Unit 3: Inheritance (8 hrs)

Specific Objectives:

- Understand and implement the concept of software reusability using inheritance in C++.

1. Introduction to Inheritance
2. Inheritance Relationship Diagram
3. Inheritance Mode: Public, Private & Protected
4. Types of Inheritance: Single, Multilevel, Hierarchical, Multiple and Hybrid
5. Ambiguity Resolution
6. Multipath Inheritance and Virtual Base Class
7. Constructor and Destructor in Derived Class
8. Subclass, Subtype and Principle of Substitutability
9. Composition and its Implementation
10. Composition Relationship Diagram
11. Software Reusability

1. Introduction to Inheritance

In object-oriented programming (OOP), inheritance is a fundamental concept that allows you to create new classes based on existing classes. It enables code reuse, promotes modularity, and facilitates the creation of hierarchies and relationships between classes.
Inheritance establishes an "**is-a**" relationship between classes, where one class, known as the child or derived class, inherits properties and behaviors from another class, called the parent or base class. The child class can extend and specialize the functionality of the parent class by adding new attributes and methods or by modifying the existing ones.

Some key points to remember when working with inheritance in C++:

- Inheritance allows a derived class to inherit attributes and member functions from a base class.
- A derived class is declared using the colon (:) followed by the access specifier and the name of the base class.
- There are three access specifiers in C++: **public**, **protected**, and **private**. These specifiers determine the accessibility of inherited members in the derived class.
- The public access specifier allows the inherited members to be accessed by external code and derived classes.
- The protected access specifier allows the inherited members to be accessed by the derived class but not by external code.
- The private access specifier restricts the accessibility of inherited members to the base class only.
- Constructors and destructors are not inherited, but they can be invoked by the derived class using the base class name.
- Inherited member functions can be overridden in the derived class to provide specialized behavior using the same function signature.
- To access the overridden member function in the base class from the derived class, you can use the scope resolution operator (::).
- Multiple inheritance allows a derived class to inherit from multiple base classes, separated by commas. However, it should be used with caution to avoid ambiguity or conflicts.

- The virtual keyword is used to enable polymorphic behavior when dealing with base and derived classes. It allows dynamic binding of functions at runtime.
- Use the virtual destructor in the base class to ensure proper cleanup of derived class objects when deleting through a base class pointer.

```
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
};
```

While any derived_class is inherited from a base_class, following things should be understood:

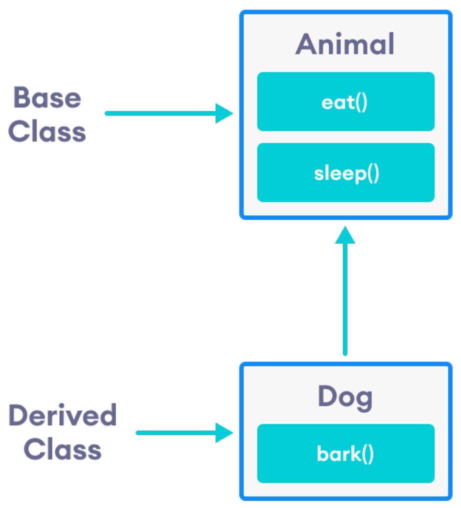
1. When a base class is privately inherited by a derived class, only the public and protected members of base class can be accessed by the member functions of derived class. This means no private member of the base class can be accessed by the objects of the derived class. Public and protected member of base class becomes private in derived class.

2. When a base class is publicly inherited by a derived class the private members are not inherited, the public and protected are inherited. The public members of base class becomes public in derived class whereas protected members of base class becomes protected in derived class.

3. When a base class is protectedly inherited by a derived class, then public members of base class becomes protected in derived class; protected members of base class becomes protected in the derived class, the private members of the base class are not inherited to derived class but note that we can access private member through inherited member function of the base class.

2. Inheritance Relationship Diagram

Inheritance is a fundamental concept in object-oriented programming (OOP) that allows classes to inherit attributes and behaviors from other classes. It establishes an "**is-a**" relationship between classes, where a subclass or derived class inherits properties from its superclass or base class.
The inheritance relationship is characterized by the following:
1. Superclass/Base class: Also known as the parent class, it is the class that provides the common attributes and behaviors to be inherited. The superclass can have its own attributes, methods, and may even be an extension of another class.
2. Subclass/Derived class: Also known as the child class, it is the class that inherits properties from its superclass. The subclass can add additional attributes and behaviors specific to itself while also having access to the attributes and behaviors inherited from the superclass.



Key points to understand about inheritance:

- Inheritance promotes code reuse and allows for the creation of specialized classes that inherit common features from a shared superclass.
- The subclass inherits all the non-private attributes and behaviors (methods) from the superclass.
- A derived class can inherit all base class methods except:
  - Constructors, destructors and copy constructors of the base class.
  - Overloaded operators of the base class.
  - The friend functions of the base class.
- The subclass can extend or modify inherited behavior by overriding methods or adding new methods specific to itself.
- Inheritance forms an "**is-a**" relationship, where a subclass is a specialized version of the superclass. For example, a `**Car**` class can be a subclass of a more general `**Vehicle**` class.
- Inheritance often follows an "**inheritance hierarchy**" or "**class hierarchy**," where multiple levels of inheritance can exist with subclasses inheriting from other subclasses.

To understand inheritance better, let's consider a practical example. Suppose we have a base *class* called "**Animal**" with *methods* like **eat()** and **sleep()**. Now, we can create a *derived class* called "**Dog**" that inherits from the "**Animal**" class. The "**Dog**" class can inherit the **name** and **age** *attributes* and the **eat()** and **sleep()** *methods* from the "**Animal**" class. Additionally, we can add new *methods* specific to dogs, such as **bark()**.

// C++ program to demonstrate inheritance

```
#include <iostream>
using namespace std;
// base class
class Animal
{
public:
    void eat()
    {
        cout << "I can eat!" << endl;
    }
    void sleep()
    {
        cout << "I can sleep!" << endl;
    }
};
// derived class
class Dog : public Animal
{
public:
    void dogDetails()
    {
        cout<<"I am a Husky. I am 2 Years old!"<<endl;
    }
    void bark()
    {
        cout << "I can bark! Woof woof!!" << endl;
    }
};
int main()
{
    Dog husky;  // Create object of the Dog class
    husky.dogDetails();  // Calling members of the derived class
    husky.eat();    // Calling members of the base class
    husky.sleep(); // Calling members of the base class
    husky.bark();  // Calling member of the derived class
    return 0;
}
```

Code Explanation:

In this C++ example, we define the base class "**Animal**" with a constructor that initializes the name and age attributes, as well as the **eat()** and **sleep()** member functions. The derived class "**Dog**" inherits publicly from the "**Animal**" class, which means it can access the protected members of the base class.

In the derived class "**Dog**", we provide its constructor that calls the base class constructor to initialize the inherited members. The "**Dog**" class also has its own method, **bark()**.

In the main function, we create an instance of the "**Dog**" class and demonstrate how it can access the inherited attributes and methods from the "Animal" class. Additionally, we invoke the **bark()** method specific to the "**Dog**" class.

Please note that in C++, the access specifiers "**protected**" and "**public**" determine the accessibility of the members in the base and derived classes. The "**protected**" members can be accessed by the derived class, while "**public**" members can be accessed by both the derived class and external code.

Example 2:

```
#include<iostream>
using namespace std;
class Area //Base class
{
protected:
    int width;
    int height;
public:
    void setWidth(int w)
    {
        width=w;
    }
```

```
  void setHeight(int h)
    {
        height=h;
    }
};
//Derived Class / Child class
class Rectangle: public Area
{
public:
    int getArea()
    {
        return width*height;
    }
};
int main()
{
    Rectangle obj;
    obj.setWidth(10);
    obj.setHeight(6);
    cout<<"The Total Area of Room = "<<obj.getArea()<<endl;
    return 0;
}
```

3. Inheritance Mode: Public, Private & Protected

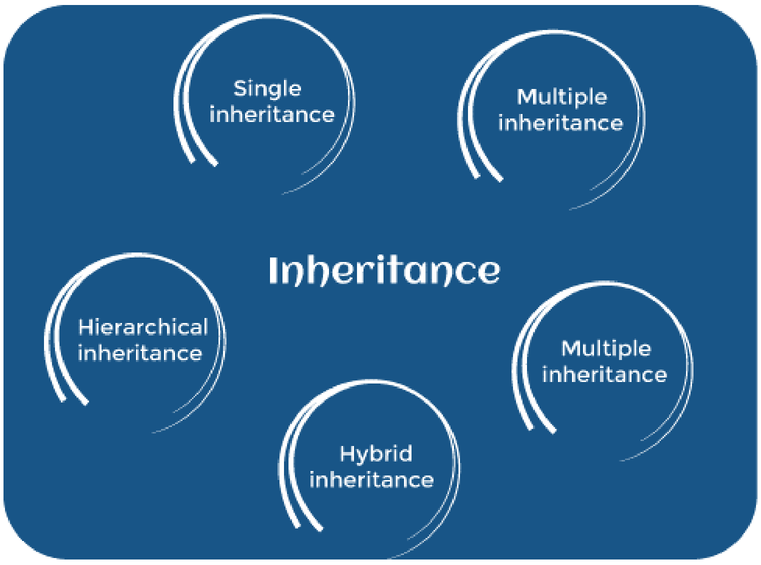
In C++, the three inheritance modes are public, private, and protected. These modes determine the accessibility of the inherited members (**methods** and **variables**) in the derived class. Here's a brief explanation of each mode:

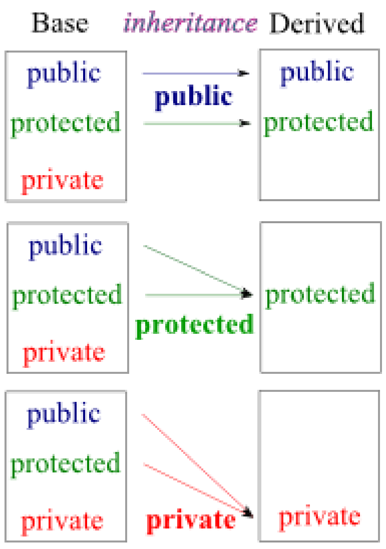
1. **Public Inheritance:**
    - In public inheritance, the public members of the base class become public members of the derived class.
    - The protected members of the base class become protected members of the derived class.
    - The private members of the base class are not accessible in the derived class.
    - This mode allows the derived class to access the public and protected members of the base class.
    - **Syntax: class Derived : public Base**

2. **Private Inheritance:**
    - In private inheritance, both the public and protected members of the base class become private members of the derived class.
    - The private members of the base class are not accessible in the derived class.
    - This mode restricts access to the members of the base class in the derived class.
    - **Syntax: class Derived : private Base**

3. **Protected Inheritance:**
    - In protected inheritance, both the public and protected members of the base class become protected members of the derived class.
    - The private members of the base class are not accessible in the derived class.
    - This mode allows the derived class to access the protected members of the base class, but restricts access to the public members.
    - **Syntax: class Derived : protected Base**

4. Types of Inheritance: Single, Multilevel, Hierarchical, Multiple and Hybrid





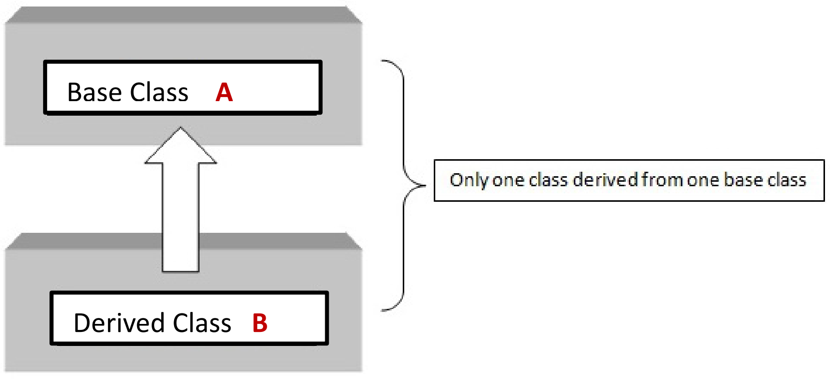
1. **Single Inheritance** – In this type of inheritance one derived class inherits from only one base class. It is the simplest form of Inheritance.
2. **Multiple Inheritance** – In this type of inheritance a single derived class may inherit from two or more than two base classes.
3. **Hierarchical Inheritance** – In this type of inheritance, multiple derived classes inherits from a single base class.
4. **Multilevel Inheritance** – In this type of inheritance the derived class inherits from a class, which in turn inherits from some other class. The Super class for one, is sub class for the other.
5. **Hybrid Inheritance** (also known as **Virtual Inheritance**) – Hybrid Inheritance is combination of Hierarchical and Multilevel Inheritance.

Note

Private members of a base class are not inherited by derived classes under inheritance. As a result, the base class's members are inaccessible to the derived class's objects. ***Only the public and protected members are inherited and accessible to the derived classes.***

1. Single Inheritance:

This is the simplest type of inheritance. In the single inheritance, one derived class can inherit property from only one base class. *For example, as explained below, the class Derived is inheriting property from only one Class Base.*

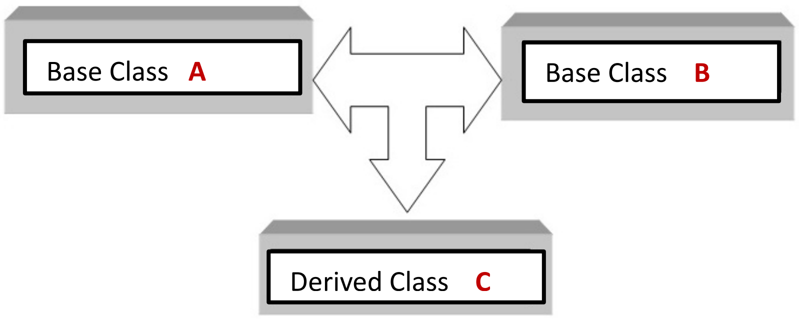


Syntax:

```
class Base
{
...
};
class Derived: access_mode Base
{
//body of Derived class which inherit property from only one base class
// access_mode can be public, private or protected
};
```

2. Multiple Inheritance:

In Multiple inheritance, a single derived class can inherit property from more than one base class. *For example, as explained below, class Derived inherits property from both Class Base1 and Class Base2.*

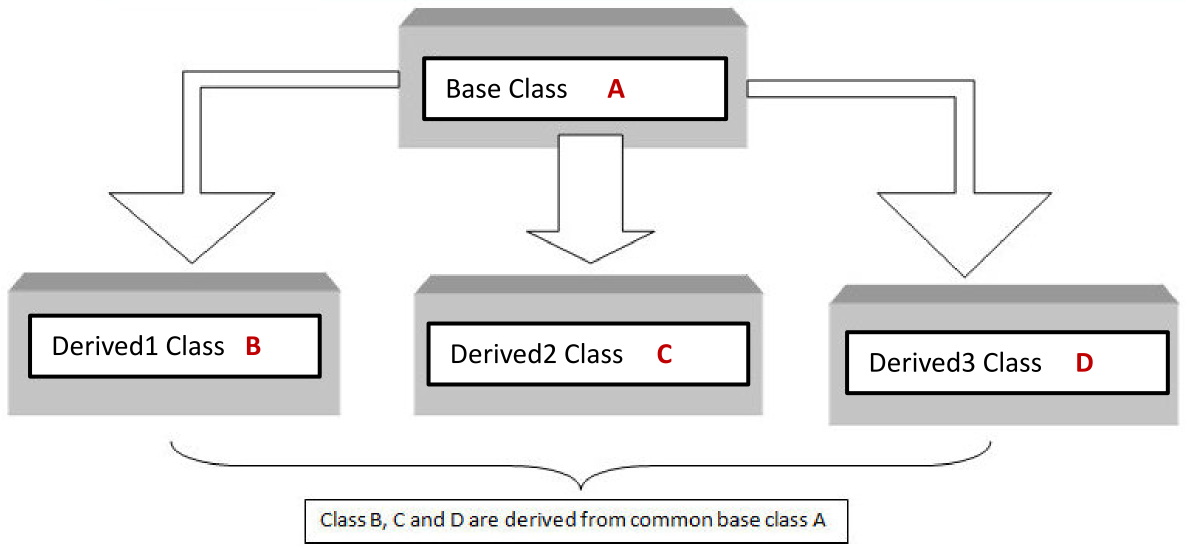


Syntax:

```
class Base1
{
...
};
class Base2
{
...
};
class Derived: access_mode Base1, access_mode Base2
{
//body of Derived class which inherit property from more than one base class
that is Base1 & Base2
};
```

3. Hierarchical Inheritance:

In hierarchical inheritance, more than one(multiple) derived classes inherit property from a single base class. *For example, as explained below, Class Derived1, Derived2 and Derived3 all three child class inherits the properties from a single class Base.*



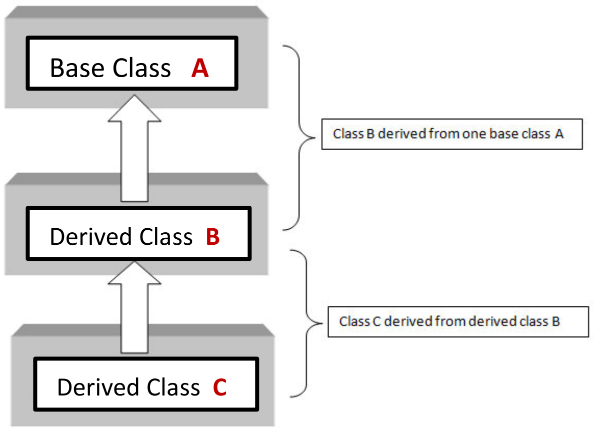
Syntax:

```
class Base
{
...
};
```

```
class Derived1: access_mode Base
{
//body of Derived1 class which inherit property from base class
};
class Derived2: access_mode Base
{
//body of Derived2 class which inherit property from Base class
};
```

4. Multilevel Inheritance:

In multilevel inheritance, the derived class inherits property from another derived class. *For example, as explained below, class Derived1 inherits property from class Base and class Derived2 inherits property from class Derived1.*

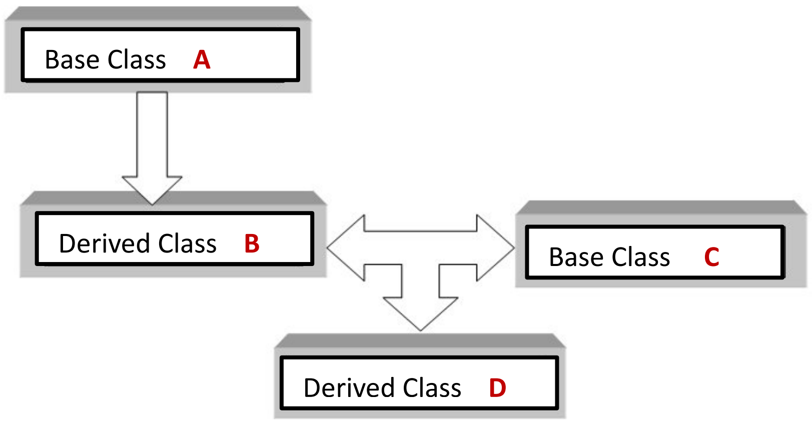

Base Class A
Derived Class B
Derived Class C
Class B derived from one base class A
Class C derived from derived class B

Syntax:

```
class Base
{
...
};
class Derived1: access_mode Base
{
//body of Derived1 class which inherit property from base class
};
class Derived2: access_mode Derived1
{
//body of Derived2 class which inherit property from Derived1 class
};
```

5. Hybrid Inheritance:

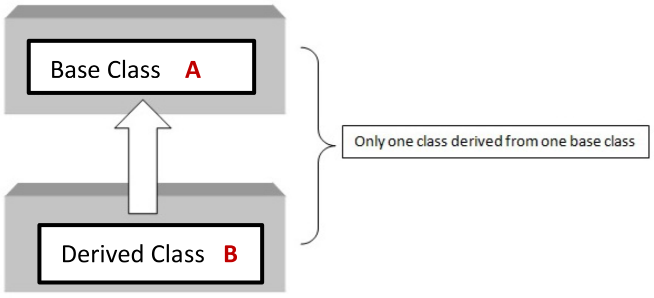
The inheritance in which the derivation of a class involves more than one form of any inheritance is called hybrid inheritance. Hybrid inheritance is a combination of both multilevel and hierarchical inheritance. It can also be called multi path inheritance.



Syntax:

```
class Base1
{
...
};
class Derived1: access_mode Base1
{
//body of Derived1 class which inherit property from the base1 class
};
class Base2
{
//body of Base2 class
};
class Derived2: access_mode Derived1, access_mode Base2
{
//body of Derived2 class which inherit property from both Derived1 and Base2
class.
};
```

Example of Single Inheritance:



```
#include <iostream>
using namespace std;
class base    //single base class
{
public:
    int x;
    void getx()
    {
        cout << "Enter the value of x = ";
        cin >> x;
    }
};
class derive : public base    //single derived class
{
private:
    int y;
public:
    void gety()
    {
        cout << "Enter the value of y = ";
        cin >> y;
    }
    void product()
    {
        cout << "Product = " << x * y;
    }
};
int main()
{
    derive obj;     //object of derived class
    obj.getx();
    obj.gety();
    obj.product();
    return 0;
}
```

Explanation of above Code:

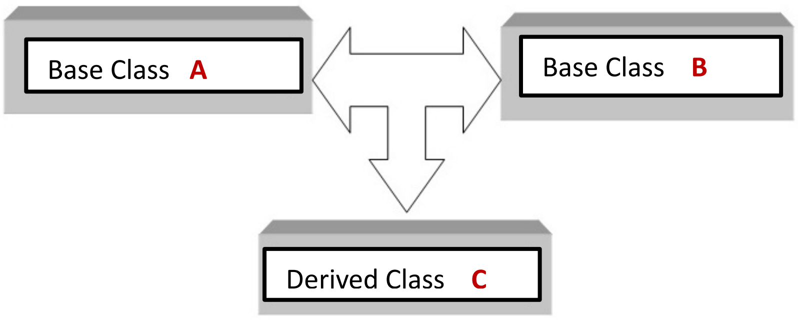
In this program class derive is publicly derived from the base class base. So, the class derive inherits all the protected and public members of base class base i.e. the protected and the public members of base class are accessible from class derive.

However private members can't be accessed, although, we haven't used any private data members in the base class.

With the object of the derived class, we can call the functions of both derived and base class.

Example of Multiple Inheritance:

```
#include<iostream>
using namespace std;
class A
{
public:
    int x;
    void getx()
    {
        cout << "enter value of x: ";
        cin >> x;
    }
};
class B
{
public:
    int y;
    void gety()
    {
        cout << "enter value of y: ";
        cin >> y;
    }
};
class C : public A, public B   //C is derived from class A and class B
{
```

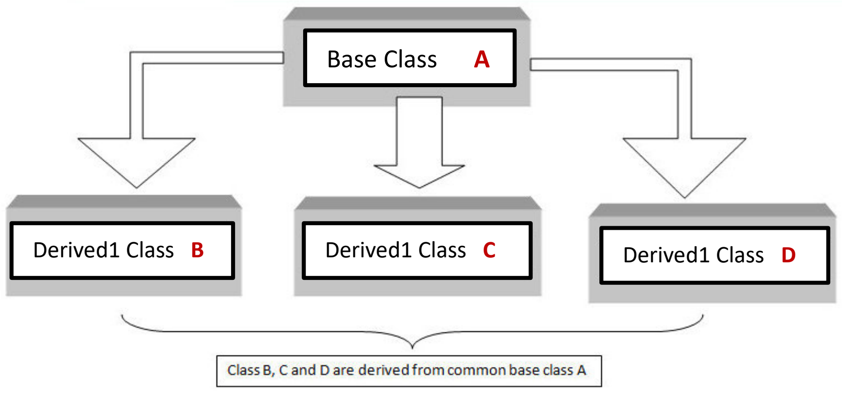


```
public:
    void sum()
    {
        cout << "Sum = " << x + y;
    }
};

int main()
{
    C obj1; //object of derived class C
    obj1.getx();
    obj1.gety();
    obj1.sum();
    return 0;
}
```

Explanation of above Code:

In the above program, there are two base class A and B from which class C is inherited. Therefore, derived class C inherits all the public members of A and B and retains their visibility. Here, we have created the object obj1 of derived class C.

Example of Hierarchical Inheritance:



```
#include <iostream>
using namespace std;
class A //single base class
{
public:
    int x, y;
    void getdata()
    {
        cout << "\nEnter value of x and y: "<<endl;
        cin >> x >> y;
    }
};
```

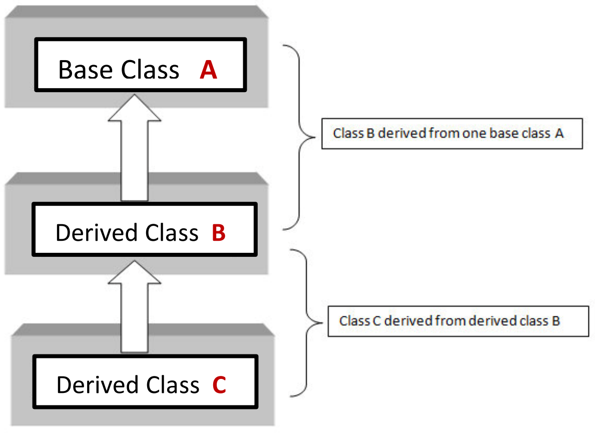
```
class B : public A        //B is derived from class base
{
public:
    void product()
    {
        cout << "\nProduct= " << x * y<<endl;
    }
};
class C : public A        //C is also derived from class base
{
public:
    void sum()
    {
        cout << "\nSum= " << x + y<<endl;
    }
};
int main()
{
    B obj1;          //object of derived class B
    C obj2;          //object of derived class C
    obj1.getdata();
    obj1.product();
    cout<<"-------------------------------"<<endl;
    obj2.getdata();
    obj2.sum();
    return 0;
}
```

Explanation of above Code:

In this example, there is only one base class A from which two class B and C are derived. Both derived class have their own members as well as base class members.

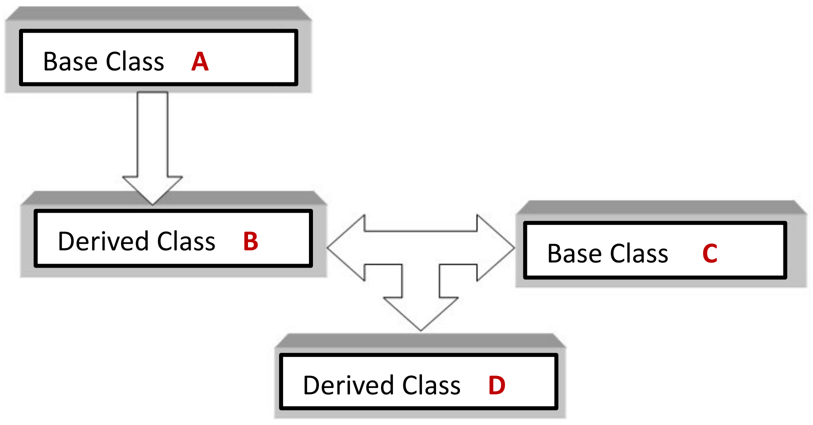
The product is calculated in the derived class B, whereas, the sum is calculated in the derived class C but both use the values of x and y from the base class.

Example of Multilevel Inheritance:


Base Class A
Derived Class B
Derived Class C
Class B derived from one base class A
Class C derived from derived class B

```
// inheritance.cpp
#include <iostream>
using namespace std;
class base //single base class
{
public:
    int x;
    void getx()
    {
        cout << "Enter value of x= "<<endl;
        cin >> x;
    }
};
class derive1 : public base // derived class from base class
{
public:
    int y;
    void gety()
    {
        cout << "Enter value of y= "<<endl;
        cin >> y;
    }
};
class derive2 : public derive1   // derived from class derive1
{
private:
    int z;
public:
    void getz()
    {
        cout << "Enter value of z= "<<endl;
        cin >> z;
    }
    void product()
    {
        cout << "Product= " << x * y * z<<endl;
    }
};
int main()
{
    derive2 obj;      //object of derived class
    obj.getx();
    obj.gety();
    obj.getz();
    obj.product();
    return 0;
}
```

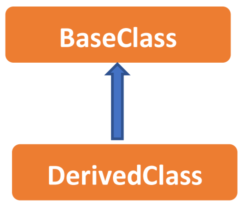
Example of Hybrid Inheritance:



```
#include <iostream>
using namespace std;
class A
{
public:
    int x;
};
class B : public A
{
public:
    B()      //constructor to initialize x in base class A
    {
        cout<<"Enter the value of x : "<<endl;
        cin>>x;
    }
};
class C
{
public:
    int y;
    C()   //constructor to initialize y
    {
        cout<<"Enter the value of y : "<<endl;
        cin>>y;
    }
};
class D : public B, public C   //D is derived from class B and class C
{
public:
    void sum()
    {
        cout << "Sum= " << x + y;
    }
};
int main()
{
    D obj;              //object of derived class D
    obj.sum();
    return 0;
}
```

Note

In inheritance diagrams, the arrow is typically shown pointing upwards from the derived class towards the base class. This convention is used to indicate the direction of inheritance, with the derived class inheriting from the base class.



In this diagram, DerivedClass is derived from BaseClass, and the arrow points upwards from DerivedClass towards BaseClass. This notation visually indicates that DerivedClass inherits from BaseClass.

5. Ambiguity Resolution

In multiple inheritances, when one class is derived from two or more base classes then there may be a possibility that the base classes have functions with the same name, and the derived class may not have functions with that name as those of its base classes. If the derived class object needs to access one of the similarly named member functions of the base classes then it results in ambiguity because the compiler gets confused about which base's class member function should be called.

In C++, ambiguity can also arise in the context of inheritance when a class inherits from multiple base classes and there are naming conflicts or ambiguous member access. This is known as **inheritance ambiguity**. To resolve it by explicitly known as **ambiguity resolution**. Here are a few examples:

Ambiguous Member Access:

```
#include <iostream>
using namespace std;
class Base1
{
public:
    void display()
    {
        cout << "Base1::display()" << endl;
    }
};

class Base2
{
public:
    void display()
    {
        cout << "Base2::display()" << endl;
    }
};
```

```
class Derived : public Base1, public Base2
{
};
```

```
//Ambiguity Resolution
obj.Base1::display();   // Calls Base1::display()
obj.Base2::display();   // Calls Base2::display()
```

```
int main()
{
    Derived obj;
    obj.display();  // Ambiguity! Both Base1::display() and Base2::display()
are accessible
    return 0;
}
```

Error: request for member 'display' is ambiguous

In this example, the Derived class inherits from both **Base1** and **Base2**, which have a member function named **display()**. When trying to access the **display()** function through an instance of **Derived**, an ambiguity occurs as both base classes provide the same function. To resolve this ambiguity, you can explicitly specify which base class's member function you want to access:

```
//Ambiguity Resolution
obj.Base1::display();   // Calls Base1::display()
obj.Base2::display();   // Calls Base2::display()
```

Naming Conflicts:

```
#include <iostream>

class Base1
{
public:
    int x;
};

class Base2
{
public:
    int x;
};

class Derived : public Base1, public Base2
{
public:
    void print()
    {
        std::cout << "Derived::x = " << x << std::endl;  // Ambiguity! Which 'x' to use?
    }
};

int main()
{
    Derived obj;
    obj.Base1::x = 5;
    obj.Base2::x = 10;
    obj.print();    // Ambiguity! Both Base1::x and Base2::x are accessible

    return 0;
}
```

```
//Ambiguity Resolution
std::cout << "Derived::x = " << Base1::x << std::endl;
// Access Base1::x
std::cout << "Derived::x = " << Base2::x << std::endl;
// Access Base2::x
```

Error: reference to 'x' is ambiguous

In this example, the **Derived** class inherits from both **Base1** and **Base2**, which have a member variable named **x**. When trying to access **x** inside the **print()** member function of **Derived**, an ambiguity occurs as there are two variables with the same name. To resolve this ambiguity, you can explicitly specify which base class's variable you want to access:

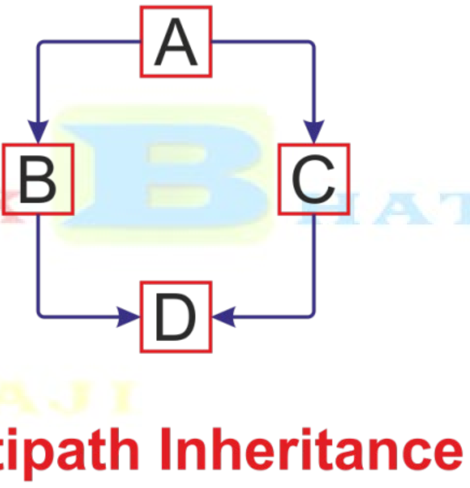
```
//Ambiguity Resolution
std::cout << "Derived::x = " << Base1::x << std::endl;  // Access Base1::x
std::cout << "Derived::x = " << Base2::x << std::endl;  // Access Base2::x
```

6. Multipath Inheritance and Virtual Base Class

Multipath inheritance refers to a situation in C++ where a derived class inherits from two or more base classes, and these base classes have a common base class as well. This can create a complex inheritance hierarchy.

It is also called as ***diamond shape inheritance*** when two classes inherit one base class and those two classes further inherited by one extreme child class then such type of inheritance is called as multipath inheritance.

Note: – This inheritance has issue with duplication of number variable inside extreme child class, as soon as the following example.



```
class A
{
public:
    int a;
};
class B : public A
{
public:
    int b;
};
class C : public A
{
public:
    int c;
};
class D : public B, public C
{
public:
    int d;
}
```

Virtual base class:

The ambiguity problem with multipath inheritance is solved by making a base class as virtual. When we make a base class as virtual then compiler will avoid the duplication of members of base class into the child class.

- To overcome the ambiguity occurred due to multipath inheritance, C++ provides the keyword virtual.
- The keyword virtual declares the specified classes virtual.
- When classes are declared as virtual, the compiler takes necessary precaution to avoid duplication of member variables.
- Thus, we make a class virtual if it is a base class that has been used by more than one derived class as their base class.

```
class A
{
public:
    int a;
};
class B : virtual public A
{
public:
    int b;
};
class C : virtual public A
{
public:
    int c;
};
class D : public B, public C
{
public:
    int d;
}
```

Ambiguity in Multipath Inheritance

```
#include<iostream>
using namespace std;
class Base
{
public:
    void print()
    {
        cout << "Base class" << endl;
    }
};
class Derived1 : public Base
{
public:
    void print()
    {
        cout << "Derived1 class" << endl;
    }
};
class Derived2 : public Base
{
public:
    void print()
    {
        cout << "Derived2 class" << endl;
    }
};
class MultiDerived : public Derived1, public Derived2{    };
int main()
{
    MultiDerived obj;
    obj.print();  // Ambiguity: Which `print` function to call?
    return 0;
}
```

```
//Ambiguity Resolution
class Derived1 : virtual public Base
```

```
//Ambiguity Resolution
class Derived2 : virtual public Base
```

```
//Ambiguity Resolution
obj.Base::print(); //Calls Base print()
obj.Derived1::print(); //Calls Derived1 print()
obj.Derived2::print(); //Calls Derived2 print()
```

Error: request for member 'print' is ambiguous

```
#include <iostream>
```

```
using namespace std;
class Base
{
public:
    int a;
};
class Derived1 : virtual public Base
{
public:
    int b;
};
```

//Ambiguity Resolution

*class Derived1 : **virtual** public Base*

```
class Derived2 : virtual public Base
{
public:
    int c;
};
class MultiDerived : public Derived1, public Derived2
{
public:
    int total;
};
int main()
{
    MultiDerived obj;
    obj.a=10; // error this is ambiguous , which a? from Derived1 or Derived2
as MultiDerived inherits from both Derived1 and Derived2
    obj.b=20;
    obj.c=30;
    obj.total=obj.a+obj.b+obj.c;
    cout<<"Total (a+b+c) : "<<obj.total<<endl;
    return 0;
}
```

//Ambiguity Resolution

*class Derived2 : **virtual** public Base*

7. Constructor and Destructor in Derived Class

In C++, a derived class inherits properties and behavior from its base class. When it comes to constructors and destructors, there are a few things to keep in mind.

1. Constructors in Derived Class:

- When you create an object of a derived class, the constructor of the derived class is called first.
- The constructor of the base class is automatically invoked before the derived class constructor.
- You can explicitly call the base class constructor within the initialization list of the derived class constructor. This ensures that the base class constructor is executed with the appropriate arguments.
- If you don't explicitly call the base class constructor, the default constructor of the base class will be called automatically.

Here's an example that demonstrates the usage of constructors in a derived class:

```
#include<iostream>
using namespace std;
class Base
{
public:
    Base(int value)
    {
        cout << "Base constructor called with value: " << value << endl;
    }
};
class Derived : public Base
{
public:
    Derived(int value) : Base(value)
    {
        cout << "Derived constructor called with value: " << value << endl;
    }
};
```

```
int main()
{
    Derived derivedObj(100);
    return 0;
}
```

---

2. Destructors in Derived Class:

- The destructor of the derived class is called before the destructor of the base class when an object is destroyed.
- You don't need to explicitly call the base class destructor since it's automatically invoked.
- You can define your own destructor in the derived class to perform additional cleanup specific to the derived class.

Here's an example that demonstrates the usage of destructors in a derived class:

```
#include<iostream>
using namespace std;
class Base
{
public:
    Base()
    {
        cout << "Base constructor called" << endl;
    }
    ~Base()
    {
        cout << "Base destructor called" << endl;
    }
};
class Derived : public Base
{
public:
    Derived()
    {
        cout << "Derived constructor called" << endl;
    }
    ~Derived()
    {
        cout << "Derived destructor called" << endl;
    }
};

int main()
{
    Derived derivedObj;
    return 0;
}
```

In summary, when working with constructors and destructors in a derived class, the base class constructor is automatically called, and the derived class constructor is responsible for calling the base class constructor if necessary. Similarly, the derived class destructor is called before the base class destructor during object destruction.

8. Subclass, Subtype and Principle of Substitutability

In C++, the terms "**subclass**," "**subtype**," and "**principle of substitutability**" are related to the concept of inheritance and polymorphism. Let's explore each of these terms:

Subclass: In C++, a subclass refers to a class that is derived from another class, known as the base class or superclass. The subclass inherits the members (variables and functions) of the base class and can also add its own members or override the inherited ones. Subclassing allows you to create a hierarchy of related classes, with each subclass inheriting and extending the behavior of its superclass.

*For example, consider a base class called "**Animal**" and a subclass called "**Dog**." The Dog class can inherit properties and methods from the Animal class, such as "**name**" and "**eat()**". It can also add its own unique members like "**bark()**" or override existing methods like "**eat()**" to provide specialized behavior for dogs.*

Subtype: Subtyping is a concept in programming languages that deals with the relationship between types. In C++, a subtype refers to a type that is derived from another type, just like a subclass is derived from a base class. The subtype inherits the characteristics of the supertype and can be used wherever the supertype is expected.
For example, if we have a function that accepts an Animal object as a parameter, we can also pass a Dog object ***(which is a subtype of Animal)*** *to that function because Dog inherits from Animal. This is possible due to the principle of substitutability.*

Principle of Substitutability: The principle of substitutability, also known as the Liskov substitution principle, is a fundamental principle of object-oriented programming. It states that objects of a derived class (**subtype**) must be able to be substituted for objects of the base class (**supertype**) without affecting the correctness of the program.

9. Composition and its Implementation

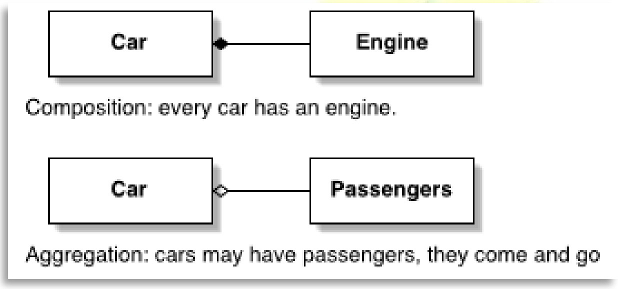
Composition is one of the fundamental approaches or concepts used in object-oriented programming (OOP) that allows you to create complex objects by combining simpler objects. It is a way to establish relationships between classes, where one class contains an instance of another class as a member variable. This allows you to build more complex and specialized objects by reusing existing classes.
In simple, Composition in C++ is defined as implementing complex objects using simpler or smaller ones. Looking around at our surroundings, we find different things built using several small components. For example, a laptop is constructed using main memory (RAM), secondary memory (Hard Drive), processor, etc. A building is created using smaller objects like bricks, sand, cement, etc. It is often good to consider complex things in terms of smaller parts and components.

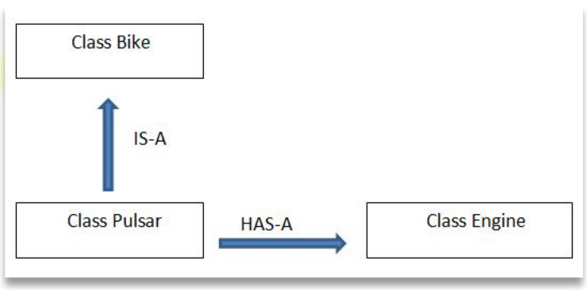
Composition in C++ is achieved by using objects and classes; therefore, it is referred to as object composition.

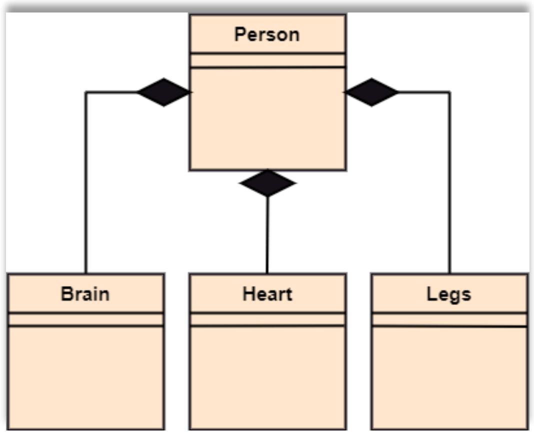
The object composition concept work on the model of **has-a** relationship among two different objects. *For example, A PC has a Core named CPU. Complex objects are often referred to as parent components or whole components, while simpler or smaller objects are often referred to as child components or part components.*

In object composition, the object created is a **part** of another object which is referred to as a **sub-object** and the **whole** object is called **super-object**. A Sub-object is destroyed when an object composition is destroyed, known as a **do and die** relationship.

- Composition is all about code reuse.
- Composition is a specialized form of aggregation.
- Creating objects of one class inside another class.
- **"Has a"** relationship
    - Bird has a beak.
    - Employee has a date of birth.
    - Employee has a date of joining.
    - Employee has an address.
- Child object does not have their lifecycle and if parent object deletes all child object will also be deleted.


Car
Engine
Composition: every car has an engine.
Car
Passengers
Aggregation: cars may have passengers, they come and go





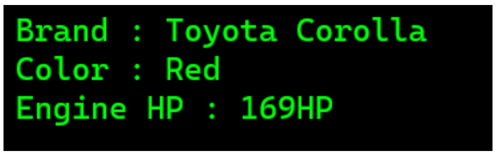
- **Person is a whole object.**
- **Brain, Heart, and Legs are part object.**
- **This whole part relationship is composition.**

- Composition is a strong relationship, because if you delete child object i.e. **Legs**, it does not affect to parent object. But if you delete the parent object i.e **Person** then whole child objects will be deleted.
- Compose object becomes a part of composer.
- Composed object cannot exist independently.

Example of Composition:

```
#include<iostream>
using namespace std;
class Engine
{
public:
    int power;
};
class Car
{
public:
    Engine e; //class Engine object as a member variable of class Car
    string brand;
    string color;
    void showDetails()
    {
        cout<<"Brand : "<<brand<<endl;
        cout<<"Color : "<<color<<endl;
        cout<<"Engine HP : "<<e.power<<"HP"<<endl;
    }
};
int main()
{
    Car c;
    c.e.power = 169;
    c.brand = "Toyota Corolla";
    c.color = "Red";
    c.showDetails();
    return 0;
}
```

Output:



```
Brand : Toyota Corolla
Color : Red
Engine HP : 169HP
```

- Class **Car** and Class **Engine** having **"has a"** relationship. →**Car has a Engine.**
- Life of Engine class depends upon the class Car, if we delete class Car then class Engine can't exist independent and hence class Engine also deleted.

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

class Engine
{
public:
    int power;
};

class Car
{
public:
    Engine e; //class Engine object as a member variable of class Car
    string brand;
    string color;

    void showDetails()
    {
        cout<<"Brand : "<<brand<<endl;
        cout<<"Color : "<<color<<endl;
        cout<<"Engine HP : "<<e.power<<"HP"<<endl;
    }
};

int main()
{
    Car *c = new Car();
    delete c; // Deallocate memory

    c->e.power = 169;
    c->brand = "Toyota Corolla";
    c->color = "Red";
    c->showDetails();

    return 0;
}
```

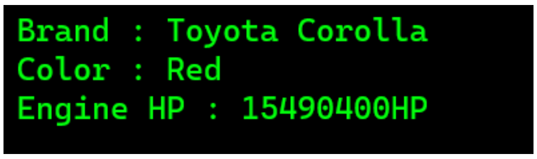
Deleting Whole object of class Car.

Output:

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

class Engine
{
public:
    int power;
};

class Car
{
public:
    Engine *e = new Engine(); //class Engine object as a member variable of
class Car
    string brand;
    string color;

    void showDetails()
    {
        delete e;
        cout<<"Brand : "<<brand<<endl;
        cout<<"Color : "<<color<<endl;
        cout<<"Engine HP : "<<e->power<<"HP"<<endl;
    }
};

int main()
{
    Car *c = new Car();
    c->e->power = 169;
    c->brand = "Toyota Corolla";
    c->color = "Red";
    c->showDetails();

    return 0;
}
```

Deleting Part object of class Engine.

Output:

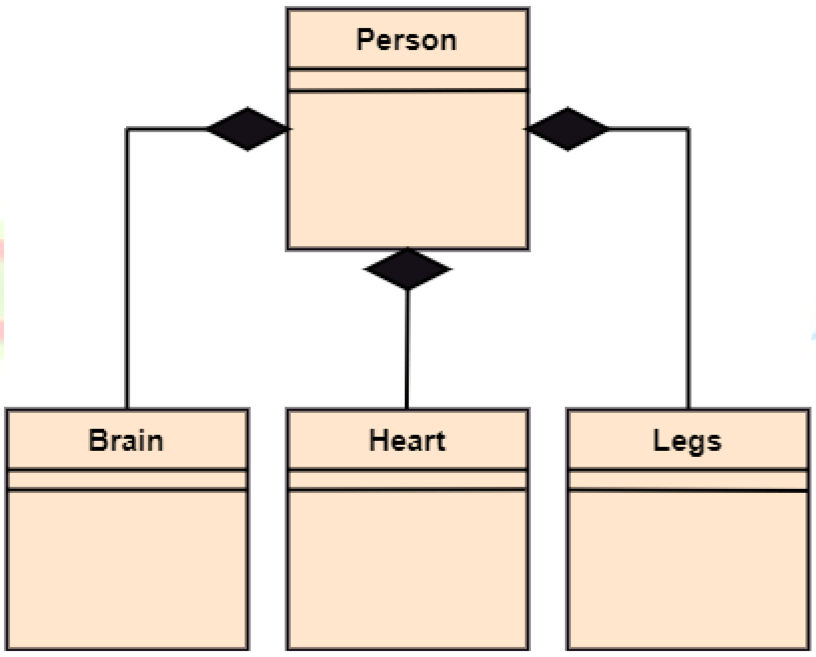


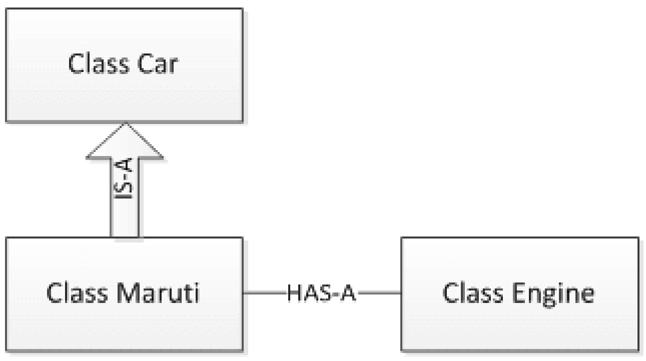
```
Brand : Toyota Corolla
Color : Red
Engine HP : 15490400HP
```

10. Composition Relationship Diagram

Composition relationship is also called a part-whole or whole-part relationship in which the part component can only be a part of a single object simultaneously. In composition relationships, the part component will be created when the object is created, and the part will be destroyed when the object is destroyed.

Composition **(HAS-A)** simply mean the use of instance variables that are references to other objects. For example, Maruti has Engine, or House has Bathroom.





11. Software Reusability

Software reusability is an important concept in software development that aims to maximize the reuse of existing code components in order to improve productivity, maintainability, and reduce development time and costs. In C++, there are several mechanisms and practices that promote software reusability.
C++ is a programming language that supports software reusability through a number of features, including:

Classes: Classes are a way of grouping related data and functions together. Once a class has been written and tested, it can be reused in other projects.

Inheritance: Inheritance allows new classes to be derived from existing classes. This means that the new class can inherit the properties and methods of the existing class. This can be a great way to reuse code and to reduce the amount of new code that needs to be written.

Templates: Templates are a way of creating generic classes and functions. This means that the same code can be used for different types of data. This can be a great way to reuse code and to make code more portable.

Note

Example: Show the simple inheritance program which shows the code reusable

THE END