UNIT 5:

Derived Classes and Inheritance

What is a Base Class?

In an object-oriented programming language, a base class is an existing class from which the other classes are determined and properties are inherited. It is also known as a superclass or parent class. In general, the class which acquires the base class can hold all its members and some further data as well.

Syntax: Class base_classname{ … }.

What is a Derived Class?

A derived class is a class that is constructed from a base class or an existing class. It has a tendency to acquire all the methods and properties of a base class. It is also known as a subclass or child class.

Syntax: Class derived_classname : access_mode base_class_name { … }.

Difference between Base Class and Derived Class in C++

S.No.	**Base Class**	**Derived Class**
1.	A base class is an existing class from which the other classes are derived and inherit the methods and properties.	A derived class is a class that is constructed from a base class or an existing class.
2.	Base class can't acquire the methods and properties of the derived class.	Derived class can acquire the methods and properties of the base class.
3.	The base class is also called superclass or parent class.	The derived class is also called a subclass or child class.

Defining of Derived Classes in C++

A derived class is defined by specifying it relationship with the base class in addition to its own details.
The general form of defining a derived class is:

```
class derived-class-name : visibility-mode base-class-name

   {

      ...

      ...   //members of derived class

      ...

   };
```

where,
class is the required keyword,
derived-class-name is the name given to the derived class,
base-class-name is the name given to the base class,
: (colon) indicates that the derived-class-name is derived from the base-class-name,
visibility-mode is optional and, if present, may be either private or public. The default visibility-mode is private. Visibility mode specifies whether the features of the base class are privately derived or publicly derived.
Examples:

```
class ABC : private XYZ    //private derivation

   {

      members of ABC
```

```
    };



class ABC : public XYZ     //public derivation

    {

        members of ABC

    };



class ABC : XYZ     //private derivation by default

    {

        members of ABC

    };
```

When a base class is **privately inherited** by a derived class, ‘public members’ of the base class becomes 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 of the derived class. They are inaccessible to the objects of the derived class. Remember, a public member of the class can be accessed by its own objects by using the dot operator. The result is that no member of the base class is accessible to the objects of the derived class in case of private derivation.

On the other hand, when the base class is **publicly inherited**, ‘public members’ of the base class becomes ‘public members’ of the derived class and therefore they are accessible to the objects of the derived class.
In both the cases, **the private members are not inherited and therefore, the private members of a base class will never become the members of its derived class**.

In inheritance, some of the base class data elements and member functions are inherited into the derived clas. We can also add our own data and member functions and thus extend the functionality of the base class. Inheritance, when used to modify and extend the capabilities of the existing classes, becomes a very powerful tool for incremental program development.

Accessing of Base class members

The base class members can be accessed by its sub-classes through access specifiers. There are three types of access specifies. They are public, private and protected.

1. Public
When the base class is publicly inherited, the public members of the base class become the derived class public members.
They can be accessed by the objects of the derived class.

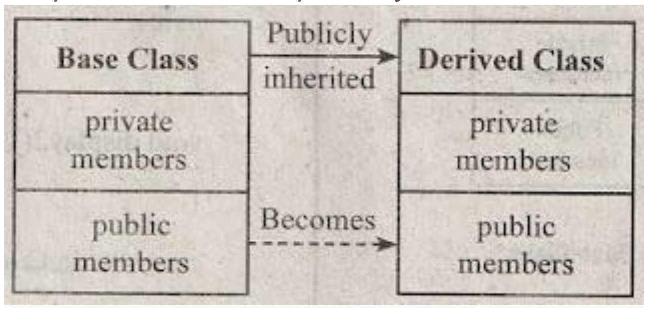


Figure: Publicly Inherited Base Class

Syntax

```
class classname
{
public:
datatype variablename;
returntype functionname();
};
```

2. Protected
When the base class is derived in protected mode, the ‘protected’ and ‘public’ members of the base class become the protected members of the derived class.
Private and protected members of a class can be accessed by,
(i) A friend function of the class.
(ii) A member function of the friend class.
(iii) A derived class member function.

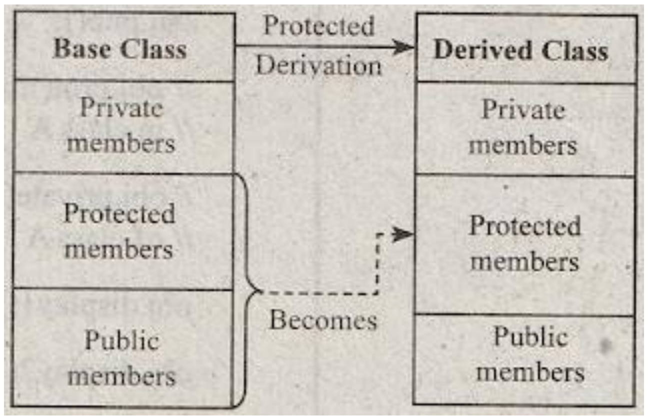


Figure: Protected Derivation of the Base Class

Also read :This Pointer in C++

Syntax

```
class classname
{
protected: :
datatype variablename;
returntype functionname();
};
```

3. Private

When a base class contains members, that are declared as private, they cannot be accessed by the derived class objects. They can be accessed only by the class in which they are defined.

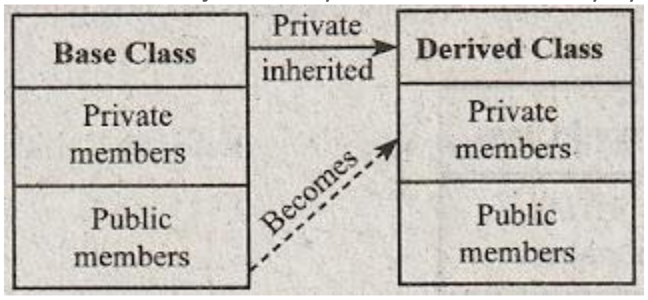


Figure: Privately inherited Base Class

Syntax

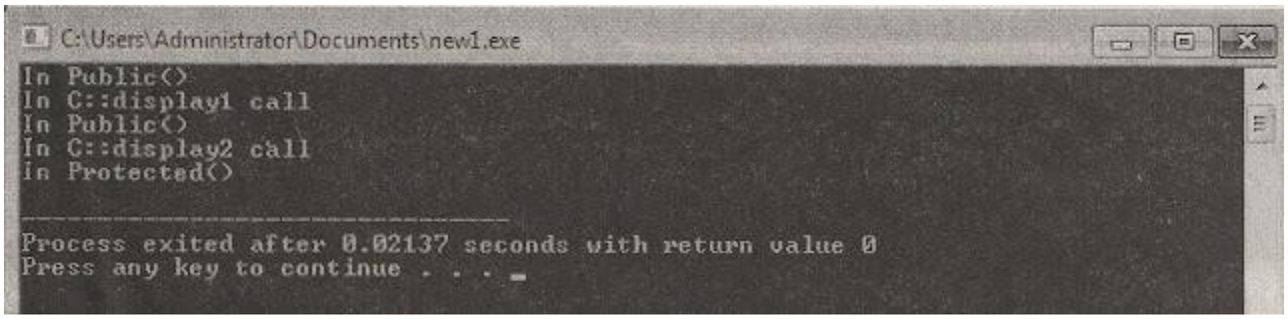
```
class classname
{
private:
datatype variablename;
returntype functionname( );
};
```

Program

```
#include <iostream>
```

```
using namespace std;
class A
{
public: pub()
{
cout << “In Public() \n”;
}
protected: prot()
{
cout << “In Protected() \n”;
}
private: priv()
{
cout << “In Pivate() \n”;
}
};
class C : public A
{
public:
void display1()
{
cout<< “In C::display1 call\n”;
pub();
}
void display2()
{
cout << “In C::display2 call\n”;
prot();
}
/*pri(Q is a private member of class A. Therefore it is an illegal access
void display3()
{
cout<<“In C::display3 call\n”;
priv();
} */
}
main()
{
C obj;
obj.pub();
// obj.prot(); illegal because it is declared as protected in class A
// obj.private(); illegal because pri() isa private member of class A
obj.display1();
obj.display2();
}
```

Output



```
C:\Users\Administrator\Documents\new1.exe
In Public()
In C::display1 call
In Public()
In C::display2 call
In Protected()

--------------------------------
Process exited after 0.02137 seconds with return value 0
Press any key to continue . . .
```

Acess Specifiers in C++

Access specifiers in C++ are used to define the level of access that a class member variable or function can have. In C++, there are three access specifiers: public, private, and protected. The public access specifier is used to specify that a class member can be accessed from anywhere, both inside and outside the class. This means that any function or object can access the public members of the class. The public members of a class are typically used to represent the interface of the class. The private access specifier is used to specify that a class member can only be accessed from within the class. This means that any function or object outside the class cannot access the private members of the class. The private members of a class are typically used to represent the implementation of the class and are hidden from the outside world.

The protected access specifier is used to specify that a class member can be accessed from within the class and its derived classes. This means that any function or object outside the class hierarchy cannot access the protected members of the class. The protected members of a class are typically used to represent the implementation of a class that should be accessible to its derived classes. The choice of access specifier to use for a class member depends on the intended use of the member. If a member variable or function needs to be accessed from outside the class, it should be declared as public. If a member variable or function needs to be used only within the class, it should be declared as private. If a member variable or function needs to be used within the class hierarchy, it should be declared as protected. One important aspect of access specifiers is encapsulation. Encapsulation is the process of hiding the implementation details of a class from the outside world. By using access specifiers, a class can control the level of access to its members and thus control the level of encapsulation.

C++ Code

```
#include <iostream>
using namespace std;
class MyClass {
public:
  int publicVar;      // can be accessed from anywhere
```

```cpp
    void publicFunc() { // can be accessed from anywhere
        cout << "This is a public function" << endl;
    }
private:
    int privateVar;     // can only be accessed from within the class
    void privateFunc() {// can only be accessed from within the class
        cout << "This is a private function" << endl;
    }
protected:
    int protectedVar;   // can only be accessed from within the class and its derived classes
    void protectedFunc(){// can only be accessed from within the class and its derived classes
        cout << "This is a protected function" << endl;
    }
};
class DerivedClass : public MyClass {
public:
    void derivedFunc() {
        // can access protectedVar and protectedFunc() from base class
        cout << "Derived function accessing protectedVar: " << protectedVar << endl;
        protectedFunc();
    }
};
int main() {
    MyClass obj;
    obj.publicVar = 10;
    obj.publicFunc();

    //obj.privateVar = 20; // error: private member cannot be accessed
    //obj.privateFunc();   // error: private member cannot be accessed
    //obj.protectedVar = 30; // error: protected member cannot be accessed
    //obj.protectedFunc();   // error: protected member cannot be accessed
    DerivedClass dObj;
    //dObj.privateVar = 40; // error: private member cannot be accessed
    //dObj.privateFunc();   // error: private member cannot be accessed
    dObj.publicVar = 50;
    dObj.publicFunc();
    //dObj.protectedVar = 60; // error: protected member cannot be accessed
    dObj.derivedFunc();     // okay: can access protected member from derived class
    return 0;
}
```

Explanation:

In this example, we have a class called MyClass with three member variables and three member functions. The member variables and functions are declared with different access specifiers. We also have a derived class called DerivedClass that inherits from MyClass. In the main function, we create an object of MyClass called obj and demonstrate that we can access the public member variables and functions, but not the private or protected ones.

Then, we create an object of DerivedClass called dObj and demonstrate that we can access the public member variables and functions, as well as the protected ones from within the derived class. However, we cannot access the private member variables or functions from the derived class. Overall, this example demonstrates how access specifiers control the level of accessibility of class members in C++.

Overriding Member Function in C++

In Object-oriented programming, Inheritance is one of the most powerful concepts. It enables a class to inherit characteristics and behaviors from another class. Overriding is a technique used in C++ programming to modify the behavior of an inherited member function in a derived class. In this article, we will see the concept of Overriding Member functions in C++ in detail.

What is Overriding Member Function?

In C++, when a Derived class inherits a Member function from its Base class, it can redefine the behavior of that function in the Derived class. This process of redefining a Base class Member Function in a Derived class is called "**Overriding**" and the redefined function is referred to as an "**Overridden Member Function**".

In other words, when a Derived class defines a Member Function with the same name and signature as a Member Function in its Base class, the Derived class's function will Override the Base class function.

How to Override a Member Function in C++?

To Override a Member Function in C++, we need to follow the steps given below:

Step 1: Define a Base Class with a Virtual Function:

A **Virtual Function** is a type of member function declared with the keyword "**Virtual**" in the base class. This indicates that the function can be Overridden in the derived class. The syntax for declaring a **Virtual Function** is as follows:

C++ code:

```cpp
class Base {
public:
   virtual void myFunction() {
      // Base class implementation
   }
};
```

Step 2: Define a Derived Class that Overrides the Virtual Function:

To Override a **Virtual Function**, we need to define a derived class that inherits from the base class and provides a new implementation of the **Virtual Function**. The syntax for defining a derived class that overrides the **Virtual Function** is as follows:

C++ Code:

```cpp
class Derived : public Base {
public:
   void myFunction() override {
      // Derived class implementation
   }
};
```

Step 3: Create Objects of the Derived Class and Call the Virtual Function:

We can create objects of the derived class and call the **Virtual Function** using a **Pointer** to the base class. The syntax for creating objects of the derived class and calling the **Virtual Function** is as follows:

C++ Code:

```cpp
Base* basePtr = new Derived();
basePtr->myFunction();
```

When we call the **Virtual Function** using a **Pointer** to the base class, the implementation of the **Virtual Function** in the derived class will be called.

Example:

Let's look at an example that demonstrates the concept of Overriding Member Functions in C++. In the above example, we have a base class or parent class called, which has a virtual function. We also have two derived classes called Circle and Square, which override the draw() function to provide their own implementation.

C++ Code:

```cpp
#include <iostream>
class Shape {
public:
    virtual void draw() {
        std::cout << "Drawing a shape" << std::endl;
    }
};

class Circle : public Shape {
public:
    void draw() override {
        std::cout << "Drawing a circle" << std::endl;
    }
};

class Square : public Shape {
public:
    void draw() override {
        std::cout << "Drawing a square" << std::endl;
    }
};
```

```

int main() {
   Shape* shapePtr = new Circle();
   shapePtr->draw();

   shapePtr = new Square();
   shapePtr->draw();

   return 0;
}
```

Output:

```
Drawing a circle
Drawing a square
```

In this example, we create objects of the ***Circle*** and ***Square*** classes and call the **draw**() function using a pointer to the base class **Shape**. Since the **draw**() function is Virtual, the implementation in the derived classes is called, and we get the output "Drawing a circle" and "Drawing a square".

Advantages of Overriding Member Function:

Polymorphism:

One of the major advantages of overriding is that it enables Polymorphism. Polymorphism allows a derived class to have multiple behaviors depending on the context in which it is used.

Code Reusability:

Overriding allows the reuse of code from the base class while allowing the derived class to modify the behavior of the base class's member functions.

Modularity:

Overriding makes it easier to maintain code because changes made in the derived class do not affect the base class.

Rules for Overriding Member Functions:

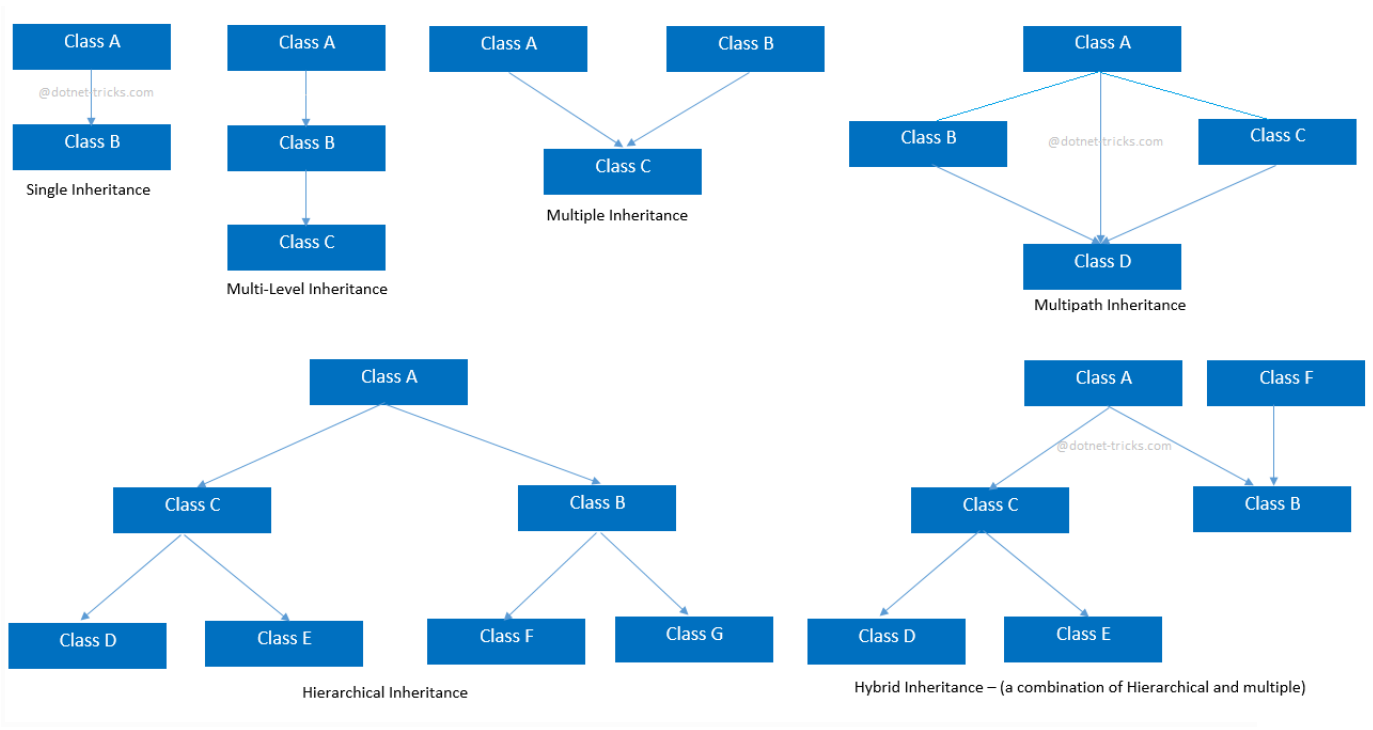
- The Overridden Function in the derived class must have the same name and arguments as the base class function.
- The Overridden Function must have the same return type as the base class function or a covariant return type.
- The access level of the Overridden Function in the derived class cannot be more restrictive than the access level of the base class function.
- The **Virtual** keyword must be used in the base class function declaration.
- The function in the base class must be declared with the same access level or higher than the derived class.

Inheritance is one of the core aspects of the fundamental called Object-Oriented Programming (OOPs) and if we need to describe then inheritance is that it provides the way of achieving code re-usability were writing the same code the multiple times, again and again, rather we can use inherit a version of the given properties of one class into the other by extending it. We will learn each type of inheritance in this article

Different Types of Inheritance

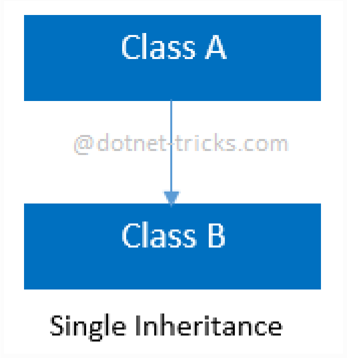
OOPs support the six different types of inheritance as given below :

1. Single inheritance
2. Multi-level inheritance
3. Multiple inheritance
4. Multipath inheritance
5. Hierarchical Inheritance
6. Hybrid Inheritance



1. Single inheritance

In this inheritance, a derived class is created from a single base class.
In the given example, Class A is the parent class and Class B is the child class since Class B inherits the features and behavior of the parent class A.



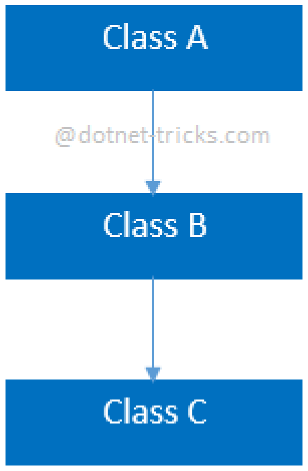
The syntax for Single Inheritance

```
//Base Class
class A
{
 public void fooA()
 {
```

```
 //TO DO:
 }
}

//Derived Class
class B : A
{
 public void fooB()
 {
 //TO DO:
 }
}
```

2. Multi-level inheritance

In this inheritance, a derived class is created from another derived class.
In the given example, class c inherits the properties and behavior of class B and class B inherits the properties and behavior of class B. So, here A is the parent class of B and class B is the parent class of C. So, here class C implicitly inherits the properties and behavior of class A along with Class B i.e there is a multilevel of inheritance.



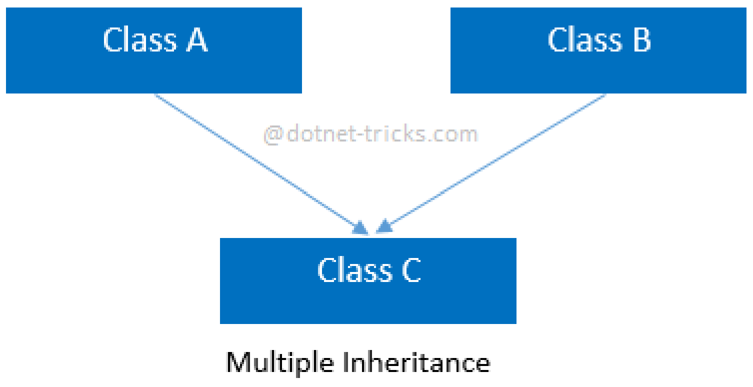
Multi-Level Inheritance

The syntax for Multi-level Inheritance

```
//Base Class
class A
{
 public void fooA()
 {
 //TO DO:
 }
}

//Derived Class
class B : A
{
 public void fooB()
 {
 //TO DO:
 }
}

//Derived Class
class C : B
{
 public void fooC()
 {
 //TO DO:
 }
}
```

3. Multiple inheritance

In this inheritance, a derived class is created from more than one base class. This inheritance is not supported by .NET Languages like C#, F#, etc., and Java Language. In the given example, class c inherits the properties and behavior of class B and class A at the same level. So, here A and Class B both are the parent classes for Class C.

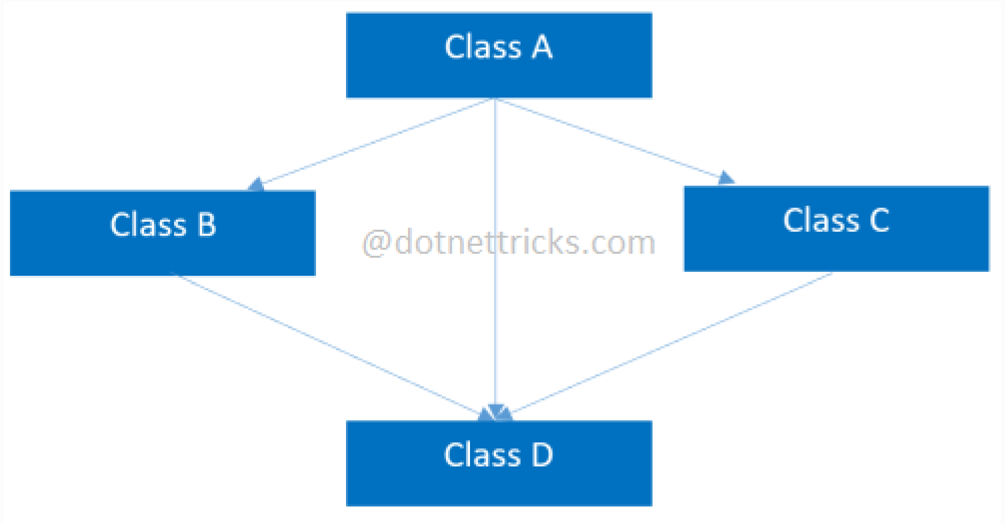


The syntax for Multiple Inheritance

```
//Base Class
class A
{
 public void fooA()
 {
 //TO DO:
 }
}

//Base Class
class B
{
 public void fooB()
 {
 //TO DO:
 }
}


//Derived Class
class C : A, B
{
 public void fooC()
 {
 //TO DO:
 }
}
```

4. Multipath inheritance

In this inheritance, a derived class is created from other derived classes and the same base class of other derived classes. This inheritance is not supported by .NET Languages like C#, F#, etc.
In the given example, class D inherits the properties and behavior of class C and class B as well as Class A. Both class C and class B inherit the Class A. So, Class A is the parent for Class B and Class C as well as Class D. So it's making it a Multipath inheritance.



Multipath Inheritance

The syntax for Multipath Inheritance

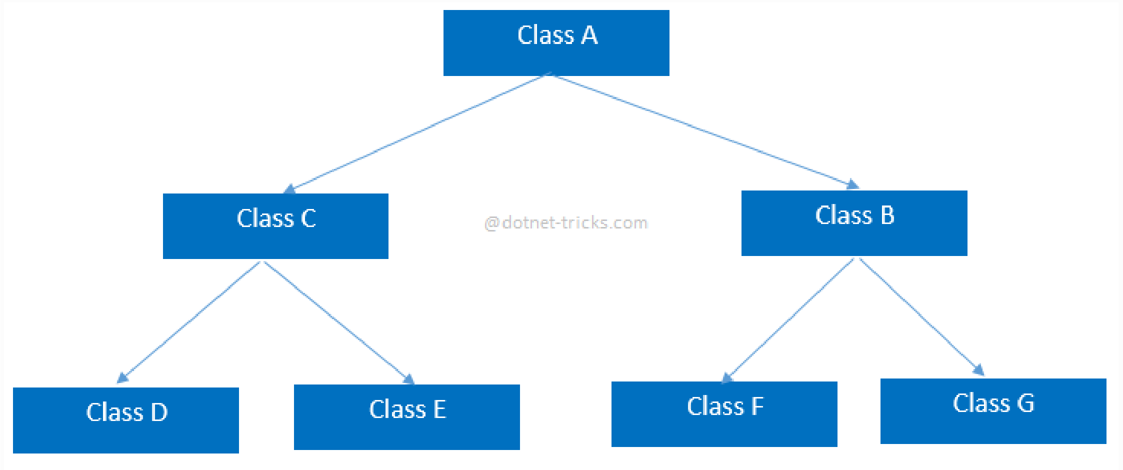
```
//Base Class
```

```
class A
{
 public void fooA()
 {
 //TO DO:
 }
}

//Derived Class
class B : A
{
 public void fooB()
 {
 //TO DO:
 }
}

//Derived Class
class C : A
{
 public void fooC()
 {
 //TO DO:
 }
}

//Derived Class
class D : B, A, C
```

```
{
 public void fooD()
 {
 //TO DO:
 }
}
```

5. Hierarchical Inheritance

In this inheritance, more than one derived class is created from a single base class and further child classes act as parent classes for more than one child class.
In the given example, class A has two children class B and class D. Further, class B and class C both are having two children - class D and E; class F and G respectively.



Hierarchical Inheritance

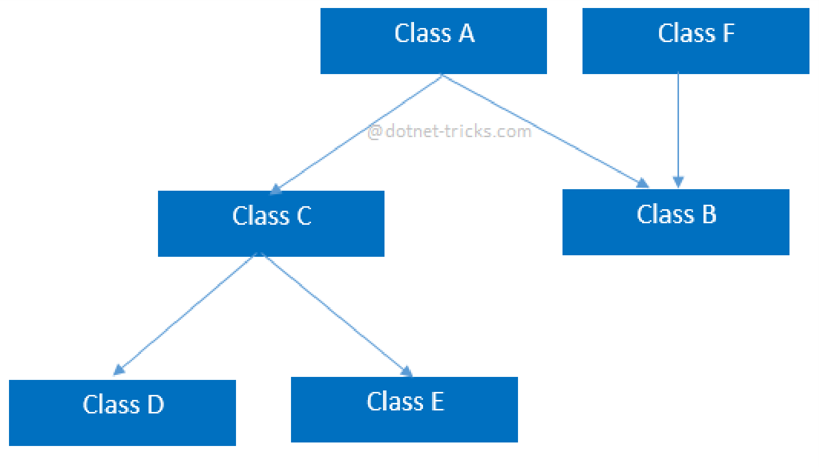
The syntax for Hierarchical Inheritance

```
//Base Class
class A
{
 public void fooA()
 {
 //TO DO:
 }
}

//Derived Class
class B : A
{
 public void fooB()
 {
 //TO DO:
 }
}

//Derived Class
class C : A
{
 public void fooC()
 {
 //TO DO:
 }
}

//Derived Class
class D : C
{
 public void fooD()
 {
 //TO DO:
 }
}
```

```
//Derived Class
class E : C
{
 public void fooE()
 {
 //TO DO:
 }
}

//Derived Class
class F : B
{
 public void fooF()
 {
 //TO DO:
 }
}

//Derived Class
class G :B
{
 public void fooG()
 {
 //TO DO:
 }
}
```

6. Hybrid inheritance

This is a combination of more than one inheritance. Hence, it may be a combination of Multilevel and Multiple inheritance or Hierarchical and Multilevel inheritance Hierarchical and Multipath inheritance, or Hierarchical, Multilevel and Multiple inheritances.
Since .NET Languages like C#, F#, etc. do not support multiple and multipath inheritance. Hence hybrid inheritance with a combination of multiple or multipath inheritances is not supported by .NET Languages.



Hybrid Inheritance – (a combination of Hierarchical and multiple)

The syntax for Hybrid Inheritance

```
//Base Class
class A
{
 public void fooA()
 {
 //TO DO:
 }
}

//Base Class
class F
{
 public void fooF()
 {
 //TO DO:
 }
}

//Derived Class
class B : A, F
{
 public void fooB()
 {
 //TO DO:
 }
}

//Derived Class
class C : A
{
 public void fooC()
 {
 //TO DO:
 }
}

//Derived Class
class D : C
{
 public void fooD()
 {
 //TO DO:
 }
}

//Derived Class
class E : C
{
 public void fooE()
 {
 //TO DO:
 }
}
```

Advantages of Inheritance

1. Reduce code redundancy.
2. Provides better code reusabilities.
3. Reduces source code size and improves code readability.
4. The code is easy to manage and divided into parent and child classes.
5. Supports code extensibility by overriding the base class functionality within child classes.
   Code usability will enhance the reliability eventually where the base class code will always be tested and debugged against the issues.

Disadvantages of Inheritance

1. In Inheritance base class and child class, both are tightly coupled. Hence If you change the code of the parent class, it will affect all the child classes.
2. In a class hierarchy, many data members remain unused and the memory allocated to them is not utilized. Hence it affects the performance of your program if you have not implemented inheritance correctly.
   Inheritance increases the coupling between the base class and the derived class. any small change in the base class will directly affect all the child classes which are extended to the parent class.

public, protected and private inheritance in C++

public, **protected,** and **private** inheritance have the following features:

- **public inheritance** makes `public` members of the base class `public` in the derived class, and the `protected` members of the base class remain `protected` in the derived class.
- **protected inheritance** makes the `public` and `protected` members of the base class `protected` in the derived class.
- **private inheritance** makes the `public` and `protected` members of the base class `private` in the derived class.

Note: `private` members of the base class are inaccessible to the derived class.

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

class PublicDerived: public Base {
  // x is public
  // y is protected
  // z is not accessible from PublicDerived
};

class ProtectedDerived: protected Base {
  // x is protected
  // y is protected
  // z is not accessible from ProtectedDerived
};

class PrivateDerived: private Base {
  // x is private
  // y is private
  // z is not accessible from PrivateDerived
};
```

Example 1: C++ public Inheritance

```
// C++ program to demonstrate the working of public inheritance

#include <iostream>
using namespace std;

class Base {
  private:
    int pvt = 1;

  protected:
    int prot = 2;

  public:
    int pub = 3;

    // function to access private member
    int getPVT() {
      return pvt;
    }
};

class PublicDerived : public Base {
  public:
    // function to access protected member from Base
    int getProt() {
      return prot;
    }
};

int main() {
  PublicDerived object1;
  cout << "Private = " << object1.getPVT() << endl;
  cout << "Protected = " << object1.getProt() << endl;
  cout << "Public = " << object1.pub << endl;
  return 0;
}
```
Run Code

Output

```
Private = 1
Protected = 2
Public = 3
```

Here, we have derived `PublicDerived` from `Base` in **public mode**.

As a result, in `PublicDerived`:

- *prot* is inherited as **protected**.
- *pub* and `getPVT()` are inherited as **public**.
- *pvt* is inaccessible since it is **private** in `Base`.

Since **private** and **protected** members are not accessible from `main()`, we need to create public functions `getPVT()` and `getProt()` to access them:

```
// Error: member "Base::pvt" is inaccessible
cout << "Private = " << object1.pvt;

// Error: member "Base::prot" is inaccessible
cout << "Protected = " << object1.prot;
```

Notice that the `getPVT()` function has been defined inside `Base`. But the `getProt()` function has been defined inside `PublicDerived`.

This is because *pvt*, which is **private** in `Base`, is inaccessible to `PublicDerived`.

However, *prot* is accessible to `PublicDerived` due to public inheritance. So, `getProt()` can access the protected variable from within `PublicDerived`.

Accessibility in public Inheritance

Accessibility	private members	protected members	public members
Base Class	Yes	Yes	Yes
Derived Class	No	Yes	Yes

Example 2: C++ protected Inheritance

```
// C++ program to demonstrate the working of protected inheritance

#include <iostream>
using namespace std;

class Base {
 private:
  int pvt = 1;

 protected:
  int prot = 2;

  public:
  int pub = 3;

  // function to access private member
  int getPVT() {
    return pvt;
  }
};

class ProtectedDerived : protected Base {
 public:
  // function to access protected member from Base
  int getProt() {
    return prot;
  }

  // function to access public member from Base
  int getPub() {
    return pub;
  }
};

int main() {
  ProtectedDerived object1;
  cout << "Private cannot be accessed." << endl;
  cout << "Protected = " << object1.getProt() << endl;
  cout << "Public = " << object1.getPub() << endl;
  return 0;
}
```

Run Code

Output

```
Private cannot be accessed.
Protected = 2
Public = 3
```

Here, we have derived `ProtectedDerived` from `Base` in **protected mode**.

As a result, in `ProtectedDerived`:

- *prot*, *pub* and `getPVT()` are inherited as **protected**.
- `pvt` is inaccessible since it is **private** in `Base`.

As we know, **protected** members cannot be directly accessed from outside the class. As a result, we cannot use `getPVT()` from `ProtectedDerived`.

That is also why we need to create the `getPub()` function in `ProtectedDerived` in order to access the *pub* variable.

```
// Error: member "Base::getPVT()" is inaccessible
cout << "Private = " << object1.getPVT();
```

```
// Error: member "Base::pub" is inaccessible
cout << "Public = " << object1.pub;
```

Accessibility in protected Inheritance

Accessibility	private members	protected members	public members
Base Class	Yes	Yes	Yes
Derived Class	No	Yes	Yes (inherited as protected variables)

Example 3: C++ private Inheritance

```
// C++ program to demonstrate the working of private inheritance

#include <iostream>
using namespace std;

class Base {
 private:
  int pvt = 1;

 protected:
  int prot = 2;

 public:
  int pub = 3;
  // function to access private member
  int getPVT() {
    return pvt;
  }
};

class PrivateDerived : private Base {
 public:
  // function to access protected member from Base
  int getProt() {
    return prot;
  }

  // function to access private member
  int getPub() {
    return pub;
  }
};

int main() {
 PrivateDerived object1;
 cout << "Private cannot be accessed." << endl;
 cout << "Protected = " << object1.getProt() << endl;
 cout << "Public = " << object1.getPub() << endl;
 return 0;
}
```
Run Code

Output

```
Private cannot be accessed.
Protected = 2
Public = 3
```

Here, we have derived `PrivateDerived` from `Base` in **private mode**.

As a result, in `PrivateDerived`:

- *prot*, `pub` and `getPVT()` are inherited as **private**.
- *pvt* is inaccessible since it is **private** in `Base`.

As we know, private members cannot be directly accessed from outside the class. As a result, we cannot use `getPVT()` from `PrivateDerived`.

That is also why we need to create the `getPub()` function in `PrivateDerived` in order to access the *pub* variable.

```
// Error: member "Base::getPVT()" is inaccessible
cout << "Private = " << object1.getPVT();

// Error: member "Base::pub" is inaccessible
cout << "Public = " << object1.pub;
```

Accessibility in private Inheritance

Accessibility	private members	protected members	public members
Base Class	Yes	Yes	Yes
Derived Class	No	Yes (inherited as private variables)	Yes (inherited as private variables)

C++ virtual function

- A C++ virtual function is a member function in the base class that you redefine in a derived class. It is declared using the virtual keyword.
- It is used to tell the compiler to perform dynamic linkage or late binding on the function.
- There is a necessity to use the single pointer to refer to all the objects of the different classes. So, we create the pointer to the base class that refers to all the derived objects. But, when base class pointer contains the address of the derived class object, always executes the base class function. This issue can only be resolved by using the 'virtual' function.
- A 'virtual' is a keyword preceding the normal declaration of a function.
- When the function is made virtual, C++ determines which function is to be invoked at the runtime based on the type of the object pointed by the base class pointer.

Rules of Virtual Function

- Virtual functions must be members of some class.
- Virtual functions cannot be static members.
- They are accessed through object pointers.
- They can be a friend of another class.
- A virtual function must be defined in the base class, even though it is not used.
- The prototypes of a virtual function of the base class and all the derived classes must be identical. If the two functions with the same name but different prototypes, C++ will consider them as the overloaded functions.
- We cannot have a virtual constructor, but we can have a virtual destructor
- Consider the situation when we don't use the virtual keyword.

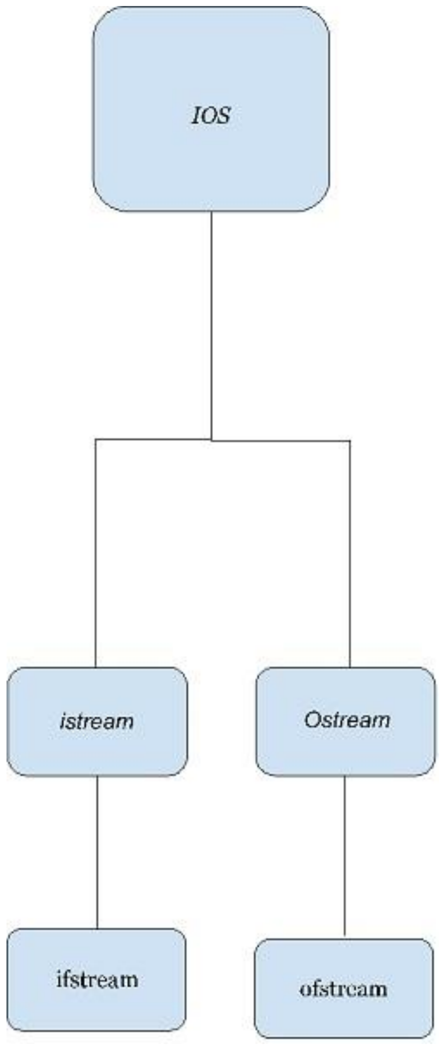
C++ Stream Classes Structure

In C++ stream refers to the stream of characters that are transferred between the program thread and i/o.

Stream classes in C++ are used to input and output operations on files and io devices. These classes have specific features and to handle input and output of the program.

The **iostream.h** library holds all the stream classes in the C++ programming language.

Let's see the hierarchy and learn about them,



Now, let's learn about the classes of the *iostream* library.

ios class − This class is the base class for all stream classes. The streams can be input or output streams. This class defines members that are independent of how the templates of the class are defined.

istream Class − The istream class handles the input stream in c++ programming language. These input stream objects are used to read and interpret the input as a sequence of characters. The cin handles the input.

ostream class − The ostream class handles the output stream in c++ programming language. These output stream objects are used to write data as a sequence of characters on the screen. cout and puts handle the out streams in c++ programming language.

Example

OUT STREAM

COUT

```
#include <iostream>
using namespace std;
int main(){
  cout<<"This output is printed on screen";
}
```

Output

This output is printed on screen

PUTS

```
#include <iostream>
using namespace std;
int main(){
  puts("This output is printed using puts");
}
```

Output

This output is printed using puts

IN STREAM

CIN

```
#include <iostream>
using namespace std;
int main(){
  int no;
  cout<<"Enter a number ";
  cin>>no;
  cout<<"Number entered using cin is "<
```

Output

```
Enter a number 3453
Number entered using cin is 3453
```

gets

```
#include <iostream>
using namespace std;
int main(){
  char ch[10];
  puts("Enter a character array");
  gets(ch);
  puts("The character array entered using gets is : ");
  puts(ch);
}
```

Output

```
Enter a character array
thdgf
The character array entered using gets is :
thdgf
```