OOPS Concepts in C++

OOPs Concept in Brief

PrepSter Tip: Consider Classes and Objects as the heart of any object-oriented language.

Following are main features of OOPS Concept in C++

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

Other Features of OOPS in C++ –

- Dynamic Binding
- Message Passing

C++ Classes and Objects

MoreA C++ class is like a blueprint for an object.

Its a collection of objects which have similar characteristics

Why Classes?

Programmers long demanded a feature in the programming language that allowed creating an interface with which they can define some common properties and methods that can be used by instances of that interface.

Let’s take an example to understand this –

Car as a class/interface, all cars in the world share some common properties –

- colour
- tyres
- mileage
- weight
- speed-limit etc

Similarly, all cars in world have common functions like – start(), brake(), stop(), reverse() etc.

With classes, we can create an interface of such common properties and methods and create thousands of instances i.e. objects to share them.

ObjectsObjects are instances of a class.
Whenever class is defined non memory is created. It is when objects of that class are created memory is allocated.

Another Example
All students in school have some common properties –

- Roll No
- DOB
- Weight
- Name
- Email ID

We can create a **class with the name student** to have all these properties defined.

If there are 1000 students in a school, we create 1000 objects unique to each student to store their details

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

// name of the class
class Student
{
   // access specifier
   public:

   // data members variables
   int rollNo, weight, age;
   string name;

   // member functions
   void displayDetails()
```

```
    {
        cout << "Roll No: " << rollNo << endl;
        cout << "Name: " << name << endl;
    }
};

int main(){

    // declare Student object s1
    Student s1;

    // assigning values
    s1.rollNo = 1;
    s1.weight = 80;
    s1.age = 21;
    s1.name = "Peter";

    // calling function for s1 object
    s1.displayDetails();

    return 0;
}
```

Some pointers to remember:

- A Class is a user defined data-type which consists of data members and member functions.
- Data members are the variables and member functions are the functions used to manipulate these variables and together these data members and member functions defines the properties and behavior of the objects in a Class.
- In the above example of class *Student*, the data member will be *rollNo*, *DOB*and *weight* .
- Member Functions will be : *displayDetails();*

Defining Member Functions

Member functions can be declared in two ways –

- Inside the class (show in previous example)
- Outside the class (Using scope resolutor)

Following is the template if you want to declare member function outside class

```
return_type class_name::functionName(arguments){

    // function working here

}
```

Note, even though the function is defined outside, it must be declared inside the class as –

```
functionName(arguments);
```

Class vs Object in C++ (Difference Between)

Class

The design concept of class is similar to that of structures and unions

- **Apt Definition of a class**: A class is a logical entity where you can write logic (statements)
- No memory gets allocated for a class

Class Can be created as follows

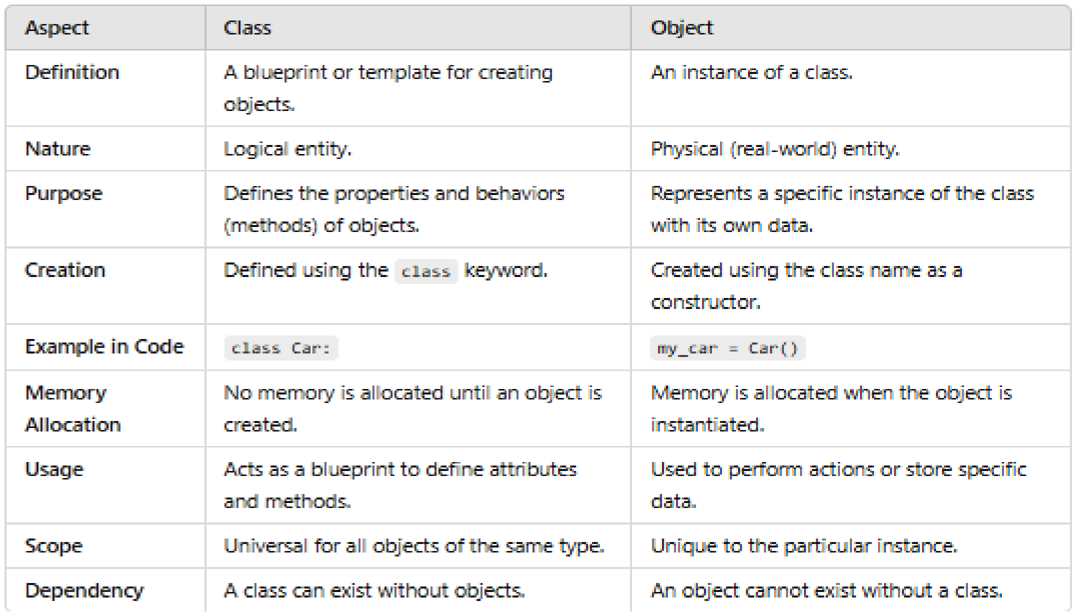
```
class name_of_class{

    // access specifiers
    public:

    // data members and member functions
    int variable1;

    void functioName1(){
        // function definition here
    }
}
```

Object

An object is a physical entity that represents memory for a class

- **Definition of an object**: The object is an instance of a class it holds the amount of memory required for the Logic present in the class

Hence you call an ***object an instance of a class or a real-world entity***



Aspect	Class	Object
Definition	A blueprint or template for creating objects.	An instance of a class.
Nature	Logical entity.	Physical (real-world) entity.
Purpose	Defines the properties and behaviors (methods) of objects.	Represents a specific instance of the class with its own data.
Creation	Defined using the `class` keyword.	Created using the class name as a constructor.
Example in Code	`class Car:`	`my_car = Car()`
Memory Allocation	No memory is allocated until an object is created.	Memory is allocated when the object is instantiated.
Usage	Acts as a blueprint to define attributes and methods.	Used to perform actions or store specific data.
Scope	Universal for all objects of the same type.	Unique to the particular instance.
Dependency	A class can exist without objects.	An object cannot exist without a class.

Encapsulation in C++

Encapsulation in C++ Language

Encapsulation is, however, different. You wrap and bind all related data together in a single form. Sort of like capsuling / binding it together.

Example:

- In a Car the engines, cooling, battery, breaking, clutching system are hosted together in the front of the car. Similarly, the emission systems are clubbed together and are at the end of the car.
- Imagine if the car fails and the engine, battery system, clutching system and brake system etc are scattered all over the car (Rather than being hosted together in the front).The mechanic will face a hard time figuring out what is wrong and will have to open the whole car open and it will take a lot more time for him to figure out what's wrong.
- Similarly, for a coder to give code more understandable and readable format he groups and encapsulates all the related data together inside a single unit.

Note: People sometimes get highly confused between data abstraction and Encapsulation. Both of them are different .

Data Abstraction: Is essentially hiding the implementation details and only giving methods to interact with the system without knowing what exactly is happening in the backend code.

Why do people get confused between Encapsulation and Data Abstraction?
With the help of encapsulation in a way we also achieve data abstraction, which is why people think that they are one and the same thing.

- Yes, while doing encapsulation i.e. grouping into single units.
- We simultaneously use access modifiers which directly help in hiding the data and managing its good architecture at the same time.

How Encapsulation is achieved in C++
Encapsulation is a state of binding/wrapping related data or code in one place.

In C++, we achieve this with classes by bundling all data members and member functions within a single class as shown below –

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

class Rect {
  public:
    int len;
    int breadth;

    int calculateArea() {
      return len * breadth;
    }
};

int main(){

    Rect obj;

    obj.len = 10;
    obj.breadth = 20;

    cout << "Area: " << obj.calculateArea();
}
```

Issues with Above Program & How Data Abstraction Solves it
We are able to directly access the variables length and breadth. Imagine if they were for User class and variables were username and password.

We need to create a protective interface, this is where data abstraction helps us.

We use access specifiers in C++ to achieve Data Abstraction.

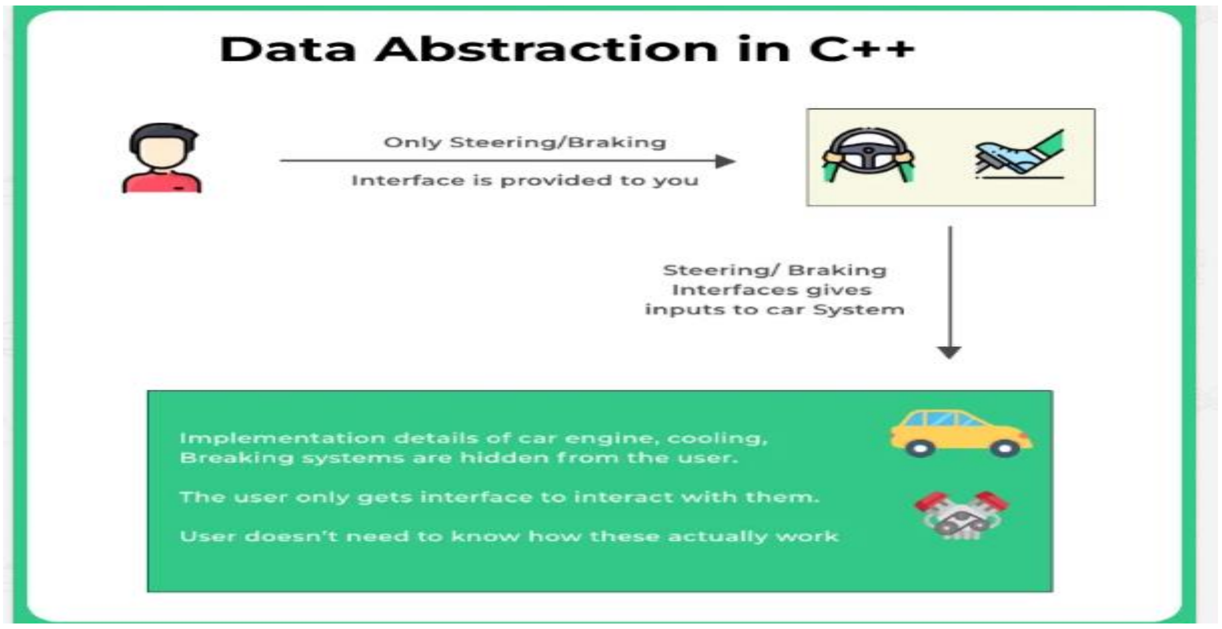
Data Abstraction in C++

What is Data Abstraction?It is essentially hiding the implementation details and only giving methods to interact with the system without knowing what exactly is happening in the backend code.

Example of Abstraction –

- When you're taking a ride in a car you only need to know that pressing on the accelerator increases the speed of the car, or pressing on the breaks slows down the car.
- But, you don't need to know the inside details of the machinery or engine details to know how all of this happens. It is taken care of by the Car manufacturer itself. They just provide you with the interface to drive the car.

Encapsulation is, however, different it is wrapping up and binging all the related data together. For a car having all the engine-related components together and breaking or clutching system together.


Data Abstraction in C++
Only Steering/Braking
Interface is provided to you
Steering/ Braking Interfaces gives inputs to car System
Implementation details of car engine, cooling, Breaking systems are hidden from the user.
The user only gets interface to interact with them.
User doesn't need to know how these actually work

Types of Abstraction

Now, there are two types of Abstraction –

- Header files

- Classes

Lets have a look at both of them below –

Header Files –
We all use header files, we import #include to use the power function. We directly use the pow() function as pow(2, 3) to get results.

But, the implementation details are hidden from us, we just get the desired output without knowing what happened in the background.

Classes –
In C++, we can define which data members & member function implementation we want to show to the outside world and which ones we want to hide.

This is done using access specifiers and getters/setters.

How do Access Specifiers help in Data Abstraction?
There are three types of access specifiers in C++

- Public
- Private
- Protected

Let's have a look at the program below. (There is some issue with the program below, that is **data abstraction has not been implemented**)

Issue with Below Program The issue with the below program is that data members can be accessed directly in the main function without any access control.

Imagine if this was your password, anybody can change, read or alter password

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

class myClass {
  // data members and functions declared public
  public:

    int x, y;

    // constructor to setup initial value at object creation
    // we will learn about constructors later
    myClass(int val1, int val2){
        x = val1;
        y = val2;
    }

};

int main(){
    myClass obj(10, 15);

    // we are directly able to access values
    obj.x = 100;
    obj.y = 200;

    cout << "x: " << obj.x << " y: " << obj.y << endl;

    return 0;
}
```

How Data Abstraction solves this?

Data abstraction solves this by –

- Making relevant data members private
- Creating function interfaces designed to do specific things allowed the coder to work with these data members

Difference between Data Abstraction and Data Encapsulation

90% of the people forget both data abstraction and encapsulation in the interviews as both of them are more or less same. But, infact they are different.

- Data Encapsulation is the process of hiding the data and programs from the outside world and essentially capsuling them together in one entity.
- Data Abstraction is the process of hiding the implementation i.e. what is happening inside a function or program from the outside world. Essentially data abstraction is achieved with the help of data encapsulation or we can say abstraction is the result of encapsulation.

1. Private members can only be access within the class and their data and implementation is hidden from the outside world

2. Public members can be accessed anywhere in the program.

Access Specifiers in C++

Access Specifiers in C++ Language

The Access Specifiers help to achieve Data Hiding in C++.Basically, access specifiers are used to hide/show data from outside world, it may cause a particle class member to not to be accessible by outside scope functions.

Types of Access Modifiers

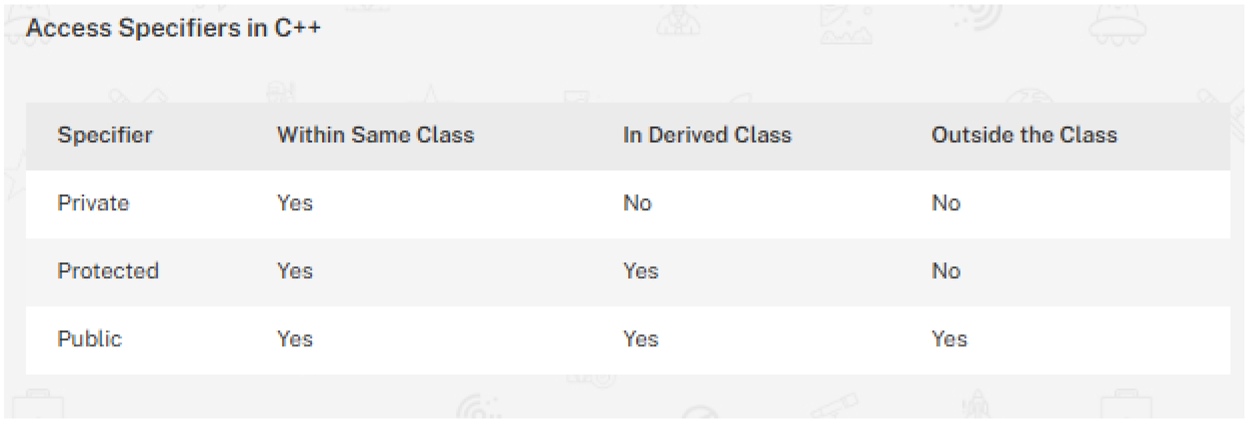
C++ has three different types of access modifiers –

1. Public
2. Private
3. Protected

PrepSter Tip – If you don't specify any keyword then, by default in C++ the class members will take private behaviour.

Example program –

```
class Rectangle{

   private:
      int length, breadth;

   public:
      int variable1, variable2;
      void getValues();
      int area();
};
```


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

1. Public Access Specifier

If any class member (data members/member functions) is declared under public then –

1. Are accessible to everyone.
2. Using the dot(.) function they can be accessed

An example of this is show below –

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

class Rectangle{

   // all below declared under public
  public:
   int length, breadth;

   int area(){
       return length * breadth;
   }
};

int main(){
   Rectangle rect1;

   // can be accessed outside the class
   rect1.length = 10;
   rect1.breadth = 20;

   cout << "Area is: " <<   rect1.area();
   return 0;
}
```

2. Private Access Specifier

If any class member is declared under private then –

1. Can only be accessed by functions declared inside the class
2. Not accessible to the outside world(class)
3. Only the member functions or the friend functions are allowed to access the private data members of a class.

An example of this is shown below –

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

class Rectangle{

  // these are a private
  private:
    int length, breadth;

  // these are public
  public:

    // area is a member function of the class thus
    // can access private data members
    int area(){
        return length * breadth;
    }
};

int main()
{
    Rectangle rect1;

    // Error
    // trying to access private member outside class
    rect1.length = 10;
    rect1.breadth = 20;

    // public member can be accessed
    cout << "Area is:" << rect1.area();

    return 0;
}
```

We can solve the above setting/fetching value issue in the following ways –

- Creating getter/setters
- Initializing values via constructors
- Setting values via public function

We will do this via setting values via public function below, however this can also be done via other methods too.

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

class Rectangle{

  // these are a private
  private:
    int length, breadth;

  // these are public
  public:

    // area is a member function of the class thus
    // can access private data members
    int area(int l, int b){

        //setting values here
        length = l;
        breadth = b;

        // return result after setting values
        return length * breadth;
    }
};

int main()
{
    Rectangle rect1;

    // public member can be accessed
    // also can be used to set values
    cout << "Area is: " << rect1.area(10, 20);

    return 0;
}
```

Protected Access Specifier

If any class members are declared under-protected then –

1. It is the same as private accessibility
2. But, derived(subclasses/child) classes can also access the protected class members.
3. These protected members are inaccessible outside. But, are accessible in the derived class as protected data members

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

// Base(parent/super) class
class Rectangle
{
  // protected data members
  protected:
    int height;
};

// Derived(child/sub) class
class Square : public Rectangle
{
  public:
    void setHeight(int h){

        // Child class is able to access
        // the inherited protected data members (height)
        // of base class
        height = h;
    }

    void displayHeight(){
        cout << "Height is: " << height << endl;
    }
};

int main() {

    Square square1;

    // member function of derived class can
    // access the protected data members of base class

    square1.setHeight(10);
    square1.displayHeight();

    return 0;
}
```

Summary

Public

Public Access Specifiers allow access –

- Within the same class
- Within the derived class
- Outside the class

Protected

Protected Access Specifiers allow access –

- Within the same class
- Within derived class as protected members

Protected Access Specifier don't allow –

- Outside the class

Private

Private Access Specifiers allow access –

- Within the same class

Protected Access Specifier don't allow –

- Within the derived class
- Outside the class

Implicit type conversion in C++

Type conversion is done in two ways in C++ one is explicit type conversion and the second is implicit type conversion. In simple words, explicit type conversion is done by the user hence known as user-defined type conversion, and implicit type conversion is done by compiler itself hence known as automatic type conversion. In this article, we will learn more about implicit type conversion in C++

Implicit data type conversion in C++

Implicit type conversion is a process that is done by the compiler itself without any human effort i.e. no external human trigger is required for the process of converting a variable of one data type to another. It is also known as automatic type conversion. If an expression contains variables of different data types so to avoid data loss the variable of smaller data type(low priority data type) is converted to the bigger data type (high priority data type).

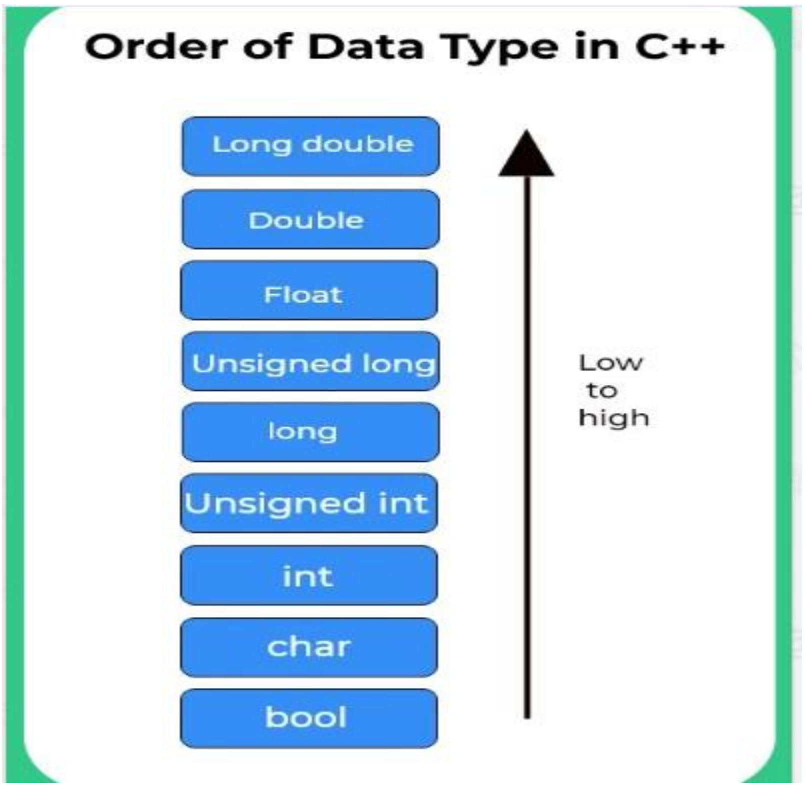
For example:

```
int a=5;
short b=2;
int c= a+b;
```

Here in integer type variable ‘c’ addition of ‘a’, an integer type variable and ‘b’, a short type variable is stored. So the compiler will implicitly convert the low priority data type i.e. short to higher priority data type i.e. integer.

Order of data type for type conversion

As we know that compiler converted variable from low priority data type to high priority data type in implicit conversion. To understand this below is a priority set for the data type starting from lowest priority data type and moving to high priority data type.



C++ program to show implicit data type conversion

```
#include <iostream>
using namespace std;
int main()
{
 int i = 10;
 char c = 'a';
 float f=1.1;
 f = f + i ;
 //i get converted to float type from integer type.

 i = i + c;
 //c get converted to int type from char type i.e. to equivalent ASCII value of 'a' i.e. 97

 cout<<"Value of i = "<< i << endl;
 cout<<"Value of c = "<< c << endl;
 cout<<"Value of f = "<< f << endl;
 return 0;
}
```

Constructor and Destructor in C++

Constructor in C++

What are Constructors Constructors are a unique class functions that do the job of initialising every object. Whenever, any object is created the constructor is called by the compiler.

- When we create an object and don't assign values the data of that object take up garbage values.
- To make sure that this doesn't happen we use constructors to assign values to them as soon as they are created or memory is assigned.

The new version of C++ automatically calls a default constructor implicitly to assign a default value in cases when we don't create a constructor ourselves.

Note: Constructors do not have a return type.

Syntax –

A constructor can be declared –

1. Inside the class
2. Outside the class (using scope resolution :: operator)

More information

1. Constructors have the same name as class name
2. They are called using classname and (); example myClass();
3. They can also be defined(initialised) outside class by using the class name and scope resolution :: operator.

Constructors in C++

Constructors in C++ Language

Syntax

Defining Internally to the class

```
class Demo {
  char c;
  public:
    Demo(){
      // Constructor declaration here
      // User defined assignments for initialisation
  }
};
```

Defining Externally to the class

```
class Demo
{
 char c;
 public:
   Demo();
     //Constructor Declared here
};

Demo::Demo()
// Constructor definition here
{
    // initialisation of values by the user
    char = "PrepInsta";
}
```

Why Constructors are used?

- When an object is created, the memory is assigned the object entities and some garbage values are assigned to those.
- These garbage values may cause hinderance in the programs, as it may be needed to only assign very specific values to entities/data.
- Constructors do not have any return argument.
- They have the same name as the name of the class and are enclosed within the class
- C++ has its own version of constructor which assigns some garbage or random values to object entities.

Types of Constructors –

The following are the types of constructors in C++ –

- Default
- Parameterized

- Copy

Default Constructors

- **Highlights –** No Passing Arguments in the constructor.

For Example –

```
#include <iostream>
using namespace std;
class Demo
{
   public:
        int i;
        Demo(){
            i=8000;
          }
};

int main()
{
  Demo  demo;
  cout << demo.i;

return 0;
}
```

Parameterised Constructors

- **Highlights –** Arguments are passed with the constructor.

For example –

```
#include <iostream>
using namespace std;
class Demo
{
  public:
    int i;
    Demo(int val)
    {
       i=val;
    }
};
int main()
```

```
{
   //setting parameterised values
   Demo demo(6000);
   Demo demo2(12000);
   cout << demo.i<<endl;;
   cout << demo2.i;

return 0;
}
```

Copy Constructors

- **Highlights –** Arguments are passed as objects and object is copied as instantiated as a new object.
- The object can be passed in multiple ways
    - Demo d2 = d1;
    - Demo d3(d1);
- While the following doesn't call the copy constructor but only assigns the value with the help of assignment operator
    - Demo d4;
    - d4 = d1

For example –

```
#include <iostream>
using namespace std;
class Demo
{
  private:
   int a, b;
   public:
   Demo()
   { }
   Demo(int a1, int b1) {
    a = a1;
    b = b1;
   }
   // Copy constructor example here
   Demo(const Demo &d2)
```

```
    {
        a = d2.a;
        b = d2.b;
   }

    int getA()
    {
        return a;

    }
    int getB()
    {
        return b;

    }
};

int main()
{
    Demo d1(5000, 6000);

    // We have called a copy constructor
    Demo d2 = d1;

    // We have called a copy constructor
    Demo d3(d1);
    Demo d4;

    // Assignment operation happens there is no call to copy constructor
    d4 = d1;

    // displaying values for both constructors
    cout << "d1.a = " << d1.getA() << ", d1.b = " << d1.getB();
    cout << "\nd2.a = " << d2.getA() << ", d2.b = " << d2.getB();

    cout << "\nd3.a = " << d3.getA() << ", d3.b = " << d3.getB();
    cout << "\nd4.a = " << d4.getA() << ", d4.b = " << d4.getB();
    return 0;
}
```

Types of Constructors in C++

Types of Constructors in C++

- Default Constructor

- Parameterized Constructor
- Copy Constructor
    - Shallow Copy
    - Deep Copy

<u>Difference between constructor and member function</u>

- Constructor name must be the same as class name but functions cannot have the same name as the class name.
- Constructors do not have a return type whereas functions must have a return type.
- Constructors are automatically called when an object is created.
- A member function can be virtual, but there is no concept of virtual constructors.
- Constructors are invoked at the time of object creation automatically and cannot be called explicitly using class objects.

1. Default Constructor in C++

- The default constructor is the constructor which doesn't take any argument. It has no parameters.
- In this case, as soon as the object is created the constructor is called which initializes its data members.
- A default constructor is so important for the initialization of object members, that even if we do not define a constructor explicitly, the compiler will provide a default constructor implicitly

Sample Code

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

class construct {
    public:
    int a, b;

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

int main()
{
    construct c;
    int sum = c.a + c.b;

    cout << "a : " << c.a << endl;
    cout << "b : " << c.b << endl;
    cout << "sum : " << sum << endl;

    return 0;
}
```

2. Parameterized Constructor in C++

- Arguments can be passed to the parameterised constructors.
- These arguments help initialize an object when it is created.
- To create a parameterized constructor, simply add parameters to it the way you would to any other function.
- When you define the constructor's body, use the parameters to initialize the object.

We can also have more than one constructor in a class and that concept is called constructor overloading.

Uses of Parameterized constructor:

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

Sample Code

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

class PrepInsta {
    private:
```

```
    int a, b;

    public:

    PrepInsta(int a1, int b1)
    {
        a = a1;
        b = b1;
    }

    int getA()
    {
        return a;
    }

    int getB()
    {
        return b;
    }
};

int main()
{
  PrepInsta obj1(10, 15);

  cout << "a = " << obj1.getA() << ", b = " << obj1.getB();

  return 0;
}
```

3. Copy Constructor in C++

- A copy constructor is a member function which initializes an object using another object of the same class.
- Whenever we define one or more non-default constructors(with parameters) for a class, a default constructor(without parameters) should also be explicitly defined as the compiler will not provide a default constructor in this case.
- An object can be initialized with another object of same type. This is same as copying the contents of a class to another class.
- For a better understanding of Copy Constructor in C++ (click here).

What is a Copy Constructor?

DefinitionThese are the special type of Constructors that takes an object as an argument and is used to copy values of data members of one object into another object.

```
class_name (class-name &){

....

}
```

- Copy Constructor is a type of constructor which is used to create a copy of an already existing object of a class type.
- It is usually of the form X (X&), where X is the class name.
- The compiler provides a default Copy Constructor to all the classes.

Code

```
#include <iostream>
using namespace std;
class PrepInsta
{
private:
    int x, y;
public:
    PrepInsta()
    { // empty default constuctor
    }

    PrepInsta(int x1, int y1)
    {
        x = x1;
        y = y1;

        cout << "Parameterized constructor called here" << endl;

    }

    // User defined Copy constructor
    PrepInsta(const PrepInsta &p2)
    {
        x = p2.x;
        y = p2.y;
        cout << "Copy constructor called here" << endl;
    }

    int getX()
    {
        return x;

    }
    int getY()
    {
        return y;

    }
};

int main()
{
    // Trying to call parameterized constructor here
    PrepInsta p1(10, 15);

    // Trying to call copy constructor here
    PrepInsta p2 = p1;

    // Trying to call Copy constructor here (Another way of doing so)
    PrepInsta p3(p1);
    PrepInsta p4;

    // Here there is no copy constructor called only assignment operator happens
    p4 = p1;

    cout << "\nFor p4 no copy constructor called only assignment operation happens\n" << endl;

    // displaying values for both constructors
    cout << "p1.x = " << p1.getX() << ", p1.y = " << p1.getY();
    cout << "\np2.x = " << p2.getX() << ", p2.y = " << p2.getY();

    cout << "\np3.x = " << p3.getX() << ", p3.y = " << p3.getY();
    cout << "\np4.x = " << p4.getX() << ", p4.y = " << p4.getY();

    return 0;
}
```

Very Important**In the above example there are two ways to call copy constructor –**

PrepInsta p2 = p1;
PrepInsta p3(p1);

However, PrepInsta p4; followed by p4 = p1; doesn't call copy constructor, its just simply and assignment operatorTypes of Copy Constructors
It is really great way of creating new object initialisations and is still widely used by programmers.

There are two types of copy constructors which are –

- Default Copy Constructors (Does Shallow Copying)
- User Defined Copy Constructors (Does Deep Copying)

We will learn more about these two in detail below –

Shallow Copy
When we do not create our own copy constructor. **C++ compiler will create a default copy constructor** for each class which does a member-wise copy between objects.

This default copy constructor does shallow copy.

NoteThis only happens in heap memory variables example pointers

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

class PrepInsta
{
   char *prepString;

   public:

   // Note : No user defined Copy constuctor created here

   PrepInsta(const char *prepString1)
   {
      //We are invoiking Dynamic memory allocation here
      prepString = new char[16];
      strcpy(prepString, prepString1);
```

```
    }

    /* concatenate method */
    void concatenate(const char *prepString1)
    {
        // using function to Concatenating two strings
        strcat(prepString, prepString1);
    }

    void print()
    {
        cout << prepString << endl;
    }
};

/* main function */
int main()
{
    PrepInsta obj1("Prep");

    //Copy constructor
    PrepInsta obj2 = obj1;

    cout << "Before concatenation - " << endl;

    obj1.print();
    obj2.print();

    // obj1 is invoking concatenate()
    // but change will also be reflected in obj2 (shallow copy)
    // see more explanation in below comments
    obj1.concatenate("Insta");

    cout << "\nAfter concatenation - " << endl;

    // as we didn't create any user defined copy constructor
    // compiler created default copy constuctor in backend on its own
    // So both obj1 & obj2 variables share same memory (shallow copy)
    // any change in obj1 also reflects in obj2
    obj1.print();
    obj2.print();


    return 0;
}
```
Any change in obj1 also reflects in obj2 (shallow copy)

Deep Copy
When we create our own user-defined copy constructor. **The individual copies of memory are created for each object** and thus there is no sharing of memory.

This user-defined copy constructor does Deep copy.

NoteThis only happens in heap memory variables example pointers

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

class PrepInsta
{
    char *prepString;

    public:
    PrepInsta (const char *str)
    {
        //We are invoking Dynamic memory allocation here
        prepString = new char[16];
        strcpy(prepString, str);
   }

   // here we are doing the change of adding additional code
   // for user defined copy constructor (Deep Copy)
    PrepInsta (const PrepInsta &x2)
    {
        //We are invoking Dynamic memory allocation here
        prepString = new char[16];
        strcpy(prepString, x2.prepString);
    }

    void concatenate(const char *prepString1)
    {
        // using function to Concatenating two strings
        strcat(prepString, prepString1);
    }

   ~PrepInsta()
    {
        delete [] prepString;
    }

   void print()
   {
        cout << prepString << endl;
   }
};

/* main function */
int main()
{
    PrepInsta p1("Prep");
    PrepInsta p2 = p1; //copy constructor

   cout << "Before concatenation - " << endl;

    p1.print();
    p2.print();

    p1.concatenate("Insta");

    cout << "\nAfter concatenation - " << endl;

    p1.print();
    p2.print();

    return 0;
}
```
Changing obj1 doesn't reflect/cause any changes in obj2(Deep Copy)

Parameterized constructor in C++

DefinitionA constructor is a special member function whose task is to initialize the objects of its class. It is special because its name is same as the class name. The constructor is invoked whenever an object of its associated class is created. A constructor constructs the values of data members of the class hence it is called as constructor.

Benefits of parameterized constructor

- Parameterized constructor helps in initializing each data member of the class to different values.
- Parameterized constructor is helpful for constructor overloading.

Parameterized Constructor (Inside Class)

Syntax
```
class class_name
{
    public: class_name(variables) //Parameterized constructor declared
    {
       // initialization of variables
    }
};
```

Default constructor in C++

There are three types of constructor in C++

- Default constructor
- Parameterized constructor
- Copy constructor

Default constructor in C++

The default constructor is a constructor that has no parameters, and even if it has parameters then all the parameters have default values. If we had not defined any constructor in the program then the compiler automatically creates one default constructor during the time of object call. This constructor which is created by the compiler implicitly does not have any parameters. As there are no arguments in the default constructor it is also known as "**zero-argument constructor**". The default constructor can be declared both inside and outside the class.

Syntax for declaring default constructor inside the class

```
class class_name
{
    public:
    class_name(); //default constructor declared.
};
```

Syntax for declaring default construct outside the class

```
class class_name
{

};
class_name :: class_name() //default constructor declared.
{

}
```

Example 1 : Program to demonstrate default constructor

```
#include <iostream>
using namespace std;
class PrepInsta
{
public:
  PrepInsta() //Default constructor.
  {
    cout<<"Learning is fun with PrepInsta"<<endl;
  }
};
int main()
{

  PrepInsta obj;//Creating object to invoke default constructor.

  return 0;
}
```

Destructors in C++

Destructors in C++ are called when –

- The object goes out of scope of execution
- Are automatically called by compiler if are not manually defined by the user

Destructors are special functions of a class

- Have the same name as the name of the class
- Are preceded with~ for example for class named as Demo the destructor will be ~Demo()
- No argument passing or copying happens unlike constructors where we can parameterize or copy

Syntax

```
class Demo
{
 public:
 ~Demo();
};
```

Example of a Destructor

```
#include <iostream>
using namespace std;
class Demo
{
 public:
 //Initialising the Constructor Destructor
 Demo()
 {
  cout << "Object has been Constructed\n";
 }
 ~Demo()
 {
  cout << "We have Destroyed the object\n";
 }
};
int main()
{
 Demo demo;
 // Constructor Called
 int temp=1;
 if(temp)
 {
  // Constructor calling happens here
  Demo prepinsta;
 }
 // The scope of the object initialised prepinsta has ended here so the destructor will be called
automatically by the system.
}
// Destructor called for Demo as demo object's out of scope
```

Copy Constructor in C++

What is a Copy Constructor?

DefinitionThese are the special type of Constructors that takes an object as an argument and is used to copy values of data members of one object into another object.

```
class_name (class-name &){

     ....

}
```

- Copy Constructor is a type of constructor which is used to create a copy of an already existing object of a class type.
- It is usually of the form X (X&), where X is the class name.
- The compiler provides a default Copy Constructor to all the classes.

Code

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

```
{
private:
    int x, y;
public:
    PrepInsta()
    { // empty default constuctor
    }

    PrepInsta(int x1, int y1)
    {
        x = x1;
        y = y1;

        cout << "Parameterized constructor called here" << endl;

    }

    // User defined Copy constructor
    PrepInsta(const PrepInsta &p2)
    {
        x = p2.x;
        y = p2.y;
        cout << "Copy constructor called here" << endl;
    }

    int getX()
    {
        return x;

    }
    int getY()
    {
        return y;

    }
};

int main()
{
    // Trying to call parameterized constructor here
    PrepInsta p1(10, 15);

    // Trying to call copy constructor here
    PrepInsta p2 = p1;

    // Trying to call Copy constructor here (Another way of doing so)
    PrepInsta p3(p1);
    PrepInsta p4;

    // Here there is no copy constructor called only assignment operator happens
    p4 = p1;

    cout << "\nFor p4 no copy constructor called only assignment operation happens\n" << endl;

    // displaying values for both constructors
    cout << "p1.x = " << p1.getX() << ", p1.y = " << p1.getY();
    cout << "\np2.x = " << p2.getX() << ", p2.y = " << p2.getY();

    cout << "\np3.x = " << p3.getX() << ", p3.y = " << p3.getY();
    cout << "\np4.x = " << p4.getX() << ", p4.y = " << p4.getY();

    return 0;
}
```

Very Important In the above example there are two ways to call copy constructor –

PrepInsta p2 = p1;
PrepInsta p3(p1);

However, PrepInsta p4; followed by p4 = p1; doesn't call copy constructor, its just simply and assignment operator

Types of Copy Constructors

It is really great way of creating new object initialisations and is still widely used by programmers.

There are two types of copy constructors which are –

- Default Copy Constructors (Does Shallow Copying)
- User Defined Copy Constructors (Does Deep Copying)

We will learn more about these two in detail below –

Shallow Copy

When we do not create our own copy constructor. **C++ compiler will create a default copy constructor** for each class which does a member-wise copy between objects.

This default copy constructor does shallow copy.

NoteThis only happens in heap memory variables example pointers

Any change in obj1 also reflects in obj2 (shallow copy)

Deep Copy
When we create our own user-defined copy constructor. **The individual copies of memory are created for each object** and thus there is no sharing of memory.

This user-defined copy constructor does Deep copy.

NoteThis only happens in heap memory variables example pointers

Changing obj1 doesn't reflect/cause any changes in obj2(Deep Copy)

Conversion Constructor in C++

Conversion Constructor in C++ Language
The compiler uses conversion constructor to convert objects from the type of the single parameter to the type of the conversion constructor's class.

Usage of conversion constructor

1. In return value of a function
2. As a parameter to a function

New way of calling Constructor
Usual way of calling a constructor is ClassName(variables).
For example: PrepInsta(int x) or PrepInsta(int x, int y)

Using conversion constructor we can call constructor by using assignment operator (=).

For Example –
1. ClassName obj = val (PrepInsta obj = 10)
2. ClassName obj = {val1, val2 ..} (PrepInsta obj = {10, 20})

Note: On many websites its given incorrectly that Constructor needs to have only 1 arguments, there are interesting ways in which we can call conversion constructor implicitly, even though it may have 0, 1 or multiple arguments

Single Value Conversion Constructor
```
#include <iostream>
using namespace std;

// base class
```

```
class Animal {

   public:
    void jump() {
        cout << "I can jump!" << endl;
    }

    void sit() {
        cout << "I can sit!" << endl;
    }
};

// derived class
class Dog : public Animal {

   public:
    void bark() {
        cout << "I can bark! Woof woof!!" << endl;
    }
};

int main() {
    // Create object of the Dog class
    Dog doggy;

    // Calling members of the base class
    doggy.jump();
    doggy.sit();

    // Calling member of the derived class
    doggy.bark();

    return 0;
}
```

Multiple Parameters Conversion Constructor

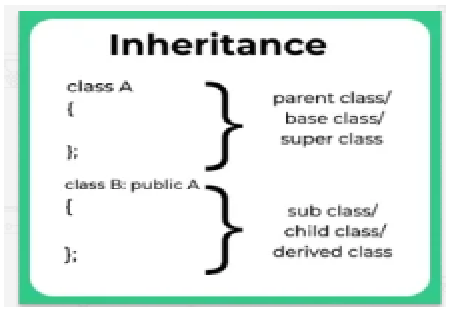
```
#include <iostream>
using namespace std;

class PrepInsta
{
    int x, y;
    public:
    void display(){
        cout << "The values are: " << x << " and " << y << endl;
    }
    // parameterized constructor
    PrepInsta(int a, int b){
        x = a;
        y = b;
    }
};

int main()
{
    PrepInsta p1(50, 50);
    p1.display();

    // Parameterized conversion constructor invoked using multiple variables
    p1 = {100, 200};
    p1.display();

    PrepInsta p2 = {200, 200};
    p2.display();

    return 0;
}
```

OOPs Advanced - 1

Inheritance in C++

Inheritance is a process in which one object acquires all the properties and behaviour of it's parent object automatically . It is the most important feature of object oriented programming and allows –

- High code re-use
- Lesser time to code
- Easier to maintain application and edit codes
- Re-use pre-defined properties and data



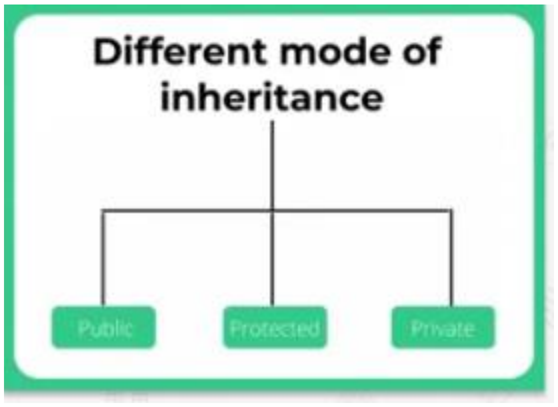
Syntax for Inheriting Declaration –

```
class NameOfDerivedClass : (Visibility mode) NameOfBaseClass
// Data Members and functions
}

In terms of Parent and Child nomenclature -
class NameOfChildClass : (Visibility mode) NameOfParentClass{
// Data Members and functions
}
```

Different Modes of Inheritance in C++

There are three different ways in which we can define relationship between base and derived class while defining inheritance –

1. **Public mode**:
    1. Public members of base class become public in derived class
    2. Protected members of base class become Protected in derived class
    3. Private members are inaccessible in derived class
2. **Protected mode**:
    1. Public members of base class become protected in derived class
    2. Protected members of base class become Protected in derived class
    3. Private members are inaccessible in derived class
3. **Private mode**:
    1. Public members of base class become private in derived class
    2. Protected members of base class become private in derived class
    3. Private members are inaccessible in derived class



Example:

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

// base class
class Animal {

   public:
    void jump() {
        cout << "I can jump!" << endl;
    }

    void sit() {
        cout << "I can sit!" << endl;
    }
};

// derived class
class Dog : public Animal {

   public:
    void bark() {
        cout << "I can bark! Woof woof!!" << endl;
    }
};

int main() {
    // Create object of the Dog class
    Dog doggy;

    // Calling members of the base class
    doggy.jump();
    doggy.sit();

    // Calling member of the derived class
    doggy.bark();

    return 0;
}
```
Benefits of Inheritance

- This helps in reduce cost for projects
- Saves time in coding
- Decreases complexity of the program
- Increases reliability

Types of inheritance in C++

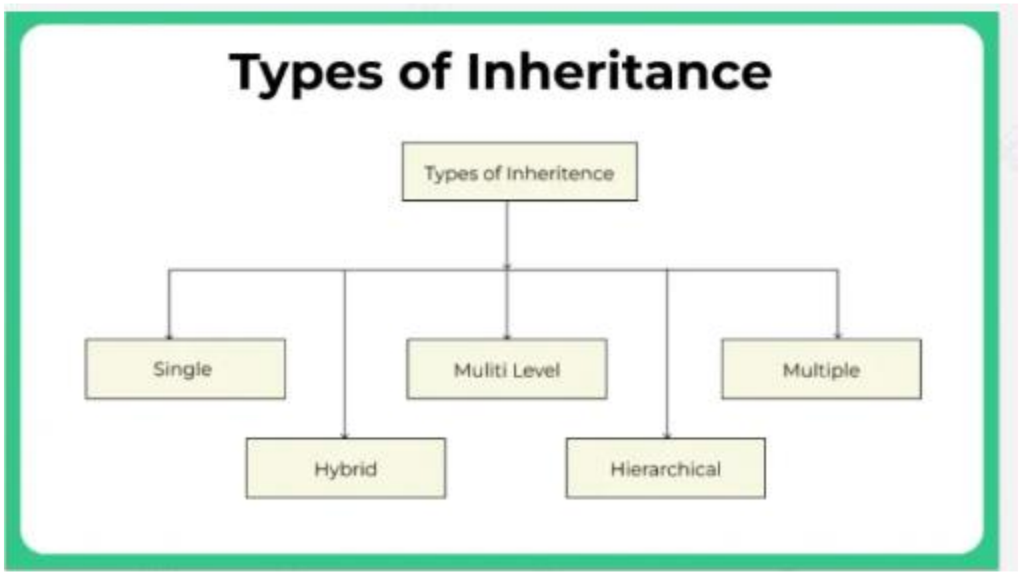
1. Single Inheritance
2. Multilevel Inheritance
3. Multiple Inheritance
4. Hierarchical Inheritance
5. Hybrid or Multipath Inheritance

Types of Inheritance in C++

Types of Inheritance in C++

Interitance is a proces in which one object acquires all the properties and behaviour of it's parent object automatically.Inheritance is of 5 types:

- Single inheritance.
- Multi-level inheritance.
- Multiple inheritance.
- Hierarchical Inheritance.
- Hybrid Inheritance.



What is Single Inheritance in C++ (Single Level)

Inheritance is the type of intelligence where software extension is possible and where you can have more properties than the previous software. In the single Inheritance a class is acquiring the properties and capabilities of from a single class.

Single Inheritance in C++

- The class that is acquiring the behaviors is called child class or derived class or subclass
- The class from which behaviors are taken is called parent class or superclass or base class

Syntax for implementing Single Inheritance

```
class //super class
{
methods;
variables;
-----;
}
class  : Acesss_Specifier  //derived class
{
Parent Methods//extended from parent
Child methods;//written by itself
}
```

Single Inheritance in Public Mode

If the derivation is done in ***public mode then all the public and protected members of the parent class will have the same public and protected access in the child class***

```
#include <iostream>
using namespace std;
class Parent                                    //parent class
{
public:
void parent_property()
    //parent class method
  {
      cout << "\ncash+gold+land+vehicles";
  }
  int p = 10;

};

class Child:public Parent      //derivation mode as public
{
public:
void child_property()
    //Child specific data
  {

      cout << "\n3bhhk Flat";

  }
// contains 2 methods property() and 1 variable P;
};

int main () {
    Child c;
    c.child_property();
    c.parent_property();          //child object accessing parent method
    Parent p;
    p.parent_property();
//p.child_property();//error:child has no such method
}
```

- In Inheritance, ***Child is superior to Parent because the child has its own data and data from the parent whereas Parent has only parent data***
- Here the child has both methods its own method child_property() and derived method parent_property()   whereas Parent class has only one method parent_property()

- Hence It is advisable to create an object for the child rather than the parent

__Private Members cannot be accessed in Child class__

Suppose If we are having 10 methods we need to give 8 methods to the child and stop 2 methods accessing to the child class, then that 2 methods are declared as private

```
#include <iostream>
using namespace std;
class Parent                                    //parent class
{
public:
void parent_property()
                                                //parent class method
  {
     cout << "\ncash+gold+land+vehicles";
  }
int p = 10;
private:
void donation_tocharity()
                                                //not given to child
  {
     cout << "\n100 crores";
  }
};

class Child:public Parent      //parent data is having public access inChild
{
public:
void child_property()//Child specific data
  {
     cout << "\n3bhhk Flat";
  }
// contains 2 methods property() and 1 variable P;
};

int main() {
    Child c;
    c.child_property();
    c.parent_property();         //child object accessing parent method
    Parent p;
}
//p.child_property();  //error:child has no such method
//p.donation_tocharity();//parents own method, error: b
```

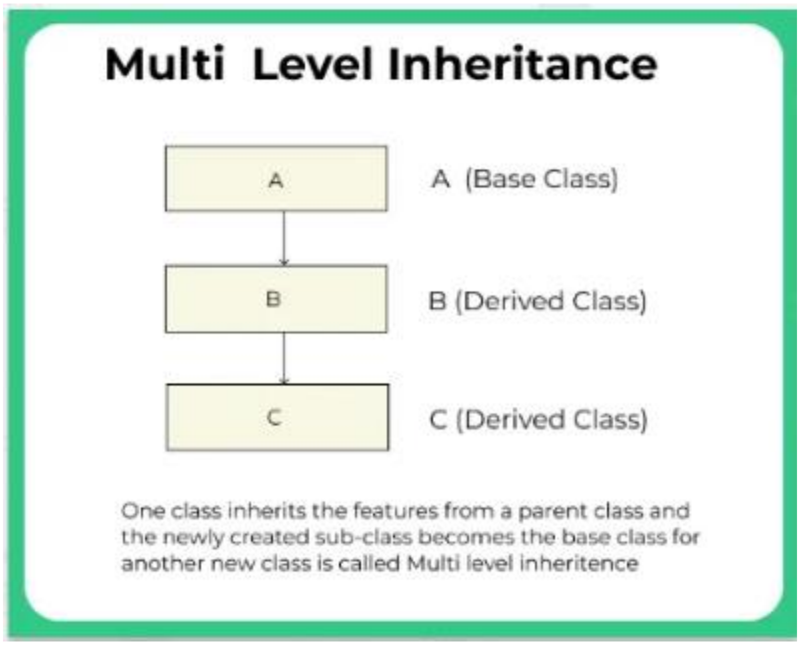
What is Multilevel Inheritance in C++

As we know that Inheritance is the process where a class is Extending the properties from another class but it is interesting to know that we can inherit a class that is already inherited which is known as multilevel inheritance.

Syntax to Implement Multilevel Inheritance:

```
class A

{
//contains Only A class  properties
...;
};
class B: public A//child of A//intermediate base class
{
....;
//contains both  A, B  class properties
};
class C: public B//child of B
{
//contains both A,B ,C class properties

-----;
};
```

- The class that is acquiring the behaviors is called child class or derived class or subclass.
- The class from which behaviors are taken is called parent class or superclass or base class.
- Here B class contains both the properties of A and B whereas C class contains the properties of B and also inherited properties of B i.e A class and also properties of C class


Multi Level Inheritance
A
A (Base Class)
B
B (Derived Class)
C
C (Derived Class)
One class inherits the features from a parent class and the newly created sub-class becomes the base class for another new class is called Multi level inheritence

Example : Program to demonstrate Multilevel Inheritance

```
#include <iostream>
using namespace std;
class grandparent                    //level 1 parent
{
public:
void gp_property()
  {
      cout << "\nproperty earned by grand father-Kandhi Buchayya";
  }
};

class parent:public grandparent
                                                    //inheriting grandparent to parent:level 2
{
public:
void p_property()
  {
      cout << "\nproperty earned by parent-Kandhi Vijay";
  }
//now it conatins 2 methods
};

class child:public parent       //level 3 child(powerful fellow)
{
public:
void c_property()
  {
      cout << "\n property earned by child itself-Kandhi Trishaank";
  }
//It contains 3 methods now
//enjoys properties of parent+grandparent+its own
};

int main() {
    child c;                                        //containss the properties of 2 classes
    c.gp_property();    //derived from grand parent
    c.p_property();                 //derived from parent
    c.c_property();                 //its own method
    parent p;
    p.gp_property();//derived from grand the parent
    p.p_property();                 //its own method
}
```

In this example, class parent contains the capabilities of both parent and grandparent whereas a child has capabilities of both parent and grandparent.

- Hence child class enjoys 3 methods gp_property(),p_property() and c_property()whereas parent class enjoys 2 methods gp_property() and p_property().
- Hence class at the lowest level has more capabilities than remaining parent classes

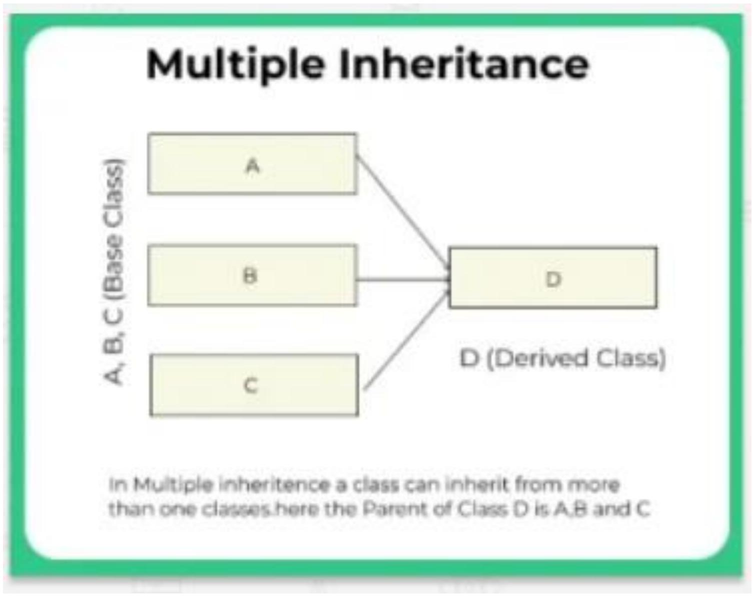
What is Multiple Inheritance in C++

The process where a single class acquiring the behaviors and capabilities from multiple classes called multiple inheritance . It is interesting to know, a child class in multiple inheritances a have more than one parent class at a time

Multiple Inheritance in C++

In Multiple Inheritance, A derived Class can have more than 1 Base Classes, thus it is deriving properties from various base Classes.

Example – In the image, D is a derived class which is deriving its properties from 3 different base classes, A, B and C.


Multiple Inheritance
A, B, C (Base Class)
A
B
C
D
D (Derived Class)
In Multiple inheritence a class can inherit from more than one classes.here the Parent of Class D is A,B and C

Syntax to Implement Multiple Inheritance

```
class parent1
{
……;
};
class parent2
{
……;
};
class child:accces1 parent1,access2 parent2//multiple parents
{
-----;//data of both parent1 +parent2+child
};
```

- The class that is acquiring the behaviors is called ***child class or derived class or subclass***
- The class from which behaviors are taken is called ***parent class or superclass or base class***
- Here parent1 and parent2 are superclasses for a single derived class child

Example Program to understand Multiple Inheritance

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

class P1                              //parent 1
{
public:
void m1()
```

```
  {
     cout << "\n" << "m1 from parent p1-3sh";
  }
};

class P2                          //parent 2
{
public:
void m2()
  {
     cout << "\n" << "m2 from parent p2-rish";
  }
};

class child:public P1, public P2
                                                 //multiple parents p1,p2
{
public:
void m3()
  {
     cout << "\n" << "m3 from childs own code-3shrish";
  }
};

int main()
{
   child c;
   c.m1();                                       //derived from parent p1
   c.m2();                                       //derived from parent p2
   c.m3();                                       //Its own Method
}
```

Here child class is having multiple parent P1 and P2 .it took m1() from p1 and m2() from P2 and its own method m3() hence child is having totally 3 methods

Ambiguity problem in multiple inheritance

Some times *two class can have the same method name and If a child class inherits multiple classes in such case, there is an ambiguity problem in naming*

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

class P1
{
public:
void m1()
```

```
  {
     cout << "\n" << "m1 from parent p1-3sh";
  }
};

class P2
{
public:
void m1()                              //same method name as P1
  {
     cout << "\n" << "m2 from parent p2-rish";
  }
};

class child:public P1, public P2
                                                  //multiple parents p1,p2
{
// which method shoud inherit m1 ()from P1 or P2?
public:
void m3()
  {
     cout << "\n" << "m3 from childs own code-3shrish";
  }
};

int main()
{
   child c;
//c.m1();//derived from parent p1
// [Error] request for member 'm1' is ambiguous
   c.m3();                                        //Its own Method and valid
}
```

- In the above example***, there is m1() in both classes P1 and P2 and child inherits both classes P1 and P2 In such it should inherit which m1() i.e m1() from class P1 or P2***. Hence ambiguity is created upon calling m1();
- Hence, for this reason, OO-designers decided not to have Multiple inheritance in Java Language

What is Hierarchical Inheritance in C++

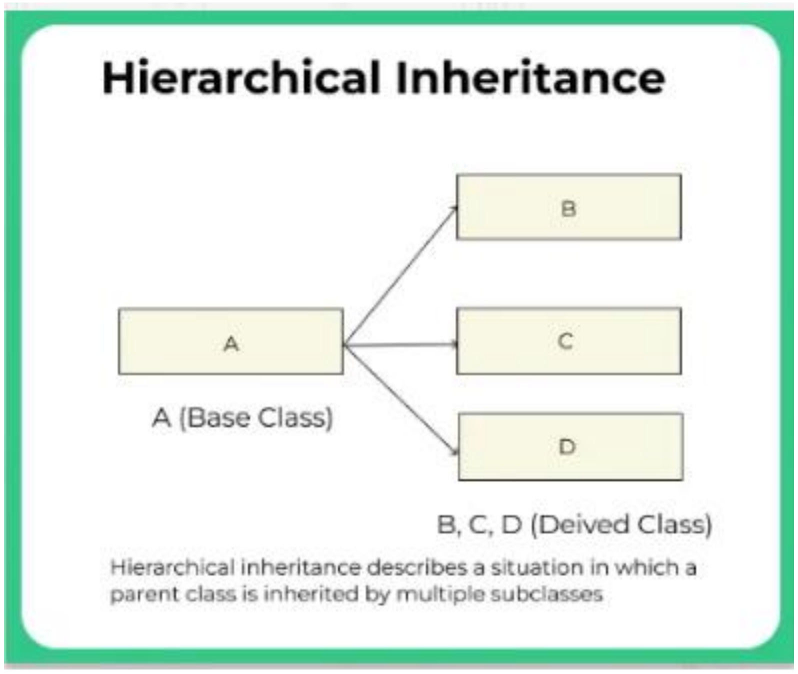
The process where multiple classes inherit the capabilities from a single base class is called Hierarchical inheritance

Syntax to Implement Hierarchical Inheritance in C++

The class that is acquiring the behaviors is called ***child class or derived class or subclass*** The class from which behaviors are taken is called ***parent class or superclass or base class***

```
class Parent//Single Parent class
{
...;
};
class Child1: Access Parent
{
----;
//child 1 derived from the parent
};
class Child2: Access Parent
{
----;
//child 2 also derived from the same parent class
};
```

Here child1 and child2 are two child classes from a common Parent class


Hierarchical Inheritance
A
B
C
D
A (Base Class)
B, C, D (Deived Class)
Hierarchical inheritance describes a situation in which a parent class is inherited by multiple subclasses

Example:program demonstrating Hierarchical inheritance

```
#include <iostream>
using namespace std;
class parent
{
  public:
```

```
  void property()
  {
      cout<<"\nProperty earned by parent";
  }
  int money=2000;
  };

class child1:public parent//child 1 extended from parent
{
  public:
  void c1_property()
  {
    cout<<"\nproperty earned by child1";
  }
};

class child2:public parent//child 2 extended from parent
{
   //child class can be empty
  //Stiill it conatins one method and variable from parent
};

main()
{
  child1 c1;
  c1.c1_property();//childs own method
  c1.property();//taken from parent
  cout<<"\nmoney got from parent:"<<c1.money;//inherited variable
  child2 c2;
  c2.property();//taken from same parent like child 1
}
```

- In the above example parent class is inherited by 2 child classes child 1 and child 2
- child1 has a total of 2 methods its own method c1_property() and inherited method property() from parent and one inherited variable m;
- The child2 has total 1 method i.e it didn't write its won methods. It has only one inherited method property() and 1 variable from the parent

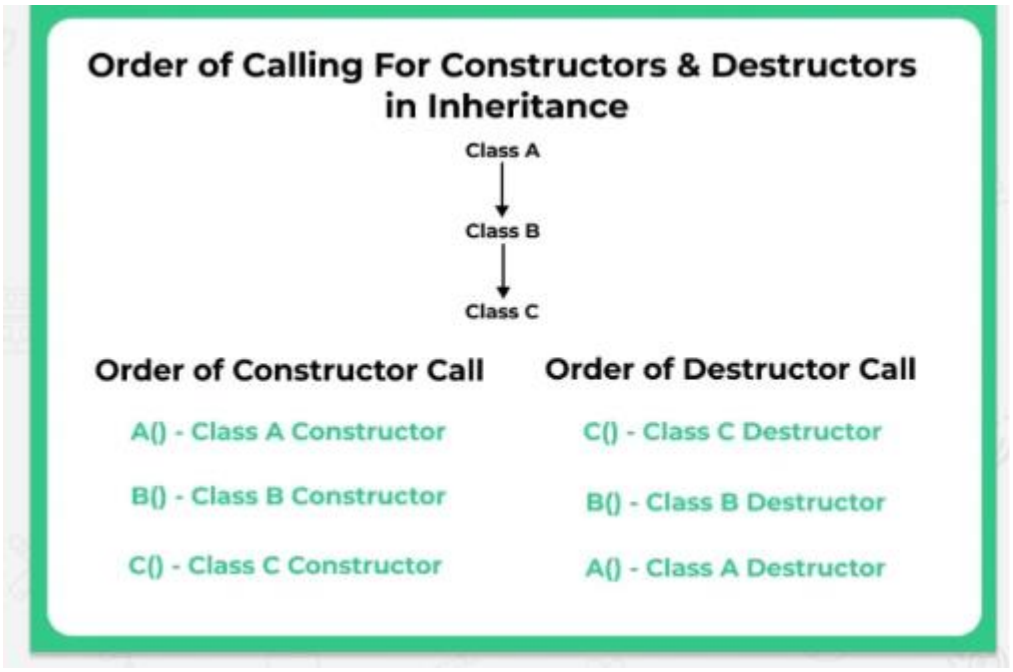
Hence It is called ***hierarchal inheritance because all the inherited class have the same level of properties and capabilities from a parent class***

Constructors and Destructors in Inheritance in C++

When we are using the constructors and destructors in the inheritance, parent class constructors and destructors are accessible to the child class hence when we create an object for the child class, constructors and destructors of both parent and child class get executed.

Constructors and Destructors in Inheritance

Let us look at below example to understand what happens –


Order of Calling For Constructors & Destructors in Inheritance
Class A
Class B
Class C
Order of Constructor Call
A() - Class A Constructor
B() - Class B Constructor
C() - Class C Constructor
Order of Destructor Call
C() - Class C Destructor
B() - Class B Destructor
A() - Class A Destructor

C++ program to the sequence of execution of constructor and destructor inheritance

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

class parent //parent class
{
   public:
   parent() //constructor
   {
      cout << "Parent class Constructor\n";
   }

   ~parent()//destructor
   {
      cout << "Parent class Destructor\n";
   }
};

class child : public parent//child class
{

   public:
   child() //constructor
   {
   cout << "Child class Constructor\n";
   }

   ~ child() //destructor
   {
   cout << "Child class Destructor\n";
   }
};

int main()
{
   //automatically executes both child and parent class
   //constructors and destructors because of inheritance
   child c;

   return 0;
}
```

Inheritance in Parametrized Constructor/ Destructor

In the case of the default constructor, it is implicitly accessible from parent to the child class but parameterized constructors are not accessible to the derived class automatically, for this reason, an explicit call has to be made in the child class constructor to access the parameterized constructor of the parent class to the child class using the following syntax

```
<class_name>:: constructor(arguments)
```

NoteWhenever you are using the parameterized constructor in the parent class it is mandatory to define a default constructor explicitly

Example program

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

class parent
{

    int x;
    public:

    // parameterized constructor
    parent(int i)
    {
        x = i;
        cout << "Parent class Parameterized Constructor\n";
    }
};
```

```
class child: public parent
{

    int y;
    public:

    // parameterized constructor
    child(int j) : parent(j)  //Explicitly calling
    {
        y = j;
        cout << "Child class Parameterized Constructor\n";
    }
};

int main()
{
    child c(10);

    return 0;
}
```

Constructor call in multiple inheritance constructors

```
class C: public A, public B;
```

Constructors are called upon the order in which they are inherited

First class A constructors are executed followed by class B constructors, then class C constructors

Polymorphism in C++

Polymorphism in C++ basically means having multiple existent forms in the program, A simple code may behave differently in different situations. For example, we have only one identity, to some we are friends, or father, student, employee etc.

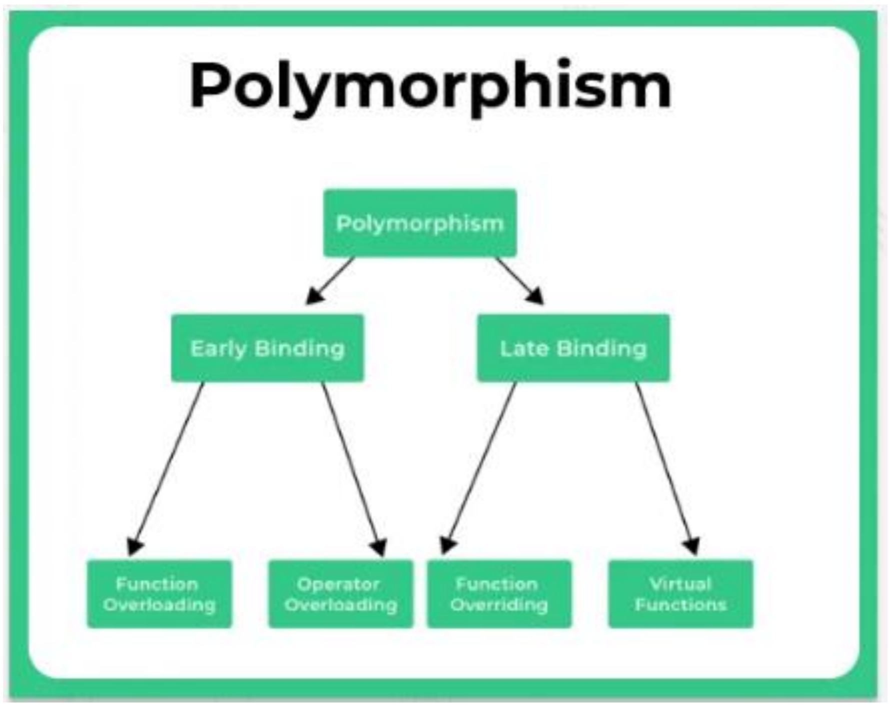
How Polymorphism works in C++

In C++, polymorphism, generally happens because of classes objects and events are related with inheritance and hierarchy. Lets see what's polymorphism in detail below –

Polymorphism is of two types –

- **Compile time polymorphism (demonstrates static/early binding)**
  - Function Overloading
  - Operator Overloading

- **Run time polymorphism (demonstrates dynamic/late binding)**
    - Function Overriding (Achieved with help of virtual keyword)



Compile Time Polymorphism

<u>Function Overloading</u>

This happens when in a given program there are multiple functions with the same name but different arguments inside it or different parameters.

For example –

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

class myClass {
   public:
   void myFunction(int a)
   {
      cout << "Accepted integer value " << a << endl;
   }

   void myFunction(double a) {
      cout << "Accepted double value " << a << endl;
```

```
    }

    void myFunction(int a, int b)
    {
        cout << "Accepted multiple values, Value1:" << a << endl;
        cout << "Accepted multiple values, Value2:" << b << endl;
    }
};

int main() {
    myClass myclassObj;

    // will go to function with int as passed on variable as value passed is int type
    myclassObj.myFunction(5);

    // will go to function with double
    myclassObj.myFunction(500.263);

    // will go to function with multiple parameters
    myclassObj.myFunction(5,3);

    return 0;
}
```

Now, in the above, the functions go to their correct argument type in myClass **this is called function overloading.**

Note1. Functions with the same name but different return types. Example void myFunction(int a), int myFunction(int a) will throw error.

2. Pointer * vs Array int fun(int *ptr); || int fun(int ptr[]); // redeclaration of fun(int *ptr) can't be overloaded.

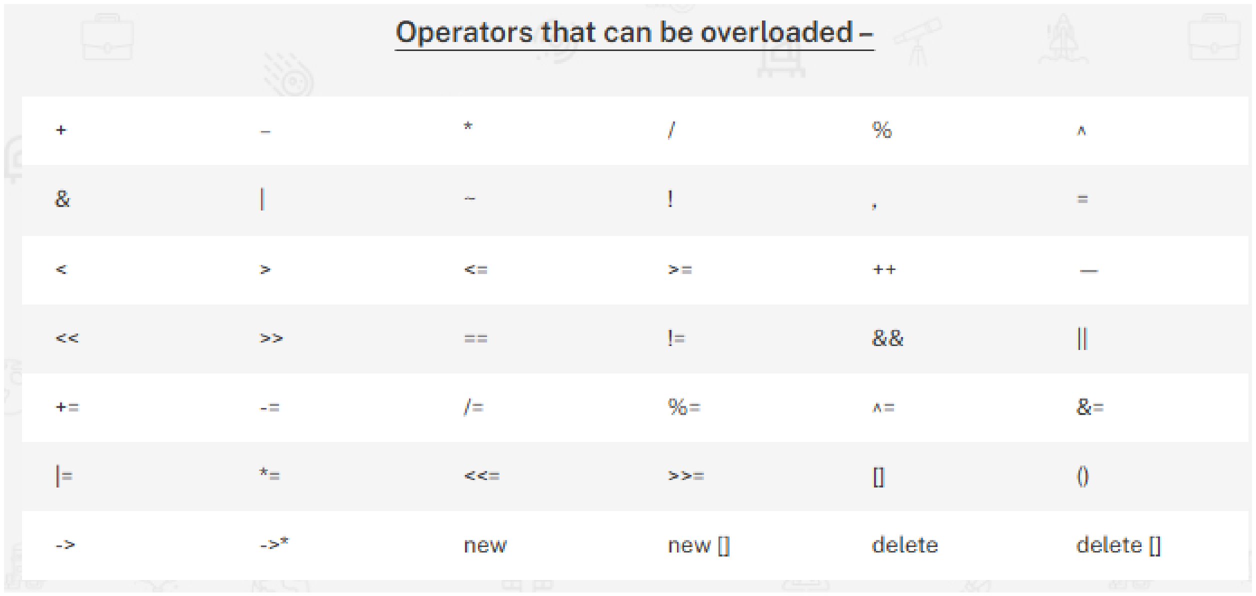
Operator Overloading

Operator overloading is possible in C++, **you can change the behaviour of an operator like +, -, & etc to do different things.**

- For example, you can use + to concatenate two strings.
- Consider two strings s1 and s2, we can concatenate two strings s1 and s2 by overloading + operator as String s, **s = s1 + s2;**

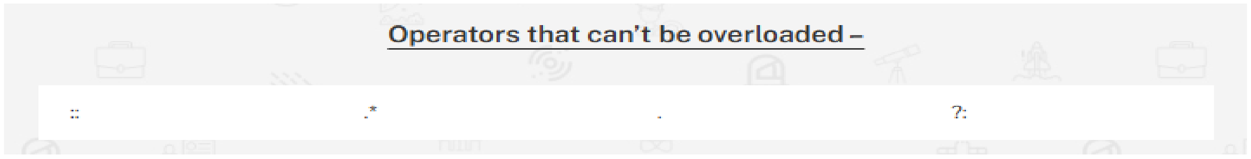
For Example – We know that ++x increments the value of x by 1. What if we want to overload this and increase the value by 100. The program below does that.

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

class Test
{
    private:
    int count;

    public:
    Test(): count(101)
    {
        //writing fuction name followed by data member and passed on value works like constructor
        // Test(): count(101) same as writing count = 101; here
    }

    // we write operator before overloading such operators
    void operator ++()
    {
        count = count+100;
    }

    void Display()
    {
        court << "Count: " << count;
    }
};

int main()
{
    Test t;

    // this calls "function void operator ++()" function
    ++t;

    t.Display();

    return 0;
}
```



Operators that can be overloaded –

+	-	*	/	%	^
&	|	~	!	,	=
<	>	<=	>=	++	--
<<	>>	==	!=	&&	||
+=	-=	/=	%=	^=	&=
|=	*=	<<=	>>=	[]	()
->	->*	new	new []	delete	delete []



Operators that can't be overloaded –

::	.*	.	?:

Runtime Polymorphism
There are two ways run time polymorphism may be achieved in C++

- Function Overriding
- Virtual Functions (Solves the problem of static resolution while working with pointers)

Note – This information is given wrong on Gks4Gks and tut point they have explained virtual functions instead at first.

Runtime Polymorphism using Function Overriding
In a case when a function is declared in both the parent class(Base Class) and child class(Derived Class).

- To decide upon which function has to be called, the program will only know at the run time.
- Let us look at the very simple program below to understand the basic

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

```

```
// This is Parent class
class Parent
{
    public:
    void print()
    {
        cout << "Parent Class printing" << endl;
    }
};

// This is Child class
class Child : public Parent
{
    public:

    // as we see that it is already declared in the parent function
    void print()
    {
        cout << "Child Class printing" << endl;
    }
};

int main()
{
    //Creating object for parent class
    Parent parent_object;

    //Creating object for child class
    //No need to get conused here its same as Child object2;
    //We are just called a default constructor
    Child child_object = Child();

    // This will go to the parent member function
    parent_object.print();

    // This will however, go to child member function
    // overrides the parent function as the object is of the child class
    child_object.print();

    return 0;
}
```

<u>What exactly happens at the backend?</u>

Even though, the child class has inherited all the functions of the parent class and has the same function definition within itself. The call is made to function of the child class as the object is of the child class.

What is this called This is behaviour shown by the child class object is called function overriding and this happens at run time. The child object has overridden the function definition of parent class and chose to exhibit its own properties.

What if?Now, what if the child class didn't have its own function declaration, in this case, it would have gone ahead with the inherited parent class definition of the print() function.

Runtime Polymorphism using Virtual Function

- What if we create a pointer to the parent class
- And assign it to the address of the base class object

Something such as this –

Parent *parent_object;

Child child_object;

parent_object = &child_object;
Now, we will expect the function called by parent_object to call the child object function right? Because the final address is pointed toward child_object location. Let us see what happens –

NoteIn below example, compile time polymorphism happens. We will force runtime polymorphism later in 2nd example below when we force it using virtual functions

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

// This is Parent class
class Parent
{
    public:
    void print()
    {
        cout << "Parent Class printing" << endl;
    }
};

// This is Child class
class Child : public Parent
{
    public:

    // as we see that it is already declared in the parent function
    void print()
    {
        cout << "Child Class printing" << endl; } }; int main() { Parent *parent_object; Child
child_object; parent_object = &child_object; // catch of the program is here // also as we are
dealing with pointers instead of . we need to use ->
    parent_object->print();
// In the above program instead of using pointers we can write
// Parent parent_object = Child();
// parent_object.print();
// this would also give same results

return 0;
}
```

What happenned above (Static Resolution) Now, this happens because of how C++ was written by its writer. Now, while run time polymorphism may exhibit dynamic (or late) binding.

But, this instance is a classic example of early binding as the print() function is set during the compilation of the program. The definition of C++ is written, forces this static call and is also called as static resolution.

Thus, the above is an example of Compile time polymorphism. Also known as early binding.

How to force into Runtime Polymorphism (Virtual Functions) The issue caused by static resolution is solved by using **virtual function** that turns the early binding into late binding i.e. compile time process to runtime process.

Forcing Runtime Polymorphism using Virtual function

If we add a virtual keyword before the function in the parent(base) class we can force runtime polymorphism (Late binding). Let us look with the code below –

Code

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

// This is Parent class
class Parent
{
    public:
    // adding virtual keyword here
    virtual void print()
    {
        cout << "Parent Class printing" << endl;
    }
};

// This is Child class
```

```
class Child : public Parent
{
    public:

    // as we see that it is already declared in the parent function
    void print()
    {
        cout << "Child Class printing" << endl;
    }
};
int main()
{
Parent *parent_object;
Child child_object;
parent_object = &child_object;

// catch of the program is here
// also as we are dealing with pointers instead of . we need to use -> parent_object->print();
 return 0;
 }
```

What exactly happenned Using the virtual keyword guarentees that the base(parent) class function is overridden and dervided(child) class function is implemented

This is called as late binding at runtime, thus runtime polymorphism is implemented. We solved the problem of static resolution which had earlier forced early binding(Compile time polymorphism)

Function Overriding in C++

What is Function Overriding?

- Using inheritance a child class can inherit features (data members/member functions) from a parent class. Imagine if, the same function is defined in both the parent class and the child class.
- Now if we call this function using the object of the child class, the function of the child class is executed.
- This is known as **function overriding** in C++. The function in child class overrides the inherited function in the Parent class
- **There are two methods to do function overriding –**

1. Generic function overriding
2. Virtual function based overriding

Advantages of function overriding in C++

As we have discussed above what is function overriding, now let’s look at some of its advantages –

- The child class having the same function as the parent class, can even have its independent implementation of that function.
- Helps in saving memory space.
- Maintain consistency, and readability of our code.
- Helps in making code re-usability easy.
- The child class can access the function of the parent class as well.

Disadvantages of function Overriding in C++

After learning things about function overriding lets have a look on its disadvantage:-

- Function overriding cannot be done within a same class. So, we need to do implement inheritance.
- Static methods can never be overridden.

```
class Base
{
   static void fun()
};

class Derived : public Base
{
   void fun() // error
};
```

Code to show Function Overriding (Method 1)

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

class Parent {
  public:
   void display() {
        cout << "Parent Function" << endl;
    }
};

class Child : public Parent {
   public:
    void display() {
        cout << "Child Function" << endl;
    }
};

int main() {
    Child childObj;
    childObj.display();

    return 0;
}
```

Using Virtual Function to Achieve Function Overriding (Method 2)

For a function display() which is defined in both parent & child classes. If we create a pointer of Parent type and point it to an object of the child class. It will –

- Always call the parent class function
- Will never override parent class function

As demonstrated in the example given below –

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

class Parent
{
public:
  void print ()
  {
    cout<<"Parent Class"<<endl;
  }
};


class Child:public Parent
{
public:

  void print ()
  {
    cout<<"Child Class"<<endl;
}};
int main ()
{
  Child c2;
  Parent &p2 = c2;
  p2.print ();
  return 0;
}
```

ReasonThe reason is because of how C++ was written by its founder Bjarne Stroustrup in 1979. What exactly happens here is called static resolution. How C++ is written forces by default makes a static call at the time compilation itself and forces early binding i.e. binding to parent class function itself.

How to solve this?

We use a virtual function in the parent class in order to ensure that the parent function is **overridden.** It is done by adding virtual keyword before function declaration in the parent class.

Code

```
#include <iostream>

using namespace std;


class Parent
{
public:

  virtual void print ()
  {
    cout << "Parent Class" << endl;
  }
};


class Child:public Parent
{
public:

  void print ()
  {
    cout << "Child Class" << endl;
  }
};
```

```
int main ()
{
  Child c2;
  Parent &p2 = c2;

  p2.print ();

  return 0;
}
```

Function overloading in C++

Function overloading is the process in which we use the same name for two or more functions defined in the class. The only difference between these functions is that they have different types of parameters or a different number of parameters.

Function Overloading in C++ Language

- A function is a set of statements that allow us to structure program in segments of code to perform individual tasks.
- For example a function created as ”int addition(int a, int b)” can be used anywhere in our program where we need to add two integer type numbers.
- Function overloading is the process in which we use the same name for two or more functions defined in the class.
- The only difference between these functions is that they have different types of parameters or a different number of parameters.
- We have to use different type or different number of arguments because it is the only way by which compiler can differentiate between two or more overloaded function.
- It is done at compile time hence it is also known as compile time polymorphism.

Advantages of function overloading in C++

As we have discussed above what is function overloading, now lets look at some of its advantage :-

- Using the function overloading concept, we can make more than one function with the same name

- Function overloading helps us to save the memory space, consistency, and readability of our code.
- Function overloading speeds up the execution of our code.
- Function overloading helps the application to load the class method based on the type of parameter.
- Function overloading makes code re-usability easy, thus it also helps to save memory.
- Function overloading makes code maintenance easy.

C++ programming code to show function overloading

```
#include <iostream>
using namespace std;
int area(int s)
{
   cout<<"Area of square="<<s*s <<endl;

   return 0;
}

int area(int l, int b)//Second definition.
{
   cout<<"Area of reactangle=" <<l*b<<endl;
   return 0;
}


int main()
{
   area(12);//Passing value for first defination.

   area(20,10);//Passing value for second defination.

}
```

<u>Disadvantages of function Overloading in C++</u>

After learning things about function overloading lets look at some of its disadvantage:-

- Function overloading process cannot be applied to the function whose declarations differ only by its return type

```
 void area()
 int area(); // error
```

- Member function declared with the same name and the same parameter types cannot be overloaded if any one of them is a static member function declaration.

```
static void area();
void area(); // error
```

What is Compile Time Polymorphism in C++

Polymorphism means to exist in many forms. There are two types of Compile Time Polymorphism in C++ which are function overloading and operator overloading.

Let us understand with an example how each works.

Function Overloading –

What is function Overloading In case we have multiple functions that have same name but, they have different arguments as parameters.

There are two different variations -

1. Change in number of arguments -
Example Function : prepinsta(int a) and prepinsta(int a, int b)

2. Change in the argument type -
Example function : prepinsta(int a) and prepinsta(float a)

- Function overloading is a concept via which we can achieve compile-time polymorphism,
- The function call and the function definition is bonded together by the compiler at compile time itself.

Let us look at an example of how this is done –

Code

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

class PrepInsta
{
    int x, y;
    double z;

    public:
    void myFunction(){
        cout << "No arguments passed here" << endl;
    }

    void myFunction(int x1){
        x = x1;
        cout << x << " was passed here (int value)" << endl;
    }

    void myFunction(double x1){
        z = x1;
        cout << z << " was passed here (double value)" << endl;
    }

    void myFunction(int x1, int y1){
        x = x1;
        y = y1;
        cout << x << " and " << y << " were passed here (int values)" << endl;
    }
};

int main(){
    PrepInsta obj;

    obj.myFunction();
    obj.myFunction(10);
    obj.myFunction(10.00);
    obj.myFunction(10,20);

    return 0;
}
```

Operator Overloading

IssueOperators don't work with objects. They only work with variables generally.

Example1 - For an object obj, We can't do obj++ to increment all data members in the class.

Example 2 - We can't do obj1 + obj2 to add all data members of two objects.

SolutionUsing operator overloading we can use these operators to -

1. To work with objects
2. To alter how these operators may work by providing our own implementation.

Example - We can alter the behaviour of an operator such as "%" or "&" or "+" etc and give our own implementation meaning and work with objects

We use keyword operator while defining an operator that we have to overload.

There are two types of operator overloading –

1. Unary Operator overloading
2. Binary Operator overloading

For example, it maybe weird but we can overload operator + to subtract rather than add two values.

```
void operator +()
  {
    z = x - y;
  }
```

Let us implement the same with an example –

Code

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

class PrepInsta {
    private:
    int x;

    public:

    // why int x1 = 0 (see explanation at the end of the code)
    PrepInsta(int x1 = 0){
        x = x1;
    }

    PrepInsta operator + (PrepInsta const &obj) {
        PrepInsta temp;

        // x of obj1 is added with x of obj2 and result it stored in
        // newly created temp object's x varaible
        // temp object is returned to be stored in obj3 in main
        temp.x = x + obj.x;
        return temp;
    }

    void print()
    {
        cout << x << endl;
    }
};

int main()
```

```
{
    PrepInsta obj1(20), obj2(10);

    PrepInsta obj3 = obj1 + obj2;
    obj3.print();

    return 0;
}
// why PrepInsta(int x1 = 0) in paramterized contructor ?
// For obj1(20) & obj2(10) we have directly values 20 & 10
// However, for temp object we didn't use parameterized contructor value
// PrepInsta(int x1 = 0) allows us to put default value to 0
// if object is created without parameterized value
// However, if parameterized value is passed (as in case of obj1(20) & obj(10))
// these will override default value 0 and place 20 and 10 as values
```

Why should you perform overloading

- Naming burden: In some situations where we need to have same functionalities but with a different type of inputs or a different number of inputs, in that situation, if you maintain separate method names for each and every method where functionality is the same the naming burden of remembering this method names increases on the user
  Example: In C language we have abs(), sabs(), labs(), fabs() methods, the functionality of all these methods is same, but still the user needs to remind all these method names
- Code readability Increases

RunTime Polymorphism in C++

The RunTime Polymorphism in C++ is achieved greatly with the help of virtual functions, it happens largely due to multiple declaration of same functions in both base and derived classes

Definition issue with C++

Let us look at this example first to understand **what is definition issue that is caused in C++** and later we will understand how virtual functions help us resolve this issue and create run time polymorphism

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

  // This function is declared in the base class as well
  void show()
  {
    cout << "Showing Derived Class" << endl; } }; int main() { Base *base_object; // pointer
reference to base class Base *base_object; // creating derived class object Derived
derived_object; base_object->show(); // mapping base object to address of derived class object
base_object = &derived_object; // since we use pointers so we use -> rather than .
  return 0;
}
```

- One would expect the derived class to be referred and print showing derived class
- It's natural to think this way as the address referred finally is of the derived class right?
- But, this doesn't happen, rather it prints showing base class

Now this happens because of how C++ is written and **it causes something called as static call or static resolution which is a classic example of early binding**.

The compiler sets the default function to be called at compile time itself and choses base class object and thus the derived class functions can never be called using pointed references between base and derived classes.

How Run Time Polymorphism is achieved?

Polymorphism is achieved by function overriding, however, this problem of early binding at compile time is resolved by invoking run time polymorphism using virtual functions.

Let us understand this with an example on how this happens.

We add virtual keyword before the function in the base class.

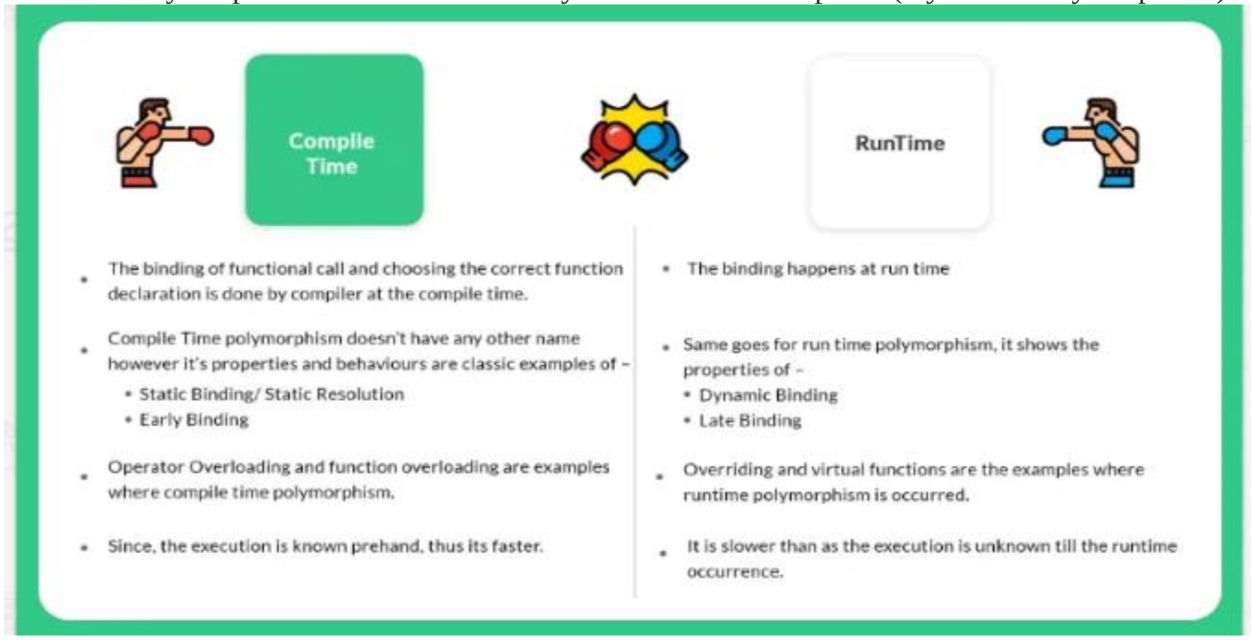
```
#include  <iostream>
using namespace std;
// This is Base class
class Base
{
public:
  virtual void show()
  {
    cout << "Showing Base Class" << endl;
  }
};
// This is Derived class function
class Derived:public Base
{
public:
// This function is declared in the base class as well
  void show()
  {
    cout << "Showing Derived Class" << endl; }}; int main() { Base* base_object; // pointer
reference to base class Base *base_object; // creating derived class object Derived
derived_object; base_object->show(); // mapping base object to address of derived class object
base_object = &derived_object; // since we use pointers so we use -> rather than .
  return 0;
}
```

Now, using virtual keyword before the parent class declaration of the function we have asked compiler to not do early binding and only resolve this at run time. This is how, run time polymorphism is achieved in C++.

Difference between Compile time and Runtime Polymorphism in C++

Now, we will talk about the major differences between compile time and runtime polymorphism in C++, we hope you understand the basics about the two fairly-

Runtime Polymorphism is also known as Dynamic Method Dispatch (Dynamic Polymorphism).


Compile Time
RunTime

- The binding of functional call and choosing the correct function declaration is done by compiler at the compile time.
- Compile Time polymorphism doesn't have any other name however it's properties and behaviours are classic examples of –
  - Static Binding/ Static Resolution
  - Early Binding
- Operator Overloading and function overloading are examples where compile time polymorphism.
- Since, the execution is known prehand, thus its faster.

- The binding happens at run time
- Same goes for run time polymorphism, it shows the properties of –
  - Dynamic Binding
  - Late Binding
- Overriding and virtual functions are the examples where runtime polymorphism is occurred.
- It is slower than as the execution is unknown till the runtime occurrence.

How Polymorphism works in C++
There are two types of polymorphism that occur in C++ which are –

- **Compile time polymorphism (demonstrates static/early binding)**
    - Function Overloading
    - Operator Overloading

Now, compile time polymorphism is classic example of how the decision happens at compile time, in other words, objects, functions and classes are binded early at compile time and are staic

For the above Reason Compile time Polymorphism is said to display properties of static resolution or early binding.

- **Run time polymorphism (demonstrates dynamic/late binding)**
    - Function Overriding (Achieved with help of virtual keyword)

The decision to bind objects, function, classes etc together is made at run time and is dynamic i.e. happens very late at runtime.

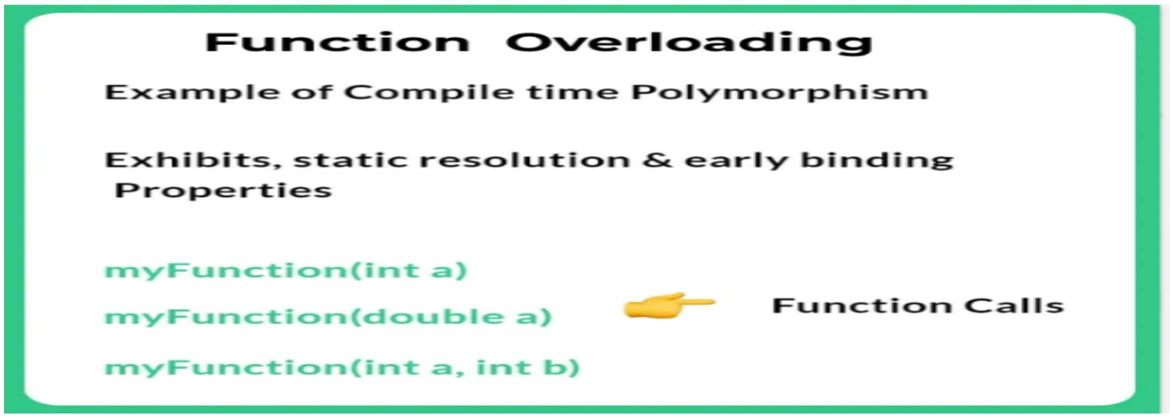
For the above Reason Runtime Polymorphism is said to display properties of dynamic resolution or late binding.

Compile Time Polymorphism
Function Overloading
When in a class there are multiple functions with same name, however the arguments passed in the function may different in number or type then the program exhibits behavior of function overloading

- For example

```
myFunction(int a, int b) and myFunction(double a, double b).
```

Let us have a look on a program to understand this.



Function Overloading

Example of Compile time Polymorphism

Exhibits, static resolution & early binding Properties

myFunction(int a)
myFunction(double a)
myFunction(int a, int b)

Function Calls

Example –

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

class PrepInsta {
  public:
  void sum(int a)
  {
    cout << "Sum of the int type variable is:" << a << endl;
  }
  void sum(double a) {
    cout << "sum of the double type variable is:" << a << endl;
  }
  void sum(int a, int b)
  {
    cout << "Sum of int type variables are:" << a + b << endl;
  }
   void sum(double a, double b)
  {
    cout << "Sum of double type variables are:" << a + b << endl;
  }
};

int main() {
  PrepInsta obj;

  obj.sum(5);
  obj.sum(500.263);
  obj.sum(5,3);
  obj.sum(5.333,3.167);

return 0;
}
```

Late Binding in C++ (Dynamic Binding)

Definition of Late Binding in C++(Dynamic Binding): Whenever the child class is not satisfied with the implementation of the parent class behavior then it can redefine that particular method with its own definition.

Late Binding in C++

Late binding in C++(Dynamic Binding) is nothing but the most popular run polymorphic technique method overloading

- We call it as late binding because method calls or identified during the execution(runtime). ***Among multiple methods in overriding, the method call is binded(attached or matched) with the required method definition at runtime***
- We call it as late because identification of method calls happen not at compilation but during the execution

C ++ program to demonstrate method overriding

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

class parent //parent class
{
public:
void property()
  {
    cout << "land+cash+gold\n";
  }
void marriage()
  {
    cout << "arranged marriage\n";
  }
};

class child:public parent       //child class
{
public:
void marriage()      //own definition of child
  {
    cout << "love marriage\n";
  }
//contains property() from parent and its own marriage()
};

int main()
{
    child c;
    c.property();
    c.marriage();
    parent p;
    p.property();
    p.marriage();

  return 0;
}
```

- Here year child class inherits two methods from parent ,property() and marriage() child is satisfied with property() but not with marriage() .hence child class had given its own implementation for marriage()
- If the child class don't inherit parent method he will not get the property().that's why there will inheritance during overriding

Static Binding (Early Binding) in C++

Defining multiple methods with the same name but difference in the number of arguments or the data type of arguments or ordering of arguments and resolving this method calls among multiple methods at compilation itself is called Static Binding or Early binding or Method overloading .

What is static Binding/ Early Binding?
Static Binding or Early Binding is most popular compile-time polymorphic technique method overloading.

It is called Static Binding or early binding because

- The compiler gets clarity among multiple methods with **the same names**.
- Knows which is the exact method to be **executed at compilation itself.**

TriviaThe compiler knows which function to call at compilation itself.

The compiler binds the connection from the function call to the actual function to be called at compile time itself.

This is why its called early binding as binding happens at compile time (before runtime)

Trivia 2 Since compiler binds the function call and function definition at compile time. Whatever values are entered / given at run time will not change function called.

Which is why its also referred to as **Static Binding**

Example Static Binding caused by Method overloading

```
// The compiler knows which function to call at compilation itself
// The compiler binds the connection from function call
// to the actual function to be called at compile time itself
// Which is why its called as early binding as binding happens
// at compile time (before runtime)
#include <iostream>
using namespace std;

class Numbers
{
    public:

    int findSum(int val1, int val2) {
        return val1 + val2;
    }

    double findSum(double val1, double val2){
        return val1 + val2;
    }

    int findSum(int val1, int val2, int val3) {
        return val1 + val2 + val3;
    }
};

int main()
{
    Numbers obj;
    cout << "Sum is " << obj.findSum(10, 20) << endl;
    cout << "Sum is " << obj.findSum(12.5, 28.3) << endl;
    cout << "Sum is " << obj.findSum(10, 20, 30) << endl;

    return 0;
}
```

How method overloading works

- Here we have Defined Multiple methods with the same name findSum()
    - obj.findSum(10, 20) calls findSum(int val1, int val2)

- obj.findSum(12.5, 28.3) calls findSum (double val1, double val2)
- obj.findSum(10, 20, 30), calls findSum(int val1, int val2, int val3)

The binding of function call via object to actual function in class happens early at the **compile time**, rather than Run Time. Hence called as **early binding**.

Since, at run time even if there are user inputs the binding won't change for these function call and bonded signature of functions this often also referred to as **Static Binding**

Why should you perform overloading?

Naming burden

In some situations where we need to have same functionalities but with a different type of inputs or a different number of inputs, in that situation

If you maintain separate method names for every method where functionality is the same the naming burden of remembering these method names increases on the user

Example: In C language we have abs(), sabs(), labs(), fabs() methods, the functionality of all these methods is same, but still the user needs to remind all these method names

Code readability

Increases code readability vastly

Points to Remember:

Impact of return type:

In method overloading, there is no impact on return type, you ***can maintain different return type or the same return type for overloaded methods***

```
void demo(int a){
   cout << a;
}

int demo(){
   int a = 2;
   return a;
}

int demo(char a){
   return a;
}
```

Ambiguity:

If you define methods with the same signature(same name, number of arguments, the order of argument, the data type of arguments)exactly the same, then it causes ambiguity, the compiler is unable to find what method is to be executed

```
int add(int a, int b){
   return a + b;
}
// error: Ambiguity in defining methods
int add(int a, int b) {
   return a + b;
}
```

Ordering of arguments

```
void add(char a, int b){
   return a + b;
}

void add(int a, char b){
   return a + b;
}
```

In the above example, overloading is valid because ***even if you have the same number of arguments but the ordering of data types is different***

Early binding and Late binding in C++

Early binding and Late binding in C++

Early Binding Early binding is a phenomenon wherein the decision to match various function calls happens at the compile time itself and the compiler directly associates the link with addresses.

Late Binding Late binding in the above problem may be solved by using virtual keyword in the base class. Let's see how this happens by using the above example itself, but only adding virtual keyword.

What exactly is binding?

For Functions –

Binding for functions means that wherever there is a function call, the compiler needs to know which **function definition** should it call?

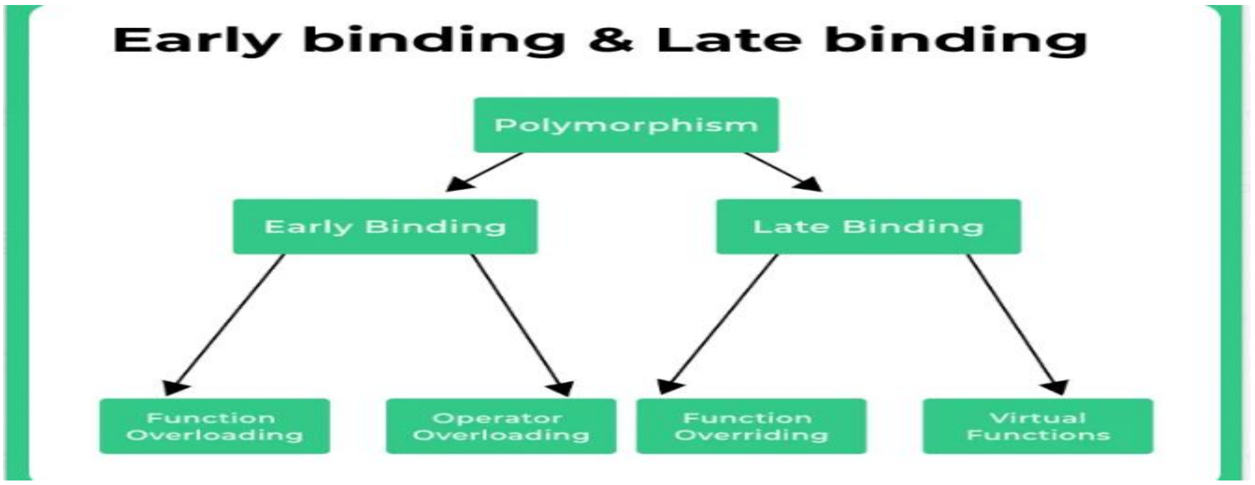
- This depends upon the signature of each function declaration & the arguments that it is taking.
- Also, compiler needs to know when this function binding happens (at compile/run time).

ExamplemyFunc(int x, int y) or myFunc(x) or myFunc(float x, float y), these are function declarations.

The compiler needs to know which declaration should it refer to whenever it sees a function call. Also when it should make the decision. At runtime or compile time?

For Variables –

It needs to know which variable should be referred to as there may be variables with the same names but different types like int, float, double (in some cases this is allowed in C++)



Early Binding

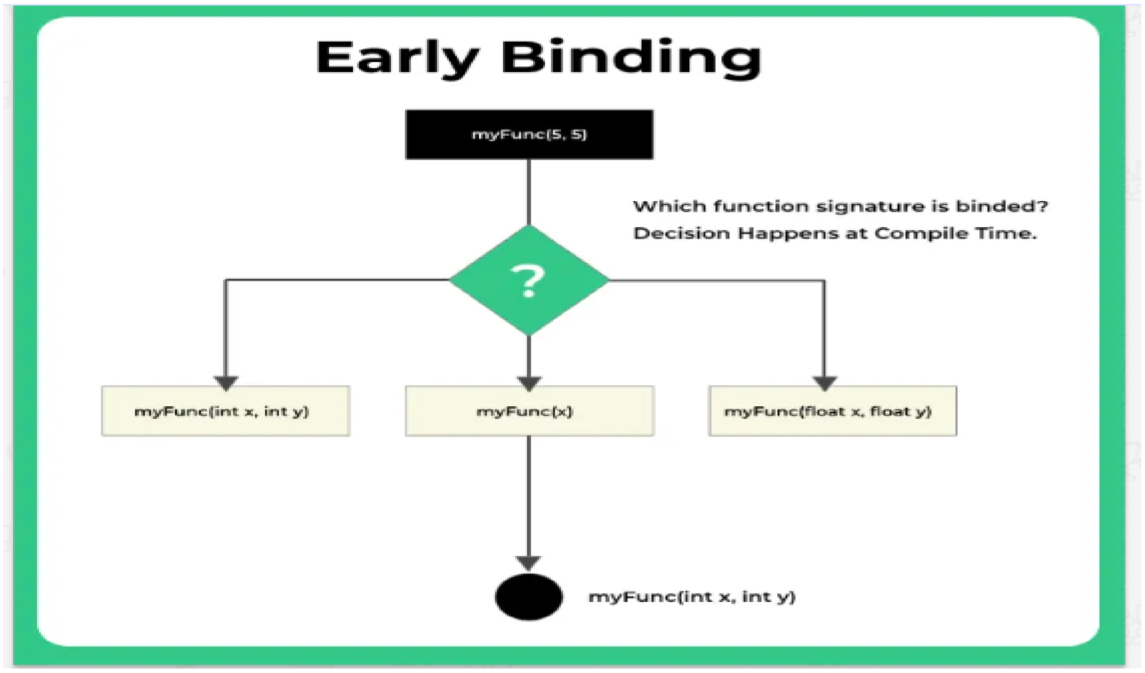
What is Early Binding?Early binding is a phenomenon wherein the decision to match various function calls happens at the compile time itself and the compiler directly associates the link with addresses.

As we know we write code in a high-level language. At the time of compilation, the following happens –

The compiler converts this HLL code into a low-level language that computers can understand, mostly machine language.

- In early binding, the compiler chooses a function declaration, out of many functions with different signatures/arguments at compile time itself.
- Thus as the name suggests the binding happens very early before the program runs.

This is why early binding is an example of compile-time polymorphism.



Code for Early Binding

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

// Defining the parent class
class Parent
{
   public:
   void print()
   {
      cout << "Parent Class" << endl;
   }
};

// Defining the child classs
class Child : public Parent
{
   public:
   // print redeclared in child class
   void print()
   {
      cout << "Child Class" << endl; } }; int main() { Parent *parent = new Child(); // catch of
the program is here // also as we are dealing with pointers instead of . we need to use ->
   parent->print();
   return 0;
}
```

Explanation:

Now, its obvious to think that the child class would be printed as eventually the address assigned is of the Child class right? since.

- Parent *parent = new Child(); is same as defining it as follows –
    - Parent *parent;
    - Child child;
    - parent = &child

And again visualising this way also leads to a conclusion that still child class should have been printed as the end address is of the child class right?

What actually happens?

Now, this isn't the case obviously and a part of the reason is because of how C++ was written by its founder Bjarne Stroustrup in 1979. **What exactly happens here is called static resolution.**

This happens because the way C++ was written which by default makes a static call at the time compilation itself and forces early binding. Now, print function is bounded at compile time.

Default behaviour to remember So remember this whenever there are pointers involved and base(parent) class object is pointed to derived class(child) address, then static resolution will happen.

Late Binding

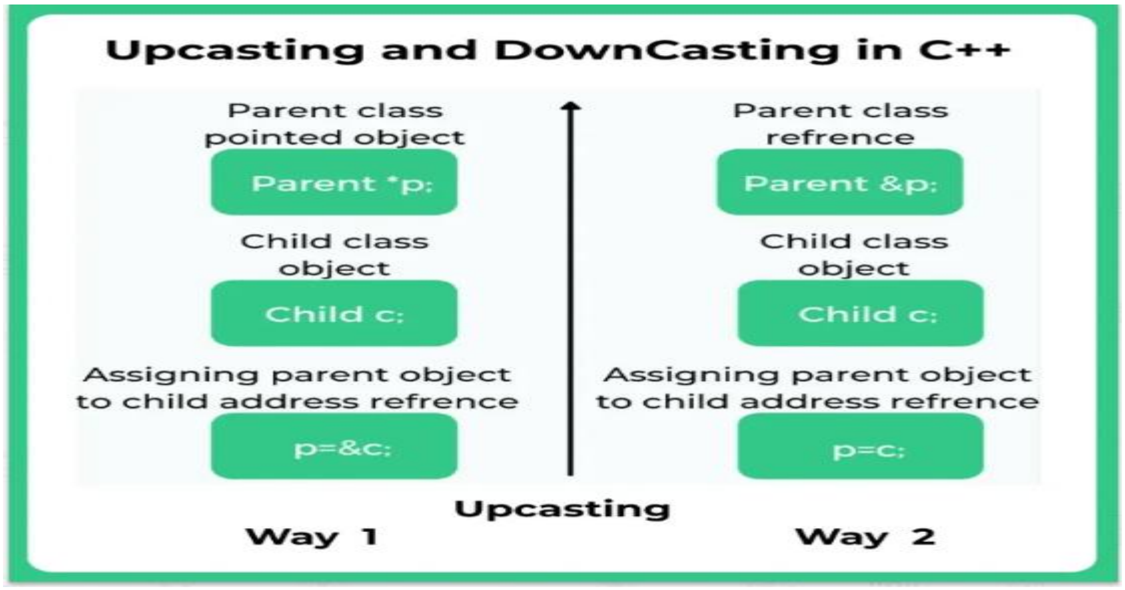
Late binding in the above problem may be solved by using virtual keyword in the base class. Let's see how this happens by using the above example itself, but only adding a virtual keyword.

Code

```
#include <iostream>
using namespace std;
 // Defining the parent class

class Parent
{
    public:
    // virtual keyword added here
    // to force late(run-time) binding
    // to ignore static resolution
    virtual void print()
    {
        cout << "Parent Class" << endl;
    }
};

// Defining the child classs
class Child : public Parent
{
    public:
    // print redeclared in child class
    void print()
    {
        cout << "Child Class" << endl;
    }
};
int main()
{
    Parent *parent = new Child();
    // catch of the program is here
    // also as we are dealing with pointers instead of . we need to use ->
    parent->print();
    return 0;
}
```

What happens really?In this case the virtual keyword added in the base class tells the compiler to not to do the binding at compile time and not force static resolution. But, rather only do the binding at runtime and thus do late binding.

Upcasting and DownCasting in C++


Upcasting and DownCasting in C++
Parent class pointed object
Parent *p;
Child class object
Child c;
Assigning parent object to child address refrence
p=&c;
Parent class refrence
Parent &p;
Child class object
Child c;
Assigning parent object to child address refrence
p=c;
Upcasting
Way 1
Way 2

Upcasting

In simple terms using upcasting allows us to treat the child class object as if it were the parent class object. There are two ways of creating upcasting relationship –

Way 1

Creating parent class pointer and assigning it to the base classes reference.

Example –

```
Parent* parentObj; // Parent class pointer
Child childObj; // Creating child class object
parentObj = &childObj; // assigning to address reference of base class object
```

Way 2

Creating Parent Classes referenced object and assigning it to the child class object

Example –

```
Parent &parentObj; // Parent class reference
Child childObj; // Creating child class object
parentObj = childObj; // direct assignment
```

Facts

- The object is not changing
- However, even with the child class objects we will only be able to access the data and function members of the parent class

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

// This is Parent class
class Parent
{
  public:
    void print()
    {
      cout << "Parent Class printing" << endl;
    }
};
```

```
// This is Child class
class Child : public Parent
{
  public:

  // as we see that it is already declared in the parent function
  void print()
  {
    cout << "Child Class printing" << endl; } }; int main() { Parent *parent_object; Child
child_object; parent_object = &child_object; // catch of the program is here // also as we are
dealing with pointers instead of . we need to use ->
  parent_object->print();

  return 0;
}
```

Now, as you see that even though the object address finally is of the child class, the function accessed is of the parent class this is because of upcasting in C++.

Downcasting

Downcasting is vice versa as of Upcasting. In upcasting we created parent class pointer reference to child class address reference.

We can not do this implicitly thus, we follow explicit form, which is –

```
// doing this will give error
Child *child = &parent;

// We have to use explicit casting as follows
Child *child = (Child) &parent;
```

Let see a program that helps us understand this –

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

// This is Parent class
class Parent
{
public:
void print()
{
cout << "Parent Class printing" << endl;
}
};

// This is Child class
```

```
class Child : public Parent
{
public:

// as we see that it is already declared in the parent function
void print()
{
cout << "Child Class printing" << endl; } }; int main() { Parent *parent_object; Child
*child_object = (Child* ) &parent_object; // catch of the program is here // also as we are
dealing with pointers instead of . we need to use ->
child_object->print();

return 0;
}
```

- In this case even though the final address is of the parent class
- The function called is of the child class
- We have to use explicit casting here

Sometimes, we also use dynamic casting but this is not necessary to learn in the current scope.

Upcasting in C++

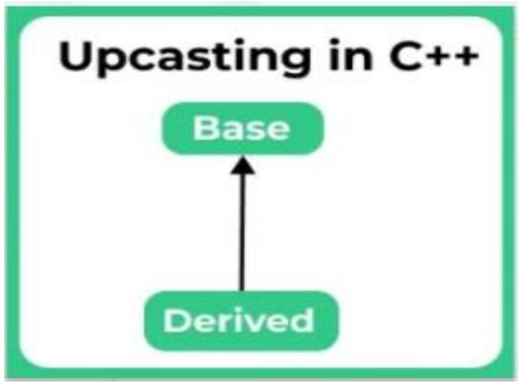
Upcasting allows public inheritance without the need of an explicit type cast.

Upcasting is a process of converting the derived class pointer to base class pointer so that anything that we can do on the base object can also be done on the derived class object.

Both upcasting and downcasting gives it possible to write complex programs with a simple syntax

Upcasting in C++
<u>Steps in upcasting</u>

- Create parent class pointer
- Create an object for child class
- Assign the address of the child object (reference) the parent class pointer
- Invoke methods using child object



Example program to demonstrate upcasting in C ++

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

class parent                                        //parent class
{
public:
void show()
  {
    cout << "Base class\n";
  }
};

class child:public parent       //child class
{

public:
void show()
  {
    cout << "Derived Class\n"; } }; int main() { parent* p; //Base class pointer child c; //Derived
class object p = &c; //assign child reference to parent class pointer c.show(); //Late Binding
Ocuurs p->show();

return 0;
}
```

Upcasting allows treating a derived type as if it was its base type i.e any changes made on derived will also show an effect on the base class

If you do upcasting you can call only access to those properties that are inherited or overridden from base class, child class properties cannot be accessible.

Downcasting in C++

Downcasting is a process of converting a base class reference to a derived class reference

What is Downcasting in C++?
The Downcasting is an opposite process to the upcasting, which converts the base class's pointer or reference to the derived class's pointer or reference. It manually cast the base class's object to the derived class's object, so we must specify the explicit typecast. The downcasting does not follow the **is- a** relation in most of the cases. It is not safe as upcasting. Furthermore, the derived class can add new functionality such as; new data members and class member's functions that use these data members.

Steps in downcasting

- Create objects for parent and child class
- Create child class pointer and assign child reference with explicit typecasting
- Using these pointer call parent and child methods

Example program to demonstrate downcasting in C++

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

class parent                                    //parent class
{
public:
void show()
  {
    cout << "Base class\n";
  }
};

class child:public parent       //child class
{
public:
void display()
  {
    cout << "Derived Class\n"; } }; int main() { parent p; child c; parent *ptr1 = &c; //impilict
type casting allowed child *ptr2 = (child*)&c; //requires explicit cast //upcasting is safe ptr1-
>show();
  ptr2->display();
  //downcasting is unsafe
  ptr2->show();

return 0;
}
```

- Downcasting requires an explicit type conversion because a derived class adds new data members and a class member functions that use this data member wouldn’t be applied to the base class
- Downcasting is not as safe as upcasting because the derived class object can be always treated as the base class object its vice versa is not right, you will get access to the memory that does not have any information of the derived class object .this is the danger with downcasting

Operator Overloading C++

Operator Overloading in C++

- We can use + to concatenate two strings together using an equation like str1 = str2 + str3, and can choose not to use concat() function.
- We can override ++ operator to not increment value by 1. But, rather by 100 possibly.

Why do we need operator overloading?

Operators work with operands. For example, we can use an increment operator with int values or int type variables like count ++.

But, we can’t use these operators with objects, directly.

ProblemConsider we want to use this with object : obj1 as obj1++ that increments all the variables inside the class by 100.

We can't do that simply as obj++ operators don't work with classes.

However, we can overload these operators and give them special meanings to run along with objects and write code to increment the value by 100 for all variables. This is done via operator overloading.

How to define Overloading?

1. We must use a keyword ‘**operator’** before the operator want to overload
2. It works as a function itself thus, a return type must also be defined

Example –

```
void operator ++()
{
  counter1 = counter1 + 1;
  counter2 = counter2 + 1;
}
```

Note – Overloaded operators will only work with objects of the class under which they are defined.

Example

Let us take an example wherein we want to –

- **Overload the ++ operator** to increment all the int variables in the class
- Increment data member variables by 100 not just simple 1 value

```
// Program to overload ++ operator to increment data member
// values by 100 each, everytime ++obj is called using operator overloading
#include <iostream>
using namespace std;

class PrepInsta
{
    int x = 1, y = 2;
    int count1 = 3, count2 = 4;

    public:
    void print()
    {
        cout << x << " " << y << " "
        << count1 << " " << count2 << " respectively"<< endl;
    }

    // ++ operator overloading defined here
    void operator ++()
    {
        x = x + 100;
        y = y + 100;
        count1 = count1 + 100;
        count2 = count2 + 100;
    }
};

int main()
{
    PrepInsta obj;

    cout << "Values initially were -" << endl;
    obj.print();

    // calling overloaded ++ operator on obj object
•       ++obj;
•
•       cout << "\nNew Values are -" << endl;
•       obj.print();
•
•       return 0;
•   }
```

Assignment Operators working with Objects are by default overloaded by C++When we write a class in C++ the compiler by default automatically creates its own version of operator overloading for the assignment operator.

What this essentially does is, consider two objects obj1 and obj2 if we simply want to copy all the data values for obj1 into obj2 we can directly use the assignment operator as obj1 = obj2

It would work the same as copy constructor only it does shallow copy which may cause problems while working with pointers, you can read more about that here

Implicit Conversion via Conversion Constructors as assignment operator is overloaded

Now, the above line may be confusing for a few people. First, we suggest reading the following page –

- Conversion Constructors (read this and then proceed further)

1. The conversion constructors are the ones that can be called with only a single argument.
2. In such cases, if we directly use the assignment operator with an object and value.
3. Can be used for implicit conversion.

Input/Output Operator Overloading in C++

Operator overloading is a type of polymorphism in which an operator is overloaded to give user defined meaning to it. Overloaded operator is used to perform operation on user-defined data type. In this article we will learn more about input/output operator overloading in C++. You can read more about operator overloading by clicking the button below.

Input/Output Operator Overloading in C++

As we know that **cout** is an object of **ostream** class and **cin** is an object **istream** class and it is important to make operator overloading function, a friend of the class because it would be called without creating an object i.e. the operators must be overloaded as a global function.

To call '<<' and '>>' operator, we must call it like, 'cout << obj1' and 'cin >> obj1'. So if we want to make them a member method, then they must be made members of ostream and istream classes, which is not a good option most of the time.

Therefore, these operators are overloaded as global functions with two parameters, cout and object of user defined class.

C++ program to show Input/Output Operator Overloading

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

class Length
{
   private:
      int kmeter;
      int meter;

   public:
      //default constructor.
      Length()
      {
         kmeter = 0;
         meter = 0;
      }
      Length(int km, int m)//overloaded constructor.
      {
         kmeter = km;
         meter = m;
      }
      //overloading '<<' operator.
      friend ostream &operator<<( ostream &output, const Length &l )
      {
         output <<l.kmeter<< "Km "<<l.meter<<"M" ;
         return output;
      }
      //overloading '>>' operator.
      friend istream &operator>>( istream  &input, Length &l )
      {
         input >> l.kmeter >> l.meter;
         return input;
      }
};

int main() {
   Length l1(1, 112), l2;

   cout << "Enter the value: " << endl; cin >> l2;
   cout << "First length: " << l1 << endl;
   cout << "Second length: " << l2 << endl;
   return 0;
}
```

Assignment Operator Overloading in C++

Overloading assignment operator in C++

- Overloading assignment operator in C++ copies all values of one object to another object.
- The object from which values are being copied is known as an instance variable.
- A non-static member function should be used to overload the assignment operator.

The compiler generates the function to overload the assignment operator if the function is not written in the class. The overloading assignment operator can be used to create an object just like the copy constructor. If a new object does not have to be created before the copying occurs, the assignment operator is used, and if the object is created then the copy constructor will come into the picture. Below is a program to explain how the assignment operator overloading works.

C++ program demonstrating assignment operator overloading

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

class Length
{
    private:
        int kmeter;
        int meter;

    public:

        Length() //default constructor
        {
            kmeter = 0;
            meter = 0;
        }
        Length(int km, int m)  //overloaded constructor.
        {
            kmeter= km;
            meter = m;
        }
        void operator = (const Length &l)
        {
            kmeter = l.kmeter;
            meter = l.meter;
        }

        //method to display length.
        void displayLength()
        {
            cout << kmeter << " Kms "<< meter << " meters" << endl;
        }
};

int main()
{
    // Assigning by overloading contructor
    Length len1(1, 100);
    Length len2(2, 200);

    cout <<"Len1 length: ";
    len1.displayLength();

    cout <<"Len2 length: ";
    len2.displayLength();

    // overloading assignment operator to copy values
    len1 = len2;

    cout << "\nLen1 Length: ";
    len1.displayLength();

    cout << "Len2 Length: ";
    len2.displayLength();

    return 0;
}
```

Function Call Operator Overloading in C++

Function call operator overloading is a sub part of operator overloading in C++. On this page, we will learn more about function call operator overloading in C++. But if you want to know more about operator overloading click the button below.

Function call operator overloading in C++

- Function call operator is overloaded by instance of the class and it is also known as function object.

- When we overload function call operator, we are not creating a new way to call a function. But, we are creating an operator function that can be passed a number of parameters.
- It only modifies how the operator is to be fetched by the object.

Syntax for function call operator overloading in C++

```
class class_name
{
    public:
        void operator ()();
};
```

Program to overload function call operator in C++

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

class Length
{
   private:
      int kmeter;
      int meter;


   public:
      Length()//Constructor.
      {
         kmeter = 0;
         meter = 0;
      }
      Length(int km, int m)
      {
         kmeter = km;
         meter = m;
      }


      Length operator()(int x, int y, int z)//Overload function call operator.
      {
         Length l;

         l.kmeter = x + z + 10;
         l.meter = y + z + 100 ;
         return l;
      }
      //Method to display length.
      void disLength() {
         cout << kmeter <<"Km " << meter << "M" << endl;
      }
};

int main()
{
   Length l1(1, 112), l2;

   cout << "First length: ";
   l1.disLength();

   l2 = l1(6, 7, 8); //Invoking function call operator.
   cout << "Second length:";
   l2.disLength();

   return 0;
}
```

Class Member Access Operator Overloading in C++

Class member access operator or member access operator is denoted by "->". Classes are given pointer like behavior using member access operator.

Class member access operator overloading in C++

- It is used to implement "smart pointers."
- Its should return pointer or an object of a class.
- The operator is generally used with pointer dereference operator which makes this combination a smart pointer.
- The derefencing pointer can be defined as a unary postfix operator.

Syntax to overload member access operator in C++

```
class class_name
{
   obj* operator->();
};
```

Program showing class member access operator overloading in C++

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

```
class Naruto {
  static int i, j;

public:
  void f() const { cout << i++ << endl; }
  void g() const { cout << j++ << endl; }
};

//Defining static member.
int Naruto::i = 1;
int Naruto::j = 2;

//Implementing a container for the above class.
class NarutoContainer {
  vector<Naruto*> a;

  public:
    void add(Naruto* naruto) {
      a.push_back(naruto);
    }
    friend class SmartPointer;
};

//Implementing smart pointer to access member of Naruto class.
class SmartPointer {
  NarutoContainer oc;
  int index;

  public:
    SmartPointer(NarutoContainer& narutoc) {
      oc = narutoc;
      index = 0;
    }

    // Returnig value indicates end of list.
    bool operator++() {
      if(index >= oc.a.size()) return false;
      if(oc.a[++index] == 0) return false;
      return true;
    }

    bool operator++(int) {
      return operator++();
    }

    //Overloading class member access operator->
    Naruto* operator->() const {
      if(!oc.a[index]) {
        return (Naruto*)0;
      }

      return oc.a[index];
    }
};

int main() {
  const int sz = 5;
  Naruto o[sz];
  NarutoContainer oc;

  for(int i = 0; i < sz; i++) { oc.add(&o[i]); } SmartPointer sp(oc); do { sp->f(); //Smart pointer
call.
    sp->g();
  } while(sp++);

  return 0;
}
```

Unary Operator Overloading in C++

Unary Operator overloading in C++ allows us to use operators with objects and provide our own unique implementation of how operators interact with objects.

Unary Operator Overloading in C++

- Overloading ++ / — operator is used to increment/decrement multiple member variables at once
- Overloading + operator is used to concatenate two strings.

NoteUnary Operator Overloading in C++ works with only single class object

Syntax:

```
return_type:: operator unary_operator_symbol(parameters)
{
// function definition
}
```

Example

```
void operator ++() // operater overloading function
{
    a = ++a;
    b = ++b;
}
```

Addition/subtraction of Complex numbers using operator overloading

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

class Complex
{
    int a, b, c;
    public:

    Complex(int a1, int b1){ // paramterized constructor
        a = a1;
        b = b1;
    }

    void operator ++(){ //operater overloading function
        a = ++a;
        b = ++b;
    }

    void operator --(){ //operater overloading function
        a = --a;
        b = --b;
    }

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

int main()
{
    Complex obj(20,15);

    ++obj;

    cout << "Increment Complex Number\n";
    obj.display();

    --obj;

    cout << "\nDecrement Complex Number\n";
    obj.display();
    return 0;
}
```

Binary Operator Overloading in C++ (With Example Programs)

Operator overloading is a compile polymorphic technique where a single operator can perform multiple functionalities. As a result, the operator that is overloaded is capable to provide special meaning to the user-defined data types as well. We can overload binary operators like +,*/, – etc to *directly manipulate the object of a class.*

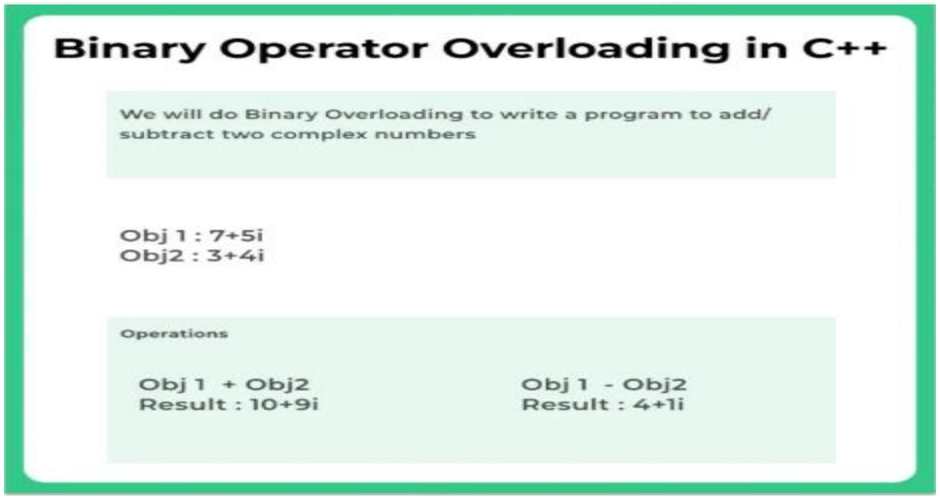
Generally asked as –

Construct a program to illustrate binary operator overloading and explain in detail.

Syntax:

```
return_type::operator binary_operator_symbol(parameters){
    // function definition
}
```

Here the operator is a keyword and binary operator symbol is the operator to be overloaded.


Binary Operator Overloading in C++
We will do Binary Overloading to write a program to add/ subtract two complex numbers
Obj 1 : 7+5i
Obj2 : 3+4i
Operations
Obj 1 + Obj2
Result : 10+9i
Obj 1 - Obj2
Result : 4+1i

Example 1

Objective: C++ Program to Add and subtract two complex numbers using Binary Operator Overloading

Here we will try to write a program and demonstrate how Binary Operator Overloading works –

In the below program we add/subtract two complex numbers

- **Complex Number 1(obj1):** 7 + 5i
- **Complex Number 2(obj2):** 3 + 4i

Operations done –

- **Operation 1:** Obj1 + Obj2
- **Operation 1:** Obj1 – Obj2

```
// Write a program to demonstrate binary operator overloading
#include <iostream>
using namespace std;
class complex
{
    int a, b;
    public:

    void get_data(){
        cout << "Enter the value of Complex Numbers a, b: "; cin >> a >> b;
    }

    complex operator+(complex ob)// overaloded operator function +
    {
        complex t;
        t.a = a + ob.a;
        t.b = b + ob.b;
        return (t);
    }

    complex operator-(complex ob)// overaloded operator function -
    {
        complex t;
        t.a = a - ob.a;
        t.b = b - ob.b;
        return (t);
    }

    void display(){
        cout << a << "+" << b << "i" << "\n";
    }
};

int main()
{
    complex obj1, obj2, result, result1;
    obj1.get_data();
    obj2.get_data();

    result = obj1 + obj2;
    result1 = obj1 - obj2;

    cout << "\n\nInput Values:\n";

•       obj1.display();
•       obj2.display();
•
•       cout << "\nResult:";
•
•       result.display();
•       result1.display();
•
•       return 0;
•   }
```

Relational Operator Overloading in C++

Relational operator overloading in C++ is frequently used to compare many of the built-in data types in C++. We can overload relational operators like >,<,>=etc to *directly manipulate the object of a class.*

Relational Operator Overloading in C++

There are various relational operators supported by C++ language like (<, >, <=, >=, ==, etc.) which can be used to compare C++ built-in data types.

You can overload any of these operators, which can be used to compare the objects of a class.

Syntax:

```
return_type operator symbol(const ClassName& object)
{
  // statements
}
// Here operator is a keyword and relational operator
// symbol is the operator to be overloaded
```

Example:

```
bool operator >(const Student& stObj)
{
  // function definition
}
```

Why Const and & used Since, we are using relational operator. We only will need to compare to two object properties.

This surely should not change the object values. Thus, safe side we are adding const passing by address. So any value, if by mistake change gives error

Example 1

Here we are Overloading > i.e. greater than operator

To know the taller between two students.

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

class Student{
    int feet = 0; //can be b/w 0 & infinity
    int inches = 0; // can be b/w 0 & 12

    public:
    void getHeight(int f, int i){
        feet = f;
        inches = i;
    }

    // const and & added check explanation above(Before code) why
    bool operator > (const Student& s2)
    {
        if(feet > s2.feet)
            return true;

        else if(feet == s2.feet && inches > s2.inches)
            return true;

        return false;
    }
};

int main()
{
    Student s1,s2;

    s1.getHeight(5,10);
    s2.getHeight(6,1);

    if(s1 > s2)
        cout << "Student 1 is taller" << endl; else if(s2 > s1)
        cout << "Student 2 is taller" << endl;
    else
        cout << "Both have equal height" << endl;


    return 0;
}
```

Overloading increment and decrement operators in C++

Overloading increment and decrement operators in C++

- The operator ***that is overloaded is capable to provide special meaning to the user-defined data types as well.***
- We can overload unary operators like ++,–etc to ***directly manipulate the object of a class***

Syntax:

```
return_type::operator increment/decrementoperator_symbol(parameters)
{
// function definition
}
```

Here operator is a keyword, increment/decrement operator symbol is the operator to be overloaded.

Overloading Prefix increment (++) operator

```
#include <iostream>
using namespace std;
class test
{
   private:
   int i;
   public:
   test(): i(0) { }//initialise data memeber
void operator ++()
{
++i;
}

void display()
{
cout << "i:" << i << "\t";
}
};

int main()
{
test t;

// Displays the value of data member i for object t
t.display();

// Invoke the  operator function void operator ++( )
++t;
// Displays the value of data member i for object t
t.display();
return 0;
}
```

Constructor Overloading in C++

Constructor Overloading in C++ is similar to Function Overloading, we can define more than one constructors for a class. It is defined by naming the constructor same as the name of the class.

Constructor Overloading in C++

The following are necessary conditions for the same –

- Name of constructor should be same as the name of the class
- The parameters passed in each variation of constructor should be different example –
  - Demo(int x) and Demo(int x, int y)
- Constructors are basically special functions of the class itself.

Implementation for Constructor Overloading

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

class MyClass {
   private:
      int x, y;

   public:
      MyClass() {
         x = 0;
         y = 0;
      }

      MyClass(int a) {
         x = a;
         y = 0;
      }

      MyClass(int a, int b) {
         x = a;
         y = b;
      }
```

```
        void print() {
            std::cout << "x = " << x << ", y = " << y << std::endl;
        }
};

int main() {
    MyClass obj1;
    MyClass obj2(5);
    MyClass obj3(3, 7);

    obj1.print();
    obj2.print();
    obj3.print();

    return 0;
}
```

OOPs Advanced – 2

Virtual Function in C++

Virtual Function in C++

Virtual functions give programmer capability to call a member function of a different class by the same function call based on a different situation.

- In other words, a virtual function is a function which gets override in a derived class and instructs c++ compiler for executing late binding on that function
- A function call is resolved at runtime in late binding and so that compiler determines the type of an object at runtime.

Virtual function syntax

```
class class_name
{
  access:
  virtual return fucntion_name(arg...)
  {
     ---;
  }
};
```

Advantage of virtual functions

Ambiguity in function calls is reduced because It can call all the required functions from parent and child class in a different context

Some points to note about virtual functions

- Members of some class virtual functions cannot be static
- The virtual function can be defined in the parent class even if it is not used
- The declaration of the virtual function of parent class in the child class must be identical, if the prototype is not identical c++ will consider them as overloaded functions
- A virtual constructor is not possible whereas a virtual destructor is possible

C++ program to demonstrate the usage of virtual keyword

Example 1:

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

class parent//parent class
{
  public:
  virtual void show()
  {
    cout << "Base class\n";
  }
};

class child:public parent//child class
{
  public:  void show()
   {
     cout << "Derived Class";
   }
};

int main()
{
  parent* p; //Base class pointer
  child c; //Derived class object
  p = &c;
  c.show(); //Late Binding Occurs
}
```

Abstract Class in C++

Abstract classes are special C++ Classes which are declared but they can not be initialised, i.e. in other words you can not create the objects for the same. However, you can derive other child classes from the abstract class in C++ and then create the objects.

- Some website refer the definition as an abstract being a class which has atleast one Pure Virtual Function in it. **Now, that is incorrect**.
- Pure Virtual functions are one way from which we can fulfil the implementation of abstract class
- But, there are other methods too in C++, which are too advanced for the topic of discussion.

For layman terms you can though consider that an abstract class can have atleast one Pure Virtual function (Don't worry we will learn what Pure Virtual Functions are, in this post)

Why we may need Abstract Classes in C++?

1. Sometimes, from a programmers point of view he may just only want to create a class for better visualization and not use to store data or use functions
2. This may also help in reducing the reducing the size of code is architecture is good

To achieve this C++ has the concept of pure virtual function and abstract classes.

For example, shape class which may be necessary to define data members that can be inherited by Rectangle or Circle class. But, the shape class can't have the definition of the calculateArea() function as we don't have specific information. But, the derived class will have their own implementation of calculateArea().

Must have feature of Abstract Classes

- We can only declare an abstract class, but can instantiate it.
- However, creation of pointed references is allowed, to support Runtime polymorphism
- Derived class generally use abstract classes to create interfaces
- They are used for upcasting

- If we use Pure Virtual functions to implement derived class then, we must implement all those functions in the derived classes without fail.
- Else by inheritance they will be abstract classes too.

Pure Virtual Functions

You must already know what Virtual functions are, they are used to support Runtime polymorphism and avoid early binding caused by static resolution at compile time.

- Any function in the base class which is preceded with keyword virtual are essentially Virtual functions.
- Pure Virtual functions are extension to Virtual functions and help us implement the concept of abstract classes properly.

The following is condition for a function to be a virtual function –

- should end with = 0;

For a virtual function be to pure virtual function it must have virtual keyword as

```
virtual void start() = 0
```

Let's have a look at the whole program and understand the same.

```
class Vehicle
{
public:
virtual void features() = 0;
};
```

Example of a virtual function would be –

```
#include<iostream>
using namespace std;
class Parent
{
  public:
  virtual void print() = 0;
};
class Child : public Parent
{
  public:
  void print()
  {
    cout<<"Printing from Child(derived) class\n";
  }
};
```

```
int main()
{
  // we will try to implement this with both
  //pointer references and normal child class objects
  Child child1;
  Parent *parent;
  Child child2;
  parent = &child2;
  child1.print();
  parent->print();
  return 0;
}
```

Pure Virtual Function in C++

Concept of Pure Virtual Function in C++

This was long demanded feature by programmers using other object-oriented languages –

<u>What programmers wanted?</u>

1. Programmers most of the time wanted to just give **good architecture** of the code and **create classes with no implementation**.
2. But, to **fully use the inheritance concept** without running the troubles of different instances of RunTime vs Compile time alternations in how code is performed.

<u>How it was solved?</u>

1. With **Pure Virtual functions**, this problem was solved as it would guarantee the same result.
2. That is implementation of the **code will always be performed by Derived (Child) classes** and they need only be declared in Base (Parent) classes.

Abstract Class

- An abstract class is any class that just has declarations but no initialisations.
- These initialisations and implementation details can be provided in the inherited derived classes.
- Pure Virtual function helps us in achieving abstract classes.

We create a pure virtual function in the base class by –

- Using virtual keyword before the function in the base class
- Assigning it to 0

```
// An abstract class
class Test
{
   // Data members of class
public:
   // Pure Virtual Function
   virtual void show() = 0;

  /* Other members */
};
```

Note – We can't create objects of Abstract Classes or classes that contain Pure Virtual Functions.

Example of Pure Virtual Function Implementation
See how a pure virtual function without any implementation is created in the base class.

The result of the pointed object reference and derived class both are the same. They both call derived class function

2. Creating Objects of Base Class (Pure Virtual Function case)

We can't create a direct object of base class containing pure virtual function which is an abstract class.

We must create objects by –

1. Derived class objects
2. Or pointer reference of base class pointed towards the object of the derived class.

This was shown above in the first example.

Creating the Object of Base class will give an error. Example of this is show below –

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

class Base
{
   public:
   virtual void display() = 0;
};
class Derived : public Base
{
    public:
    void display(){
        cout << "Inside the Derived Class\n";
    }
};
int main()
{
    // IMPORTANT
    // this will cause error as we have created
    // object of base class
    Base base;
    base.display();

    // this alone wont have created any errors
    Derived derived;
    derived.display();

    // this also won't have created any errors
    // base class pointer referencing derived class object
    Base *base;
    Derived derived2;
    base = &derived2;
    base->display();

    return 0;
}
```

3. Overriding base class Pure Virtual Function

Overriding the base class pure virtual function is necessary as if we don’t do that. Then, the derived class also is classified as an abstract class let us see what error comes if we do this –

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

class Base
{
    public:
    virtual void display() = 0;
};
class Derived : public Base
{
    public:
    // not defining base class virtual function (pure) here
    // thus overriding doesn't happen
    // error
};

int main()
{
    // this will cause error as we have created
    // object of base class
    Derived derived;
    derived.display();

    return 0;
}
```

4. The base class can have it’s own constructor

Let’s see an example for this –

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

class Base
{
    public:
    int x, y;

    virtual void display() = 0;
    // base class can have its own constructor
    // even though we won't be able to create
    // objects of base class
    Base(){
        x=1; y=1;
    }
};

class Derived : public Base
{
    public:
    Derived(){
        x=10; y=10;
    }
    void display()
    {
        cout << "Values are -\n" << x << " and " << y;
}
};

int main()
{
    Derived derived;
    derived.display();

    return 0;
}
```

5. Illegal definition of Pure Virtual Function

Inline Definition of Pure Virtual Functions is illegal. But, still, you can define it outside the class

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

class Base
{
    public:
    virtual void display() = 0;
};

// Pure Virtual definition, can define
// and wont cause error but wont be printed
void Base :: display()
{
    cout << "Pure Virtual definition\n";
}

class Derived : public Base
{
    public:
    void display(){
        cout << "Inside the Derived Class\n";
   }
};
int main()
{
    Derived derived1;
    derived1.display();

    return 0;
}
```

Virtual function v/s Pure Virtual function in C++

The main difference between virtual function and pure virtual function is that virtual function has its definition in the parent class and the child class which is inheriting it can redefine the definition whereas ***a pure virtual function will not have any definition and all the inherited child classes must give a new definition***

Let’s discuss the top five differences between virtual function and pure virtual function and establish a clear distinction and purpose of each

Virtual Function

Definition

A virtual function has its ***definition in the base class***

Declaration

```
virtual funct_name(parameter_list)
{. . . . . ;
}
```

Child classes

All the child classes ***may or may not redefine the virtual function definition of the parent class***

Effect

They are hierarchical in nature and do not affect the compilation process ***if any of the child class do not override the virtual function of a parent class then it’s allowed***

Abstract class

abstract class is not possible

Example program

```
#include<iostream>
using namespace std;
class B {
public:
  virtual void m()//virtual function
  {
    cout<<" In Child \n";
  }
};
class D: public B {
  public:
  void m() {
    cout<<"In Child \n";
  }
};
int main()
{
  D d; // An object of class D
  B *b= &d;// A pointer of type B* pointing to d
  b->m();// prints"D::m() called
}
```

Pure Virtual Function

Definition

Pure virtual functions ***will not have any definition in the base class***

Declaration

```
virtual funct_name(parameter_list)=0;
```

Child class

All ***child classes must override the virtual function of the parent class***

Effect

If ***any of the child class failed to override the virtual function of the base class then a compilation error will occur***

Abstract class

If a ***class contains at least one pure virtual function then it is abstract***

Example program

```
#include<iostream>
using namespace std;
class B {
  public:
  virtual void m() = 0; // Pure Virtual Function
};

class D:public B {
  public:
  void s() {
    cout << " Virtual Function in child class\n";
  }
};

int main() {
  B *b;
  D dobj; // An object of class D
  b = &dobj;// A pointer of type B* pointing to dobj
  b->s();// prints D::s() called
}
```

Virtual Base Class in C++

Why we need virtual base Classes?

The below program will give error

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

class Father : public GrandParent
{
    //print function inherited from GrandParent
};

class Mother : public GrandParent
```

```
{
   //print function inherited from GrandParent
};

//multiple inheritance
class Child : public Father,public Mother
{
   //print function inherited two times from Father & Mother both
   //creating error and ambiguity.

};

int main()
{
   Child c;

   c.print(); // will generate error here
   //print function is inherited two times from Grandparent via Father and mother both
   //This causes ambiguity for the compiler

   return 0;
}
```

Why the error is there?

The error is caused as

- Both mother and father classes inherit **print function** from GrandFather class.
- Child class inherits the print function twice from both mother and father
- This creates two instances of same function in child class which causes ambiguity in compiler on which function to call and is disallowed as an error.

How to solve the below issue?

Check the below example to understand, all we will be changing is adding keyword virtual before classes Father and mother

NoteVirtual base class concept is different from Virtual Keyword in Polymorphism

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

class GrandParent
{
   public:
   void print()
   {
      cout<<"Hello" << endl;
   }
};

class Father : virtual public GrandParent
{
   //print function inherited from GrandParent
};

class Mother : virtual public GrandParent
{
   //print function inherited from GrandParent
};

//multiple inheritance
class Child : public Father,public Mother
{
   //print function only inherited once as both mother and father
   //have virtual keywords before them

};

int main()
{
   Child c;

   c.print();
   //No Ambiguity now

   return 0;
}
```

Exception Handling in C++

Exception handling is performed in C++ using **try**, **catch** and **throw**. These help in making sure that the whole program runs completely, even if some minute runtime errors that may occur due to logical, boundary and other problem in the program.

Example :
If at run time an arithmetic operation happening has operational input that is dividing by 0.

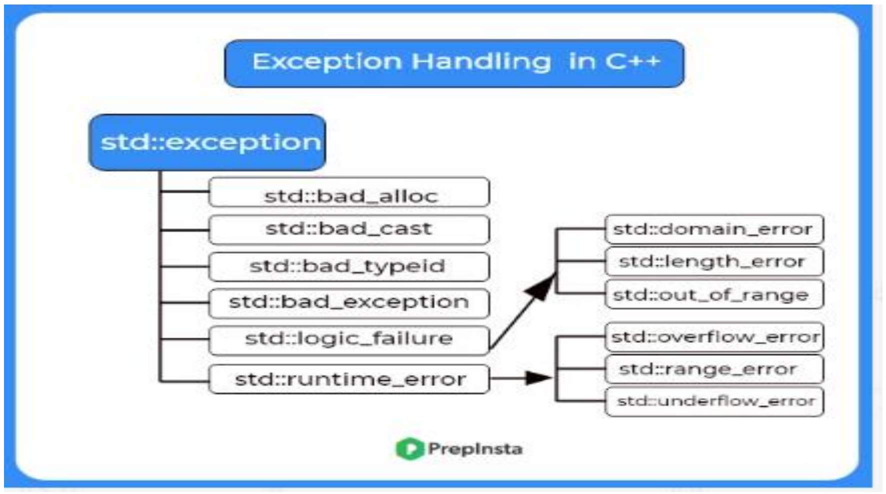
When we talk about errors there are the following two majorly –

1. **Run Time errors –** These are called **exceptions** that we can handle using try, catch and throw keywords. (Ex – Dividing by 0)

2. **Compile time errors –** These are the error that stop the program at compile time itself. (Example – Using string without including string library in the program headers)

For Exceptions there are again two different types which are –

- **Synchronous –** Caused by how the program is written logically.
- **Asynchronous –** Which happens because of reasons which are beyond programmers control. Example Disk Failure



How this works

1. Try – This is the part of code, where a programmer will assume an error to occur. The programmer will surround the code inside a try block.
2. Catch – This is the part of the code, where the programmer is catching and identifying which exception has occurred and is trying to handle it.
3. Throw – While the system may notify which type of exception has occurred. Sometimes, logically you may want to notify an error that is specific to your code logic. These is where you throw exception.

Let's look at the basic structure of try and catch, which is essentially you trying a code that may throw an error and catch is capturing the type of error that has occurred.

```
try {
  // code that is tried
} catch( NameOfException e1 ) {
  // catch block,
} catch(NameOfException e2 ) {
  // catch block
} catch(NameOfException eN ) {
  // catch block
}
```

Example –

Let’s write a code for division by 0

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

double divide(int value1, int value2) {
   if( value2 == 0 ) {
      throw "Exception happens division by 0";
   }
   return (value1/value2);
}

int main () {
   int a = 200;
   int b = 0;
   double temp;

   try {
      temp = divide(a, b);
      cout << temp << endl;
   } catch (const char* message) {
     cerr << message << endl;
   }

   return 0;
}
```

Program Flow

It is very important to understand the flow of the program –

- Try is executed
  - If Exception occurs or throw statement occurs
  - Immediate jump to catch statement, which then is executed.

- If no exception or thor statement
    - Then catch statement is never executed.

The following program will help you understand the flow –

```
#include <iostream>
using namespace std;
int main()
{
  int temp = 0;

  cout << "(Here 1) Currently before the try statement" <<endl;
  try
  {
    cout << "(Here 2)Currently inside the try statement" << endl;
    if (temp == 0)
    {
      throw temp;
      cout << "(Here 0) Since, this is after throw statement, flow jumps to catch and this never
executed" << endl;
    }
  }
  catch (int x ) {
    cout << "(Here 3) Flow transferred to catch block" <<endl;
  }

  cout << "(Here 4) Try catch and throw blocks executed back to int main block";
  return 0;
}
```

Output -

```
(Here 1) Currently before the try statement
(Here 2)Currently inside the try statement
(Here 3) Flow transferred to catch block
(Here 4) Try catch and throw blocks executed back to int main block
```

Catch All

There is a catch all exception which is catch(…) keyword in block. Let us look how it works.

```
#include <iostream>
using namespace std;
int main()
{
  try
  {
    throw "PrepInsta";
  }
  catch (int x)
  {
    cout << "Integer was caught " << x;
  }
  catch (...)
  {
    cout << "Default Exception was called here\n" <<endl;
  }
  return 0;
}
```

Templates in C++

Template is a mechanism where one function or class operates on different data types instead of creating separate class/function for each type

Function templates

Templates applied with functions are **function templates**

Demerits of overloaded function

- Program consumes more disk space
- Time consuming as the function body is replicated with different data types
- Presence of error in one function needs correction in all overloaded functions

Merits of Templates

- Source Code reusability which in turn promotes readability, reduces redundancy, and increases code flexibility
- A single name for function template can be represented for many data types

Syntax

```
return type function name([template-name var1, template-name var2])
{
  Body;
}
```

- Here template name act as template argument type

- Template argument is substituted whenever a specific data type is declared, we substitute the template argument

Type 1:

```
#include <iostream>

using namespace std;

template

T sum( T x, T y)

{

    return x + y;

}

int main()

{

               cout << "Sum : " << sum(3, 5) << endl;

                cout << "Sum : " << sum(3.0, 5.2) << endl;

                cout << "Sum : " << sum(3.24234, 5.24144) << endl;

                return 0;

}
```

Class Template

- Templates can be extended at class level and creating class templates is similar to that of function templates
- Class templates are generally used for data storage classes [containers]

Syntax

```
Template
class
{
  body;
};
```

Difference between function and class templates

- Class Template argument can be used at anywhere in the class specification where as function template argument can be used only within the function specification when there is a reference to the types
- Function templates are instantiated when a function call is encountered where as class templates are instantiated by defining an object using template argument
- Class templates can be used by all member functions of a class

```
class-nameobject ;
#include<iostream>
using namespace std;
template
class Student{
  T marks1;
  T marks2;
  public:
  Student( T m1, T m2 )
  {
    marks1 = m1;
    marks2 = m2;
  }
  T totalMarks()
  {
    return marks1 + marks2;
  }
  T percent()
  {
    return (totalMarks()/200)*100;
  };
int main()
{
  Student s1 ( 48 59 );
  Student s2 ( 65.5, 96.4 );
  cout << "Total marks of student1 : " << s1.totalMarks() << endl;
  cout << "Total marks of student2 : " << s2.totalMarks() << endl;
  cout << "percent of student1 : " << s1.percent() << endl;
  cout << "percent of student2 : " << s2.percent() << endl;
  return 0;
}
```

Interfaces in C++

Interfaces in C++

- Sometimes you need not disclose all the details(implementation) of your project to the user i.e just need to present him the set of services in the form of options in an interface.
- For example, in an ATM machine, you will be shown the list of options provided by that Bank that is withdrawal, deposit but not the implementation of backend code of those services
- This Hiding of implememntataion details wherever required is called as an interface

Syntax

Interfaces are implemented using abstract classes, a class is made abstract by ***declaring at least one of its functions as a pure virtual function***, the pure virtual function must be declared under public access

```
class class_name
{
  public:
  // pure virtual function
  virtual return-type func_name() = 0;
};
```

A pure virtual function is specified by using (=0) in its declaration

Example program demonstrating interfaces in C++

Consider the following example where parent class provides an interface to the child class to implement a function called getArea()

```
#include<iostream>
using namespace std;
// parent class
class Shape
{
  public:
  // pure virtual function providing interface framework.
  virtual int getArea() = 0;
  void setWidth(int w) {
    width = w;
  }
  void setHeight(int h) {
   height = h;
  }
  protected:
  int width;
  int height;
};
// child classes for implementation
```

```
class Rectangle: public Shape {
  public:
  int getArea() {
   return (width * height);
  }
};
// child classes for implementation
class Triangle: public Shape {
  public:
  int getArea() {
    return (width * height)/2;
  }
};
int main()
{
  Rectangle Rect;
  Triangle Tri;
  Rect.setWidth();
  Rect.setHeight(4);
  // display the area of the object.
  cout << "Total Rectangle area: " << Rect.getArea() << endl;

  Tri.setWidth(8);
  Tri.setHeight(4);

  // Print the area of the object.
  cout << "Total Triangle area: " << Tri.getArea() << endl;
  return 0;
}
```

Point to remember about abstract classes

- An abstract class cannot be used and ***not allowed to create objects***, just it is used as an interface if you try to create an object of an abstract class that it was a compile time error
- ***A child class that is implementing a parent abstract class must define all the pure virtual functions of parent class in the child*** then only child class can be allowed to create objects and use them

Why interfaces

- In OOP's design system an abstract base class will provide a ***common generalized interface that is appropriate and can be used for all external applications of similar type***
- Interface mechanism ***allows applications to add new software updates and other changes to the existing system easily*** even after a software system has been defined

Files and Streams in C++

There are 3 standard file handling streams as follows

1. ***Ofstream***: It is an output file stream which is used to ***create files for writing data*** to the files
2. ***Ifstream:*** it represents input file stream it is used for ***reading data from the files***
3. ***fstream***: It is viewed as a combination of both ofstream and ifstream i.e. it has ***capabilities of both ofstream and ifstream***

All the stream classes are specially designed to manipulate disk files

Modes for file opening

File modes will tell ***what is the purpose of opening a file i.e whether you need to read or write, or what operation*** you need to perform on that file, etc.

- ***ios::app***: Append mode
- ***ios::ate***: Open a file for output and read/write control to the end of the file
- ***ios::in***: Open file for reading
- ***ios::out:*** Open file for writing
- ***ios::trunk***: When any file already exists, its existing contents will be deleted before file opening

Opening and closing a file in C plus plus

Before ***working with file programs in C++ the programmer must have clarity about the following things***

- A ***name*** to be given for the file
- ***Data type and structure*** of the file
- Purpose ***(reading, writing data***)
- Opening method(***file mode***)
- ***Closing the file*** (after completing all operations)

<u>Opening a file</u>

- Before manipulating and performing operations on that file the general operation that is to be performed is opening that particular file
- A file is opened using a stream and any input and output operation performed in the stream will be also applied to the physical file that it is pointing to

In this program, the file that we are working with must be created in the directory and it is capable of performing both read and write operations

Example

```
#include <bits/stdc++.h>
using namespace std;
int main()
{
  fstream file;
  file.open ("example.txt", ios::out | ios::in );
  return 0;
}
```

Closing a file

C++ automatically terminates all the file resources and streams associated and closes all open files when the program terminates but it’s always a good programming practice to explicitly close the files

```
#include <bits/stdc++.h>
using namespace std;
int main()
{
  ofstream file;
  file.open ("prepinsta.txt");
  file.close();
  return 0;
}
```

Mostly used functions for file handling in C plus plus

- ***open()***: To ***create a file***
- ***close()****: To **close an existing file***
- ***get()***: To ***read a single character*** from the file
- ***put()***: To ***write a single character*** in the file
- ***read()***: To ***read data from*** a file
- ***write()***: To ***write data int***o a file

C++ program to demonstrate for reading from and writing to a file

We use << and >> to write and read from a file respectively. Let’s see an example.

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

int  main()
{

  char text[200];

  fstream file;
  file.open ("prepinsta.txt", ios::out | ios::in );

  cout << "Write text to be written on file." << endl;
  cin.getline(text, sizeof(text));

  // Writing to file
  file << text << endl;

  // Reding from file
  file >> text;
  cout << text << endl;

  //closing the file
  file.close();
  return 0;
}
```

Dynamic memory

Dynamic Memory Allocation in C++

Allocation of memory at run time based on the requirement of the user is called dynamic memory allocation.Dynamic memory allocation helps us to avoid wastage of storage and hence it is very important to understand it.

Dynamic Memory Allocation in C++

- Dynamic memory allocation can be done using new operator in C++
- In all the Dynamic Memory, allocation happens in Heap Area as *heap can extend or shrink at any time*

New operator

Using the new keyword, the required number of bytes of memory is allocated at run time.

Syntax:

```
data_type *ptr=new datatype(value);
```

Example

```
int *p=new int(100);//4 bytes allocated
```

Delete operator

The memory allocated using new can be *released and taken by at any time* in between the program using ***delete*** keyword

Syntax:

```
 delete ptr; // Release memory pointed by pointer-variable
```

Here, ptr is the pointer that points to the data object created by new. Example:

```
delete p;
```

To free the dynamically allocated array pointed by pointer-variable:

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

Example:

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

Example program to demonstrate new and delete

```
/* program to demonstrate new and delete*/
#include  <iostream>
using namespace std;
int main()
{
   int *p;
   float *q;
   char *r;

   p=new int(10);//allocates 2 bytes and the pass the address to p
   q=new float(1.5);//allocates 4 bytes and the pass the address to q
   r=new char('x');//allocates 1 bytes and the pass the address to r

   cout<<"*p="<<*p<<" *q="<<*q<<" *r="<<*r<<endl;;
   delete p;// release the memory allocated to p
   cout<<"*p="<<*p<<" *q="<<*q<<" *r="<<*r<<endl;;

return 0;
}
```

In the above program, 2 bytes are allocated and the value of 10 is stored and the address passed to ‘p’, similarly for ‘q’ and ‘r’. in the end, p is deleted i.e the memory is taken back, so ‘p’ no more points to that address .hence we get a garbage value if we access any memory after deleting it

Normal Array Declaration vs Using new

- Normal arrays are deallocated by the compiler (If the array is local, then deallocated when the function returns or completes).
- However, dynamically allocated arrays always remain there until either they are deallocated by programmer or program terminates.

Program to demonstrate array creation using new

```
/* Program to demostrate array creation using new */
#include <iostream>
using namespace std;

int main()
{
  int n;
  cout << "how many elements?:"; cin >> n;

  int *p = new int[n];                    // creates n bytes in heap
  cout << "\nenter elelments:";

  for (int i = 0; i < n; i++) cin >> p[i];

  cout << "\nelements are:";

  for (int i = 0; i < n; i++)

  cout << *(p + i) << "\t";

  delete[]p; //release the memory

  cout <<  "\n" <<  *(p + 0) << "\n";   //garbage value
  cout << *(p + 2) << "\n";   //garbage value

 return 0;
}
```

What happens if enough memory is not available at runtime?

- In rare cases required memory may be available at runtime In such case program may terminate abnormally
- To avoid this we can make use of an exception defined in ***std::bad_alloc***, in which case it returns a NULL pointer and a user-defined message can be displayed to avoid confusion to the programmer
- Therefore, it may be a good idea to check for the pointer variable whether the required memory is available or not

```
int *p = new(nothrow) int;
if (!p)
{
cout << "Sorry:Memory not avaialble to allocate\n";
}
```

Need for dynamic memory allocation

Static Memory Allocation

- Allocating memory at design time or compile time is known as Static memory allocation.
- In Static memory allocation Memory once allocated ***neither extended or deleted,***
- Due to which there is a chance of wastage or shortage of memory
- Non-static and local variables get memory allocated on Stack Area

Example:

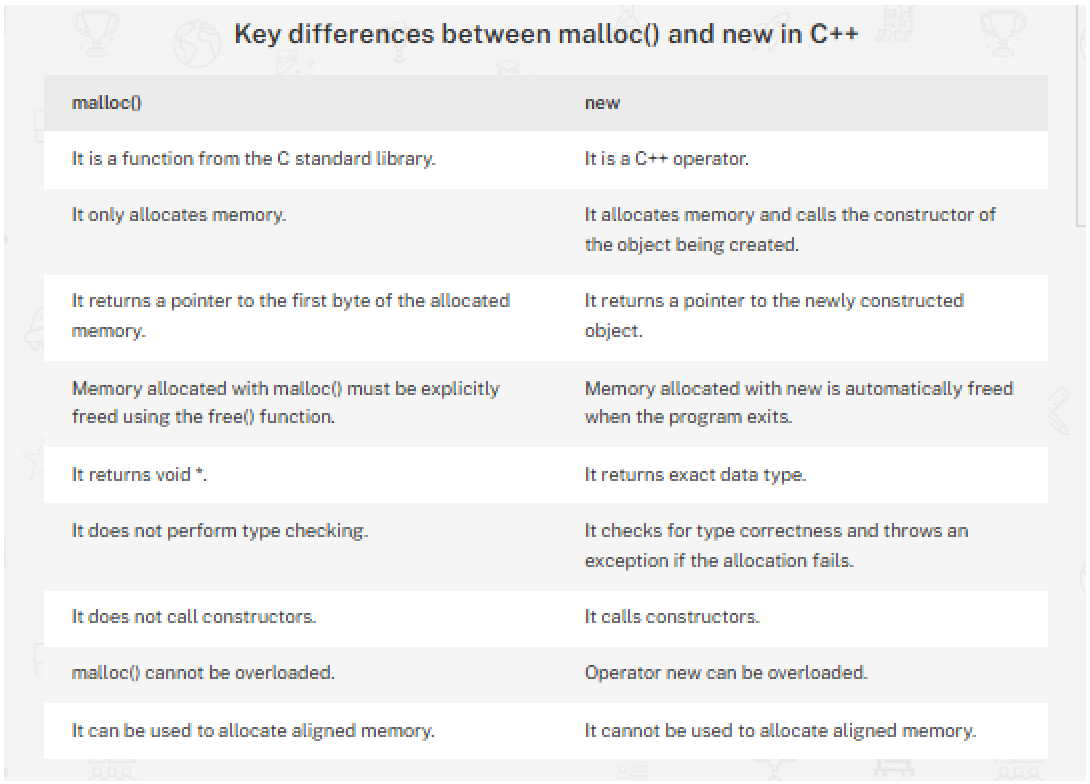
```
int a[5];//allocates 5xsize(int)=10bytes
int c[5]={1,2,3};//4 bytes wasted as space for 2 elements is unused
int b[2]={1,2,3};//need 4 bytes more
```

- Static memory ***once allocated cant be deleted*** by the programmer in between, it is deleted only by the compiler at the end
- In large Programs in real time projects after usage giving the memory back and ***having free space would enhance the performance***

C++ malloc() vs new

In C++, malloc() and new are similar in that they both dynamically allocate memory on the heap, but new also constructs an object in that memory and has additional functionality.

Here we will discuss the key difference between malloc() and operator new.



Key differences between malloc() and new in C++

malloc()	new
It is a function from the C standard library.	It is a C++ operator.
It only allocates memory.	It allocates memory and calls the constructor of the object being created.
It returns a pointer to the first byte of the allocated memory.	It returns a pointer to the newly constructed object.
Memory allocated with malloc() must be explicitly freed using the free() function.	Memory allocated with new is automatically freed when the program exits.
It returns void *.	It returns exact data type.
It does not perform type checking.	It checks for type correctness and throws an exception if the allocation fails.
It does not call constructors.	It calls constructors.
malloc() cannot be overloaded.	Operator new can be overloaded.
It can be used to allocate aligned memory.	It cannot be used to allocate aligned memory.

Note: In summary, malloc() and new both can be used to allocate memory dynamically in C++, but new provides automatic memory management and does not require manual calculation of memory size, making it easier to use and less error-prone than malloc().

Example of malloc()

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

int main()
{
  int *p = (int *)malloc(sizeof(int));     // Allocate memory for an integer
  if (p == NULL)
    {
      cout << "Memory allocation failed\n";
      return 1;
    }

  *p = 5;                                  // Assign a value to the memory
  cout << "The value of the integer is: " << *p << endl;

  free(p);                                 // Free the allocated memory

  return 0;
}
```

Example of new

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

class Test
{
public:
  Test(int x)
  {
    value = x;
    cout << "Test object created with value " << value << endl;
  }
   ~Test()
  {
    cout << "Test object destroyed with value " << value << endl;
  }
  int getValue()
  {
    return value;
  }

private:
  int value;
};

int main()
{
  Test *p = new Test(5);  // Allocate memory for Test object and call constructor

  cout << "The value of the Test object is: " << p->getValue() << endl;

  delete p;                  // Free the allocated memory and call destructor

  return 0;
}
```

New keyword in C++

New keyword in C++ is a very helpful way to improve a program. In case of static memory allocation once the memory is allocated it cannot be deleted and unused memory gets wasted whereas dynamic memory allocation enables the programmer to manually create and delete (give back to store) the memory at any point of time

New keyword in C++

Dynamic Memory Allocation: The advantage of dynamic memory mechanism used to locate memory at the time of execution without any wastage of memory block.

In C++ dynamic memory concept is supported by new and delete operators, here we will discuss the new keyword in C++ in detail.

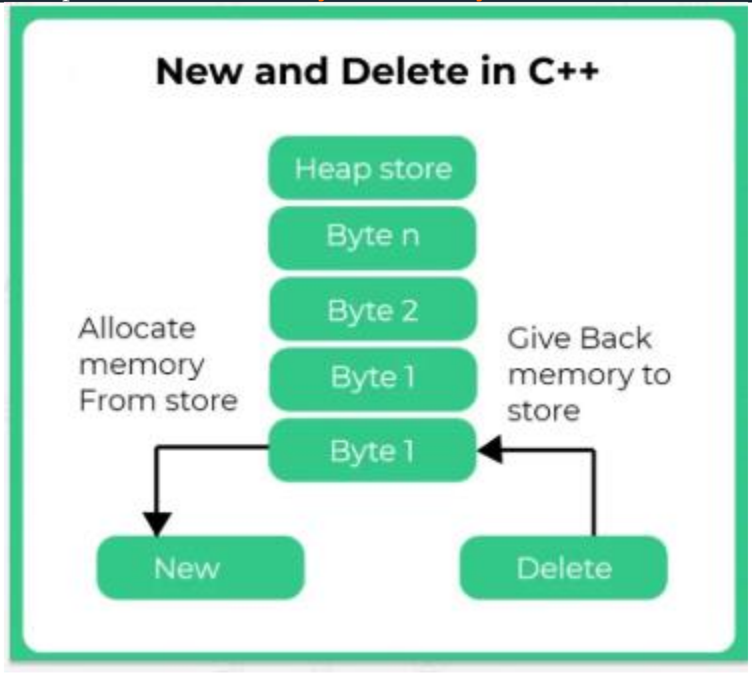
New: The new operator allocates the required bytes of memory by returning a pointer to the allocated memory

Syntax

```
datatype *varname=new datatype[size]
```

Example

```
int *p =new int[7] //14 bytes= 7 *2 bytes
```



- For example, if the size of the integer is 2, 14 bytes of memory is returned by the new operator for n specified as 7
- initially pointer variable p contents of base address i.e first byte of memory

C++ program to create memory dynamically to store some data and display the data

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

int main()
{
  int *p, i, n;

  cout << "enter the size:";
  cin >> n;

  p = new int[n];                    //allocates  n*sizeof(int)  bytes
    if (p == NULL)                              //in rare case if available memory not available
    {
        cout << "\nmemory alloacation failed";
        exit (1);
    }

  cout << "\nenter the elements\n";

  for (int i = 0; i < n; i++)
   {
      cin >> *(p + i);
   }
  cout << "elements are:\n";

  for (int i = 0; i < n; i++)
   {
      cout << *(p + i) << "\t";
   }
return 0;
}
```

Heap memory

Dynamic memory allocation always takes place in a memory area called heap, because heap Data Structure has a property to grow and shrink at any point of time i.e memory can be deleted and allocated at any point of time as per the requirement.

This Pointer

Each individual object in C++ is given access to its own address via an important pointer, which is known as **this** pointer. The this pointer can be defined as an implicit parameter to all the member functions. Hence, when used within a member function, it may be used to refer to the invoking object.

Friend functions are not given access to a **this** pointer. The reason behind this is that friends are not members of a class. Only member functions are allowed to have a **this** pointer.

```
#include <iostream>

using namespace std;

class Room {
  public:
    // Constructor is being defined here
    Room(float a = 5.0, float b = 4.0) {
      cout <<"Constructor called." << endl;
      length = a;
      breadth = b;
    }
    float Area() {
      return length * breadth;
    }
    int Check(Room room) {
      return this->Area() > box.Volume();
    }

  private:
    float length;
    float breadth;
};

int main(void) {
  Room Room1(2.5, 2.1, 5.2);    // Room1 Declaration
  Room Room2(7.0, 5.0, 4.4);    // Room2 Declaration

  if(Room1.compare(Room2)) {
    cout << "Room2 is smaller than Room1" <<endl;
  } else {
    cout << "Room2 is equal to or larger than Room1" <<endl;
  }

  return 0;
}
```