of the reasons for runtime errors is bad input.
- Second, the program that I have given you requires an internet connection but if you are not providing an internet connection then the program cannot continue.
- Third, the program that I have given to you needs some files on your computer system but you have deleted those files or my program is using a printer but you do not have a printer. So, these are the list of problems. **These are nothing but the unavailability of resources**.

All the things that are outside the program or not in the control of the program. Those are in the control of the user.

What happens in the case of runtime errors?

In case of runtime errors, the program crashes or program stops abnormally without completing its execution. Suddenly the program will stop, that's how the runtime errors are dangerous for the user. So, who is responsible? The user is responsible because as a developer I did my job perfectly and the user is not providing proper resources or proper input so the program is failing. Now let us give you some real-life examples.

What can we call these runtime errors?

These runtime errors are called exceptions. Why we are using the term exception? Suppose I have given you a perfect program and it will always run perfectly except in some conditions. Those conditions are giving a bad input or not providing resources. So, the program will not run in those cases. That's why we call these exceptional cases or exceptions.

What are Exceptions?

An exception is nothing but a situation in which we get the runtime error. Let us see in automobile engineering. If there is no fuel in the car then the car will indicate that there is less fuel or it is in a reserved condition so you must find the closest fuel station or gas station.

In the same way. the programs should also behave like that. If there is bad input then the program should not stop suddenly. It should ask the user that **'this input is not proper please enter another type of Input and try again"**. If there is no internet connection then the program should not stop suddenly but it should give a prompt to the user that **"we found that there is no internet connection please connect to the internet and run it again"**. So, **the idea is the program should not terminate under such a situation but guide the user to solve the problem.**

Because who is there to resolve it? The programmer will not come to the client-side and check for the errors and remove them. The program itself can give a proper and meaningful message to the user so that users can remove this type of problem.

Note :- Giving a proper message to the user and informing them about the exact problem. And also providing him guidance to solve that problem, that's it. This is the objective of exception handling.

How to Implement Exception Handling in C++?

```
try {
  // Block of code to try
  throw exception; // Throw an exception when a problem arise
}
catch () {
  // Block of code to handle errors
}
```

Inside the try block, if there is any error then it will jump to the catch block and execute the statements inside the catch block.

- **try**: It represents a block of code that can throw an exception.
- **catch**: It represents a block of code that is executed when a particular exception is thrown.
- **throw**: It is used to throw an exception inside a try block

```cpp
#include<iostream>
using namespace std;
int Division (int a, int b) throw (int){
    if (b == 0)
        throw 1;
    return a / b;
}
int main(){
    int x, y, z;
    cout << "Enter two Numbers:";
    cin >> x >> y;
    try{
        z = Division (x, y);
        cout << z << endl;
    }
    catch (int e){
        cout << "Division by zero " << e << endl;
    }
    cout << "Bye" << endl;
}
//Output
Enter two Numbers:20 0
Division by zero 1
Bye
```

Throwing Exceptions from C++ constructors :-

An exception should be thrown from a C++ constructor whenever an object cannot be properly constructed or initialized. Since there is no way to recover from failed object construction, an exception should be thrown in such cases.

Since C++ constructors do not have a return type, **it is not possible to use return in the codes. Therefore, the best practice is for constructors to throw an exception to signal failure.**

```cpp
#include<iostream>
using namespace std;
class Rectangle{
private:
    int length;
    int breadth;
public:
    Rectangle(int l, int b){
        if (l < 0 || b < 0){
            throw 1;
        }
        else{
            length = l;
            breadth = b;
        }
    }
    void Display()
    {
        cout<<"Length: "<<length<<" Breadth: "<<breadth;
    }
};
int main(){
    try{
        Rectangle r1(10, -5);
        r1.Display();
    }
    catch (int num){
        cout<<"Rectangle Object Creation Failed";
    }
}
//Output
Rectangle Object Creation Failed
```

Key Points regarding exception handling :-

- An exception in C++ is thrown by using the throw keyword from inside the try block. The throw keyword allows the programmer to define custom exceptions.
- Multiple handlers (catch expressions) can be chained – each one with a different exception type. Only the handler whose argument type matches the exception type in the throw statement is executed.
- C++ does not require a finally block to make sure resources are released if an exception occurs.

Using Multiple catch blocks in C++ :-

```cpp
#include<iostream>
#include<conio.h>
using namespace std;
int main(){
    int arr[3] ={-1, 2};
    for (int i = 0; i < 2; i++){
        int num = arr[i];
        try{
            if (num > 0) throw 1;
            else throw 'a';
        }
        catch (int ex){
             cout << "Integer Exception" << endl;
        }
        catch (char ex){
             cout << "Character Exception" << endl;
        }
    }
    return 0;
}
```

Generic Catch Block (Catch - All) in C++ :-

The following example contains a generic catch block to catch any uncaught errors/exceptions. catch(…) block takes care of all types of exceptions.

```cpp
#include <iostream>
#include<conio.h>
using namespace std;
int main(){
    int arr[3] = { -1, 2, 5 };
    for (int i = 0; i < 3; i++){
        int num = arr[i];
        try{
            if(num == -1) throw 1;
            else if(num == 2) throw 'a';
            else throw "Generic";
        }
        catch(int ex){
            cout<<"Integer Exception"<<endl;
        }
        catch (char ex){
            cout<<"Character Exception"<<endl;
        }
        //Generic catch block
        catch (...){ // to catch anytime of exceptions
            cout<<"Generic Exception"<<endl;
        }
    }
    return 0;
}
```

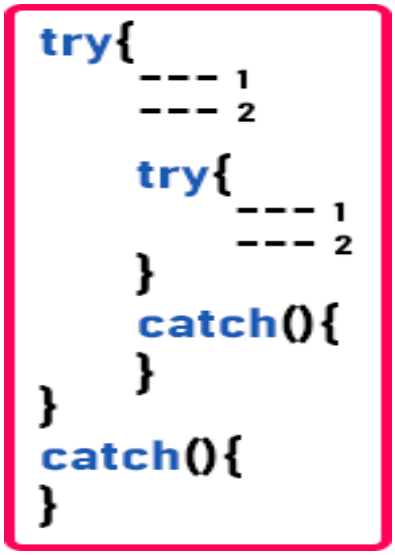
Note : If we write catch-all first, then all the exceptions will be handled here only. The lower catch blocks will never be executed. So, the catch-all block must be the last block.

Throwing User-Defined Type in C++ :-

```cpp
#include<iostream>
#include<conio.h>
using namespace std;
class myExp{
    //Nothing is here
};
int main (){
    int num = 2;
    try{
        if (num == 1) throw 1;
        else if (num == 2) throw myExp();
        else if (num == 3) throw "Unknown Exception";
        else cout<<"Value"<<num<<endl;
    }
    catch(int ex){
        cout<<"Integer Exception"<<endl;
    }
    catch (myExp e){
        cout<<"myExp Exception"<<endl;
    }
    catch (...){
        cout<<"Unknown Exception"<<endl;
    }
    return 0;
}
int main (){
    int num = 2;
    try{
        if (num == 1) throw 1;
        else if (num == 2) throw myExp();
        else if (num == 3) throw "Unknown Exception";
        else cout<<"Value"<<num<<endl;
    }
    catch(int ex){
        cout<<"Integer Exception"<<endl;
    }
    catch(myExp e){
        cout<<"myExp Exception"<<endl;
    }
    catch(...){
        cout<<"Unknown Exception"<<endl;
    }
    return 0;
}
```

Can we have a Try Block inside a Try block?

Yes, we can have a try block inside another try block in C++. The following diagram shows the syntax of writing nested try blocks in C++.



Example to Understand Try-Catch Blocks in C++:

```
class MyExp1{};

class MyExp2 : public MyExp1{};
```

So, we have these two classes. And MyExp2 is publicly inherited from the MyExp1 class. So, MyExp1 is a parent class and MyExp2 is a child class.

```
try{
    ---
    ---
    ---
}
catch(MyExp1 e){
}
catch(MyExp2 e){
}
```

```
try{
    ---
    ---
    ---
}
catch(MyExp2 e){
}
catch(MyExp1 e){
}
```

As you can see, we have written catch blocks for both types of exceptions. So, is this the correct format of catch block? No. We have written the catch block for the parent class first and then for the child class. **We must write catch blocks for the child class first and then for the parent class as shown in the below image.**

Inheriting Exception Class from Built-in Exception Class :-

If you are throwing your own class exception then better extend your class from the built-in exception class in C++ as follows. **But it is not mandatory.**

```
class MyException: public exception {
};
```

Overriding the exception class what method in C++ :-

After inheriting our custom exception class from the built-in exception class, do we have to override anything? Yes, we can override one method which is **"what"** method as follows. But this is not mandatory as in our previous example we have not overridden the what method and it is working fine.

```
class MyException : public exception{
        public:
                char * what(){
                        return "My Custom Exception";
                }
};
```

How to make the function throws something in C++?

```
int Division(int a, int b) throw (MyException)
{
      if (b == 0)
          throw MyException();
      return a / b;
}
```

```
int Division(int a, int b) throw (int)
{
        if (b == 0)
            throw 1;
        return a / b;
}
```

So, whatever the type of value you are throwing, you can mention that in the brackets. **And if there are more values then you can mention them with commas as follows:**

```
int Division(int a, int b) throw (int, MyException)
{
        if (b == 0)
            throw 1;
        return a / b;
}
```

```cpp
#include<iostream>
using namespace std;
#include <exception>
class MyException:public exception{
public:
    char * what(){
        return "My Custom Exception";
    }
};
int Division(int a, int b) throw (int, MyException){
    if (b == 0) throw 1;
    if (b == 1) throw MyException();
    return a / b;
}
int main(){
    int x = 10, y = 1, z;
    try{
        z = Division(x, y);
        cout<<z<<endl;
    }
    catch(int x){
        cout<<"Division By Zero Error"<<endl;
    }
    catch (MyException ME){
        cout<<"Division By One Error"<<endl;
        cout<<ME.what ()<<endl;
    }
    cout<<"End of the Program"<<endl;
}
```

Can we use cout instead of throw?

If you use cout, the user will know the error. **And if you use throw, it will inform the calling function about the error.**

Are return and throw the same?

Return is for returning results. The throw is for reporting an error. If you change their roles then the roles of Try and Catch will also change.

Template in C++

Template in C++ allows us to define Generic Functions and Generic Classes. That means using a Template we can implement Generic Programming in C++. They are going to work with a variety of data types. Templates in C++ can be represented in two ways. They are as follows.

- **Function templates**
- **Class templates**

```
int max(int a, int b){
        return a > b ? a : b;
}
```

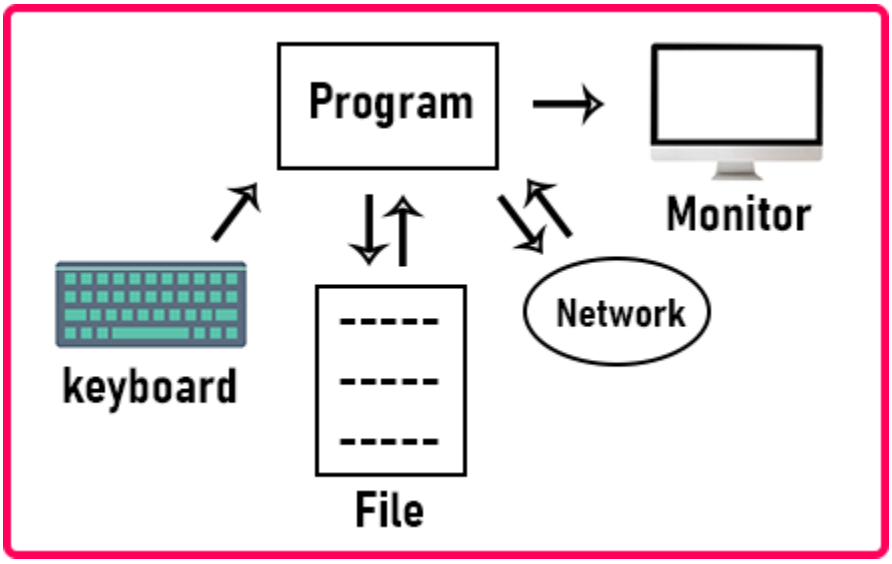
```
T max(T a, T b){
        return a > b ? a : b;
}
```

```cpp
#include<iostream>
using namespace std;
template <class T, class R>
void Add (T x, R y){
    cout << x + y << endl;
}
int main(){
    //Integer and Integer
    Add (4, 24);
    //Float and Float
    Add (25.7f, 67.6f);
    //Integer and double
    Add (14, 25.5);
    //Float and Integer
    Add (25.7f, 45);
    return 0;
}
//Output
28
93.3
39.5
70.7
```

```cpp
#include <iostream>
using namespace std;
template<class T>
class Stack{
    private:
        T * stk;
        int top, size;
    public:
        Stack (int sz){
            size = sz;
            top = -1;
            stk = new T[size];
        }
        void Push(T x);
        T Pop();
};
template<class T>
void Stack<T>::Push(T x){
    if (top == size - 1)
        cout<<"Stack is Full"<<endl;
    else{
        top++;
        stk[top] = x;
        cout<<x <<" Added to Stack"<<endl;
    }
}
template<class T>
T Stack<T>::Pop(){
    T x = 0;
    if (top == -1)
        cout<<"Stack is Empty"<<endl;
    else{
        x = stk[top];
        top--;
        cout<<x<<" Removed from Stack"<<endl;
    }
    return x;
}
int main(){
    //Stack of Integer
    Stack<float> stack1(10);
    stack1.Push(10), stack1.Push(23),stack1.Push(33);
    stack1.Pop();
    //Stack of double
    Stack<double> stack2(10);
    stack2.Push(10.5), stack2.Push(23.7), stack2.Push(33.8);
    stack2.Pop();
    return 0;
}
```

File Handling in C++

Before learning File Handling in C++, let us first learn about Streams in C++. **A stream is a flow of data or a flow of characters.** Streams are a way to perform input and output operations in C++. They are objects that allow you to read or write a sequence of characters, **typically from or to a file**. Streams are used to perform these operations in a flexible and efficient manner.



If we have a program, then the program may be getting the data from the keyboard or program may be sending the data to the monitor or the program may be accessing the data from an external file.

istream: Represents an input stream and is used to read data from an input source (e.g., a file or the console).

ostream: Represents an output stream and is used to write data to an output destination (e.g., a file or the console).

iostream: Represents an input/output stream and can be used to both read and write data.

Now similarly, for file access also, there are classes available that are ifstream for input stream and ofstream for the output stream.

For input stream from the keyboard, we have already a built-in object present in iostream header file that is **cin** and also an extraction operator that is **">>"**. So directly we can use cin with extraction operator. **Cin is an object of istream class**. And we have another object that is **cout** that we commonly used for displaying the text on the screen. **cout is an object of ostream**. cout is used with the insertion operator **"<<"**.

```
#include <fstream>
int main(){
        ofstream outfile("my.txt");
}
```

So, this object outfile will associate with the file my.txt. Now whatever you will write into this outfile object it will get dropped into that file. **It's like you have connected a pipe from your program to that file on the disk. So, whatever you drop in the outfile it will get stored in the file.**

Next thing, when we were opening this file through outfile object, the file should be already existing. **If it is existing then outfile will open the file and if it is not existing then it will create a new file with the name my.txt.**

Next point, if the already file is existing and **it is having some content then what will happen to that content?** The outfile will truncate that content or remove the content. Suppose my.txt has some content, my.txt: 100 200 300

So, we have these numbers in the my.txt file. **These values will be removed and fresh content will be written.** If you want to append the new content after the old content then there is a mode available that is **ios::app.** 'app' stands for append.

So like this mode, there are two modes available that are ios::app and **ios::trunc. 'trunc' stands for truncate.**

After writing the content, you must close the file by writing the following statement. **outfile.close();**

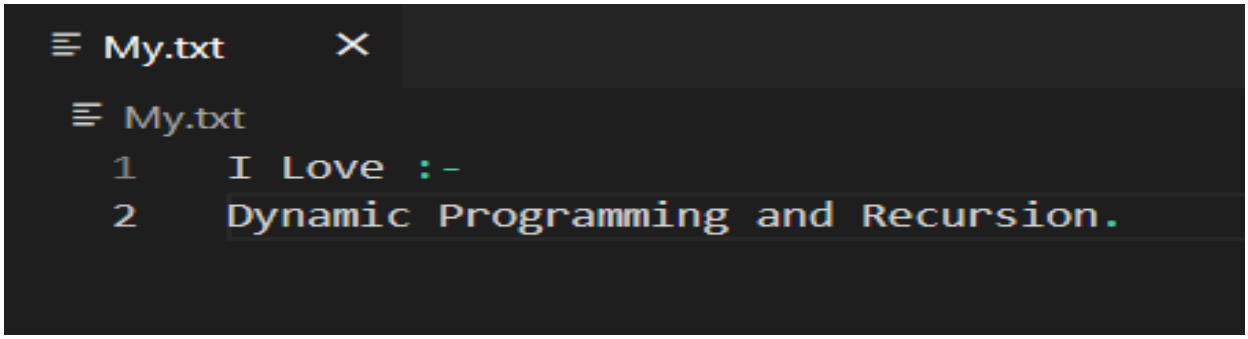
This statement will close the stream and the file will be free from the program. It is important to close the file.

See sometimes, you have connected a pen drive or memory card to your laptop and you have kept one of the files open and you try to eject the card. Then you will get a message that "some program is using your memory card or pen drive". So, you cannot eject it. It means the operating system knows that some program is using some file so it will not allow you to eject. **It means that files are being occupied or that resource is in use.** So, when the program is not used then it should be released. If the program ends, then

automatically the resource will be released. So, it is a good practice to close the resource when you have finished.

```cpp
#include<iostream>
#include<fstream>
using namespace std;
int main(){
    ofstream outfile("My.txt", ios::trunc);
    outfile<<"I Love :- "<<endl;
    outfile<<"Dynamic Programming and Recursion."<<endl;
    outfile.close();
    return 0;
}
```

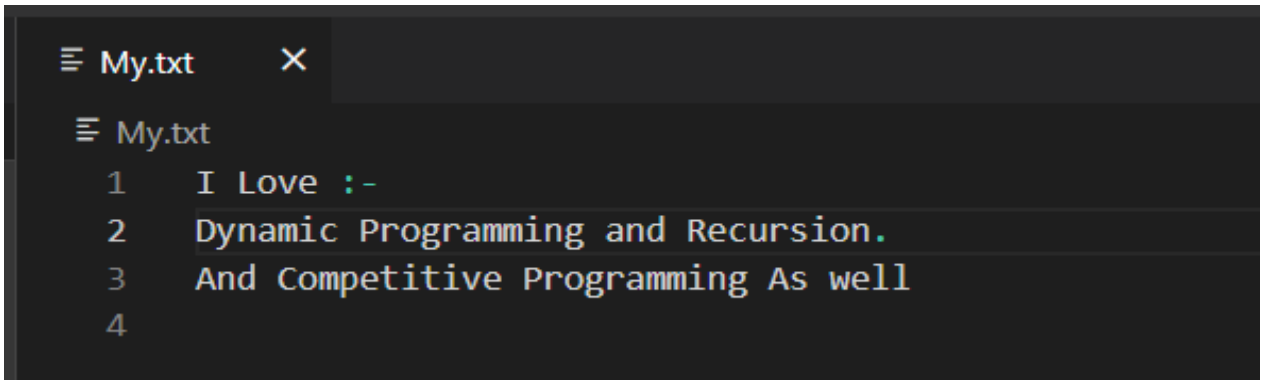
Output:-



Now, if you don't want to remove the old data instead you want to append the new data below the old data, then you need to use the ios::app

```cpp
#include<iostream>
#include<fstream>
using namespace std;
int main(){
    ofstream outfile("My.txt", ios::app);
    outfile<<"And Competitive Programming As well"<<endl;
    outfile.close();
    return 0;
}
```

Output:-



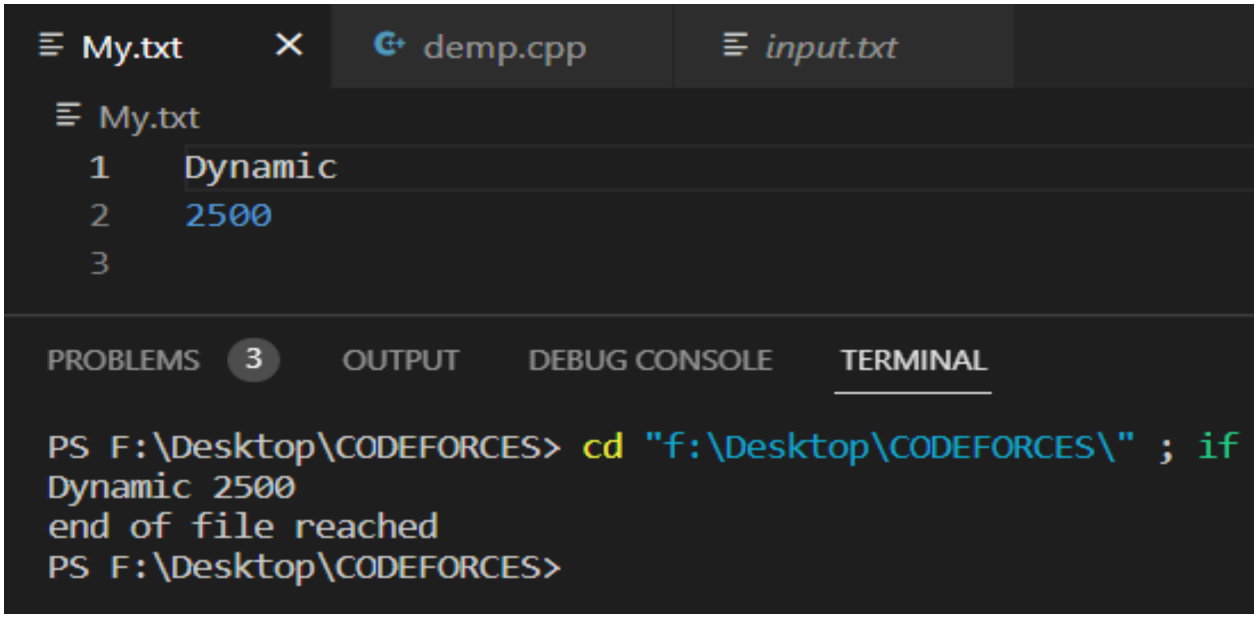
How to Open a File in C++?

```cpp
int main() {
        ifstream infile;
        infile.open("my.txt");
        if (!infile)
                cout << "File Cannot be opened";
}
```

So, the infile.open("my.txt") statement will open the file. Now one important thing, when you are reading from a file then the file must be existing. It will not create a new file so the file must be existing at the location where the program is saved.

Next, we have checked for the file status. **if(!infile)** statement will check for whether the file is open or not.
There is another method for checking the file status.
infile.is_open(), this method will return true if the file is opened otherwise it will return false.

```cpp
#include <iostream>
#include <fstream>
using namespace std;
int main(){
    ifstream infile;
    infile.open("my.txt");
    string str;
    int x;
    infile>>str;
    infile>>x;
    infile>>x;
    cout<<str<<" "<<x<<endl;
    if (infile.eof()){
        cout<<"end of file reached";
    }
    infile.close();
}
```



What is eof function in C++?

One important thing, once we have finished reading the content, we have reached the end of the file (eof). So sometimes we need to check whether we have reached the end or not. **infile.eof(),** this statement will return true if we are at the end of the file. So, we have written the print statement "end of file reached".

Example to Understand File Reading in C++:

```cpp
#include<iostream>
#include<fstream>
using namespace std;
int main(){
    ifstream ifs;
    ifs.open ("my.txt");
    if (ifs.is_open())
        cout<<"File is Opened"<<endl;
    string name;
    int roll;
    string branch;
    ifs >> name >> roll >> branch;
    ifs.close();
    cout<<"Name: "<<name<<endl;
    cout<<"Roll: "<<roll<<endl;
    cout<<"Branch: "<<branch<<endl;
    if(ifs.eof())
        cout<<"End of File Reached";
}
```

My.txt | demp.cpp | input.txt

My.txt

```
Aman_Babu
2019017
CSE

```

PROBLEMS 3 OUTPUT DEBUG CONSOLE TERMINAL

```
PS F:\Desktop\CODEFORCES> cd "f:\Desktop\CODEFORCES\" ; if ($?) { g++ demp.cp
File is Opened
Name: Aman_Babu
Roll: 2019017
Branch: CSE
End of File Reached
PS F:\Desktop\CODEFORCES>
```

Serialization in C++ :- Serialization is the process of storing and retrieving the state of an object. Suppose we want to store the student information in a file and again we want to retrieve the information of a student from a file. While working with Serialization the class must have a default constructor.

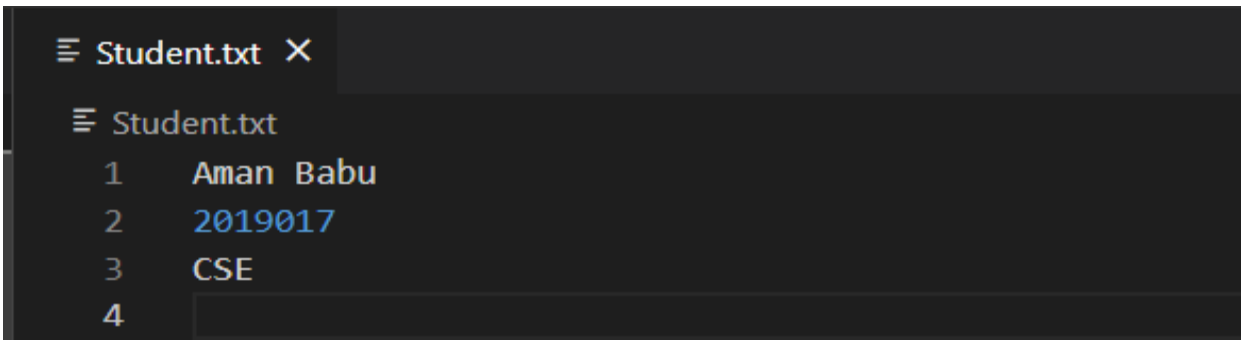
```cpp
#include<iostream>
#include<fstream>
using namespace std;
class Student {
public:
    string name;
    int rollno;
    string branch;
    Student(){}
    Student(string s, int roll_no, string t){
        name = s;
        rollno = roll_no;
        branch = t;
    }
    friend ofstream & operator << (ofstream & ofs, Student s);
    friend ifstream & operator >> (ifstream & ifs, Student & s);
};
int main(){
    Student s1("Aman Babu", 2019017, "CSE");
    ofstream ofs("Student.txt", ios::trunc);
    ofs<<s1.name<<endl;
    ofs<<s1.rollno<<endl;
    ofs<<s1.branch<<endl;
}
```

Student.txt

```
1  Aman Babu
2  2019017
3  CSE
4
```

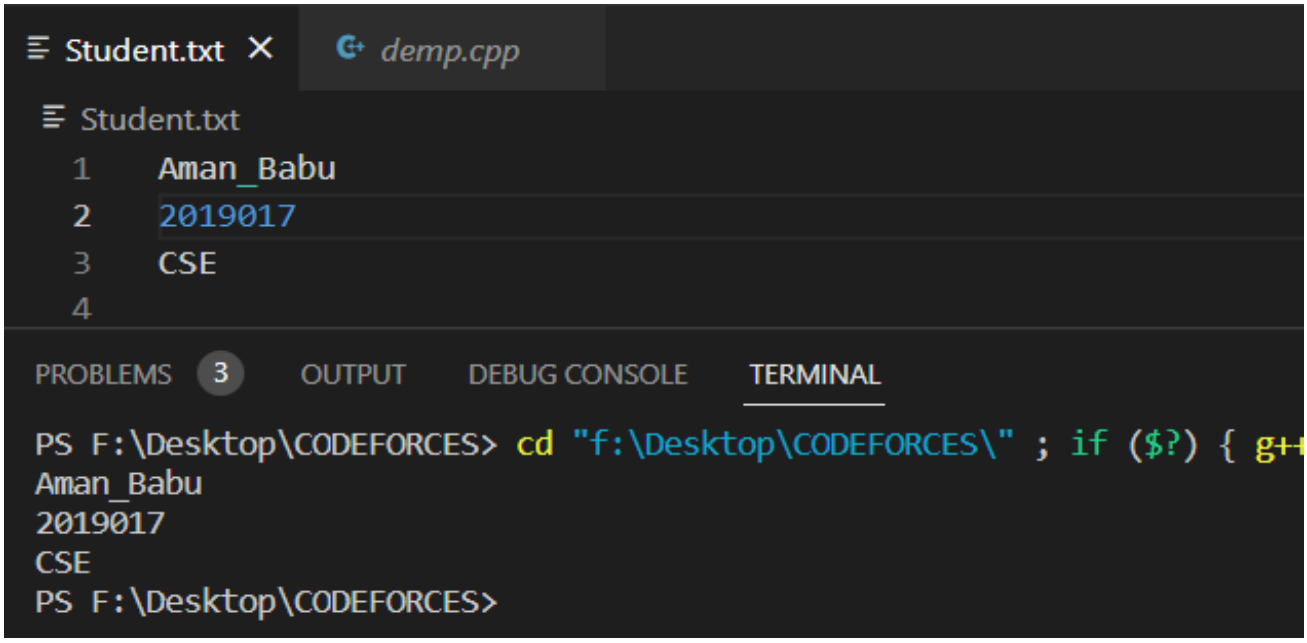
Why we have stored all the values individually? If we can store the complete object with name, rollno and branch then we can say that we have stored the object. So, **if we can store the whole object at once then we can store an object as well as we can retrieve the object. Yes.** This is possible.We have to overload the operator of the ofs stream in the student class

```cpp
#include<iostream>
#include<fstream>
using namespace std;
class Student{
public:
    string name;
    int rollno;
    string branch;
    Student(){}
    Student(string n, int r, string b){
        name = n;
        rollno = r;
        branch = b;
    }
    friend ofstream & operator<<(ofstream &ofs, Student s);
    friend ifstream & operator>>(ifstream &ifs, Student &s);
};
ofstream & operator<<(ofstream & ofs, Student s){
    ofs<<s.name<<endl;
    ofs<<s.rollno<<endl;
    ofs<<s.branch<<endl;
    return ofs;
}
int main(){
    Student s1("Aman Babu", 2019017, "CSE");
    ofstream ofs("Student.txt", ios::trunc);
    ofs<<s1;
    ofs.close ();
}
```


Student.txt
Student.txt
1 Aman Babu
2 2019017
3 CSE
4

Retrieving the State of an Object in C++:-

```cpp
#include<iostream>
#include<fstream>
using namespace std;
class Student{
public:
    string name;
    int rollno;
    string branch;
    Student(){}
    Student(string n, int r, string b){
        name = n;
        rollno = r;
        branch = b;
    }
    friend ofstream & operator<<(ofstream &ofs, Student s);
    friend ifstream & operator>>(ifstream &ifs, Student &s);
};
ofstream & operator<<(ofstream & ofs, Student s){
    ofs<<s.name<<endl;
    ofs<<s.rollno<<endl;
    ofs<<s.branch<<endl;
    return ofs;
}
ifstream & operator >> (ifstream & ifs, Student & s){
    ifs>>s.name>>s.rollno>>s.branch;
    return ifs;
}
int main(){
    Student s1("Aman_Babu", 2019017, "CSE");
    ofstream ofs("Student.txt", ios::trunc);
    ofs<<s1;
    ofs.close ();
    Student s2;
    ifstream ifs("Student.txt");
    ifs>>s2;
    cout<<s2.name<<endl<<s2.rollno<<endl<<s2.branch<<endl;
    ifs.close();
}
```



Text and Binary Files in C++:

There are two types of files:

- **Text Files**
- **Binary Files**

Difference between Text and Binary Files in C++ with Examples:
Text files are human-readable whereas binary files are machine-readable.
Suppose we write a number that is 13. Then how is it written in the text file and binary file? Let's see the difference.

13 is an integer value. The binary form of 13 is 1101. So how many bytes do integers take? In most compilers, an integer takes 4 bytes but to make our explanation easy, we consider an integer takes 2 bytes. So, suppose 13 (integer) is taking 2 bytes then how many bits are there in 1101? **0000 0000 0000 1101**

Total 16 bits of a binary number. So, the same binary form is stored in the binary file with all 16 bits. That's why we called this a binary file.

Then what are text files? 13 will not be written in the text file. it will convert into ASCII. **The ASCII code of 1 is 49 and for 3 the ASCII code is 51.** Then what is the binary form of 49 and 51?
49 - 110001, 51 - 110011
These are the binary forms of 49 and 51. ASCII codes take 8 bits of binary so, **49 - 00110001, 51 - 00110011**

0011 0001 0011 0011 (This will be stored in the text file that is)
(ASCII code of 1) + (ASCII code of 3) = 49 + 51 =
(binary code of 49) + (binary code of 51) = 0011000100110011

Both text and binary files stored the binary number. Then how text files are human-readable?
Suppose we have a text file that has stored 13. When we open this file in notepad then what will notepad do? For every 8 bits, it will be converted into ASCII and then display that symbol.

0011 0001 0011 0011

So, for this binary number, the first 8 bits will be converted into ASCII which is 1 and the next 8 bits will be converted into ASCII which is 3. So, 1 and 3 will be displayed in a notepad.
And what about binary files? Suppose we have a binary file that contains the following code.

0000 0000 0000 1101

In a binary file, the first 8 bits will be taken. **In this case, all the bits are 0, and 0 is the ASCII code of some unknown garbage symbol. So, we will get some junk characters.** If you open this file in notepad then it might not show any meaningful symbols because the first 8 bits are not making any meaningful ASCII code. That's it. We cannot read and understand it because that is a pure binary form or we can say it is machine-understandable.

If you are reading from the text file then you can use the insertion and extraction operator for reading and writing the data in the form of text.
And if you want to read and write it in the form of binary then the first thing in C++ that you have to use is **iso::binary mode**. And also there are functions available for reading and writing that are **read()** and **write()**.
read() is available in the file input stream and write() is available in the file output stream.

Which file is faster? Binary file or text file?
The binary file is faster than the text file. Because text file needs conversion (symbols to ASCII and ASCII to binary). But in the binary file, there is no conversion required.

Which file takes more space?

Text files will take more space and binary will take less space. Suppose we have the 4-digit number so a text file will take 4 bytes but a binary file will take 2 bytes.

Manipulators in C++

Manipulators are helping functions in C++ that are used to modify the input/output stream. What it means, **it will not modify the value of a variable, it will only modify the streams or format of streams using the insertion (<<) and extraction (>>) operators.**

- Manipulators are special functions that can be included in the I/O statement to alter the format parameters of a stream.
- Manipulators are operators that are used to format the data display.
- To access manipulators, the file iomanip should be included in the program.

Manipulators are used for enhancing streams or formatting streams. For writing the data, we can adopt some formats. For example, a common manipulator that we used is the endl that is used for the endline. Instead of endl, we can also say that cout << "\n";

This will also print a new line. So, endl is a manipulator which is used for formatting stream. So, it is useful for formatting output streams.

Integer Manipulators in C++:-

hex – It will display the data in hexadecimal.

oct – It will display data in the octal form.

dec – To display data in decimal form.

Float Manipulators in C++:-

fixed: It will show in the fixed floating-point number.

scientific: It will display the number in scientific form or exponent form.

There are other manipulators also available.

setw – It will set some amount of space for displaying the data. For example,

```cpp
#include<iostream>
#include<iomanip>
using namespace std;
int main(){
    cout<<"Hex 163: "<<hex<<163<<"\n";
    cout<<"Oct 163: "<<oct<<163<<"\n";
    cout<<"Dec 163: "<<dec<<163<<"\n";
    cout<<"Fixed Manipulator: "<<fixed<<162.6454 <<endl;
    cout<<"Scientific Manipulator: "<<scientific<<162.6454<<"\n";
    cout<<setw(10)<<"World";
    return 0;
}
```

Output:-

```
PROBLEMS 3   OUTPUT   DEBUG CONSOLE   TERMINAL

PS F:\Desktop\CODEFORCES> cd "f:\Desktop\CODEFORCES\" ; if ($?)
Hex 163: a3
Oct 163: 243
Dec 163: 163
Fixed Manipulator: 162.645400
Scientific Manipulator: 1.626454e+002
     World
```

Constants in C++

As the name suggests, Constants in C++ are given to such variables or values which cannot be modified or changed once they are defined. That means they are fixed values in a program, unlike the variables whose value can be altered.

There are two ways to define constants in C++.

- By using the const keyword
- By using #define preprocessor

```
int main(){
    const int x = 10;
    x++;
    cout << x;
}
```

```cpp
#include<iostream>
using namespace std;
int main(){
    const int x = 10;
    x++;
    cout << x;
    return 0;
}
```

Output:-

```
PROBLEMS 4   OUTPUT   DEBUG CONSOLE   TERMINAL

PS F:\Desktop\CODEFORCES> cd "f:\Desktop\CODEFORCES\" ; if ($?) { g
demp.cpp: In function 'int main()':
demp.cpp:5:6: error: increment of read-only variable 'x'
     x++;
      ^~
PS F:\Desktop\CODEFORCES>
```

These variables are also called read-only variables.

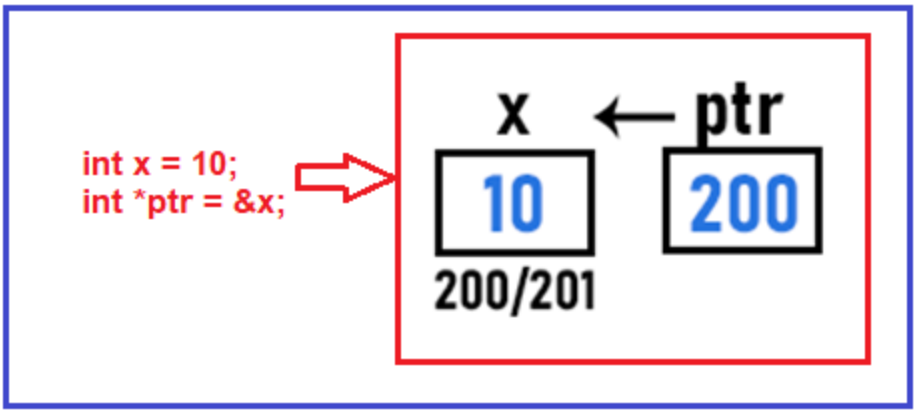
#define x 10

What is the difference between #define and const in C++?

- #define is a preprocessor directive and it is performed before the compilation process starts. But const is an identifier that will consume memory that cannot be modified.
- **#define is just a symbolic constant and const is a constant identifier.**
- **#define will not consume memory, but const will consume memory according to the data type.**
- #define is not a part of the language, it is a pre-compiler whereas const is a part of the language that is part of the compiler.

Constant Pointers in C++:-

```
int main()
{
        int x = 10;
        int *ptr = &x;
        ++*ptr;
        cout << x;
}
```



Then we have written ++*ptr. So, *ptr will increment the value of x from 10 to 11 as follows.

But

```
int main(){
        int x = 10;
        const int *ptr = &x;
        ++*ptr;
        cout << x;
}
```

In this case, the pointer ptr can point to x and **access or read x, but it cannot modify the value x. So here, ++*ptr is not allowed.** So, by using the constant pointer, we cannot modify the data.

This can be written in other ways also, like,

int const *ptr = &x; OR const int *ptr = *x;

Suppose we have one more variable y and it is having a value of 30 (int y = 30;). So, can we point ptr at y? **Yes. By writing, ptr = &y,** ptr will point at y. So, we can make the pointer point to some other data, but you cannot modify that. Here also, we cannot modify the value of y.

```
#include<iostream>
using namespace std;
int main(){
    int x = 10;
    int y = 30;
    const int *ptr = &x;
    ptr = &y; //Allowed
    //++*ptr; //Not Allowed
    cout << *ptr;
}
//output
30
```

Another Usage of Constants in C++:

- **int * const ptr = &x;**

Now ptr is a constant pointer to an integer. So, who is the constant here? Pointer ptr is constant here. What does it mean? It means once the pointer ptr is pointing at x, it cannot be

changed. After that, you will be unable to change it to another variable. **We cannot write ptr = &y** because ptr is constant. So, the address in that pointer cannot be changed.

```cpp
#include<iostream>
using namespace std;
int main(){
    int x = 10;
    int y = 30;
    int * const ptr = &x;
    ++*ptr; // Allowed
    ptr = &y; //Not Allowed
    cout<<*ptr;
}
```

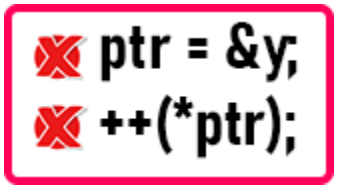
Output :-

```
PROBLEMS 4   OUTPUT   DEBUG CONSOLE   TERMINAL

PS F:\Desktop\CODEFORCES> cd "f:\Desktop\CODEFORCES\" ; if ($?) { g
demp.cpp: In function 'int main()':
demp.cpp:8:12: error: assignment of read-only variable 'ptr'
     ptr = &y; //Not Allowed
           ^
PS F:\Desktop\CODEFORCES>
```

Two constants in C++:-

- **const int * const ptr = &x;**

Two constants in C++:-



We cannot modify the data by writing ++ *ptr and we cannot assign the pointer to another date like ptr = &y.

```cpp
#include<iostream>
using namespace std;
int main(){
    int x = 10;
    int y = 30;
    const int * const ptr = &x;
    ++*ptr; //const int * const ptr = &x; Allowed
    ptr = &y; //Not Allowed
    cout<<*ptr;
    return 0;
}
```

Output :-

```
PROBLEMS 5   OUTPUT   DEBUG CONSOLE   TERMINAL

PS F:\Desktop\CODEFORCES> cd "f:\Desktop\CODEFORCES\" ; if ($?) { g++ demp.cpp
demp.cpp: In function 'int main()':
demp.cpp:7:8: error: increment of read-only location '*(const int*)ptr'
     ++*ptr; //const int * const ptr = &x; Allowed
        ^~~
demp.cpp:8:12: error: assignment of read-only variable 'ptr'
     ptr = &y; //Not Allowed
           ^
PS F:\Desktop\CODEFORCES>
```

There are 3 types of constant pointers in C++.

- **The pointer can be constant:** Pointer cannot be modified i.e. ptr = &y is not valid here.

- **Data can be constant:** Pointer cannot modify the data i.e. ++*ptr is not valid.
- **Both the pointer and data can be constant:** Here pointer cannot be modified i.e. i.e. ptr = &y is not valid, and data is also can't be modify i.e. ++*ptr is not valid.

Const Keyword in C++ Functions:-

```cpp
#include<iostream>
using namespace std;
class Demo{
public:
    int x = 10;
    int y = 20;
    void Display() const{
        x++; //Not Allowed
        cout << x << " " << y << endl;
    }
};
int main(){
    Demo d;
    d.Display();
}
```

Output:-

```
PROBLEMS 4  OUTPUT  DEBUG CONSOLE  TERMINAL

PS F:\Desktop\CODEFORCES> cd "f:\Desktop\CODEFORCES\" ; if ($?) { g++ demp.cpp -o
demp.cpp: In member function 'void Demo::Display() const':
demp.cpp:8:10: error: increment of member 'Demo::x' in read-only object
        x++; //Not Allowed
         ^~
PS F:\Desktop\CODEFORCES>
```

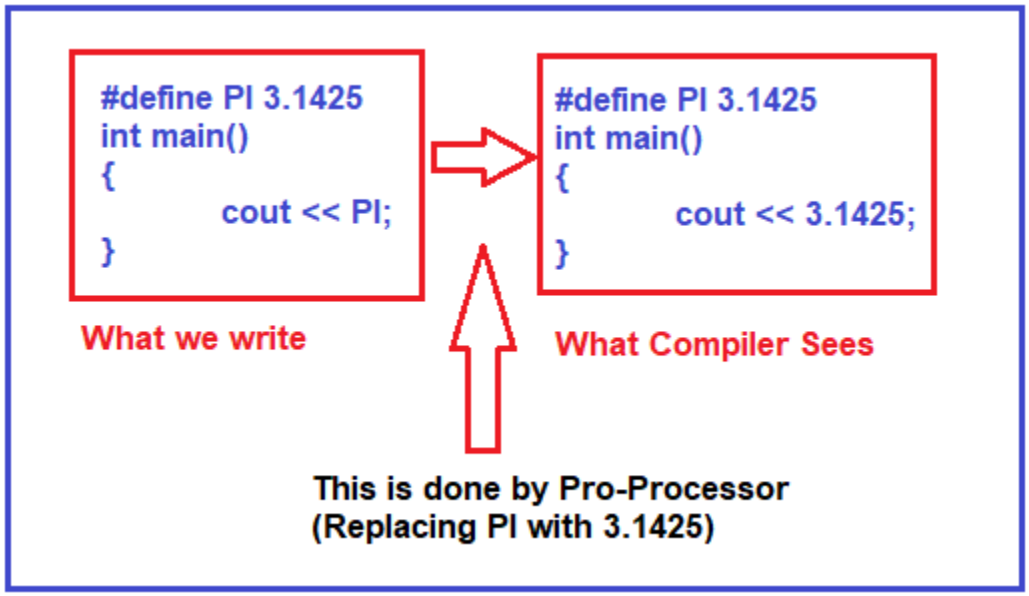
Another use of const keyword:-

```
void fun(const int &x, const int &y){
        ---
}
```

So, parameters can also be made constant. Now a and b cannot be changed through function fun. We cannot write x++ or y++ here.

Preprocessor Directives in C++

Macros or Preprocessor Directives in C++ are instructions to the compiler. **We can give some instructions to the compiler so that before the compiler starts compiling a program,** it can follow those instructions and perform those instructions. The most well-known Macros or Preprocessor Directives that we used in our program is **#define.**

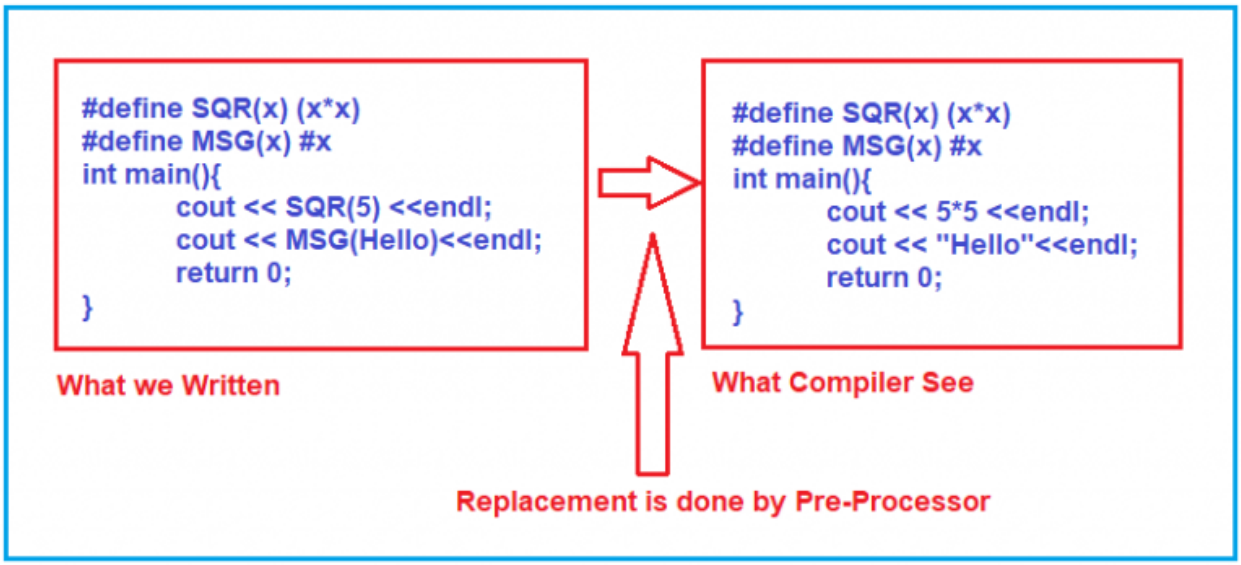


#define c cout

Now, can we write c << 10? Yes. What will happen? This c will be replaced by cout before the compilation.

Define Function using #define Preprocessor Directive in C++:-

```
#define SQR(x) (x*x)
#define MSG(x) #x
int main(){
        cout << SQR(5) <<endl;
        cout << MSG(Hello)<<endl;
        return 0;
}
```



With #x, we created another function, MSG(x). Whatever the parameters we send in MSG, that will be converted into a string.

So, if we write cout << MSG(Hello), then it means "Hello". So, we have given Hello without double-quotes. But MSG (Hello) will be replaced by "Hello".

#ifndef Directive in C++, #ifndef. It means if not defined.

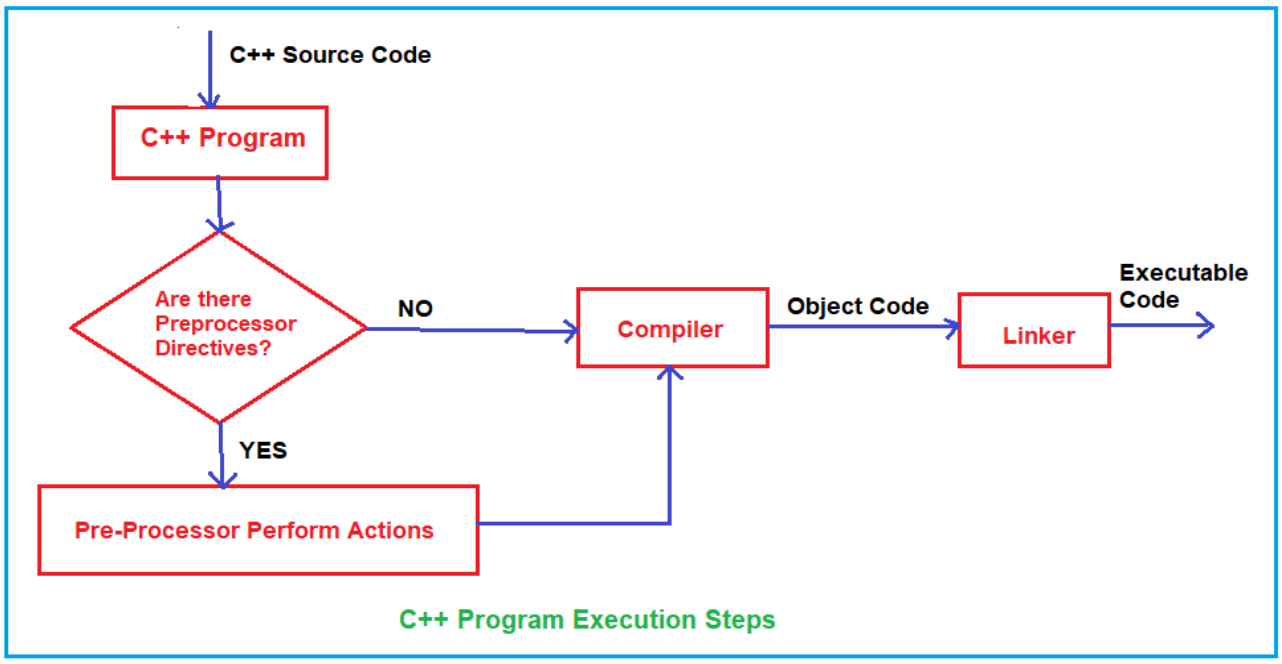
```
#ifndef
        #define PI 3.1425
#endif
```

This PI is defined if it is not already defined, then only it will be defined, otherwise, it will not be defined again.

```cpp
#include <iostream>
using namespace std;
#define max(x, y) (x > y ? x : y)
#ifndef PI
    #define PI 3.1425
#endif
int main(){
    cout<<PI<<endl;
    cout<<max(121, 125)<<endl;
    return 0;
}
```

```
PROBLEMS 3   OUTPUT   DEBUG CONSOLE   TERMINAL

PS F:\Desktop\CODEFORCES> cd "f:\Desktop\CODEFORCES\" ; if ($?) { g++ d
3.1425
125
PS F:\Desktop\CODEFORCES>
```

How a C++ Program Executes?



Namespaces in C++ :- Namespaces are used for removing name conflicts in C++. If you are writing multiple functions with the same name but are not overloaded, they are independent functions. They are not a part of the class.

```
void fun (){
        cout << "First";
}
void fun(){
        cout << "Second";
}
int main (){
        fun();
}
```

So, we have two functions of name fun and one main function. The main function is to call the function fun. So, which fun function will be called? First fun or second fun? **First of all, the compiler will not compile our program. The compiler will say that we are redefining the same function multiple times.**

But we want the same function but we want to remove this ambiguity. We have to remove the name conflicts. So, for this, we can use namespaces in C++. Let us define our namespace as follows:

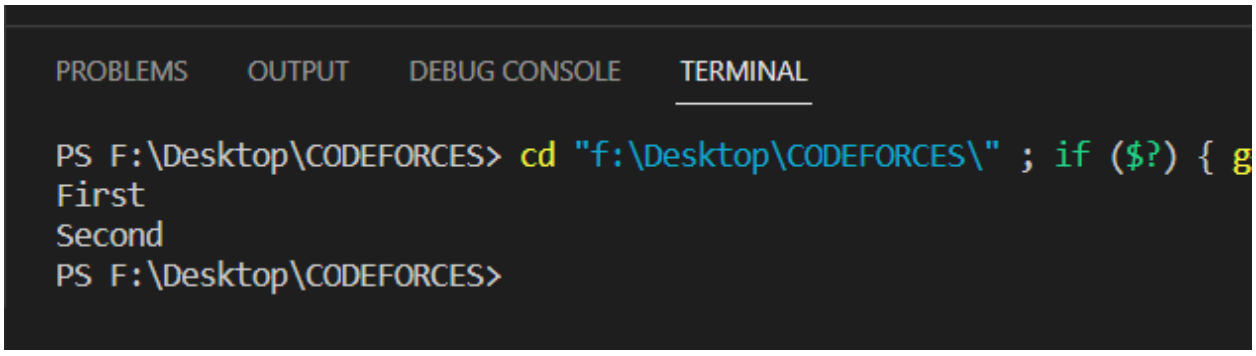
```
namespace First{
        void fun (){
                cout << "First";
        }
}

namespace Second{
        void fun(){
                cout << "Second";
        }
}
```

```
First::fun();
Second::fun();
```

So first we have to write the namespace, then the scope resolution operator, and then the function name.

```cpp
#include <iostream>
using namespace std;
namespace First{
    void fun(){
        cout<<"First"<<endl;
    }
}
namespace Second{
    void fun(){
        cout<<"Second"<<endl;
    }
}
int main(){
    First::fun ();
    Second::fun ();
    return 0;
}
```

Output :-



A namespace is a container that holds a set of identifiers. **These identifiers can be variables, functions, or classes,** and they are used to organize and structure code. A namespace helps to prevent naming conflicts by providing a way to group related identifiers together and give them a unique name.

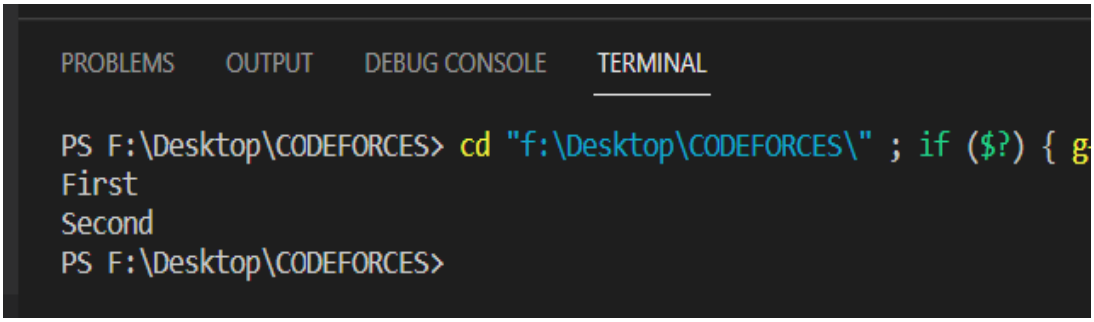
using namespace First;

Now when we call the function fun anywhere in the program, it will call fun inside the first namespace. If you still want to call the second namespace function, then you can write,

Second::fun();

```cpp
#include <iostream>
using namespace std;
namespace First{
    void fun(){
        cout<<"First"<<endl;
    }
}
namespace Second{
    void fun(){
        cout<<"Second"<<endl;
    }
}
using namespace First;
int main(){
    fun();
    Second::fun();
    return 0;
}
```

Output:-



In our C++ program, we have been using namespace std, so there is one namespace std that has the cin and cout objects. So that's why we just write a single line using namespace std; and we can use cin and cout objects. Otherwise, we have to write like this,

std::cout << "Hello";

Destructors in C++ :-

A destructor in C++ is also a member function like a constructor which is also invoked automatically when the object goes out of scope or we can also explicitly call the destructor to destroy the object by making a call to delete.

A destructor has the same name as the class name as a constructor, but to differentiate between them the destructor function is preceded by a tilde (~) symbol

```
class Test{
        public:
                Test(){
                        cout << "Test Object Created" << endl;
                }
                ~Test(){
                        cout << "Test Object Destroyed" << endl;
                }
};
```

When the Destructor function will be Called in C++?

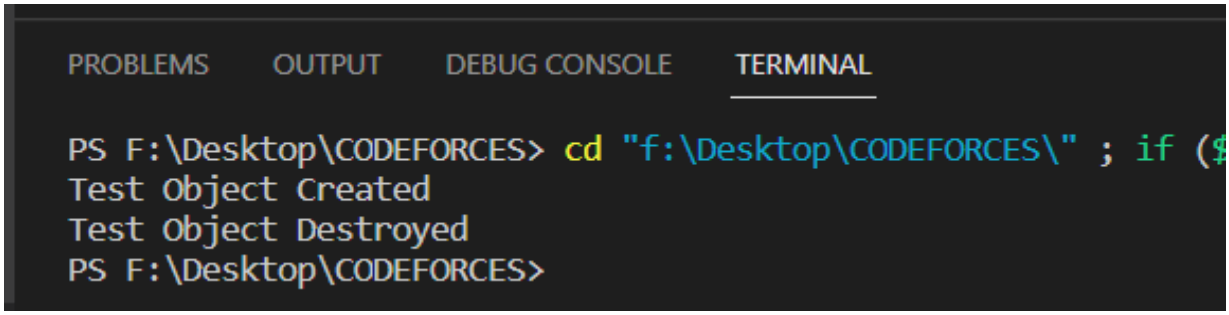
This destructor function will be called when the object will be destroyed. We know once the main function ends, automatically all the objects that are associated with the program get destroyed. **So, the constructor is called when the object is created and the destructor is called when the object is destroyed.**

```
Test *obj = new Test();
delete obj;
```

As you can see in the above example, we have written delete obj. So, when we are creating an object by writing a **new Test()**, the constructor will be called. And when we delete that object by writing delete obj then destructor will be called.

```cpp
#include <iostream>
using namespace std;
class Test{
public:
    Test(){
        cout<<"Test Object Created" << endl;
    }
    ~Test(){
        cout<<"Test Object Destroyed" << endl;
    }
};
int main(){
    Test *obj = new Test();
    delete obj;
    return 0;
}
```

Output:-



If an object is created dynamically then it should also be deleted when not required. So, when we say delete obj, the destructor function will be called.

Is it Mandatory to Define a Destructor in C++?

No, it is not mandatory to define a destructor in C++. If as a programmer we do not define a destructor, the compiler will provide a default one automatically. And for many classes, this compiler-provided default destructor is sufficient. **We only need to define a destructor explicitly in our class when the class handles external resources that need to be released.**

What is the use of Destructor in C++?

The constructor is used for initialization purposes. It is also used for allocating resources. Similarly a destructor is used for deallocating resources or releasing resources. we have to deallocate the resources. Which type of resources? When you are acquiring any external things like heap memory, file, network connection, etc. these are all external sources.

```cpp
#include <iostream>
#include <fstream>
using namespace std;
class Test{
    int *p;
    ifstream fis;
    Test(){
        p = new int[10];
        fis.open ("my.txt");
    }
    ~Test(){
        delete[]p;
        fis.close ();
    }
};
```

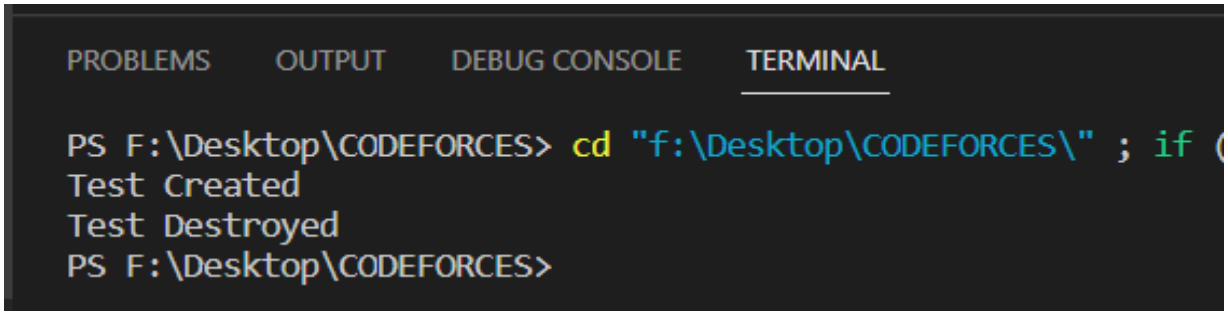
Here we have a class called Test. Inside this Test class, we have a pointer p. We have allocated this p dynamically inside the constructor. Then inside the destructor, we must delete the allocated memory otherwise this will cause memory leak problems.

Can we overload the constructor? Can we have multiple constructors? Yes. Can we have multiple destructors? No. Can the constructor or destructor return anything? No. All the rules of constructors are followed on destructor except destructor cannot be overloaded. A destructor can be virtual also.

Example to Understand Static Allocation of an object in C++

```cpp
#include <iostream>
using namespace std;
class Test{
public:
    Test(){
        cout<<"Test Created"<<endl;
    }
    ~Test(){
        cout<<"Test Destroyed"<<endl;
    }
};
void fun(){
    Test t;
}
int main(){
    fun();
}
```

Output:-



Example to Understand Dynamic Allocation of Object in C++

```cpp
#include <iostream>
using namespace std;
class Test{
private:
    int *p;
public:
    Test(){
        p = new int[10];
        cout<<"Test Created"<<endl;
    }
    ~Test(){
        delete[]p;
        cout<<"Test Destroyed"<<endl;
    }
};
void fun(){
    Test *t = new Test ();
    delete t;
}
int main(){
   fun();
}

```

Output:-

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL

PS F:\Desktop\CODEFORCES> cd "f:\Desktop\CODEFORCES\" ; if ($
Test Created
Test Destroyed
PS F:\Desktop\CODEFORCES>
```

Note :- If we are not writing delete []p then the destructor will not be called.

Rules to be Followed While Declaring Destructors in C++:-

- C++ Destructors do not accept arguments.
- You cannot return value from a destructor even if a void is not allowed. So, the destructor should not have any return type.
- Destructors in C++ cannot be declared const, volatile, or static. **However, destructors in C++ can be invoked for the destruction of objects which are declared as const, volatile, or static.**
- Destructors in C++ can be declared as virtual.

When Destructor Function Called in C++?

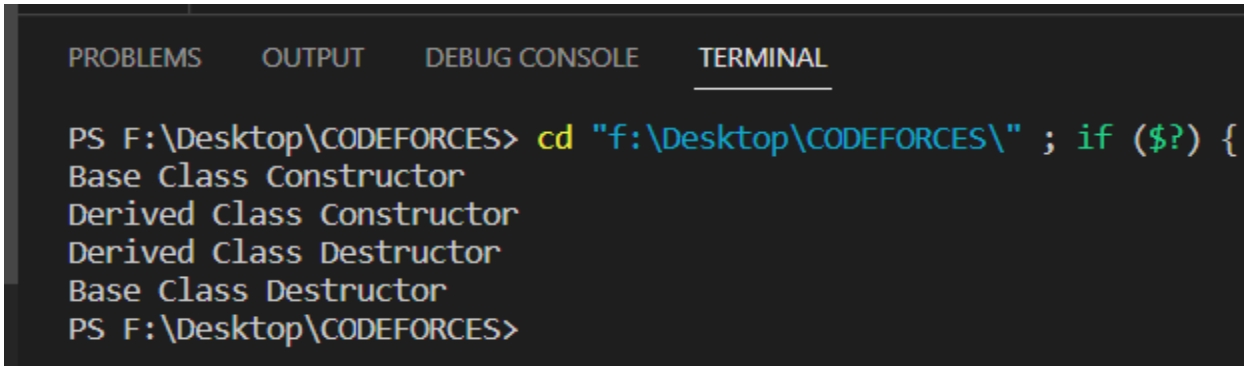
- A local (automatic) object with block scope goes out of scope.
- An object allocated using the new operator is explicitly deallocated using delete.
- The lifetime of a temporary object ends.
- A program ends and global or static objects exist.
- The destructor is explicitly called using the destructor function's fully qualified name.

Virtual Destructors in C++ :-

Example to Understand Destructors inheritance in C++:

```cpp
#include <iostream>
using namespace std;
class Base{
public:
    Base(){
        cout<<"Base Class Constructor"<<endl;
    }
    ~Base(){
        cout<<"Base Class Destructor"<<endl;
    }
};
class Derived:public Base{
public:
    Derived(){
        cout<<"Derived Class Constructor"<<endl;
    }
    ~Derived(){
        cout<<"Derived Class Destructor"<<endl;
    }
};
int main(){
    Derived d;
    return 0;
}
```

Output :-



Here, we have a Base class. It is having two public members functions that are Base() and ~Base() i.e. one public constructor and one public destructor.

Next, we have a Derived class which is inherited from the Base class. It is also having two public member functions that are constructor and destructor.

How are the Destructors Called in C++?

First, the destructor of the Derived class will be called then the destructor of the Base class will be called. The below messages will be printed on the screen.

Derived Class Destructor

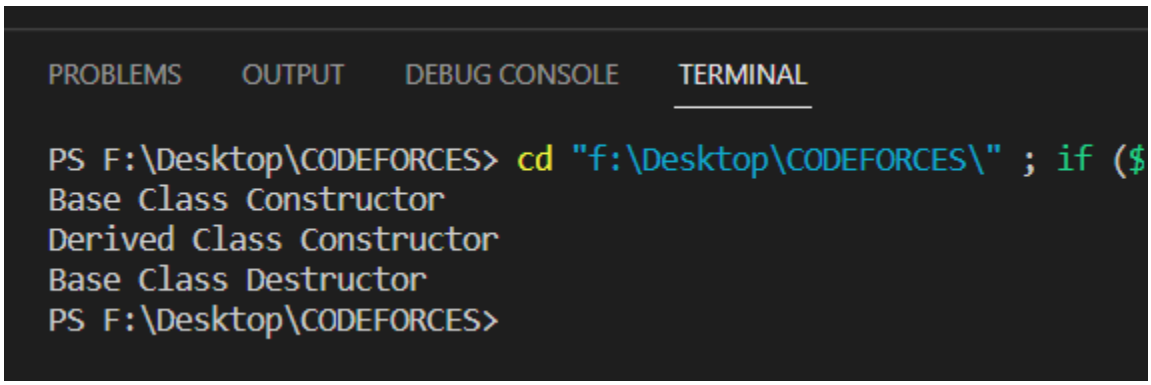
Base Class Destructor

So, if you are creating an object of a derived class and when it is being destroyed, first, the destructor of the Derived class will be called then the destructor of the Base class will be called.

So, in contrast to the constructor, destructors are called from the bottom-up approach.

```cpp
#include <iostream>
using namespace std;
class Base{
public:
    Base(){
        cout<<"Base Class Constructor"<<endl;
    }
    ~Base(){
        cout<<"Base Class Destructor"<<endl;
    }
};
class Derived:public Base{
public:
    Derived(){
        cout<<"Derived Class Constructor"<<endl;
    }
    ~Derived(){
        cout<<"Derived Class Destructor"<<endl;
    }
};
int main(){
    Base *p = new Derived();
    delete p;
    return 0;
}
```

Output :-



Virtual Destructors in C++:

A Virtual Destructor in C++ is used to release the memory space which is allocated by the derived class object while deleting the object of the derived class using a base class pointer object.

Here, inside the Main method, we have a Base class pointer and the object is of the Derived class.

Then on the next line, we are deleting p because dynamically created memory should be deleted when not required using the delete operator.

So new Derived object will be destroyed. Which constructor will be called? Remember in C++, the function calls are depending upon the pointer, not upon the object.

So, the pointer p is of the Base class. So only the Base class destructor will be called. The derived class destructor will not be called.

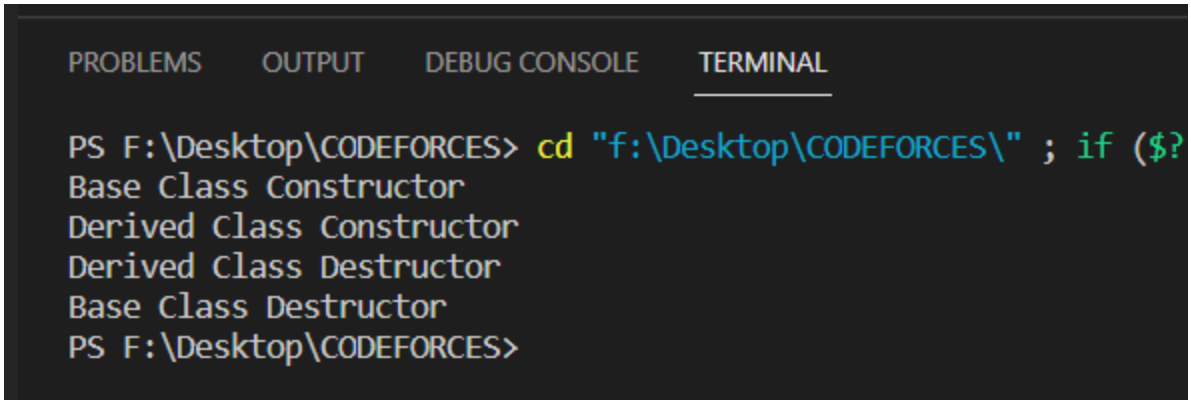
But suppose you want the destructor of the derived class should also be called and then the Base class destructor called. then we have to write virtual keyword before the name of Base class Destructor as,

```
virtual ~Base ()
{
        cout << "Base Class Destructor" << endl;
}
```

If we write virtual keyword there, then even though the pointer is of Base class and object is of Derived class and when the object is deleted then first, **the destructor of Derived class will be called and then the destructor of Base class will be called.**

```cpp
#include <iostream>
using namespace std;
class Base{
public:
    Base(){
        cout<<"Base Class Constructor"<<endl;
    }
    virtual ~Base(){
        cout<<"Base Class Destructor"<<endl;
    }
};
class Derived:public Base{
public:
    Derived(){
        cout<<"Derived Class Constructor"<<endl;
    }
    ~Derived(){
        cout<<"Derived Class Destructor"<<endl;
    }
};
int main(){
    Base *p = new Derived();
    delete p;
    return 0;
}
```

Output:-



So, this is how virtual destructors work.
If you don't make the Base class destructor virtual and you are using a pointer of the Base class and object of the Derived class **then only the resources acquired by the Base class code will be released. But the resources acquired by Derived class code will not be released because that will not execute.**
So, this must also execute. If you are using the Base class pointer and Derived class object then you should make the Base class destructor virtual.

Pure Virtual Destructors in C++ :-
The only difference between Virtual Destructor and Pure Virtual Destructor in C++ is that the **pure virtual destructor will make the class Abstract,** hence we cannot create an object of that class but we can create a pointer of that class.
There is no requirement to implement pure virtual destructors in the derived classes as we are going to create instances of the derived class which are going to be pointed by the Base class pointer.

```cpp
#include <iostream>
using namespace std;
class Base{
public:
    Base(){
        cout<<"Base Class Constructor"<<endl;
    }
    virtual ~Base() = 0;
};
Base::~Base(){
    cout<<"Base Class Destructor"<<endl;
}
class Derived:public Base{
public:
    Derived(){
        cout<<"Derived Class Constructor"<<endl;
    }
    ~Derived(){
        cout<<"Derived Class Destructor"<<endl;
    }
};
int main(){
    Base *p = new Derived();
    delete p;
    return 0;
}
```

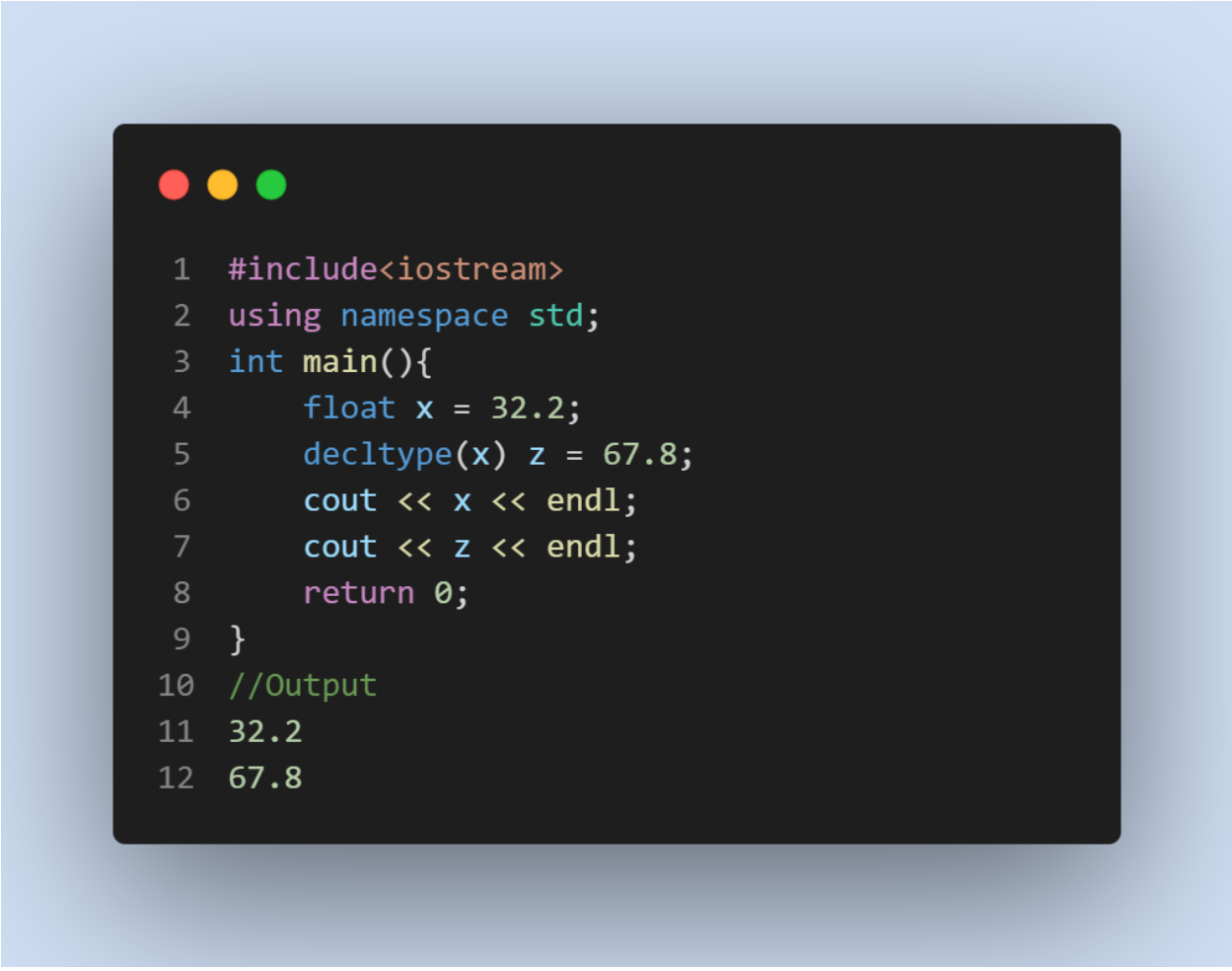
Output:-

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL

PS F:\Desktop\CODEFORCES> cd "f:\Desktop\CODEFORCES\" ; if ($?) { g++ de
Base Class Constructor
Derived Class Constructor
Derived Class Destructor
Base Class Destructor
PS F:\Desktop\CODEFORCES>
```

Some features of C++ 11

Auto Keyword in C++ :- The auto keyword is a very powerful and useful feature of C++ 11. When a programmer uses the library functions or some functions of the built-in classes then he/she doesn't have to know about the data type. We can simply use the auto declaration to do it automatically. It saves the programmer's time.

Decltype in C++ 11:-



```cpp
#include<iostream>
using namespace std;
int main(){
    float x = 32.2;
    decltype(x) z = 67.8;
    cout << x << endl;
    cout << z << endl;
    return 0;
}
//Output
32.2
67.8
```

Final Keyword in C++

Mainly the final keyword is used for two purposes. They are as follows :-

- Restrict Class Inheritance
- Restrict Method Overriding

Restrict Class Inheritance :-

```cpp
#include<iostream>
using namespace std;
class Parent final{
    void show(){}
};
class Child:Parent{
    void show (){}
};
int main(){
    return 0;
}
Error :-
demp.cpp:6:7: error: cannot derive from 'final'
base 'Parent' in derived type 'Child'
 class Child:Parent{
       ^~~~~
```

Restrict Method Overriding :-

```
1  #include<iostream>
2  using namespace std;
3  class Parent{
4      virtual void show() final{}
5  };
6  class Child : Parent{
7      void show(){}
8  };
9  int main(){
10     return 0;
11 }
12 demp.cpp:4:18: error: overriding
13 final function 'virtual void Parent::show()'
14      virtual void show() final{}
15                   ^~~~
```

Lambda Expressions :- Lambda Expressions is useful for defining unnamed functions in C++. So, we can define the function wherever we like. They are more likely inline functions but they will have the features of the function. Let us see the syntax for a lambda expression or unnamed function in C++.

```
[ Capture_List ] ( Parameter_List ) -> Return_Type { body };
```

```cpp
#include <iostream>
using namespace std;
int main(){
    [](){cout << "Hello"<<endl;}();
    [](int x, int y) {cout << "Sum: " << x + y <<endl;}(10, 5);
    int x = [](int x, int y) {return x+y;} (10, 5);
    cout<<x;
}
```

Output :-

```
Hello
Sum: 15
15
```

So, for calling this function we can directly put the parenthesis '()' after the curly braces.

Another way of Calling Unnamed Function in C++ :-

```cpp
#include <iostream>
using namespace std;
int main(){
    auto F =  [](){cout << "Hello";};
    F();
}
```

Writing Return Type using Lambda Expression in C++:

By default, the C++ compiler will identify the return type, so we don't have to mention it or if you want then you can mention it.

```
#include <iostream>
using namespace std;
int main(){
    int S = [] (int x, int y) -> int{return x + y;}(10, 5);
    cout<<S;
}
```

Output: 15

Accessing local variables of a function inside the unnamed function:

```
#include <iostream>
using namespace std;
int main(){
    int x = 3;
    int y = 6;
    [x, y] () {cout << x << " " << y;}();
}
```

[x, y] () {cout << ++x << " " << ++y;}; Can we write like this? No. We cannot modify these captured variables. For modifying, we have to use a reference for capturing variables as follows:

[&x, &y] () {cout << ++x << " " << ++y;}; Now we can modify the x and y values. By writing reference, we can modify the local variables inside the lambda expression. The complete example code is given below.

If we want to access all the things in this scope then just write a reference inside the capture list as follows:

[&] () {cout << ++x << " " << ++y;};

```cpp
#include<iostream>
using namespace std;
template<typename t>
void fun(t r){
    r();
}
int main(){
    cout << "Function 1:" << endl;
    [](){cout << "Hello";}();

    cout << endl << "Function 2:" << endl;
    int z = [](int x, int y) {return x+y;} (30, 32);
    cout << "Sum: " << z;

    cout << endl << "Function 3:" << endl;
    int x = 5;
    int y = 6;

    auto F = [&x, &y](){cout << ++x << " " << ++y << endl;};
    cout << "x & y : ";
    F();

    cout << endl << "Function 4:" << endl;
    auto S = [x, y](){cout << "x * y: " << x*y << endl;};
    fun(S);
    return 0;
}

```

Output :-

```
Function 1:
Hello
Function 2:
Sum: 62
Function 3:
x & y : 6 7

Function 4:
x * y: 42
```

Disadvantages of Pointers :-

- Uninitialized Pointers
- The pointer may cause a memory leak
- Dangling Pointers

Uninitialized pointers in C++ :-

```
int *p;
```

if we have declared a pointer then we should not use that pointer unless we have initialized it.

```
*p = 25;
```

This means that we want to store the value '25' wherever the pointer is pointing. But the question here is where the pointer is pointing? In, 'int *p' it is not pointing anywhere. Just we have declared. Then what is the address in 'p' here?

Some default garbage addresses may be in 'p', some random addresses that may belong to a program or may not belong to a problem. So, it is an invalid address as we have not made a pointer to point to some particular location. So first of all, make it point to some location then we can access it. There are 3 methods of doing it.

1st Method :-

```
int x = 10;          int *p = &x;
```

If we have some variable 'x' then, Now the pointer is pointing to this known variable 'x' which is already declared.

2nd Method :-

```
int *p = (int*) 0x5628;
```

We can assign some addresses using some hexadecimal codes but that address has to be type casted as an integer pointer. So, can we directly assign some addresses to a pointer? Yes, if we are sure that the address belongs to the program so this type of initialization is also allowed. **This is not commonly used. This is mostly used in systems programming**

3rd Method :-

```
int *p = new int[5];
```

We can dynamically allocate some memory and assign that to a pointer. If we don't write size and write only 'int' then it will allocate just one integer so either to an existing variable.

Memory Leak :-

This is related to a pointer as well as heap memory. As we have already discussed heap memory, when we are allocating heap memory then when we don't require that, we should deallocate it. If we don't deallocate it then we say that memory is leaked from that total set of memory.

```
int *p = new int[3];                p = NULL;
```

Here we have a pointer and I have allocated heap memory of some size. Then after some time, we don't need this memory. So, we simply say 'p = NULL', then point P will not be pointing on that memory. We should not do this unless we have explicitly deleted the memory.So first of all, say delete 'p' then only, make 'p' as null.

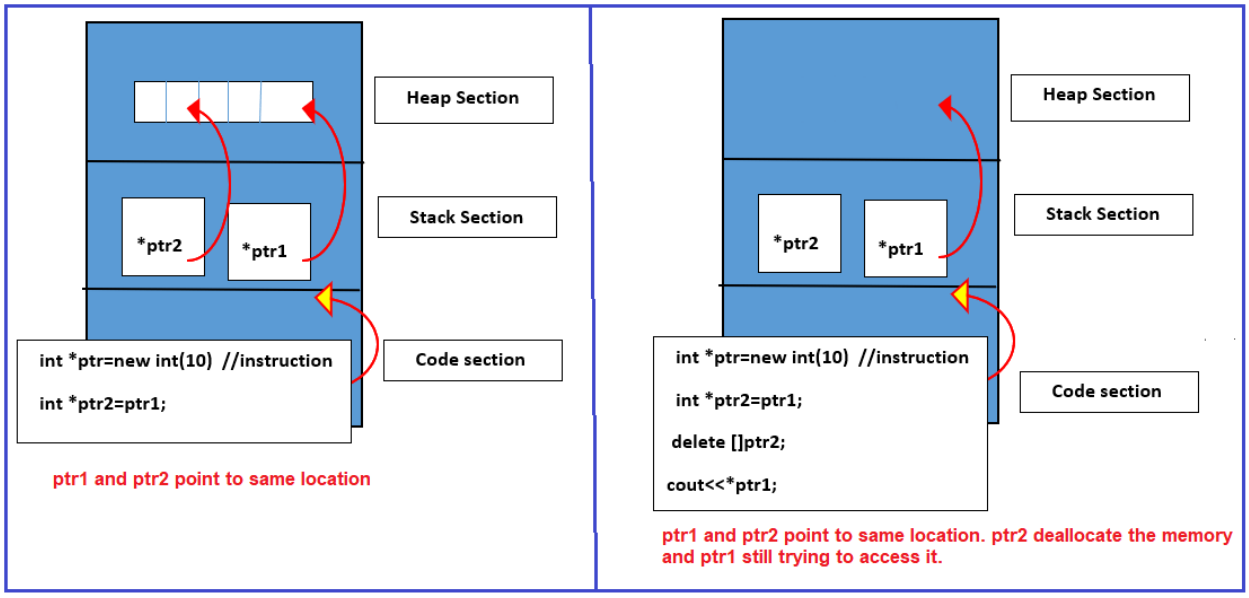
```
delete []p;

p = NULL;
```

Now here is one more thing that we can write 'p = 0' also or write 'p = nullptr'.

In modern C++ it is suggested to use 'nullptr'. You should avoid using null. So back to this memory leak, the conclusion is you must delete the memory when you are not using it before making a pointer to null. Now let us move to the third problem which is the dangling pointer.

Dangling Pointer in C++ :-

So uninitialized pointers mean, the pointer is never initialized, dangling pointer means the pointer was initialized but memory is deallocated. We should avoid these three types of problems while writing programs or developing applications. Actually, these problems are caused due to negligence of beginner programmers. Expert programmers may check all these things thoroughly only before delivering a program or before developing software.




```cpp
#include<iostream>
using namespace std;
int main() {
    int *ptr1=new int(10);
    int *ptr2=ptr1;
    delete []ptr2;
    cout<<*ptr1;
    return 0;
}
```

The improper use of pointers is often at the root of many security problems. When an application behaves in unpredictable ways, it may not seem to be a security issue, at least in terms of unauthorized access.
Pointers can do arithmetic, References can't: **Memory access via pointer arithmetic is fundamentally unsafe and for safeguarding**, Java has a robust security model and disallows pointer arithmetic for this reason.

What are Smart Pointers in C++?

If you are creating anything inside the heap memory then for accessing the heap memory, we need to use pointers. The problem with heap memory is that when you don't need it then you must deallocate the memory. And mostly the programmer shows laziness in writing the code for the deallocation of objects from heap memory which causes severe problems like memory leaks which will cause the program to crash.

```cpp
#include<iostream>
using namespace std;
class Rectangle{
private:
    int Length;
    int Breadth;
public:
    Rectangle(int l, int b){
        Length = l;
        Breadth = b;
    }
    int Area(){
        return Length * Breadth;
    }
};

int Fun(int l, int b){
    Rectangle *p = new Rectangle(l, b);
    int area = p->Area();
    delete p;
    return area;
}
int main(){
    while(1){
        int Result = Fun(10, 20);
        cout << Result << endl;
    }
    return 0;
}
```

Using Smart Pointers in C++:

If we declare the smart pointer then they will automatically deallocate the object when the smart pointer is going out of the scope. Let us show you how we can declare smart pointers in C++.

```
int Fun(int l, int b)
{
        unique_ptr<Rectangle> p(new Rectangle(l, b));
        int area = p->Area();
        return area;
}
```

Example to Understand unique_ptr in C++ :-

If you are using unique_ptr, if an object is created and a pointer is pointing to that object then only one pointer can point to that object. So, we cannot share this object with another pointer. But we can transfer the control from one pointer to another pointer by removing p1. So unique_ptr means upon an object at a time only one pointer will be pointing.

```cpp
#include<iostream>
#include<memory>
using namespace std;
class Rectangle{
private:
    int Length;
    int Breadth;
public:
    Rectangle(int l, int b){
        Length = l;
        Breadth = b;
    }
    int Area(){
        return Length * Breadth;
    }
};

int main(){
    unique_ptr<Rectangle> ptr1(new Rectangle(10,5));
    cout<<ptr1->Area()<<endl;

    unique_ptr<Rectangle> ptr2;
    ptr2=move(ptr1);

    cout<<ptr1->Area()<<endl;
    cout<<ptr2->Area()<<endl;
    return 0;
}
```

Output: 50

Shared_ptr :-

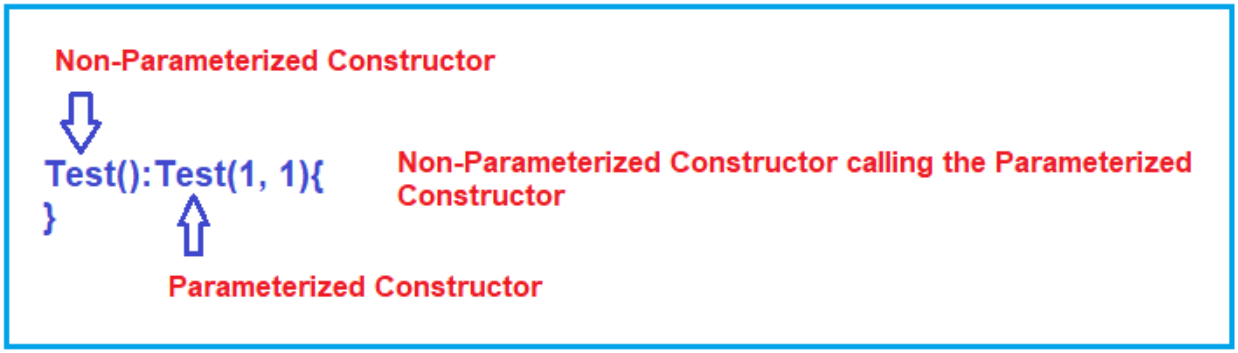
Just like how we have used unique_ptr, the same way we have to use shared_ptr. More than one pointer can point to one object. **This pointer maintains a Ref_count that is a reference counter. Suppose 3 pointers are pointing on a single object the Ref_count will be 3.** So shared means an object can be used by more than one pointer. If we remove one pointer then Ref_count will be reduced by 1. We can know the value of Ref_count by using the use_count() function.

```cpp
#include<iostream>
#include<memory>
using namespace std;
class Rectangle{
private:
    int Length;
    int Breadth;
public:
    Rectangle(int l, int b){
        Length = l;
        Breadth = b;
    }
    int Area(){
        return Length * Breadth;
    }
};

int main(){
    shared_ptr <Rectangle> ptr1 (new Rectangle(10, 5));
    cout << ptr1->Area() << endl;

    shared_ptr <Rectangle> ptr2;
    ptr2 = ptr1;

    cout << "ptr1 " << ptr1->Area() << endl;
    cout << "ptr1 " << ptr2->Area() << endl;
    cout << ptr1.use_count() << endl;
}
//Output
50
ptr1 50
ptr1 50
2
```

Weak_ptr :-

It is also the same as shared_ptr. Here also more than one pointer can point to a single object. But it will not maintain Ref_count. So that's why it is known as weak_ptr. So, the pointer will not have a strong hold on the object. The reason is if the pointers are holding the object and requesting other objects, they may form a deadlock between the pointers. So, to avoid deadlock, weak_ptr is useful. So, it doesn't have Ref_count so it is more like unique_ptr but it allows the pointer to share an object, so it is more like shared_ptr. It is in between unique and shared which is not strict. It doesn't bother how many pointers are pointing at an object.

InClass Initializer and Delegation of Constructors :-

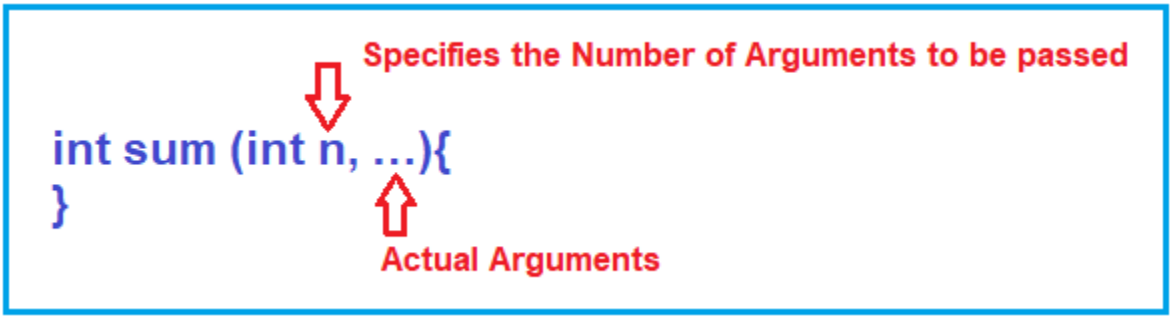


```cpp
#include<iostream>
using namespace std;
class Test{
private:
    int x = 15, y = 30, z;
public:
    Test(int a, int b){
        x = a;
        y = b;
    }
    //using constructor delegation
    Test():Test(35, 75){
        z = 10;
    }
    void Display(){
        cout<<"x : "<<x<<", y : "<<y<<", z : "<<z;
    }
};
int main(){
    Test obj;
    obj.Display();
}
```

Ellipsis in C++:

Ellipsis is used for taking a variable number of arguments in a function. For example, if we want to write a function for finding the sum of elements or sum of integers then we want our function to work with a different number of parameters like,

Sum (10, 20) = 30 Sum (30, 54, 23) = 107



How to access Elements using Ellipsis in C++?

For accessing the elements, there is a class available in C++ that is va_list. For a better understanding, please have a look at the following code. What are the important instructions in this function?

va_list(), va_start(), va_arg() and va_end().

```
int Sum(int n, ...){
        va_list list;
        va_start(list, n);
        int s = 0;

        for(int i = 0; i < n; i++)
                s += va_arg(list, int);

        va_end(list);
        return s;
}
```

```cpp
#include<iostream>
#include<cstdarg>
using namespace std;
int sum (int n, ...){
    va_list list;
    va_start (list, n);
    int x;
    int s = 0;
    for (int i = 0; i < n; i++){
        x = va_arg (list, int);
        s += x;
    }
    return s;
}
int main(){
    cout << sum (3, 12, 24, 36) << endl;
    cout << sum (7, 13, 26, 39, 52, 65, 78, 81) << endl;
}
//Output
72
354
```