B.C.A. Semester – 4

BCA-404

Object Oriented Programming Using C++

UNIT - 4

Introduction to 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.

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:

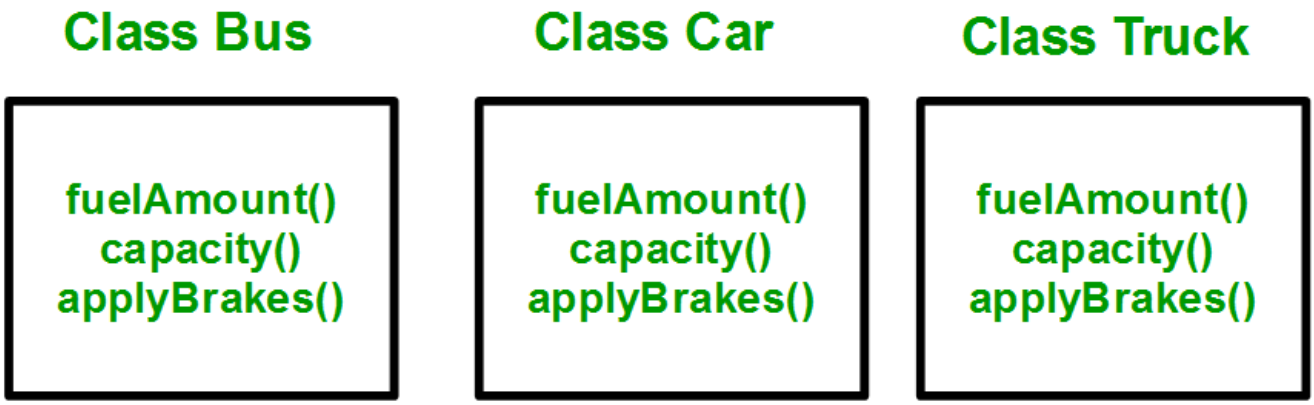
Why and when to use inheritance?

Modes of Inheritance

Types of Inheritance

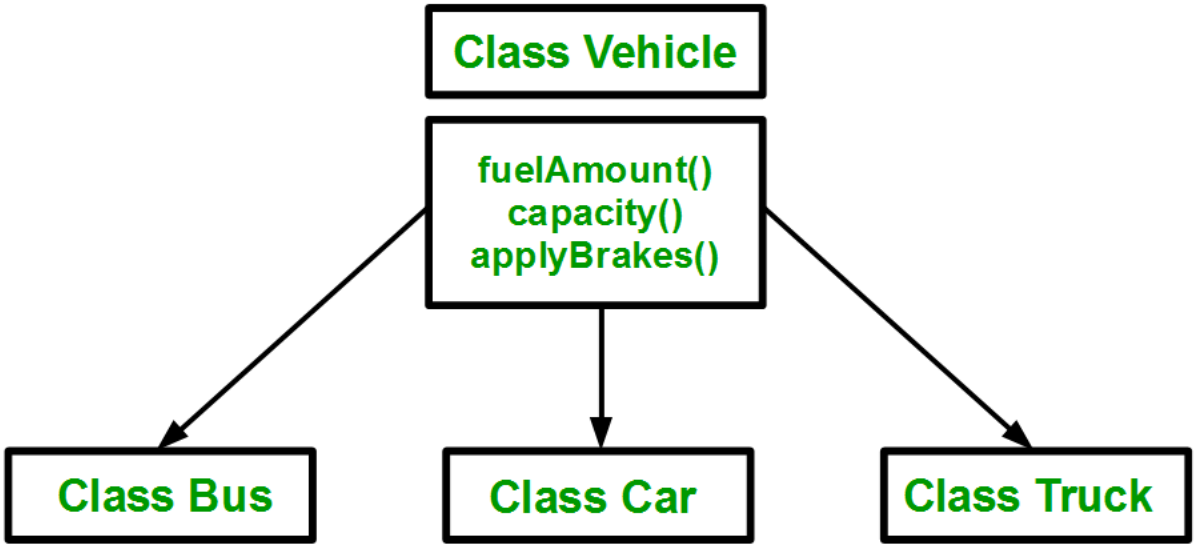
Why and when to use inheritance?

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;
      cout << "Enter the Name:";
```

```
        cin >> name;
    }

    void display_p()
    {
        cout << endl <<"Id: "<< id << "\nName: " << name <<endl;
    }
};

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

public:
    void set_s()
    {
        set_p();
        cout << "Enter the Course Name:";
        cin >> course;
        cout << "Enter the Course Fee:";
        cin >> fee;
    }

    void display_s()
    {
        display_p();
        cout <<"Course: "<< course << "\nFee: " << fee << endl;
```

```
    }
};


int 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

Id: 101
Name: Dev
Course: GCS
Fee: 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;
    cout<<"Enter the Name:";
    cin>>name;
}

void Person::display_p()
{
    cout<<endl<<"id: "<< id<<"\nName: "<<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:";
    cin>>course;
    cout<<"Enter the Course Fee:";
    cin>>fee;
}

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

int 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
Id: 101
Name: Dev
Course: GCS
Fee: 70000
```

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.

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.

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.

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

```
// 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:-

Single inheritance

Multilevel inheritance

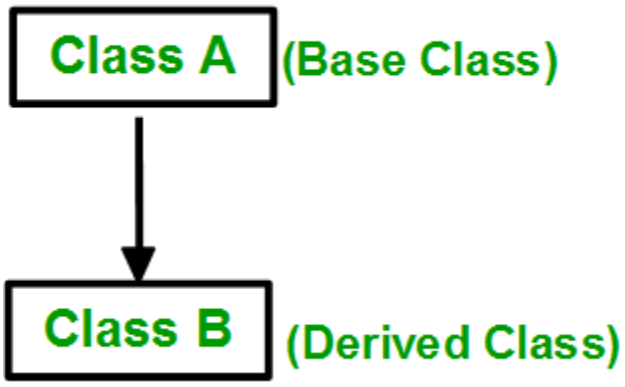
Multiple inheritance

Hierarchical inheritance

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:
```

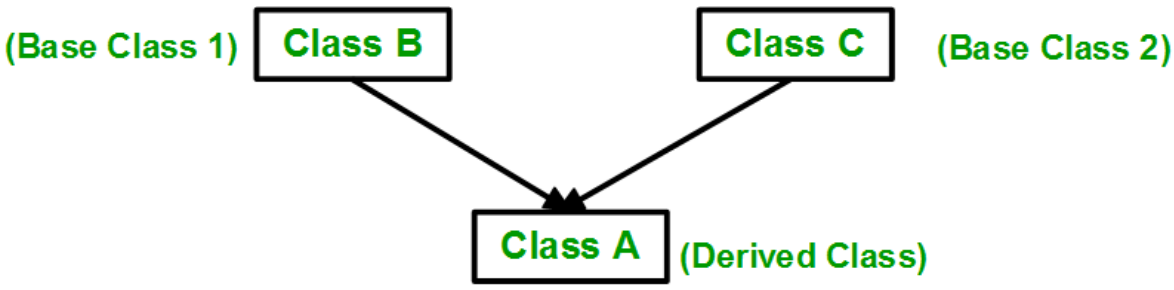
```
    void set_A(int x)
    {
        a=x;
    }

    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++

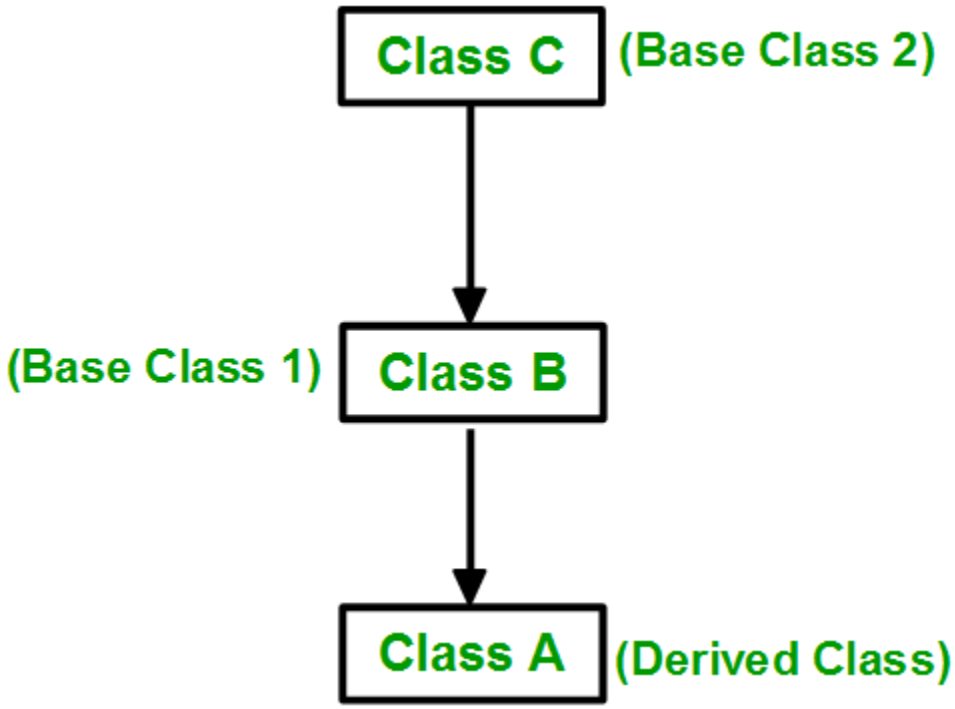
```
// Example:


#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

```
// 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;
}
```

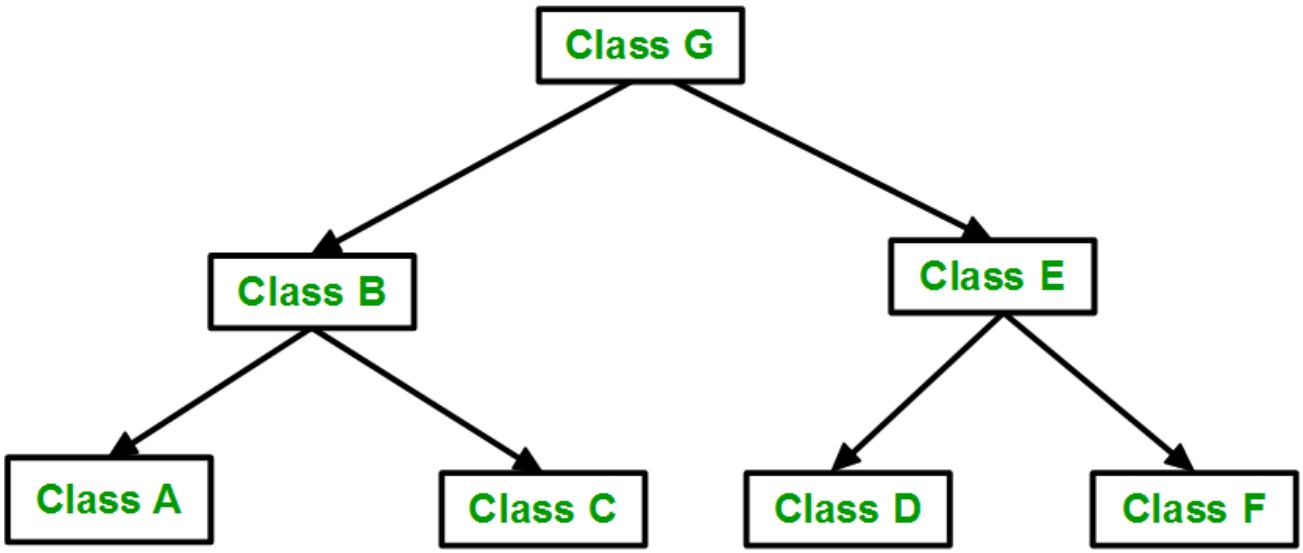
Output

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 {
};
```

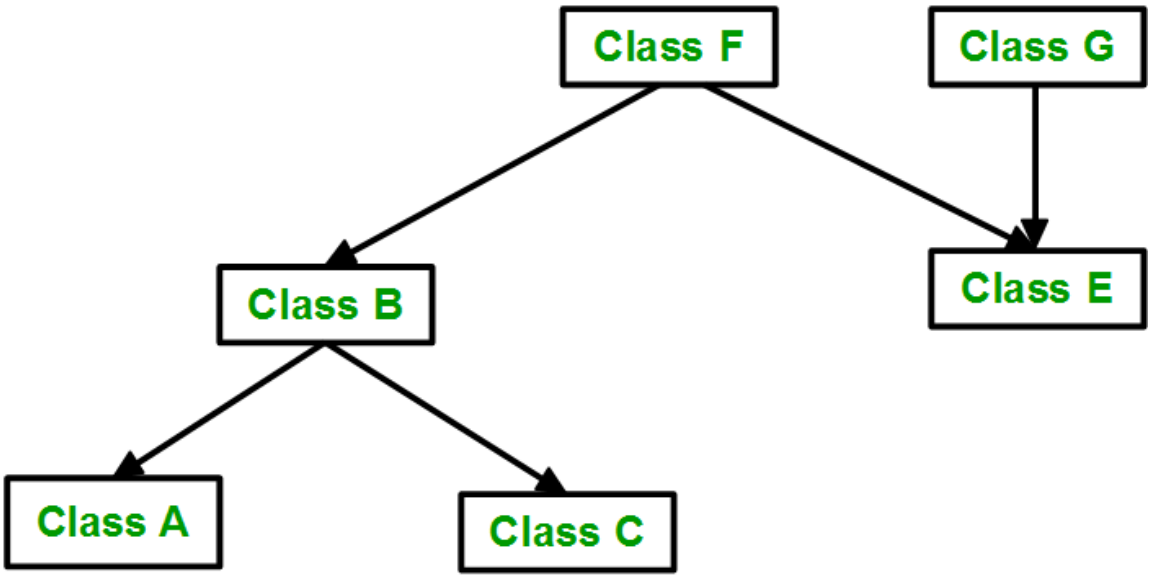
```
// main function
int main()
{
    // Creating object of sub class will
    // invoke the constructor of base class.
    Car obj1;
    Bus obj2;
    return 0;
}
```

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

```
// 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 : 4

- # Virtual Function in C++

A virtual function (also known as virtual methods) is a member function that is declared within a base class and is re-defined (overridden) by a derived class. When you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class's version of the method.

Virtual functions ensure that the correct function is called for an object, regardless of the type of reference (or pointer) used for the function call.

They are mainly used to achieve Runtime polymorphism.

Functions are declared with a virtual keyword in a base class.

The resolving of a function call is done at runtime.

Rules for Virtual Functions

The rules for the virtual functions in C++ are as follows:

Virtual functions cannot be static.

A virtual function can be a friend function of another class.

Virtual functions should be accessed using a pointer or reference of base class type to achieve runtime polymorphism.

The prototype of virtual functions should be the same in the base as well as the derived class.

They are always defined in the base class and overridden in a derived class. It is not mandatory for the derived class to override (or re-define the virtual function), in that case, the base class version of the function is used.

A class may have a virtual destructor but it cannot have a virtual constructor.

Compile time (early binding) VS runtime (late binding) behavior of Virtual Functions

Consider the following simple program showing the runtime behavior of virtual functions.

C++

```
// C++ program to illustrate
// concept of Virtual Functions

#include <iostream>
using namespace std;

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

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

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

```
    void show() { cout << "show derived class\n"; }
};


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

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

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

    return 0;
}
```

Output

print derived class

show base class

Explanation: Runtime polymorphism is achieved only through a pointer (or reference) of the base class type. Also, a base class pointer can point to the objects of the base class as well as to the objects of the derived class. In the above code, the base class pointer ‘bptr’ contains the address of object ‘d’ of the derived class.

Late binding (Runtime) is done in accordance with the content of the pointer (i.e. location pointed to by pointer) and Early binding (Compile-time) is done according to the type of pointer since the print() function is declared with the virtual keyword so it will be bound at runtime (output is print derived class as the pointer is pointing to object of derived class) and show() is non-virtual so it will be bound during compile time (output is show base class as the pointer is of base type).

Note: If we have created a virtual function in the base class and it is being overridden in the derived class then we don’t need a virtual keyword in the derived class, functions are automatically considered virtual functions in the derived class.

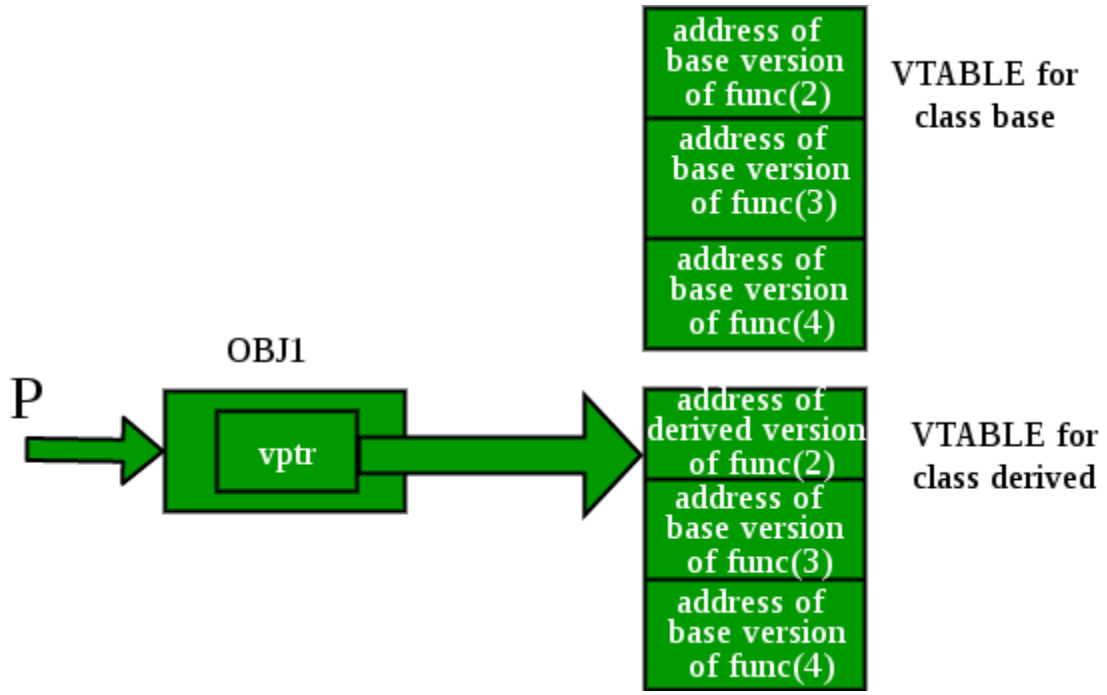
Working of Virtual Functions (concept of VTABLE and VPTR)

As discussed here, if a class contains a virtual function then the compiler itself does two things.

If an object of that class is created then a virtual pointer (VPTR) is inserted as a data member of the class to point to the VTABLE of that class. For each new object created, a new virtual pointer is inserted as a data member of that class.

Irrespective of whether the object is created or not, the class contains as a member a static array of function pointers called VTABLE. Cells of this table store the address of each virtual function contained in that class.

Consider the example below:



C++

```
// C++ program to illustrate
// working of Virtual Functions
#include <iostream>
```

```
using namespace std;

class base {
public:
   void fun_1() { cout << "base-1\n"; }
   virtual void fun_2() { cout << "base-2\n"; }
   virtual void fun_3() { cout << "base-3\n"; }
   virtual void fun_4() { cout << "base-4\n"; }
};

class derived : public base {
public:
   void fun_1() { cout << "derived-1\n"; }
   void fun_2() { cout << "derived-2\n"; }
   void fun_4(int x) { cout << "derived-4\n"; }
};

int main()
{
   base* p;
   derived obj1;
   p = &obj1;

   // Early binding because fun1() is non-virtual
   // in base
   p->fun_1();
```

```
    // Late binding (RTP)
    p->fun_2();

    // Late binding (RTP)
    p->fun_3();

    // Late binding (RTP)
    p->fun_4();

    // Early binding but this function call is
    // illegal (produces error) because pointer
    // is of base type and function is of
    // derived class
    // p->fun_4(5);

    return 0;
}
```

Output

base-1

derived-2

base-3

base-4

Explanation: Initially, we create a pointer of the type base class and initialize it with the address of the derived class object. When we create an object of the derived class, the compiler creates a pointer as a data member of the class containing the address of VTABLE of the derived class.

A similar concept of Late and Early Binding is used as in the above example. For the fun_1() function call, the base class version of the function is called, fun_2() is overridden in the derived class so the derived class version is called, fun_3() is not overridden in the derived class and is a virtual function so the base class version is called, similarly fun_4() is not overridden so base class version is called.

Note: fun_4(int) in the derived class is different from the virtual function fun_4() in the base class as prototypes of both functions are different.

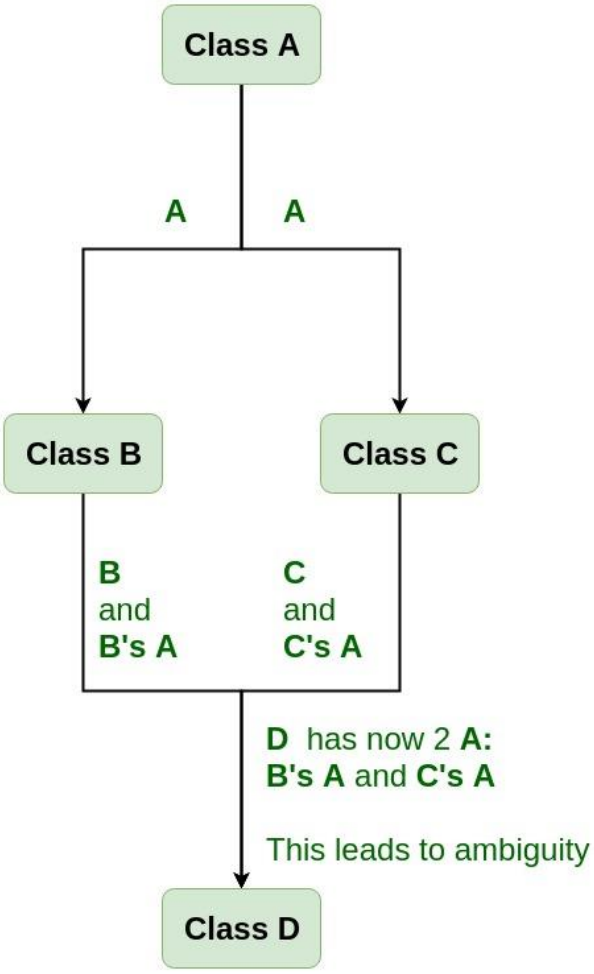
Limitations of Virtual Functions

Slower: The function call takes slightly longer due to the virtual mechanism and makes it more difficult for the compiler to optimize because it does not know exactly which function is going to be called at compile time.

Difficult to Debug: In a complex system, virtual functions can make it a little more difficult to figure out where a function is being called from

Virtual base class in C++

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



inheritances.

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

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

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

CPP14

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

class A {
public:
   void show()
   {
      cout << "Hello form A \n";
   }
};

class B : public A {
};

class C : public A {
};

class D : public B, public C {
```

```
};


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

Compile Errors:

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

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

Syntax for Virtual Base Classes:

Syntax 1:

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

Syntax 2:

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

Note:
virtual can be written before or after the public. Now only one copy of data/function member will be copied to class C and class B and class A becomes the virtual base class. Virtual base classes offer a way to save space and avoid ambiguities in class hierarchies that use multiple inheritances. When a base class is specified as a virtual base, it can act as an indirect base more than once without duplication of its data members. A single copy of its data members is shared by all the base classes that use virtual base.

Example 1

CPP14

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

class A {
public:
   int a;
   A() // constructor
   {
      a = 10;
   }
};

class B : public virtual A {
};

class C : public virtual A {
};

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

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

    return 0;
}
```

Output

a = 10

Explanation :
The class A has just one data member a which is public. This class is virtually inherited in class B and class C. Now class B and class C use the virtual base class A and no duplication of data member a is done; Classes B and C share a single copy of the members in the virtual base class A.

Example 2:

CPP14

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

class A {
public:
    void show()
    {
        cout << "Hello from A \n";
    }
};
```

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

class C : public virtual A {
};

class D : public B, public C {
};

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

Output

Hello from A

What is an abstract class in C++?

By definition, a C++ abstract class must include at least one pure virtual function. Alternatively, put a function without a definition. Because the subclass would otherwise turn into an abstract class in and of itself, the abstract class's descendants must specify the pure virtual function.

Broad notions are expressed using abstract classes, which can then be utilized to construct more specific classes. You cannot make an object of the abstract class type. However, pointers and references can be used to abstract class types. When developing an abstract class, define at least one pure virtual feature. A virtual function is declared using the pure specifier (= 0) syntax.

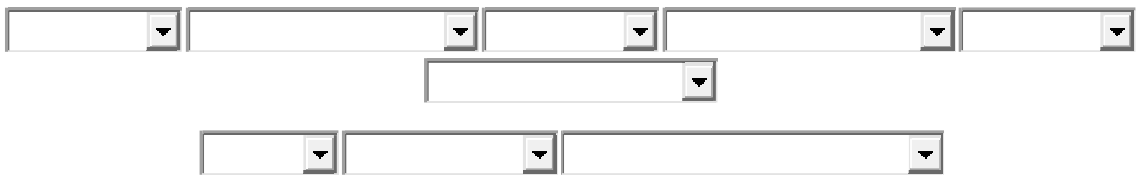
Consider the example of the virtual function. Although the class's objective is to provide basic functionality for shapes, elements of type shapes are far too general to be of much value. Because of this, the shape is a good candidate for an abstract class:

Code

```
C-lass classname //abstract class
{
//data members
public:
//pure virtual function
/* Other members */
};
```

What are the characteristics of abstract class?

Although the Abstract class type cannot be created from scratch, it can have pointers and references made to it. A pure virtual function can exist in an abstract class in addition to regular functions and variables. Upcasting, which lets derived classes access their interface, is the main usage of abstract classes. Classes that descended from an abstract class must implement all pure virtues.



What are the restrictions to abstract class?

The following uses of abstract classes are not permitted:

1. **Conversions made consciously**

2. **Member data or variables**
3. **Types of function output**
4. **Forms of debate**

It is unknown what happens when a pure virtual method is called explicitly or indirectly by the native code function Object () of an abstract class. Conversely, abstract group constructors and destructors can call additional member functions.

Although the constructors of the abstract class are allowed to call other member functions, if they either directly or indirectly call a pure virtual function, the outcome is unknown. But hold on! What exactly is a pure virtual function?

Let's first examine virtual functions in order to comprehend the pure virtual function.

A member function that has been redefined by a derived class from a base class declaration is referred to as a virtual function.

- # Pointers

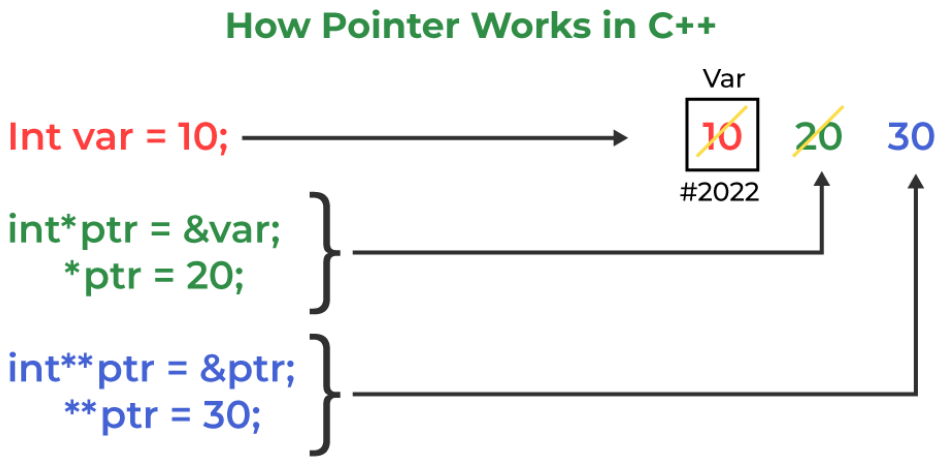
Pointers are symbolic representations of addresses. They enable programs to simulate call-by-reference as well as to create and manipulate dynamic data structures. Iterating over elements in arrays or other data structures is one of the main use of pointers.

The address of the variable you're working with is assigned to the pointer variable that points to the same data type (such as an int or string).

Syntax:

datatype *var_name;

int *ptr; // ptr can point to an address which holds int data



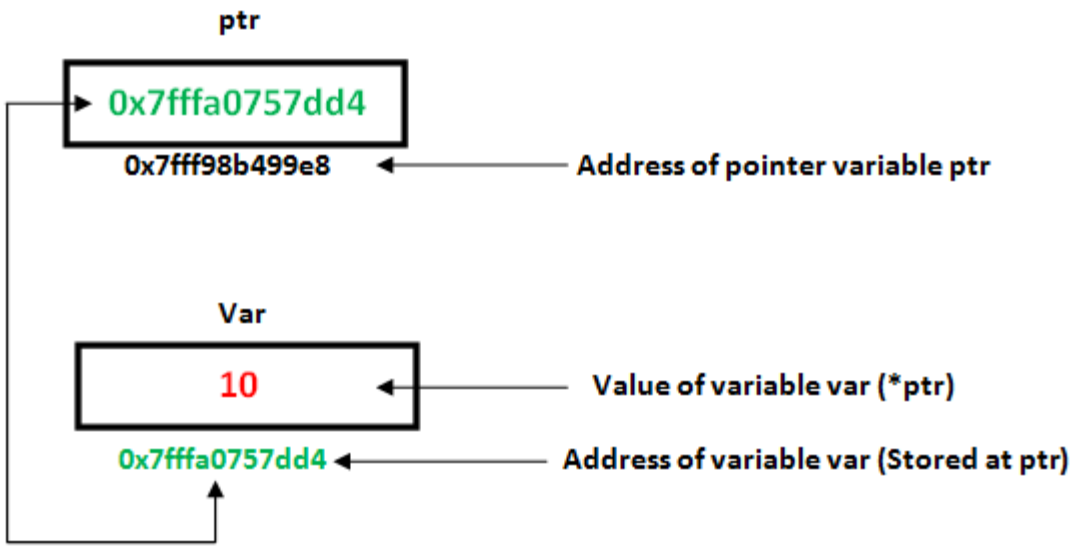
How to use a pointer?

Define a pointer variable

Assigning the address of a variable to a pointer using the unary operator (&) which returns the address of that variable.

Accessing the value stored in the address using unary operator (*) which returns the value of the variable located at the address specified by its operand.

The reason we associate data type with a pointer is that it knows how many bytes the data is stored in. When we increment a pointer, we increase the pointer by the size of the data type to which it points.



C++

```
// C++ program to illustrate Pointers

#include <bits/stdc++.h>
using namespace std;
void geeks()
{
    int var = 20;

    // declare pointer variable
    int* ptr;

    // note that data type of ptr and var must be same
    ptr = &var;

    // assign the address of a variable to a pointer
    cout << "Value at ptr = " << ptr << "\n";
    cout << "Value at var = " << var << "\n";
    cout << "Value at *ptr = " << *ptr << "\n";
}
// Driver program
int main()
{
  geeks();
  return 0;
}
```

Output

Value at ptr = 0x7ffe454c08cc

Value at var = 20

Value at *ptr = 20

References and Pointers

There are 3 ways to pass C++ arguments to a function:

Call-By-Value

Call-By-Reference with a Pointer Argument

Call-By-Reference with a Reference Argument

C++

```
// C++ program to illustrate call-by-methods

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

// Pass-by-Value
int square1(int n)
{
    // Address of n in square1() is not the same as n1 in
    // main()
    cout << "address of n1 in square1(): " << &n << "\n";

    // clone modified inside the function
    n *= n;
    return n;
}
```

```
// Pass-by-Reference with Pointer Arguments
void square2(int* n)
{
   // Address of n in square2() is the same as n2 in main()
   cout << "address of n2 in square2(): " << n << "\n";

   // Explicit de-referencing to get the value pointed-to
   *n *= *n;
}
// Pass-by-Reference with Reference Arguments
void square3(int& n)
{
   // Address of n in square3() is the same as n3 in main()
   cout << "address of n3 in square3(): " << &n << "\n";

   // Implicit de-referencing (without '*')
   n *= n;
}
void geeks()
{
   // Call-by-Value
   int n1 = 8;
   cout << "address of n1 in main(): " << &n1 << "\n";
   cout << "Square of n1: " << square1(n1) << "\n";
   cout << "No change in n1: " << n1 << "\n";

   // Call-by-Reference with Pointer Arguments
```

```
    int n2 = 8;
    cout << "address of n2 in main(): " << &n2 << "\n";
    square2(&n2);
    cout << "Square of n2: " << n2 << "\n";
    cout << "Change reflected in n2: " << n2 << "\n";

    // Call-by-Reference with Reference Arguments
    int n3 = 8;
    cout << "address of n3 in main(): " << &n3 << "\n";
    square3(n3);
    cout << "Square of n3: " << n3 << "\n";
    cout << "Change reflected in n3: " << n3 << "\n";
}
// Driver program
int main() { geeks(); }
```

Output

```
address of n1 in main(): 0x7fffa7e2de64
address of n1 in square1(): 0x7fffa7e2de4c
Square of n1: 64
No change in n1: 8
address of n2 in main(): 0x7fffa7e2de68
address of n2 in square2(): 0x7fffa7e2de68
Square of n2: 64
Change reflected in n2: 64
address of n3 in main(): 0x7fffa7e2de6c
address of n3 in square3(): 0x7fffa7e2de6c
Square of n3: 64
```

Change reflected in n3: 64

In C++, by default arguments are passed by value and the changes made in the called function will not reflect in the passed variable. The changes are made into a clone made by the called function. If wish to modify the original copy directly (especially in passing huge object or array) and/or avoid the overhead of cloning, we use pass-by-reference. Pass-by-Reference with Reference Arguments does not require any clumsy syntax for referencing and dereferencing.

Function pointers in C

Pointer to a Function

Array Name as Pointers

An array name contains the address of the first element of the array which acts like a constant pointer. It means, the address stored in the array name can't be changed. For example, if we have an array named val then val and &val[0] can be used interchangeably.

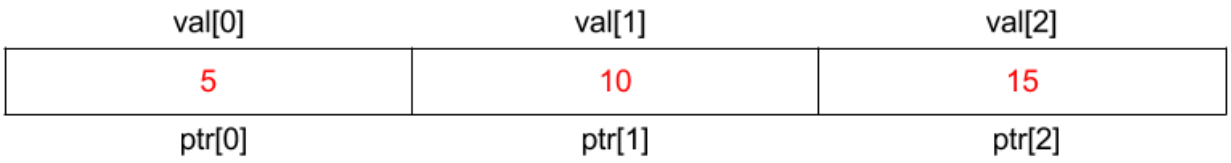
C++

```
// C++ program to illustrate Array Name as Pointers
#include <bits/stdc++.h>
using namespace std;
void geeks()
{
   // Declare an array
   int val[3] = { 5, 10, 20 };

   // declare pointer variable
   int* ptr;

   // Assign the address of val[0] to ptr
   // We can use ptr=&val[0];(both are same)
   ptr = val;
   cout << "Elements of the array are: ";
```

```
    cout << ptr[0] << " " << ptr[1] << " " << ptr[2];
}
// Driver program
int main() { geeks(); }
```

Output

Elements of the array are: 5 10 20



If pointer ptr is sent to a function as an argument, the array val can be accessed in a similar fashion. Pointer vs Array

Pointer Expressions and Pointer Arithmetic

A limited set of arithmetic operations can be performed on pointers which are:

incremented (++)

decremented (—)

an integer may be added to a pointer (+ or +=)

an integer may be subtracted from a pointer (– or -=)

difference between two pointers (p1-p2)

(Note: Pointer arithmetic is meaningless unless performed on an array.)

C++

```
// C++ program to illustrate Pointer Arithmetic
#include <bits/stdc++.h>
using namespace std;
void geeks()
```

```
{
    // Declare an array
    int v[3] = { 10, 100, 200 };

    // declare pointer variable
    int* ptr;

    // Assign the address of v[0] to ptr
    ptr = v;

    for (int i = 0; i < 3; i++) {
        cout << "Value at ptr = " << ptr << "\n";
        cout << "Value at *ptr = " << *ptr << "\n";

        // Increment pointer ptr by 1
        ptr++;
    }
}

// Driver program
int main() { geeks(); }
```

Output

Value at ptr = 0x7ffe5a2d8060

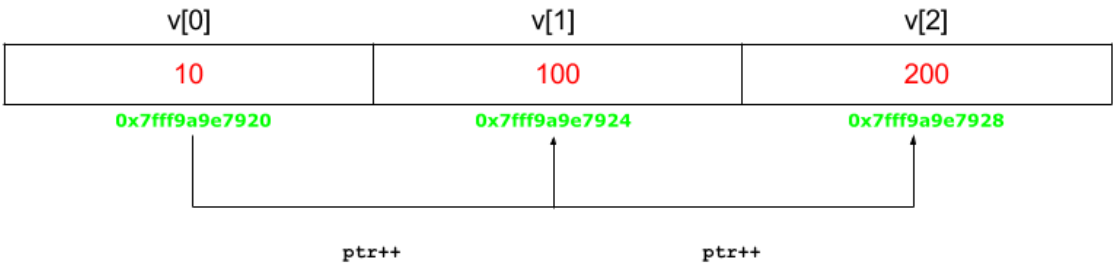
Value at *ptr = 10

Value at ptr = 0x7ffe5a2d8064

Value at *ptr = 100

Value at ptr = 0x7ffe5a2d8068

Value at *ptr = 200



Advanced Pointer Notation

Consider pointer notation for the two-dimensional numeric arrays. consider the following declaration

int nums[2][3] = { { 16, 18, 20 }, { 25, 26, 27 } };

In general, nums[i][j] is equivalent to *(*(nums+i)+j)

Pointer Notation	Array Notation	Value
*(*nums)	nums[0] [0]	16
*(*nums+1)	nums[0] [1]	18
*(*nums+2)	nums[0] [2]	20
((nums + 1))	nums[1] [0]	25
((nums + 1)+1)	nums[1] [1]	26
((nums + 1)+2)	nums[1] [2]	27

Pointers and String literals

String literals are arrays containing null-terminated character sequences. String literals are arrays of type character plus terminating null-character, with each of the elements being of type const char (as characters of string can't be modified).

This declares an array with the literal representation for "geek", and then a pointer to its first element is assigned to ptr. If we imagine that "geek" is stored at the memory locations that start at address 1800, we can represent the previous declaration as:

'g'	'e'	'e'	'k'	'\0'
1800	1801	1802	1803	1804

As pointers and arrays behave in the same way in expressions, ptr can be used to access the characters of a string literal. For example:

char x = *(ptr+3);

char y = ptr[3];

Here, both x and y contain k stored at 1803 (1800+3).

Pointers to pointers

In C++, we can create a pointer to a pointer that in turn may point to data or another pointer. The syntax simply requires the unary operator (*) for each level of indirection while declaring the pointer.

char a;

char *b;

char ** c;

a = 'g';

b = &a;

c = &b;

Here b points to a char that stores 'g' and c points to the pointer b.

Void Pointers

This is a special type of pointer available in C++ which represents the absence of type. Void pointers are pointers that point to a value that has no type (and thus also an undetermined length and undetermined dereferencing properties). This means that void pointers have great flexibility as they can point to any data type. There is a payoff for this flexibility. These pointers cannot be directly dereferenced. They have to be first transformed into some other pointer type that points to a concrete data type before being dereferenced.

C++

```
// C++ program to illustrate Void Pointer
```

```
#include <bits/stdc++.h>
using namespace std;
void increase(void* data, int ptrsize)
{
    if (ptrsize == sizeof(char)) {
        char* ptrchar;

        // Typecast data to a char pointer
        ptrchar = (char*)data;

        // Increase the char stored at *ptrchar by 1
        (*ptrchar)++;
        cout << "*data points to a char"
             << "\n";
    }
    else if (ptrsize == sizeof(int)) {
        int* ptrint;

        // Typecast data to a int pointer
        ptrint = (int*)data;

        // Increase the int stored at *ptrchar by 1
        (*ptrint)++;
        cout << "*data points to an int"
             << "\n";
    }
}
```

```
void geek()
{
    // Declare a character
    char c = 'x';

    // Declare an integer
    int i = 10;

    // Call increase function using a char and int address
    // respectively
    increase(&c, sizeof(c));
    cout << "The new value of c is: " << c << "\n";
    increase(&i, sizeof(i));
    cout << "The new value of i is: " << i << "\n";
}
// Driver program
int main() { geek(); }
```

Output

*data points to a char

The new value of c is: y

*data points to an int

The new value of i is: 11

Invalid pointers

A pointer should point to a valid address but not necessarily to valid elements (like for arrays). These are called invalid pointers. Uninitialized pointers are also invalid pointers.

int *ptr1;

int arr[10];

int *ptr2 = arr+20;

Here, ptr1 is uninitialized so it becomes an invalid pointer and ptr2 is out of bounds of arr so it also becomes an invalid pointer. (Note: invalid pointers do not necessarily raise compile errors)

NULL Pointers

A null pointer is a pointer that point nowhere and not just an invalid address. Following are 2 methods to assign a pointer as NULL;

int *ptr1 = 0;

int *ptr2 = NULL;

Advantages of Pointers

Pointers reduce the code and improve performance. They are used to retrieve strings, trees, arrays, structures, and functions.

Pointers allow us to return multiple values from functions.

In addition to this, pointers allow us to access a memory location in the computer's memory.

- ## This pointers

To understand 'this' pointer, it is important to know how objects look at functions and data members of a class.

Each object gets its own copy of the data member.

All-access the same function definition as present in the code segment.

Meaning each object gets its own copy of data members and all objects share a single copy of member functions. Then now question is that if only one copy of each member function exists and is used by multiple objects, how are the proper data members are accessed and updated? The compiler supplies an implicit pointer along with the names of the functions as 'this'. The 'this' pointer is passed as a hidden argument to all nonstatic member function calls and is available as a local variable within the body of all nonstatic functions. 'this' pointer is not available in static member functions as static member functions can be called without any object (with class name). For a class X, the type of this pointer is 'X* '. Also, if a member function of X is declared as const, then the type of this pointer is 'const X *' (see this GFact) In the early version of C++ would let 'this' pointer to be changed; by doing so a programmer could change which object a method was working on. This feature was eventually removed, and now this in C++ is an r-value. C++ lets object destroy themselves by calling the following code :

```
delete this;
```

As Stroustrup said ‘this’ could be the reference than the pointer, but the reference was not present in the early version of C++. If ‘this’ is implemented as a reference then, the above problem could be avoided and it could be safer than the pointer. Following are the situations where ‘this’ pointer is used: 1) When local variable’s name is same as member’s name

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

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

int main()
{
  Test obj;
```

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

Output:

x = 20

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

2) To return reference to the calling object

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

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

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

class Test
{
private:
  int x;
```

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

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

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

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

Output:

x = 10 y = 20

Virtual function vs Pure virtual function in C++

Before understanding the differences between the virtual function and pure virtual function in C++, we should know about the virtual function and pure virtual function in C++.

What is virtual function?

Virtual function is a member function that is declared within the base class and can be redefined by the derived class.

Let's understand through an example.

```
1.  #include <iostream>
2.  using namespace std;
3.  class base
4.  {
5.     public:
6.     void show()
7.     {
8.        std::cout << "Base class" << std::endl;
9.     }
10. };
11. class derived1 : public base
12. {
13.    public:
14.    void show()
15.    {
16.       std::cout << "Derived class 1" << std::endl;
17.    }
18. };
19. class derived2 : public base
20. {
21.    public:
22.    void show()
23.    {
24.       std::cout << "Derived class 2" << std::endl;
25.    }
26. };
27. int main()
28. {
```

```
29.     base *b;
30.     derived1 d1;
31.     derived2 d2;
32.     b=&d1;
33.     b->show();
34.     b=&d2;
35.     b->show();
36.     return 0;
37. }
```

In the above code, we have not used the virtual method. We have created a base class that contains the show() function. The two classes are also created named as '**derived1**' and '**derived2**' that are inheriting the properties of the base class. Both 'derived1' and 'derived2' classes have redefined the show() function. Inside the main() method, pointer variable 'b' of class base is declared. The objects of classes derived1 and derived2 are d1 and d2 respectively. Although the 'b' contains the addresses of d1 and d2, but when calling the show() method; it always calls the show() method of the base class rather than calling the functions of the derived1 and derived2 class.

To overcome the above problem, we need to make the method as virtual in the base class. Here, virtual means that the method exists in appearance but not in reality. We can make the method as virtual by simply adding the virtual keyword preceeding to the function. In the above program, we need to add the virtual keyword that precedes to the show() function in the base class shown as below:

```
1. virtual void show()
2. {
3.     std::cout << "Base class" << std::endl;
4. }
```

Once the above changes are made, the output would be:

Important points:

- It is a run-time polymorphism.
- Both the base class and the derived class have the same function name, and the base class is assigned with an address of the derived class object then also pointer will execute the base class function.

- If the function is made virtual, then the compiler will determine which function is to execute at the run time on the basis of the assigned address to the pointer of the base class.

What is pure virtual function?

A pure virtual function is a virtual function that has no definition within the class. Let's understand the concept of pure virtual function through an example.

In the above pictorial representation, shape is the base class while rectangle, square and circle are the derived class. Since we are not providing any definition to the virtual function, so it will automatically be converted into a pure virtual function.

Characteristics of a pure virtual function

- A pure virtual function is a "do nothing" function. Here "do nothing" means that it just provides the template, and derived class implements the function.
- It can be considered as an empty function means that the pure virtual function does not have any definition relative to the base class.
- Programmers need to redefine the pure virtual function in the derived class as it has no definition in the base class.
- A class having pure virtual function cannot be used to create direct objects of its own. It means that the class is containing any pure virtual function then we cannot create the object of that class. This type of class is known as an abstract class.

Syntax

There are two ways of creating a virtual function:

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

or

1. `virtual void display() {}`

Let's understand through an example.

```
1. #include <iostream>
2. using namespace std;
```

```
3.  // Abstract class
4.  class Shape
5.  {
6.      public:
7.      virtual float calculateArea() = 0; // pure virtual function.
8.  };
9.  class Square : public Shape
10. {
11.     float a;
12.     public:
13.     Square(float l)
14.     {
15.         a = l;
16.     }
17.     float calculateArea()
18.     {
19.         return a*a;
20.     }
21. };
22. class Circle : public Shape
23. {
24.     float r;
25.     public:
26.
27.     Circle(float x)
28.     {
29.         r = x;
30.     }
31.     float calculateArea()
32.     {
33.         return 3.14*r*r ;
34.     }
35. };
36. class Rectangle : public Shape
```

```
{
    float l;
    float b;
    public:
    Rectangle(float x, float y)
    {
       l=x;
       b=y;
    }
    float calculateArea()
    {
        return l*b;
    }
};
int main()
{

    Shape *shape;
    Square s(3.4);
    Rectangle r(5,6);
    Circle c(7.8);
    shape =&s;
    int a1 =shape->calculateArea();
    shape = &r;
    int a2 = shape->calculateArea();
    shape = &c;
    int a3 = shape->calculateArea();
    std::cout << "Area of the square is " <<a1<< std::endl;
    std::cout << "Area of the rectangle is " <<a2<< std::endl;
    std::cout << "Area of the circle is " <<a3<< std::endl;
    return 0;
}
```

Differences between the virtual function and pure virtual function

Virtual function	Pure virtual function
A virtual function is a member function of base class which can be redefined by derived class.	A pure virtual function is a member function of base class whose only declaration is provided in base class and should be defined in derived class otherwise derived class also becomes abstract.
Classes having virtual functions are not abstract.	Base class containing pure virtual function becomes abstract.

Syntax:
`virtual<func_type><func_name>()`
`{`
`// code`
`}`
	Syntax:
	`virtual<func_type><func_name>()`
	`= 0;`

Definition is given in base class.	No definition is given in base class.
Base class having virtual function can be instantiated i.e. its object can be made.	Base class having pure virtual function becomes abstract i.e. it cannot be instantiated.
If derived class do not redefine virtual function of base class, then it does not affect compilation.	If derived class do not redefine virtual function of base class, then no compilation error but derived class also becomes abstract just like the base class.
All derived class may or may not redefine virtual function of base class.	All derived class must redefine pure virtual function of base class otherwise derived class also becomes abstract just like base class.