Object Oriented Programming in C++

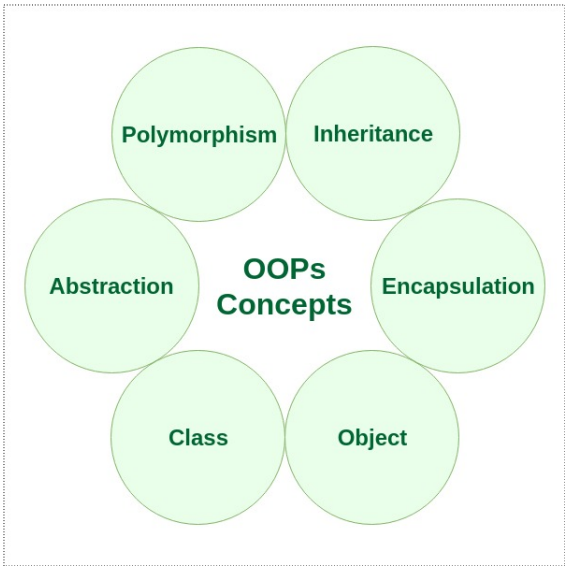
Difficulty Level : Easy • Last Updated : 11 Mar, 2023

Object-oriented programming – As the name suggests uses objects in programming. Object-oriented programming aims to implement real-world entities like inheritance, hiding, polymorphism, etc. in programming. The main aim of OOP is to bind together the data and the functions that operate on them so that no other part of the code can access this data except that function.

There are some basic concepts that act as the building blocks of OOPs i.e.

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

Characteristics of an Object-Oriented Programming Language



Class

The building block of C++ that leads to Object-Oriented programming is a Class. It is a user-defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class. A class is like a blueprint for an object. For Example: Consider the Class of Cars. There may be many cars with different names and brands but all of them will share some common properties like all of them will have 4 wheels, Speed Limit, Mileage range, etc. So here, the Car is the class, and wheels, speed limits, and mileage are their properties.

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

We can say that a **Class in C++** is a blueprint representing a group of objects which shares some common properties and behaviors.

Object

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

C++

```
// C++ Program to show the syntax/working of Objects as a
// part of Object Oriented PProgramming
#include <iostream>
using namespace std;

class person {
    char name[20];
    int id;

public:
    void getdetails() {}
};

int main()
{

    person p1; // p1 is a object
    return 0;
}
```

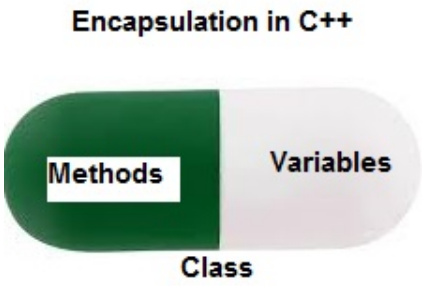
Objects take up space in memory and have an associated address like a record in pascal or structure or union. When a program is executed the objects interact by sending messages to one another. Each object contains data and code to manipulate the data. Objects can interact without having to know details of each other's data or code, it is sufficient to know the type of message accepted and the type of response returned by the objects.

To know more about C++ Objects and Classes, refer to this article – C++ Classes and Objects

Encapsulation

In normal terms, Encapsulation is defined as wrapping up data and information under a single unit. In Object-Oriented Programming, Encapsulation is defined as binding together the data and the functions that manipulate them. Consider a real-life example of encapsulation, in a company, there are different sections like the accounts section, finance section, sales section, etc. The finance section handles all the financial transactions and keeps records of all the data related to finance. Similarly, the sales section handles all the sales-related activities and keeps records of all the sales. Now there may arise a situation when for some reason an official from the finance section needs all the data about sales in a particular month. In this case, he is not allowed to directly access the data of the sales

section. He will first have to contact some other officer in the sales section and then request him to give the particular data. This is what encapsulation is. Here the data of the sales section and the employees that can manipulate them are wrapped under a single name "sales section".



Encapsulation in C++

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

To know more about encapsulation, refer to this article – Encapsulation in C++

Abstraction

Data abstraction is one of the most essential and important features of object-oriented programming in C++. Abstraction means displaying only essential information and hiding the details. Data abstraction refers to providing only essential information about the data to the outside world, hiding the background details or implementation. Consider a real-life example of a man driving a car. The man only knows that pressing the accelerator will increase the speed of the car or applying brakes will stop the car but he does not know how on pressing the accelerator the speed is actually increasing, he does not know about the inner mechanism of the car or the implementation of an accelerator, brakes, etc. in the car. This is what abstraction is.

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

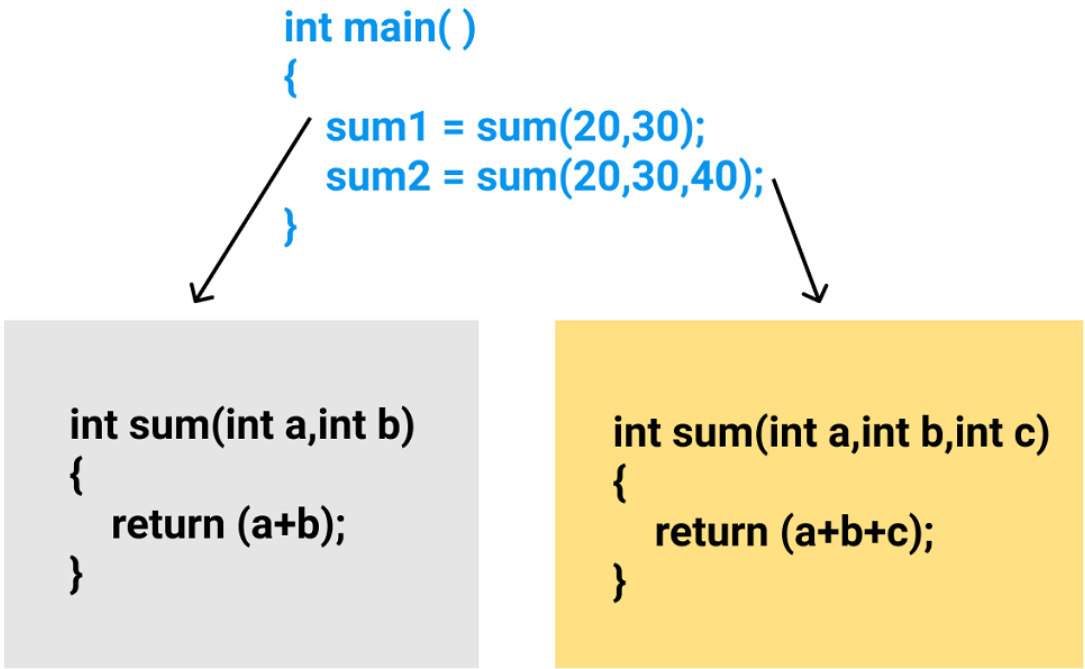
To know more about C++ abstraction, refer to this article – Abstraction in C++

Polymorphism

The word polymorphism means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form. A person at the same time can have different characteristics. A man at the same time is a father, a husband, and an employee. So the same person possesses different behavior in different situations. This is called polymorphism. An operation may exhibit different behaviors in different instances. The behavior depends upon the types of data used in the operation. C++ supports operator overloading and function overloading.

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

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



Polymorphism in C++

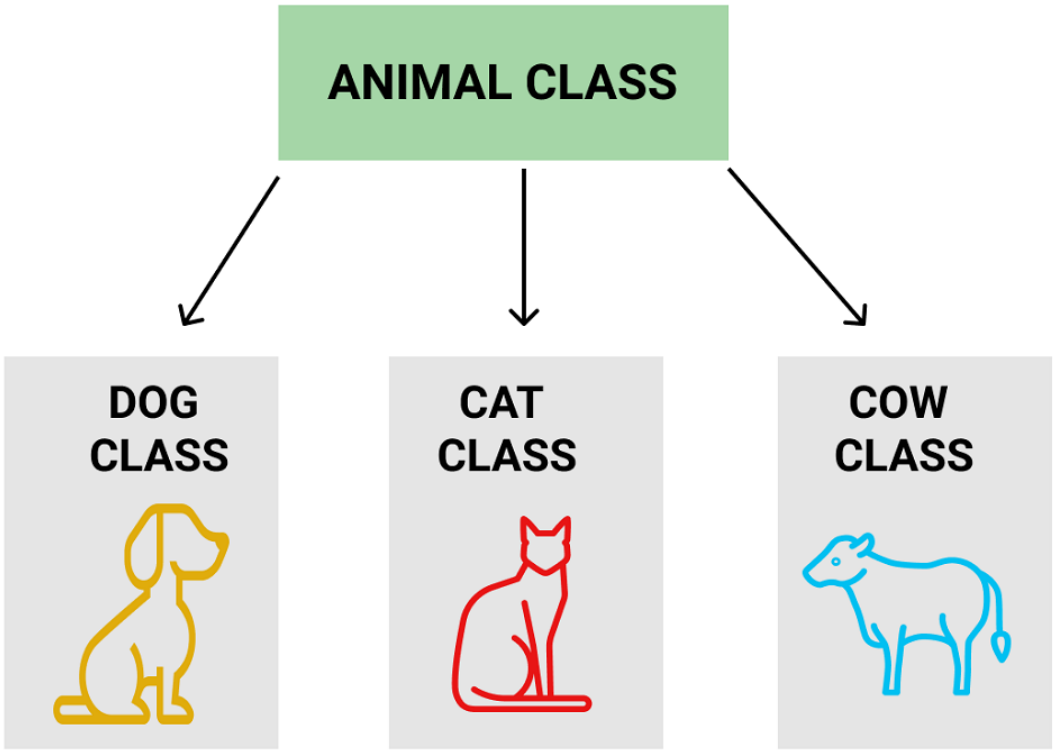
To know more about polymorphism, refer to this article – Polymorphism in C++

Inheritance

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

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

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



Inheritance in C++

To know more about Inheritance, refer to this article – Inheritance in C++

Dynamic Binding

In dynamic binding, the code to be executed in response to the function call is decided at runtime. C++ has virtual functions to support this. Because dynamic binding is flexible, it avoids the drawbacks of static binding, which connected the function call and definition at build time.

Example:

C++

```
// C++ Program to Demonstrate the Concept of Dynamic binding
// with the help of virtual function
#include <iostream>
using namespace std;

class GFG {
public:
    void call_Function() // function that call print
    {
        print();
    }
    void print() // the display function
    {
        cout << "Printing the Base class Content" << endl;
    }
};
class GFG2 : public GFG // GFG2 inherit a publicly
{
public:
    void print() // GFG2's display
    {
        cout << "Printing the Derived class Content"
             << endl;
    }
};
int main()
{
    GFG geeksforgeeks; // Creating GFG's pbject
    geeksforgeeks.call_Function(); // Calling call_Function
    GFG2 geeksforgeeks2; // creating GFG2 object
    geeksforgeeks2.call_Function(); // calling call_Function
                                    // for GFG2 object
    return 0;
}
```

Output

```
Printing the Base class Content
Printing the Base class Content
```

As we can see, the print() function of the parent class is called even from the derived class object. To resolve this we use virtual functions.

Message Passing

Objects communicate with one another by sending and receiving information. A message for an object is a request for the execution of a procedure and therefore will invoke a function in the receiving object that generates the desired results. Message passing

involves specifying the name of the object, the name of the function, and the information to be sent.

Related Articles:

- Classes and Objects
- Inheritance
- Access Modifiers
- Abstraction

This article is contributed by **Vankayala Karunakar**. Please write comments if you find anything incorrect, or if you want to share more information about the topic discussed above.

1.45k

Related Articles

1. Why C++ is partially Object Oriented Language?
2. OOPs | Object Oriented Design
3. Can a C++ class have an object of self type?
4. Object Slicing in C++
5. Preventing Object Copy in C++ (3 Different Ways)
6. Where is an object stored if it is created inside a block in C++?
7. cerr - Standard Error Stream Object in C++
8. How to add reference of an object in Container Classes
9. Dynamic initialization of object in C++
10. Object Delegation in C++

Previous Next

C++ Classes and Objects

Difficulty Level : Easy • Last Updated : 16 Feb, 2023

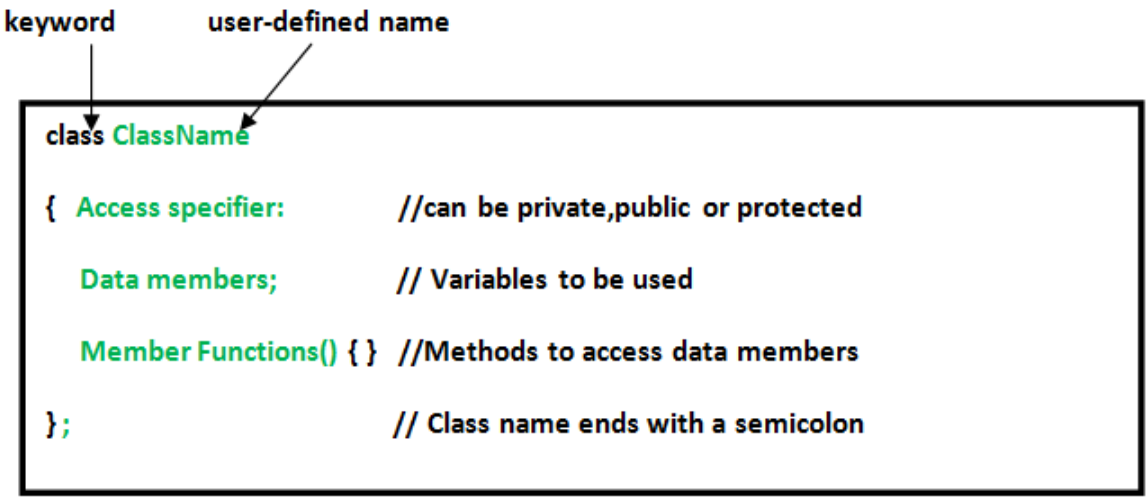
Class: A class in C++ is the building block that leads to Object-Oriented programming. It is a user-defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class. A C++ class is like a blueprint for an object. For Example: Consider the Class of **Cars**. There may be many cars with different names and brand but all of them will share some common properties like all of them will have *4 wheels*, *Speed Limit*, *Mileage range* etc. So here, Car is the class and wheels, speed limits, mileage are their properties.

- A Class is a user defined data-type which has data members and member functions.
- Data members are the data variables and member functions are the functions used to manipulate these variables and together these data members and member functions defines the properties and behavior of the objects in a Class.
- In the above example of class *Car*, the data member will be *speed limit*, *mileage* etc and member functions can be *apply brakes*, *increase speed* etc.

An **Object** is an instance of a Class. When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated.

Defining Class and Declaring Objects

A class is defined in C++ using keyword class followed by the name of class. The body of class is defined inside the curly brackets and terminated by a semicolon at the end.



Declaring Objects: When a class is defined, only the specification for the object is defined; no memory or storage is allocated. To use the data and access functions defined in the class, you need to create objects. **Syntax:**


```
ClassName ObjectName;
```

Accessing data members and member functions: The data members and member functions of class can be accessed using the dot('.') operator with the object. For example if the name of object is *obj* and you want to access the member function with the name *printName()* then you will have to write *obj.printName()* .

Accessing Data Members

The public data members are also accessed in the same way given however the private data members are not allowed to be accessed directly by the object. Accessing a data member depends solely on the access control of that data member. This access control is given by Access modifiers in C++. There are three access modifiers : **public, private and protected**.

CPP

```
// C++ program to demonstrate accessing of data members
#include <bits/stdc++.h>
using namespace std;
```

```
class Geeks {
    // Access specifier
public:
    // Data  Members
    string geekname;
    // Member Functions()
    void printname() { cout << "Geekname is:" << geekname; }
};
int main()
{
    // Declare an object of class geeks
    Geeks obj1;
    // accessing data member
    obj1.geekname = "Abhi";
    // accessing member function
    obj1.printname();
    return 0;
}
```

Output

```
Geekname is:Abhi
```

Member Functions in Classes

There are 2 ways to define a member function:

- Inside class definition
- Outside class definition

To define a member function outside the class definition we have to use the scope resolution :: operator along with class name and function name.

CPP

```
// C++ program to demonstrate function
// declaration outside class

#include <bits/stdc++.h>
using namespace std;
class Geeks
{
    public:
    string geekname;
    int id;

    // printname is not defined inside class definition
    void printname();

    // printid is defined inside class definition
    void printid()
    {
```

```
        cout <<"Geek id is: "<<id;
    }
};

// Definition of printname using scope resolution operator ::
void Geeks::printname()
{
    cout <<"Geekname is: "<<geekname;
}
int main() {

    Geeks obj1;
    obj1.geekname = "xyz";
    obj1.id=15;

    // call printname()
    obj1.printname();
    cout << endl;

    // call printid()
    obj1.printid();
    return 0;
}
```

Output:

```
Geekname is: xyz
Geek id is: 15
```

Note that all the member functions defined inside the class definition are by default **inline**, but you can also make any non-class function inline by using keyword inline with them. Inline functions are actual functions, which are copied everywhere during compilation, like pre-processor macro, so the overhead of function calling is reduced. Note: Declaring a friend function is a way to give private access to a non-member function.

Constructors

Constructors are special class members which are called by the compiler every time an object of that class is instantiated. Constructors have the same name as the class and may be defined inside or outside the class definition. There are 3 types of constructors:

- Default constructors
- Parameterized constructors
- Copy constructors

CPP

```cpp
// C++ program to demonstrate constructors

#include <bits/stdc++.h>
using namespace std;
class Geeks
{
    public:
    int id;

    //Default Constructor
    Geeks()
    {
        cout << "Default Constructor called" << endl;
        id=-1;
    }

    //Parameterized Constructor
    Geeks(int x)
    {
        cout <<"Parameterized Constructor called "<< endl;
        id=x;
    }
};
int main() {

    // obj1 will call Default Constructor
    Geeks obj1;
    cout <<"Geek id is: "<<obj1.id << endl;

    // obj2 will call Parameterized Constructor
    Geeks obj2(21);
    cout <<"Geek id is: " <<obj2.id << endl;
    return 0;
}
```

Output:

```
Default Constructor called
Geek id is: -1
Parameterized Constructor called
Geek id is: 21
```

A **Copy Constructor** creates a new object, which is exact copy of the existing object. The compiler provides a default Copy Constructor to all the classes. Syntax:

```
class-name (class-name &){}
```

Destructors

Destructor is another special member function that is called by the compiler when the scope of the object ends.

CPP

```
// C++ program to explain destructors

#include <bits/stdc++.h>
using namespace std;
class Geeks
{
    public:
    int id;

    //Definition for Destructor
    ~Geeks()
    {
        cout << "Destructor called for id: " << id <<endl;
    }
};

int main()
  {
    Geeks obj1;
    obj1.id=7;
    int i = 0;
    while ( i < 5 )
    {
        Geeks obj2;
        obj2.id=i;
        i++;
    } // Scope for obj2 ends here

    return 0;
  } // Scope for obj1 ends here
```

Output:

```
Destructor called for id: 0
Destructor called for id: 1
Destructor called for id: 2
Destructor called for id: 3
Destructor called for id: 4
Destructor called for id: 7
```

Interesting Fact (Rare Known concept)

Why do we give semicolon at the end of class ?

Many people might say that its a basic syntax and we should give semicolon at end of class as its rule define in cpp . But the main reason why semi-colons is there at end of class is compiler checks if user is trying to create an instance of class at the end of it .

Yes just like structure , Union we can also create the instance of class at the end just before the semicolon. As a result once execution reaches at that line it create class and allocates memory to your instance

---

C++

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

class Demo{
   int a, b;
    public:
    Demo()   // default constructor
    {
        cout << "Default Constructor" << endl;
    }
    Demo(int a, int b):a(a),b(b)  //parameterised constructor
    {
        cout << "parameterized constructor -values" << a  << " "<< b << endl;
    }

}instance;


int main() {

    return 0;
}
```

Output

```
 Default Constructor
```

We can see that we have created class instance of Demo with name "instance" , as a result output we can see is Default Constructor is called.

Similarly we can also call the parameterized constructor just by passing values here

---

C++

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

class Demo{
    public:
```

```
    int a, b;
    Demo()
    {
        cout << "Default Constructor" << endl;
    }
    Demo(int a, int b):a(a),b(b)
    {
        cout << "parameterized Constructor values-" << a << " "<< b << endl;
    }


}instance(100,200);


int main() {

    return 0;
}
```

Output

```
parameterized Constructor values-100 200
```

So by creating instance just before the semicolon , we can create the Instance of class

Pure Virtual Destructor **Related Articles:**

- Multiple Inheritance in C++
- C++ Quiz

This article is contributed by **Abhirav Kariya**. If you like GeeksforGeeks and would like to contribute, you can also write an article using write.geeksforgeeks.org or mail your article to review-team@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

771

Related Articles

1. Classes and Objects in Java
2. Catching Base and Derived Classes as Exceptions in C++ and Java
3. Pure Virtual Functions and Abstract Classes in C++

Inheritance in C++

Difficulty Level : Easy • Last Updated : 17 Feb, 2023

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

Inheritance is a feature or a process in which, new classes are created from the existing classes. The new class created is called "derived class" or "child class" and the existing class is known as the "base class" or "parent class". The derived class now is said to be inherited from the base class.

When we say derived class inherits the base class, it means, the derived class inherits all the properties of the base class, without changing the properties of base class and may add new features to its own. These new features in the derived class will not affect the base class. The derived class is the specialized class for the base class.

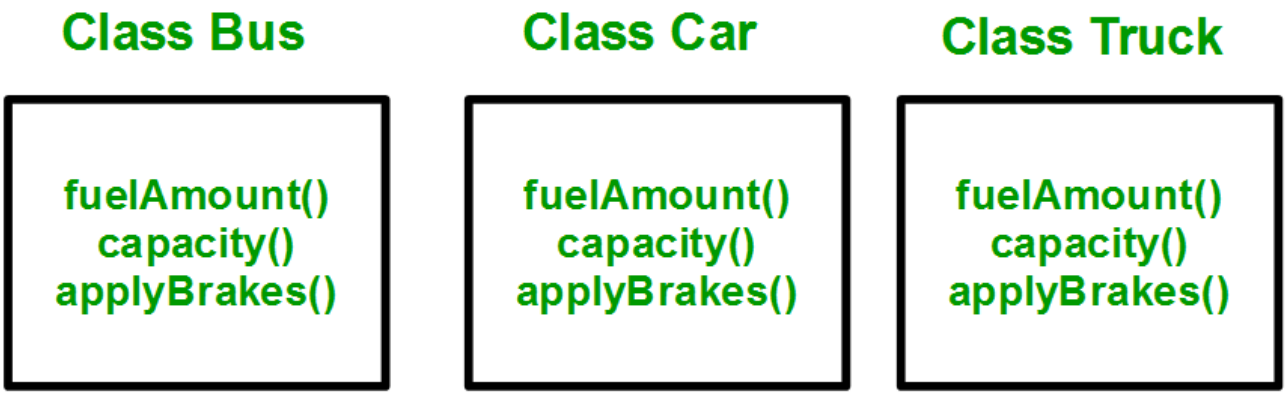
- **Sub Class:** The class that inherits properties from another class is called Subclass or Derived Class.
- **Super Class:** The class whose properties are inherited by a subclass is called Base Class or Superclass.

The article is divided into the following subtopics:

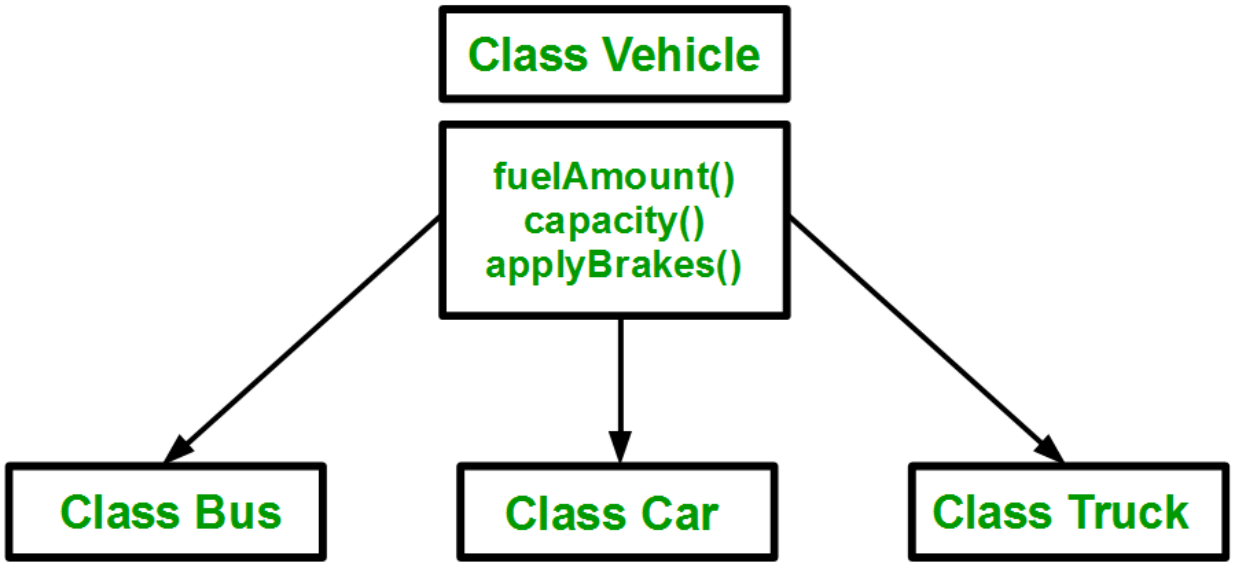
- Why and when to use inheritance?
- Modes of Inheritance
- Types of Inheritance

Why and when to use inheritance?

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



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



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

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

Derived Classes: A Derived class is defined as the class derived from the base class.

Syntax:

```
class  <derived_class_name> : <access-specifier> <base_class_name>
{
        //body
}
```

Where

class – keyword to create a new class

derived_class_name – name of the new class, which will inherit the base class

access-specifier – either of private, public or protected. If neither is specified, PRIVATE is taken as default

base-class-name – name of the base class

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

Example:

1. class ABC : private XYZ //private derivation
   { }
2. class ABC : public XYZ //public derivation
   { }
3. class ABC : protected XYZ //protected derivation
   { }

4. class ABC: XYZ //private derivation by default
{ }

Note:

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

o On the other hand, when the base class is publicly inherited by the derived class, public members of the base class also become the public members of the derived class. Therefore, the public members of the base class are accessible by the objects of the derived class as well as by the member functions of the derived class.

C++

```
// Example: define member function without argument within the class

#include<iostream>
using namespace std;

class Person
{
    int id;
    char name[100];

    public:
        void set_p()
        {
            cout<<"Enter the Id:";
            cin>>id;
            fflush(stdin);
            cout<<"Enter the Name:";
            cin.get(name,100);
        }

        void display_p()
        {
            cout<<endl<<id<<"\t"<<name<<"\t";
        }
};

class Student: private Person
{
    char course[50];
    int fee;

    public:
    void set_s()
        {
```

```
            set_p();
            cout<<"Enter the Course Name:";
            fflush(stdin);
            cin.getline(course,50);
            cout<<"Enter the Course Fee:";
            cin>>fee;
        }

        void display_s()
        {
            display_p();
            cout<<course<<"\t"<<fee<<endl;
        }
};

main()
{
    Student s;
    s.set_s();
    s.display_s();
    return 0;
}
```

Output:

```
Enter the Id: 101
Enter the Name: Dev
Enter the Course Name: GCS
Enter the Course Fee:70000

101      Dev     GCS    70000
```

C++

```
// Example: define member function without argument outside the class

#include<iostream>
using namespace std;

class Person
{
    int id;
    char name[100];

    public:
        void set_p();
        void display_p();
};

void Person::set_p()
{
```

```
    cout<<"Enter the Id:";
    cin>>id;
    fflush(stdin);
    cout<<"Enter the Name:";
    cin.get(name,100);
}

void Person::display_p()
{
    cout<<endl<<id<<"\t"<<name;
}

class Student: private Person
{
    char course[50];
    int fee;

    public:
        void set_s();
        void display_s();
};

void Student::set_s()
{
    set_p();
    cout<<"Enter the Course Name:";
    fflush(stdin);
    cin.getline(course,50);
    cout<<"Enter the Course Fee:";
    cin>>fee;
}

void Student::display_s()
{
    display_p();
    cout<<"\t"<<course<<"\t"<<fee;
}

main()
{
    Student s;
    s.set_s();
    s.display_s();
    return 0;
}
```

Output

```
Enter the Id:Enter the Name:Enter the Course Name:Enter the Course Fee:
0    t    0
```

C++

```
// Example: define member function with argument outside the class

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

class Person
{
    int id;
    char name[100];

    public:
        void set_p(int,char[]);
        void display_p();
};

void Person::set_p(int id,char n[])
{
    this->id=id;
    strcpy(this->name,n);
}

void Person::display_p()
{
    cout<<endl<<id<<"\t"<<name;
}

class Student: private Person
{
    char course[50];
    int fee;
    public:
    void set_s(int,char[],char[],int);
    void display_s();
};

void Student::set_s(int id,char n[],char c[],int f)
{
    set_p(id,n);
    strcpy(course,c);
    fee=f;
}


void Student::display_s()
{
    display_p();
    cout<<"t"<<course<<"\t"<<fee;
}

main()
{
    Student s;
```

```
    s.set_s(1001,"Ram","B.Tech",2000);
    s.display_s();
    return 0;
}
```

CPP

```
// C++ program to demonstrate implementation
// of Inheritance

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

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

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

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

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

    return 0;
}
```

Output

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

Output:

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

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

Modes of Inheritance: There are 3 modes of inheritance.

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

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

CPP

```cpp
// C++ Implementation to show that a derived class
// doesn't inherit access to private data members.
// However, it does inherit a full parent object.
class A {
public:
    int x;

protected:
    int y;

private:
    int z;
};

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

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

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

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

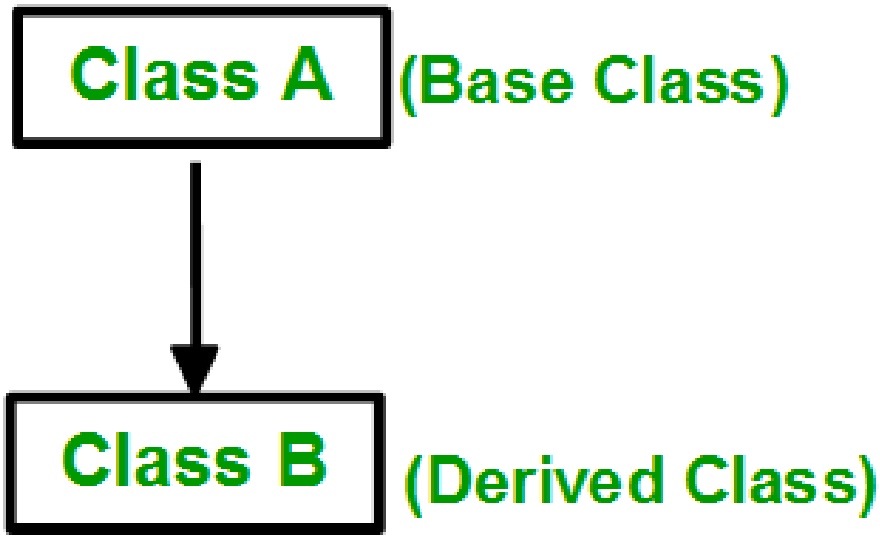
Base class member access specifier	Type of Inheritence		
	Public	Protected	Private
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Not accessible (Hidden)	Not accessible (Hidden)	Not accessible (Hidden)

Types Of Inheritance:-

1. Single inheritance
2. Multilevel inheritance
3. Multiple inheritance
4. Hierarchical inheritance
5. Hybrid inheritance

Types of Inheritance in C++

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



Syntax:

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

OR

class A
{
... .. ...
};

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

CPP

```
// C++ program to explain
// Single inheritance
#include<iostream>
using namespace std;

// base class
class Vehicle {
  public:
    Vehicle()
    {
      cout << "This is a Vehicle\n";
```

```
    }
};

// sub class derived from a single base classes
class Car : public Vehicle {

};

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

Output

```
This is a Vehicle
```

C++

```
// Example:

#include<iostream>
using namespace std;

class A
{
    protected:
    int a;

    public:
        void set_A()
        {
            cout<<"Enter the Value of A=";
            cin>>a;

        }
        void disp_A()
        {
            cout<<endl<<"Value of A="<<a;
        }
};


class B: public A
{
    int b,p;
```

```
    public:
        void set_B()
        {
            set_A();
            cout<<"Enter the Value of B=";
            cin>>b;
        }

        void disp_B()
        {
            disp_A();
            cout<<endl<<"Value of B="<<b;
        }

        void cal_product()
        {
            p=a*b;
            cout<<endl<<"Product of "<<a<<" * "<<b<<" = "<<p;
        }

};

main()
{

    B _b;
    _b.set_B();
    _b.cal_product();

    return 0;

}
```

Output:- Enter the Value of A= 3 3 Enter the Value of B= 5 5 Product of 3 * 5 = 15

C++

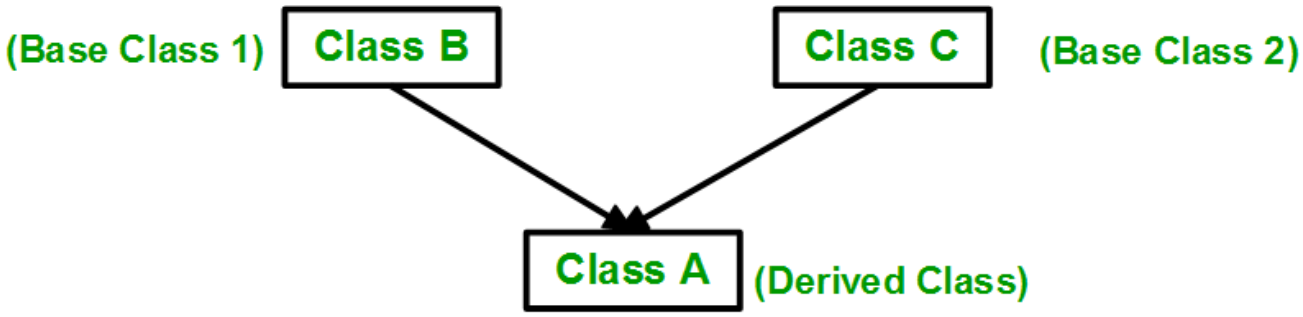
```
// Example:

#include<iostream>
using namespace std;

class A
{
    protected:
    int a;

    public:
        void set_A(int x)
        {
             a=x;
        }
```

```
        void disp_A()
        {
            cout<<endl<<"Value of A="<<a;
        }
};

class B: public A
{
    int b,p;

    public:
        void set_B(int x,int y)
        {
            set_A(x);
            b=y;
        }

        void disp_B()
        {
            disp_A();
            cout<<endl<<"Value of B="<<b;
        }

        void cal_product()
        {
            p=a*b;
            cout<<endl<<"Product of "<<a<<" * "<<b<<" = "<<p;
        }

};

main()
{
    B _b;
    _b.set_B(4,5);
    _b.cal_product();

    return 0;
}
```

Output

```
Product of 4 * 5 = 20
```

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



Syntax:

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

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

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

CPP

```
// C++ program to explain
// multiple inheritance
#include <iostream>
```

```
using namespace std;

// first base class
class Vehicle {
public:
    Vehicle() { cout << "This is a Vehicle\n"; }
};

// second base class
class FourWheeler {
public:
    FourWheeler()
    {
        cout << "This is a 4 wheeler Vehicle\n";
    }
};

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

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

Output

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

C++

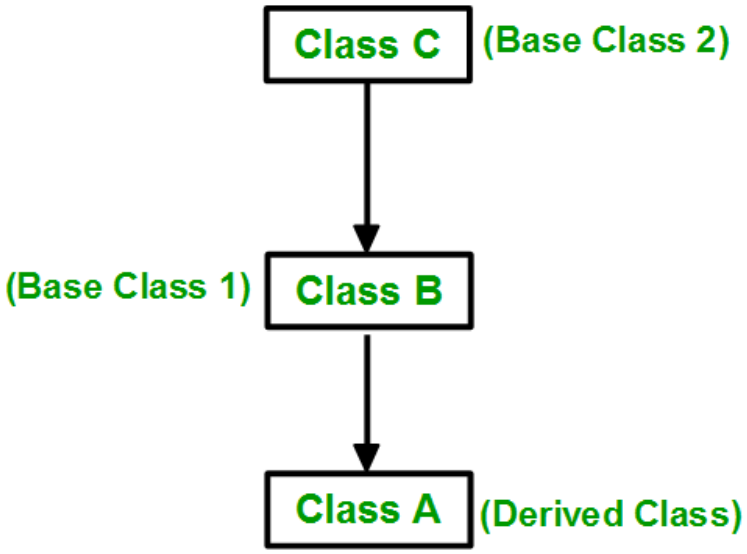
```
// Example:

#include<iostream>
using namespace std;

class A
{
              protected:
              int a;

              public:
                  void set_A()
                  {
                        cout<<"Enter the Value of A=";
```

```
                        cin>>a;

                    }

                    void disp_A()
                    {
                        cout<<endl<<"Value of A="<<a;
                    }
};

class B: public A
{
          protected:
               int b;

          public:
                 void set_B()
               {
                   cout<<"Enter the Value of B=";
                    cin>>b;
                }


               void disp_B()
                 {
                    cout<<endl<<"Value of B="<<b;
                 }
};

class C: public B
{
      int c,p;

      public:
          void set_C()
          {
                 cout<<"Enter the Value of C=";
                 cin>>c;
           }

           void disp_C()
           {
                 cout<<endl<<"Value of C="<<c;
           }

            void cal_product()
             {
                  p=a*b*c;
                 cout<<endl<<"Product of "<<a<<" * "<<b<<" * "<<c<<" = "<<p;
              }
};

main()
{

    C _c;
    _c.set_A();
```

```
    _c.set_B();
    _c.set_C();
    _c.disp_A();
    _c.disp_B();
    _c.disp_C();
    _c.cal_product();

    return 0;

}
```

To know more about it, please refer to the article Multiple Inheritances.

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



Syntax:-

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

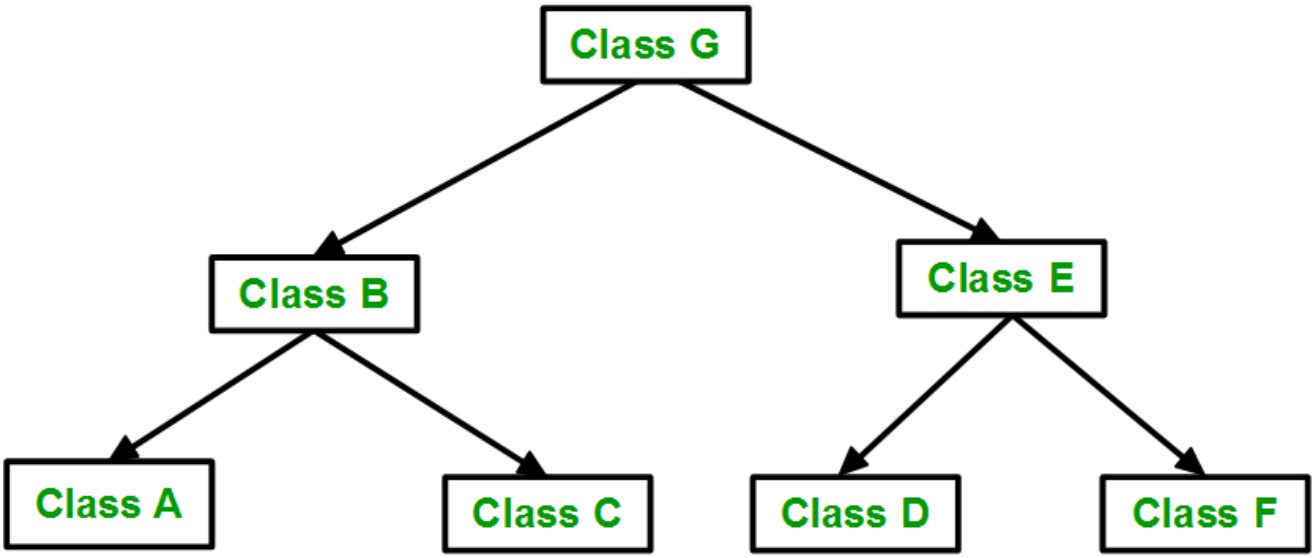
CPP

```cpp
// C++ program to implement
// Multilevel Inheritance
#include <iostream>
using namespace std;

// base class
class Vehicle {
public:
    Vehicle() { cout << "This is a Vehicle\n"; }
};

// first sub_class derived from class vehicle
class fourWheeler : public Vehicle {
public:
    fourWheeler()
    {
        cout << "Objects with 4 wheels are vehicles\n";
    }
};
// sub class derived from the derived base class fourWheeler
class Car : public fourWheeler {
public:
    Car() { cout << "Car has 4 Wheels\n"; }
};

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

Output

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

4. Hierarchical Inheritance: In this type of inheritance, more than one subclass is inherited from a single base class. i.e. more than one derived class is created from a single base class.



Syntax:-

```
class A
{
    // body of the class A.
}
class B : public A
{
    // body of class B.
}
class C : public A
{
    // body of class C.
}
class D : public A
{
    // body of class D.
}
```

CPP

```
// C++ program to implement
// Hierarchical Inheritance
#include <iostream>
using namespace std;

// base class
class Vehicle {
public:
    Vehicle() { cout << "This is a Vehicle\n"; }
};
```

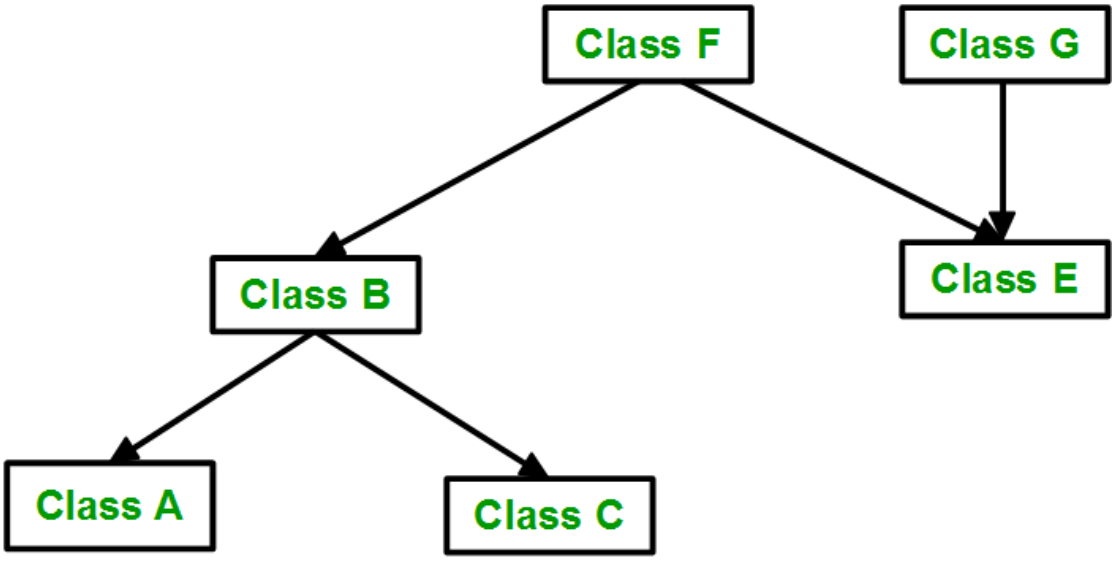
```
// first sub class
class Car : public Vehicle {
};

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

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

Output

```
This is a Vehicle
This is a Vehicle
```

5. Hybrid (Virtual) Inheritance: Hybrid Inheritance is implemented by combining more than one type of inheritance. For example: Combining Hierarchical inheritance and Multiple Inheritance.

Below image shows the combination of hierarchical and multiple inheritances:



CPP

```
// C++ program for Hybrid Inheritance

#include <iostream>
using namespace std;

// base class
class Vehicle {
public:
    Vehicle() { cout << "This is a Vehicle\n"; }
};

// base class
class Fare {
public:
    Fare() { cout << "Fare of Vehicle\n"; }
};

// first sub class
class Car : public Vehicle {
};

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

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

Output

```
This is a Vehicle
Fare of Vehicle
```

C++

```
// Example:

#include <iostream>
using namespace std;

class A
{
    protected:
    int a;
```

```
    public:
    void get_a()
    {
       cout << "Enter the value of 'a' : ";
       cin>>a;
    }
};

class B : public A
{
    protected:
    int b;
    public:
    void get_b()
    {
       cout << "Enter the value of 'b' : ";
       cin>>b;
    }
};
class C
{
    protected:
    int c;
    public:
    void get_c()
    {
        cout << "Enter the value of c is : ";
        cin>>c;
    }
};

class D : public B, public C
{
    protected:
    int d;
    public:
    void mul()
    {
         get_a();
         get_b();
         get_c();
         cout << "Multiplication of a,b,c is : " <<a*b*c;
    }
};


int main()
{
    D d;
    d.mul();
    return 0;
}
```

6. A special case of hybrid inheritance: Multipath inheritance:

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

Example:

CPP

```cpp
// C++ program demonstrating ambiguity in Multipath
// Inheritance

#include <iostream>
using namespace std;

class ClassA {
public:
    int a;
};

class ClassB : public ClassA {
public:
    int b;
};

class ClassC : public ClassA {
public:
    int c;
};

class ClassD : public ClassB, public ClassC {
public:
    int d;
};

int main()
{
    ClassD obj;

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

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

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

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

    cout << "\n b : " << obj.b;
    cout << "\n c : " << obj.c;
    cout << "\n d : " << obj.d << '\n';
```

```
}
```

Output

```
a from ClassB  : 10
a from ClassC  : 100
b : 20
c : 30
d : 40
```

Output:

```
a from ClassB : 10
a from ClassC : 100
b : 20
c : 30
d : 40
```

In the above example, both ClassB and ClassC inherit ClassA, they both have a single copy of ClassA. However Class-D inherits both ClassB and ClassC, therefore Class-D has two copies of ClassA, one from ClassB and another from ClassC.
If we need to access the data member of ClassA through the object of Class-D, we must specify the path from which a will be accessed, whether it is from ClassB or ClassC, bcoz compiler can't differentiate between two copies of ClassA in Class-D.

There are 2 Ways to Avoid this Ambiguity:

1) Avoiding ambiguity using the scope resolution operator: Using the scope resolution operator we can manually specify the path from which data member a will be accessed, as shown in statements 3 and 4, in the above example.

CPP

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

Note: Still, there are two copies of ClassA in Class-D.

2) Avoiding ambiguity using the virtual base class:

CPP

```
#include<iostream>

class ClassA
{
  public:
    int a;
};

class ClassB : virtual public ClassA
{
  public:
    int b;
};

class ClassC : virtual public ClassA
{
  public:
    int c;
};

class ClassD : public ClassB, public ClassC
{
  public:
    int d;
};

int main()
{
    ClassD obj;

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

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

    cout << "\n a : " << obj.a;
    cout << "\n b : " << obj.b;
    cout << "\n c : " << obj.c;
    cout << "\n d : " << obj.d << '\n';
}
```

Output:

```
 a : 100
 b : 20
 c : 30
 d : 40
```

According to the above example, Class-D has only one copy of ClassA, therefore, statement 4 will overwrite the value of a, given in statement 3.

This article is contributed by **Harsh Agarwal**. If you like GeeksforGeeks and would like to contribute, you can also write an article using write.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

977

Related Articles

1. Inheritance and Friendship in C++
2. Inheritance and Constructors in Java
3. Multiple Inheritance in C++
4. Does overloading work with Inheritance?
5. OOP in Python | Set 3 (Inheritance, examples of object, issubclass and super)
6. Java and Multiple Inheritance
7. Difference between Containership and Inheritance in C++
8. Difference between Single and Multiple Inheritance in C++
9. Difference between Inheritance and Polymorphism
10. Runtime Polymorphism in various types of Inheritance in C++

Previous Next

Article Contributed By :

GeeksforGeeks

Vote for difficulty

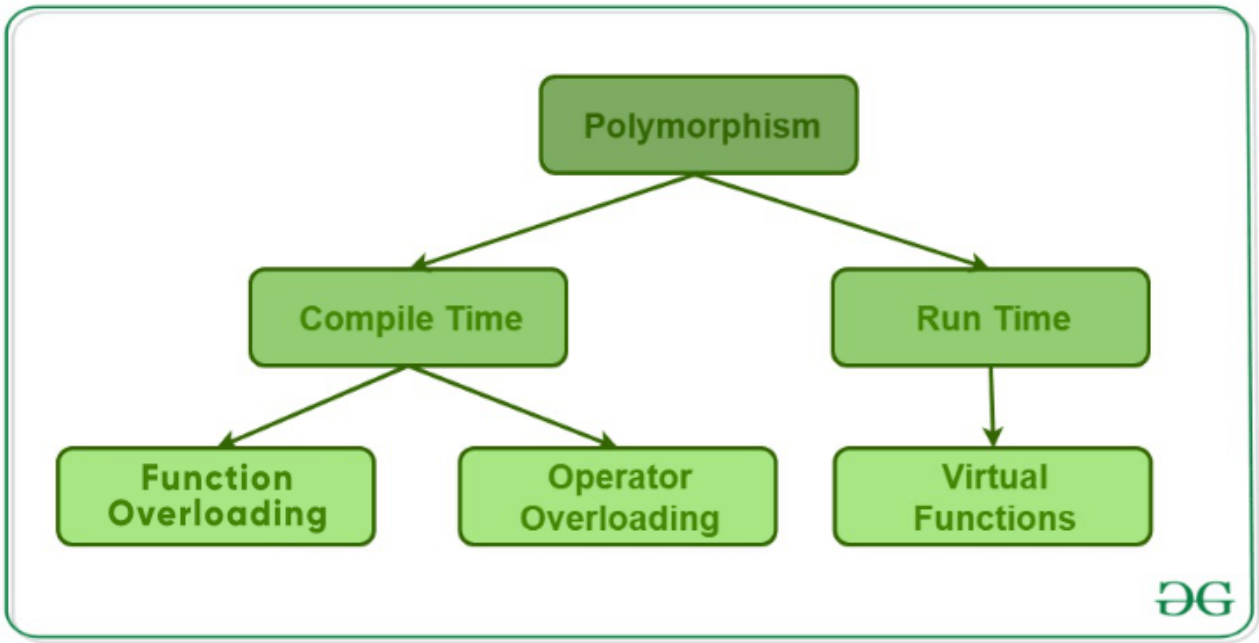
C++ Polymorphism

Difficulty Level : Easy • Last Updated : 03 Apr, 2023

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

Types of Polymorphism

- **Compile-time Polymorphism.**
- **Runtime Polymorphism.**



Types of Polymorphism

1. Compile-Time Polymorphism

This type of polymorphism is achieved by function overloading or operator overloading.

A. Function Overloading

When there are multiple functions with the same name but different parameters, then the functions are said to be **overloaded,** hence this is known as Function Overloading. Functions can be overloaded by **changing the number of arguments** or/and **changing the type of arguments**. In simple terms, it is a feature of object-oriented programming providing many functions to have the same name but distinct parameters when numerous tasks are listed under one function name. There are certain Rules of Function Overloading that should be followed while overloading a function.

Below is the C++ program to show function overloading or compile-time polymorphism:

C++

```
// C++ program to demonstrate
// function overloading or
// Compile-time Polymorphism
#include <bits/stdc++.h>

using namespace std;
class Geeks {
public:

    // Function with 1 int parameter
    void func(int x)
    {
        cout << "value of x is " <<
                  x << endl;
    }

    // Function with same name but
    // 1 double parameter
    void func(double x)
    {
```

```
        cout << "value of x is " <<
                x << endl;
    }

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

// Driver code
int main()
{
    Geeks obj1;

    // Function being called depends
    // on the parameters passed
    // func() is called with int value
    obj1.func(7);

    // func() is called with double value
    obj1.func(9.132);

    // func() is called with 2 int values
    obj1.func(85, 64);
    return 0;
}
```

Output

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

Explanation: In the above example, a single function named function **func()** acts differently in three different situations, which is a property of polymorphism.
To know more about this, you can refer to the article – Function Overloading in C++.

B. Operator Overloading

C++ has the ability to provide the operators with a special meaning for a data type, this ability is known as operator overloading. For example, we can make use of the addition operator (+) for string class to concatenate two strings. We know that the task of this operator is to add two operands. So a single operator '+', when placed between integer operands, adds them and when placed between string operands, concatenates them.

Below is the C++ program to demonstrate operator overloading:

CPP

```cpp
// C++ program to demonstrate
// Operator Overloading or
// Compile-Time Polymorphism
#include <iostream>
using namespace std;

class Complex {
private:
    int real, imag;

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

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

// Driver code
int main()
{
    Complex c1(10, 5), c2(2, 4);

    // An example call to "operator+"
    Complex c3 = c1 + c2;
    c3.print();
}
```

Output

```
12 + i9
```

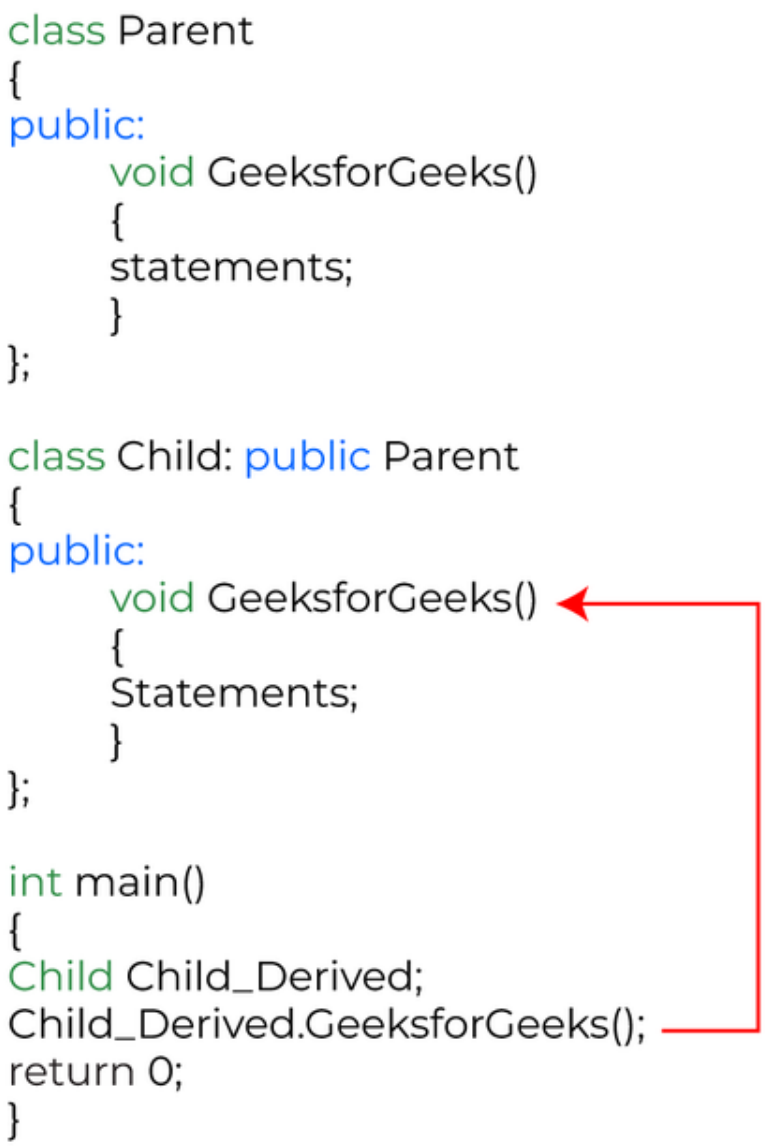
Explanation: In the above example, the operator '+' is overloaded. Usually, this operator is used to add two numbers (integers or floating point numbers), but here the operator is made to perform the addition of two imaginary or complex numbers.
To know more about this one, refer to the article – Operator Overloading.

2. Runtime Polymorphism

This type of polymorphism is achieved by **Function Overriding**. Late binding and dynamic polymorphism are other names for runtime polymorphism. The function call is resolved at runtime in runtime polymorphism. In contrast, with compile time polymorphism, the compiler determines which function call to bind to the object after deducing it at runtime.

A. Function Overriding

Function Overriding occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be overridden.



Function overriding Explanation

Below is the C++ program to demonstrate function overriding:

C++

```
// C++ program for function overriding
#include <bits/stdc++.h>
using namespace std;

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

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

class derived : public base {
public:

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

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

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

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

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

    return 0;
}
```

Output

```
print derived class
show base class
```

Runtime Polymorphism with Data Members

Runtime Polymorphism can be achieved by data members in C++. Let's see an example where we are accessing the field by reference variable which refers to the instance of derived class.

C++

```
// C++ program for  function overriding with data members
#include <bits/stdc++.h>
using namespace std;

#include <iostream>
using namespace std;
class Animal {                                  //  base class declaration.
    public:
    string color = "Black";
};
class Dog: public Animal                       // inheriting Animal class.
{
 public:
    string color = "Grey";
};
  //Driver code
int main(void) {
     Animal d= Dog();  //accessing the field by reference variable which refers to der
    cout<<d.color;
}
```

Output

```
black
```

Virtual Function

A virtual function is a member function that is declared in the base class using the keyword virtual and is re-defined (Overridden) in the derived class.

Some Key Points About Virtual Functions:

- Virtual functions are Dynamic in nature.
- They are defined by inserting the keyword "**virtual**" inside a base class and are always declared with a base class and overridden in a child class
- A virtual function is called during Runtime

Below is the C++ program to demonstrate virtual function:

C++

```
// C++ Program to demonstrate
// the Virtual Function
#include <iostream>
using namespace std;

// Declaring a Base class
class GFG_Base {

public:
    // virtual function
    virtual void display()
    {
        cout << "Called virtual Base Class function" <<
                "\n\n";
    }

    void print()
    {
        cout << "Called GFG_Base print function" <<
                "\n\n";
    }
};

// Declaring a Child Class
class GFG_Child : public GFG_Base {

public:
    void display()
    {
        cout << "Called GFG_Child Display Function" <<
                "\n\n";
    }

    void print()
    {
        cout << "Called GFG_Child print Function" <<
                "\n\n";
    }
};
```

```
// Driver code
int main()
{
    // Create a reference of class GFG_Base
    GFG_Base* base;

    GFG_Child child;

    base = &child;

    // This will call the virtual function
    base->GFG_Base::display();

    // this will call the non-virtual function
    base->print();
}
```

Output

```
Called virtual Base Class function


Called GFG_Base print function
```

This article is contributed by **Harsh Agarwal**. If you like GeeksforGeeks and would like to contribute, you can also write an article using write.geeksforgeeks.org or mail your article to review-team@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks. Please write comments if you find anything incorrect, or if you want to share more information about the topic discussed above.

704

Related Articles

1. Virtual Functions and Runtime Polymorphism in C++
2. Difference between Inheritance and Polymorphism
3. Runtime Polymorphism in various types of Inheritance in C++
4. C++ Error - Does not name a type
5. Execution Policy of STL Algorithms in Modern C++
6. C++ Program To Print Pyramid Patterns

Encapsulation in C++

Difficulty Level : Easy • Last Updated : 18 Feb, 2023

Encapsulation in C++ is defined as the wrapping up of data and information in a single unit. In Object Oriented Programming, Encapsulation is defined as binding together the data and the functions that manipulate them.

Consider a real-life example of encapsulation, in a company, there are different sections like the accounts section, finance section, sales section, etc. Now,

- The finance section handles all the financial transactions and keeps records of all the data related to finance.
- Similarly, the sales section handles all the sales-related activities and keeps records of all the sales.

Now there may arise a situation when for some reason an official from the finance section needs all the data about sales in a particular month.

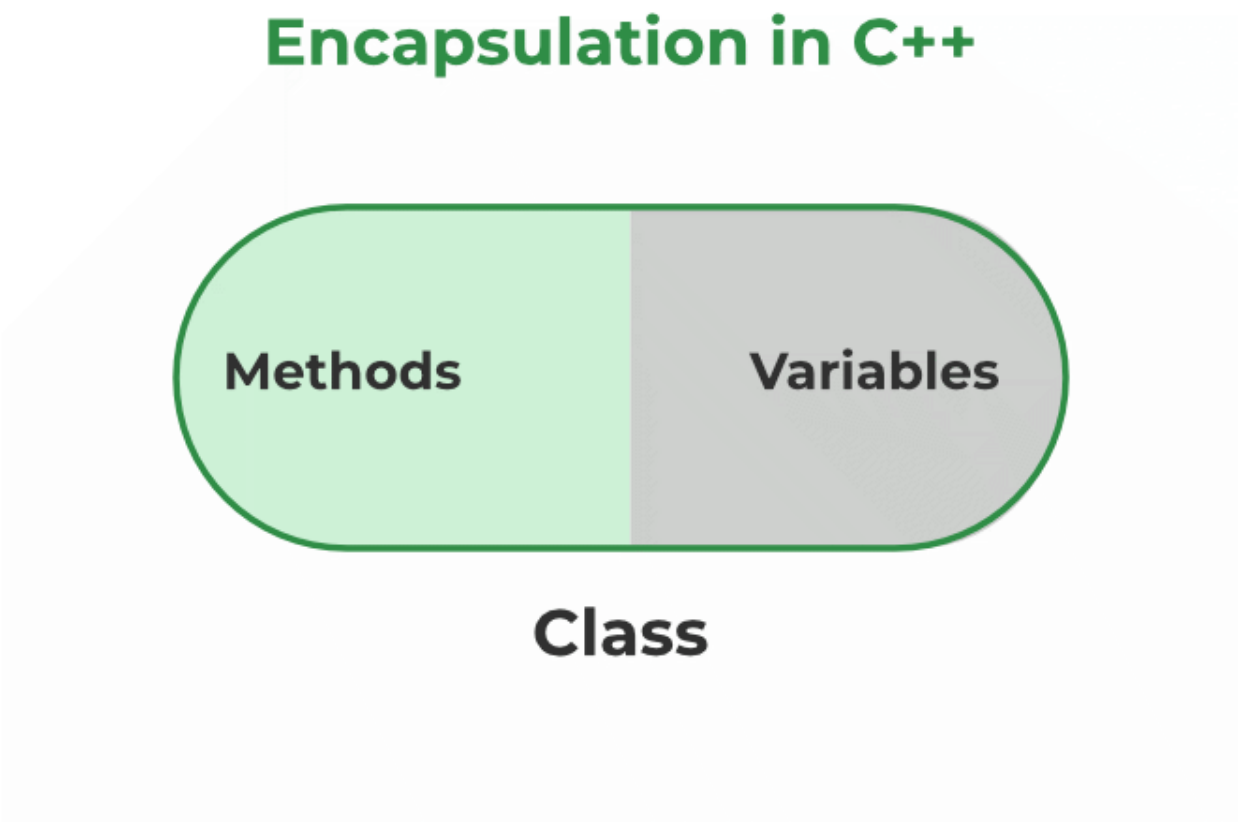
In this case, he is not allowed to directly access the data of the sales section. He will first have to contact some other officer in the sales section and then request him to give the particular data.

This is what **Encapsulation** is. Here the data of the sales section and the employees that can manipulate them are wrapped under a single name "sales section".

Features of Encapsulation

Below are the features of encapsulation:

1. We can not access any function from the class directly. We need an object to access that function that is using the member variables of that class.
2. The function which we are making inside the class must use only member variables, only then it is called *encapsulation*.
3. If we don't make a function inside the class which is using the member variable of the class then we don't call it encapsulation.
4. Increase in the security of data
5. It helps to control the modification of our data members.



Encapsulation also leads to data abstraction. Using encapsulation also hides the data, as in the above example, the data of the sections like sales, finance, or accounts are hidden from any other section.

Simple Example of C++:

C++

```
#include <iostream>
#include <string>

using namespace std;

class Person {
  private:
    string name;
    int age;
  public:
    Person(string name, int age) {
      this->name = name;
      this->age = age;
    }
    void setName(string name) {
      this->name = name;
    }
    string getName() {
      return name;
    }
    void setAge(int age) {
      this->age = age;
    }
    int getAge() {
      return age;
    }
};

int main() {
  Person person("John Doe", 30);

  cout << "Name: " << person.getName() << endl;
  cout << "Age: " << person.getAge() << endl;

  person.setName("Jane Doe");
  person.setAge(32);

  cout << "Name: " << person.getName() << endl;
  cout << "Age: " << person.getAge() << endl;

  return 0;
}
```

Output:

Name: John Doe

Age: 30

Name: Jane Doe

Age: 32

In C++, encapsulation can be implemented using **classes** and access modifiers.

Example:

C++

```
// C++ program to demonstrate
// Encapsulation
#include <iostream>
using namespace std;

class Encapsulation {
private:
    // Data hidden from outside world
    int x;

public:
    // Function to set value of
    // variable x
    void set(int a) { x = a; }

    // Function to return value of
    // variable x
    int get() { return x; }
};

// Driver code
int main()
{
    Encapsulation obj;
    obj.set(5);
    cout << obj.get();
    return 0;
}
```

Output

```
5
```

Explanation: In the above program, the variable **x** is made private. This variable can be accessed and manipulated only using the functions get() and set() which are present inside the class. Thus we can say that here, the variable x and the functions get() and set() are bound together which is nothing but encapsulation.

C++

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

// declaring class
```

```
class Circle {
    // access modifier
private:
    // Data Member
    float area;
    float radius;

public:
    void getRadius()
    {
        cout << "Enter radius\n";
        cin >> radius;
    }
    void findArea()
    {
        area = 3.14 * radius * radius;
        cout << "Area of circle=" << area;
    }
};
int main()
{
    // creating instance(object) of class
    Circle cir;
    cir.getRadius(); // calling function
    cir.findArea(); // calling function
}
```

Output

```
Enter radius
Area of circle=0
```

Role of Access Specifiers in Encapsulation

Access specifiers facilitate Data Hiding in C++ programs by restricting access to the class member functions and data members. There are three types of access specifiers in C++:

- **Private**
- **Protected**
- **Public**

By **default**, all data members and member functions of a class are made **private** by the compiler.

Points to Consider

As we have seen in the above example, access specifiers play an important role in implementing encapsulation in C++. The process of implementing encapsulation can be sub-divided into two steps:

1. Creating a class to encapsulate all the data and methods into a single unit.
2. Hiding relevant data using access specifiers.

This article is contributed by **Harsh Agarwal**. If you like GeeksforGeeks and would like to contribute, you can also write an article using write.geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks. Please write comments if you find anything incorrect, or if you want to share more information about the topic discussed above.

478

Related Articles

1. Difference between Abstraction and Encapsulation in C++
2. C++ Error - Does not name a type
3. Execution Policy of STL Algorithms in Modern C++
4. C++ Program To Print Pyramid Patterns
5. Jagged Arrays in C++
6. Introduction to Parallel Programming with OpenMP in C++
7. Hollow Half Pyramid Pattern Using Numbers
8. 30 OOPs Interview Questions and Answers (2023)
9. C++ <cstdio>
10. C++ <cstring>

Previous Next

Article Contributed By :

GeeksforGeeks

Abstraction in C++

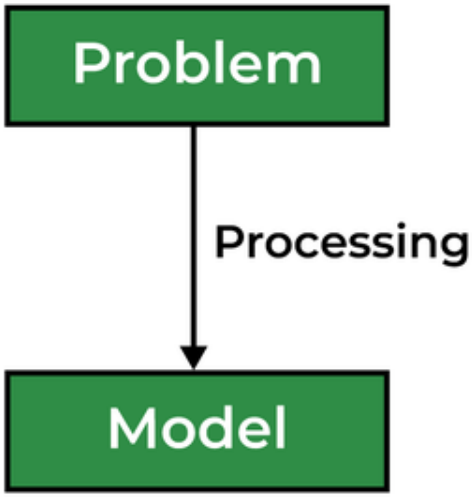
Difficulty Level : Easy • Last Updated : 23 Dec, 2022

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

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

Types of Abstraction:

1. **Data abstraction –** This type only shows the required information about the data and hides the unnecessary data.
2. **Control Abstraction –** This type only shows the required information about the implementation and hides unnecessary information.



Abstraction using Classes

We can implement Abstraction in C++ using classes. The class helps us to group data members and member functions using available access specifiers. A Class can decide which data member will be visible to the outside world and which is not.

Abstraction in Header files

One more type of abstraction in C++ can be header files. For example, consider the pow() method present in math.h header file. Whenever we need to calculate the power of a number, we simply call the function pow() present in the math.h header file and pass the numbers as arguments without knowing the underlying algorithm according to which the function is actually calculating the power of numbers.

Abstraction using Access Specifiers

Access specifiers are the main pillar of implementing abstraction in C++. We can use access specifiers to enforce restrictions on class members. For example:

- Members declared as **public** in a class can be accessed from anywhere in the program.
- Members declared as **private** in a class, can be accessed only from within the class. They are not allowed to be accessed from any part of the code outside the class.

We can easily implement abstraction using the above two features provided by access specifiers. Say, the members that define the internal implementation can be marked as private in a class. And the important information needed to be given to the outside world can be marked as public. And these public members can access the private members as they are inside the class.

Example:

C++

```
// C++ Program to Demonstrate the
// working of Abstraction
#include <iostream>
using namespace std;

class implementAbstraction {
private:
    int a, b;

public:
    // method to set values of
    // private members
    void set(int x, int y)
    {
        a = x;
        b = y;
    }

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

int main()
{
    implementAbstraction obj;
    obj.set(10, 20);
    obj.display();
    return 0;
}
```

Output

```
a = 10
b = 20
```

You can see in the above program we are not allowed to access the variables a and b directly, however, one can call the function set() to set the values in a and b and the

function display() to display the values of a and b.

Advantages of Data Abstraction

- Helps the user to avoid writing the low-level code
- Avoids code duplication and increases reusability.
- Can change the internal implementation of the class independently without affecting the user.
- Helps to increase the security of an application or program as only important details are provided to the user.
- It reduces the complexity as well as the redundancy of the code, therefore increasing the readability.

This article is contributed by **Harsh Agarwal**. If you like GeeksforGeeks and would like to contribute, you can also write an article using write.geeksforgeeks.org or mail your article to review-team@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks. Please write comments if you find anything incorrect, or if you want to share more information about the topic discussed above.

C++

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

class Vehicle
{
    public:
        void company()
        {
            cout<<"GFG\n";
        }
    public:
        void model()
        {
            cout<<"SIMPLE\n";
        }
    public:
        void color()
        {
            cout<<"Red/GREEN/Silver\n";
        }
    public:
        void cost()
        {
            cout<<"Rs. 60000 to 900000\n";
        }
    public:
        void oil()
        {
            cout<<"PETRO\n";
```

```
        }
    private:
        void piston()
        {
            cout<<"4 piston\n";
        }
    private:
        void manWhoMade()
        {
            cout<<"Markus Librette\n";
        }
};
int main()
{

    Vehicle obj;
    obj.company();
    obj.model();
    obj.color();
    obj.cost();
    obj.oil();
}
```

Output

```
GFG
SIMPLE
Red/GREEN/Silver
Rs. 60000 to 900000
PETRO
```

435

Related Articles

1. Difference between Abstraction and Encapsulation in C++
2. C++ Error - Does not name a type
3. Execution Policy of STL Algorithms in Modern C++
4. C++ Program To Print Pyramid Patterns
5. Jagged Arrays in C++
6. Introduction to Parallel Programming with OpenMP in C++

Function Overloading in C++

Difficulty Level : Easy • Last Updated : 16 Mar, 2023

Function overloading is a feature of object-oriented programming where two or more functions can have the same name but different parameters. When a function name is overloaded with different jobs it is called Function Overloading. In Function Overloading "Function" name should be the same and the arguments should be different. Function overloading can be considered as an example of a polymorphism feature in C++.

If multiple functions having same name but parameters of the functions should be different is known as Function Overloading.
If we have to perform only one operation and having same name of the functions increases the readability of the program.
Suppose you have to perform addition of the given numbers but there can be any number of arguments, if you write the function such as a(int,int) for two parameters, and b(int,int,int) for three parameters then it may be difficult for you to understand the behavior of the function because its name differs.

The parameters should follow any one or more than one of the following conditions for Function overloading:

- Parameters should have a different type

add(int a, int b)
add(double a, double b)

Below is the implementation of the above discussion:

C++

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


void add(int a, int b)
{
  cout << "sum = " << (a + b);
}

void add(double a, double b)
{
    cout << endl << "sum = " << (a + b);
}

// Driver code
int main()
{
    add(10, 2);
    add(5.3, 6.2);

    return 0;
}
```

Output

```
sum = 12
sum = 11.5
```

- Parameters should have a different number

add(int a, int b)

add(int a, int b, int c)

Below is the implementation of the above discussion:

C++

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

void add(int a, int b)
{
  cout << "sum = " << (a + b);
}

void add(int a, int b, int c)
{
    cout << endl << "sum = " << (a + b + c);
}

// Driver code
int main()
{
    add(10, 2);
    add(5, 6, 4);

    return 0;
}
```

Output

```
sum = 12
sum = 15
```

- Parameters should have a different sequence of parameters.

add(int a, double b)

add(double a, int b)

Below is the implementation of the above discussion:

C++

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

```
void add(int a, double b)
{
    cout<<"sum = "<<(a+b);
}

void  add(double a, int b)
{
    cout<<endl<<"sum = "<<(a+b);
}

// Driver code
int main()
{
    add(10,2.5);
    add(5.5,6);

      return 0;
}
```

Output

```
sum = 12.5
sum = 11.5
```

Following is a simple C++ example to demonstrate function overloading.

CPP

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

void print(int i) {
  cout << " Here is int " << i << endl;
}
void print(double  f) {
  cout << " Here is float " << f << endl;
}
void print(char const *c) {
  cout << " Here is char* " << c << endl;
}

int main() {
  print(10);
  print(10.10);
  print("ten");
  return 0;
}
```

Output

```
Here is int 10
Here is float 10.1
Here is char* ten
```

C++

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

void add(int a, int b)
{
    cout<<"sum ="<<(a+b);
}

void  add(int a, int b,int c)
{
    cout<<endl<<"sum ="<<(a+b+c);
}


main()
{

    add(10,2);
    add(5,6,4);
    return 0;
}
```

C++

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

void add(int a, double b)
{
    cout<<"sum ="<<(a+b);
}
void  add(double a, int b)
{
    cout<<endl<<"sum ="<<(a+b);
}

main()
{
    add(10,2.5);
    add(5.5,6);
    return 0;
           }
```

How does Function Overloading work?

- *Exact match*:- (Function name and Parameter)
- *If a not exact match is found:-*

  ->Char, Unsigned char, and short are promoted to an int.

  ->Float is promoted to double

- *If no match is found*:

  ->C++ tries to find a match through the standard conversion.

- *ELSE ERROR* 😕

1. Function overloading and return type
2. Functions that cannot be overloaded in C++
3. Function overloading and const keyword
4. Function Overloading vs Function Overriding in C++

Recent articles on function overloading in C++

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

325

Related Articles

1. error: call of overloaded 'function(x)' is ambiguous | Ambiguity in Function overloading in C++
2. Function Overloading vs Function Overriding in C++
3. Function Overloading and Return Type in C++
4. Function overloading and const keyword
5. Overloading function templates in C++
6. Advantages and Disadvantages of Function Overloading in C++
7. Overloading of function-call operator in C++

Operator Overloading in C++

Difficulty Level : Medium • Last Updated : 05 Mar, 2023

Operator overloading is a compile-time polymorphism. It is an idea of giving special meaning to an existing operator in C++ without changing its original meaning.

In C++, we can make operators work for user-defined classes. This means C++ has the ability to provide the operators with a special meaning for a data type, this ability is known as operator overloading. For example, we can overload an operator '+' in a class like String so that we can concatenate two strings by just using +. Other example classes where arithmetic operators may be overloaded are Complex Numbers, Fractional Numbers, Big integers, etc.

Example:

```
int a;
float b,sum;
sum = a + b;
```

Here, variables "a" and "b" are of types "int" and "float", which are built-in data types. Hence the addition operator '+' can easily add the contents of "a" and "b". This is because the addition operator "+" is predefined to add variables of built-in data type only.

Example:

C++

```
// C++ Program to Demonstrate the
// working/Logic behind Operator
// Overloading
class A {
    statements;
};

int main()
{
    A a1, a2, a3;

    a3 = a1 + a2;

    return 0;
}
```

In this example, we have 3 variables "a1", "a2" and "a3" of type "class A". Here we are trying to add two objects "a1" and "a2", which are of user-defined type i.e. of type "class A" using the "+" operator. This is not allowed, because the addition operator "+" is predefined to operate only on built-in data types. But here, "class A" is a user-defined type, so the compiler generates an error. This is where the concept of "Operator overloading" comes in.

Now, if the user wants to make the operator "+" add two class objects, the user has to redefine the meaning of the "+" operator such that it adds two class objects. This is done by using the concept of "Operator overloading". So the main idea behind "Operator overloading" is to use C++ operators with class variables or class objects. Redefining the meaning of operators really does not change their original meaning; instead, they have been given additional meaning along with their existing ones.

Example:

C++

```
// C++ Program to Demonstrate
// Operator Overloading
#include <iostream>
using namespace std;

class Complex {
private:
    int real, imag;

public:
    Complex(int r = 0, int i = 0)
```

```
    {
        real = r;
        imag = i;
    }

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

int main()
{
    Complex c1(10, 5), c2(2, 4);
    Complex c3 = c1 + c2;
    c3.print();
}
```

Output

```
12 + i9
```

What is the difference between operator functions and normal functions?

Operator functions are the same as normal functions. The only differences are, that the name of an operator function is always the operator keyword followed by the symbol of the operator, and operator functions are called when the corresponding operator is used.

Example:

C++

```
#include <iostream>
using namespace std;
class Complex {
private:
    int real, imag;

public:
    Complex(int r = 0, int i = 0)
    {
        real = r;
        imag = i;
    }
    void print() { cout << real << " + i" << imag << endl; }
```

```
    // The global operator function is made friend of this
    // class so that it can access private members
    friend Complex operator+(Complex const& c1,
                             Complex const& c2);
};
Complex operator+(Complex const& c1, Complex const& c2)
{
    return Complex(c1.real + c2.real, c1.imag + c2.imag);
}
int main()
{
    Complex c1(10, 5), c2(2, 4);
    Complex c3
        = c1
          + c2; // An example call to "operator+"
    c3.print();
    return 0;
}
```

Output

```
12 + i9
```

Can we overload all operators?

Almost all operators can be overloaded except a few. Following is the list of operators that cannot be overloaded.

```
sizeof
typeid
Scope resolution (::)
Class member access operators (.(dot), .* (pointer to member operator))
Ternary or conditional (?:)
```

Operators that can be overloaded

We can overload

- **Unary operators**
- **Binary operators**
- **Special operators** ([ ], (), etc)

But, among them, there are some operators that cannot be overloaded. They are

- **Scope resolution operator (:** 🙂
- **Member selection operator**
- **Member selection through *

Pointer to a member variable

- **Conditional operator (? 🙂**
- **Sizeof operator sizeof()**

Operators that can be overloaded	Examples
Binary Arithmetic	+, -, *, /, %
Unary Arithmetic	+, -, ++, –
Assignment	=, +=,*=, /=,-=, %=
Bitwise	& , | , << , >> , ~ , ^
De-referencing	(->)
Dynamic memory allocation, De-allocation	New, delete
Subscript	[ ]
Function call	()
Logical	&, | |, !
Relational	>, < , = =, <=, >=

Why can't the above-stated operators be overloaded?

1. sizeof – This returns the size of the object or datatype entered as the operand. This is evaluated by the compiler and cannot be evaluated during runtime. The proper incrementing of a pointer in an array of objects relies on the sizeof operator implicitly. Altering its meaning using overloading would cause a fundamental part of the language to collapse.

2. typeid: This provides a CPP program with the ability to recover the actually derived type of the object referred to by a pointer or reference. For this operator, the whole point is to uniquely identify a type. If we want to make a user-defined type 'look' like another type,

polymorphism can be used but the meaning of the typeid operator must remain unaltered, or else serious issues could arise.

3. Scope resolution (::): This helps identify and specify the context to which an identifier refers by specifying a namespace. It is completely evaluated at runtime and works on names rather than values. The operands of scope resolution are note expressions with data types and CPP has no syntax for capturing them if it were overloaded. So it is syntactically impossible to overload this operator.

4. Class member access operators (.(dot), .* (pointer to member operator)): The importance and implicit use of class member access operators can be understood through the following example:

Example:

C++

```
// C++ program to demonstrate operator overloading
// using dot operator
#include <iostream>
using namespace std;

class ComplexNumber {
private:
    int real;
    int imaginary;

public:
    ComplexNumber(int real, int imaginary)
    {
        this->real = real;
        this->imaginary = imaginary;
    }
    void print() { cout << real << " + i" << imaginary; }
    ComplexNumber operator+(ComplexNumber c2)
    {
        ComplexNumber c3(0, 0);
        c3.real = this->real + c2.real;
        c3.imaginary = this->imaginary + c2.imaginary;
        return c3;
    }
};
int main()
{
    ComplexNumber c1(3, 5);
    ComplexNumber c2(2, 4);
    ComplexNumber c3 = c1 + c2;
    c3.print();
    return 0;
}
```

Output

```
5 + i9
```

Explanation:

The statement ComplexNumber c3 = c1 + c2; is internally translated as ComplexNumber c3 = c1.operator+ (c2); in order to invoke the operator function. The argument c1 is implicitly passed using the **'.'** operator. The next statement also makes use of the dot operator to access the member function print and pass c3 as an argument.

Besides, these operators also work on names and not values and there is no provision (syntactically) to overload them.

5. Ternary or conditional (?:): The ternary or conditional operator is a shorthand representation of an if-else statement. In the operator, the true/false expressions are only evaluated on the basis of the truth value of the conditional expression.

```
conditional statement ? expression1 (if statement is TRUE) : expression2
(else)
```

A function overloading the ternary operator for a class say ABC using the definition

```
ABC operator ?: (bool condition, ABC trueExpr, ABC falseExpr);
```

would not be able to guarantee that only one of the expressions was evaluated. Thus, the ternary operator cannot be overloaded.

Important points about operator overloading

1) For operator overloading to work, at least one of the operands must be a user-defined class object.

2) Assignment Operator: Compiler automatically creates a default assignment operator with every class. The default assignment operator does assign all members of the right side to the left side and works fine in most cases (this behavior is the same as the copy constructor). See this for more details.

3) Conversion Operator: We can also write conversion operators that can be used to convert one type to another type.

Example:

C++

```
// C++ Program to Demonstrate the working
```

```
// of conversion operator
#include <iostream>
using namespace std;
class Fraction {
private:
    int num, den;

public:
    Fraction(int n, int d)
    {
        num = n;
        den = d;
    }

    // Conversion operator: return float value of fraction
    operator float() const
    {
        return float(num) / float(den);
    }
};

int main()
{
    Fraction f(2, 5);
    float val = f;
    cout << val << '\n';
    return 0;
}
```

Output

```
 0.4
```

Overloaded conversion operators must be a member method. Other operators can either be the member method or the global method.

4) Any constructor that can be called with a single argument works as a conversion constructor, which means it can also be used for implicit conversion to the class being constructed.

Example:

C++

```
// C++ program to demonstrate can also be used for implicit
// conversion to the class being constructed
#include <iostream>
using namespace std;

class Point {
private:
    int x, y;
```

```
public:
    Point(int i = 0, int j = 0)
    {
        x = i;
        y = j;
    }
    void print()
    {
        cout << "x = " << x << ", y = " << y << '\n';
    }
};

int main()
{
    Point t(20, 20);
    t.print();
    t = 30; // Member x of t becomes 30
    t.print();
    return 0;
}
```

Output

```
x = 20, y = 20
x = 30, y = 0
```

Quiz on Operator Overloading

356

Related Articles

1. Operator Overloading '<<' and '>>' operator in a linked list class
2. Rules for operator overloading
3. Overloading Subscript or array index operator [] in C++
4. C++ | Operator Overloading | Question 10
5. Overloading New and Delete operator in c++
6. C++ Program to concatenate two strings using Operator Overloading