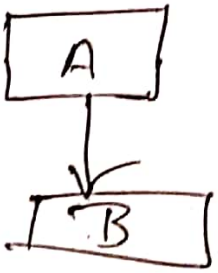
Unit-4 Inheritance

4.1 Concept of Inheritance

Inheritance allows a new class (derived class / child class) to acquire the properties and methods of an existing class (base class / parent class / Super class).

Types of Inheritance in C++:

① Single inheritance:-
A derived class inherits the properties & methods of one base class.



where 'A' is the base class
'B' is the derived class.

Syntax:

```
Class Baseclass(A)
{
  // members of the A class
};
Class DerivedClass(B): accessSpecifier
                       Baseclass
{
  // member of the derived class
};
```

A simple Example to illustrate Single inheritance:

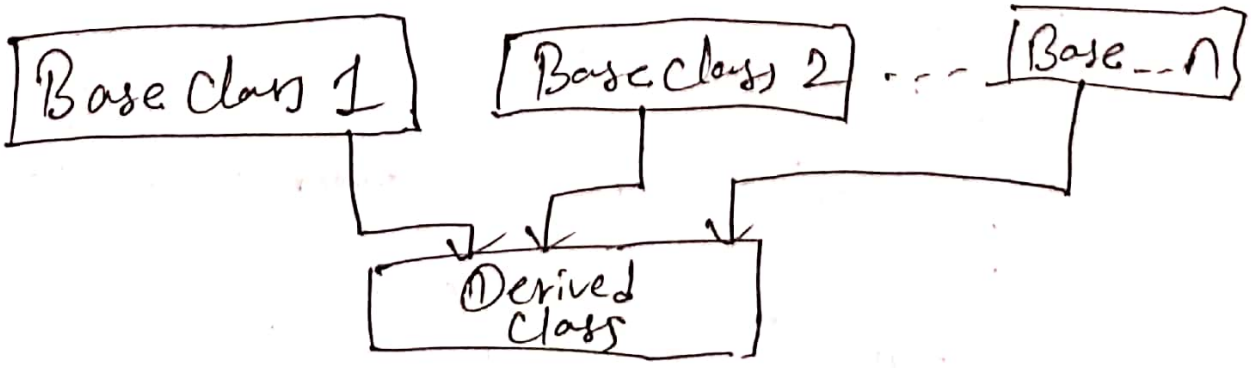
```
#include <iostream>
using namespace std;
// Base class
Class A
{
  public:
   void show()
   {
    cout << "Base class function called" <<endl;
   }
};
// Derived class
 Class B : public A {
 public:
 void display()
 {
  cout<< "Derived Class function called";
 }
};
 int main()
 {
    B obj; // creating object of derived class
    obj.show();
    obj.display();
     return 0;
 }
```

output:
Base class function called
Derived class function called

2. Multiple Inheritance: A derived class inherits from more than one base class. This allows the derived class to have access to members of multiple base classes.

Syntax:

```
Class A{
    // members of the first base class
};
Class B{
    // members of the second base class
};
Class C: accessSpecifier1 A, accessSpecifer2 B{
    // members of the derived class
};
```



Multiple inheritance is the process of deriving a new class that inherits the attributes from two or more classes.

Example of Multiple inheritance

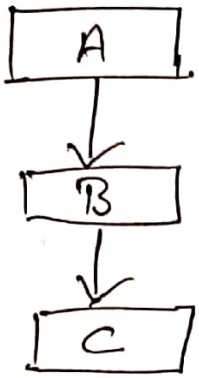
```
#include <iostream>
using namespace std;
//first Base class
class Base1{
public:
  void showBase1()
  {cout << "Base1class function called"<<endl;
};
//second Base class
class Base2{
public:
  void showBase2()
  {cout<<"Base 2 class function called";
  }
};
//Derived class inheriting from Base1 and Base2
class Derived: public Base1, public Base2{
public:
  void showDerived()
  {cout<<"Derived Class function called"<<endl;
  };
  int main()
{Derived obj;
 obj.ShowBase1();
 obj.showBase2()
 obj.showDerived();
 return 0;
}
```

output:

```
Base 1 Class function called
Base 2 Class function called
Derived Class function called
```

3. Multilevel Inheritance:-

A class is derived from another derived class.



A process of deriving a class from another derived class.

Syntax:

```
Class A{
  // members of the base class A
};
Class B : accessspecifier A {
  // members of the derived class B
};
class C : accessspecifie B{
  // members of the derived class C
};
```

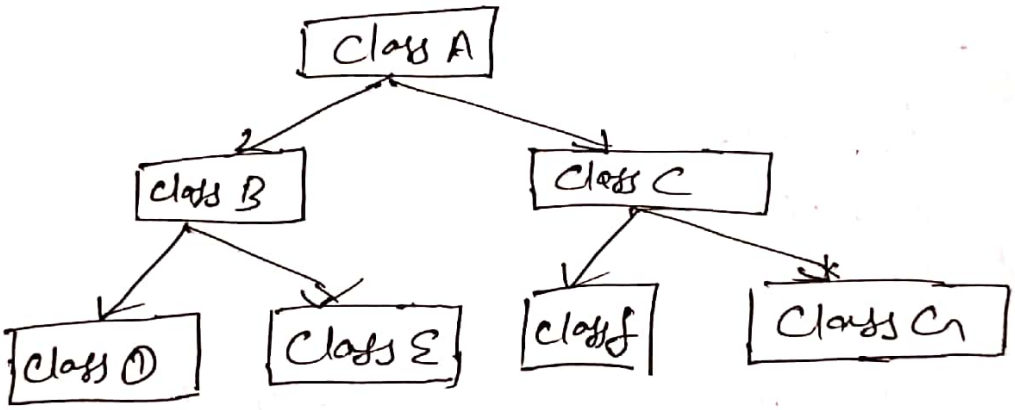
Example: C++ program to demonstrate multilevel inheritance.

```
#include <iostream>
using namespace std;
class Base{
public:
  void showBase()
  {cout << "Base class function called";
  }
};
class Derived1 : public Base
{public:
  void showDerived1()
  {cout << "Derived1 class function called";
  }
};
class Derived1.1 : public Derived1
{public:
  void showDerived1.1()
  {cout<<"Derived 1.1 class function
        Called";
  };
  int main()
  Derived1.1 obj;
  obj.showBase();
  obj.showDerived1();
  obj.showDerived1.1();
  return 0;
}
```

4) Hierarchical Inheritance

Hierarchical Inheritance is a type of inheritance where multiple derived classes inherit from a single base class. In this model, the base class acts as a common parent for all derived classes, which share the characteristic and behaviors of the base class but can also introduce their own specific attributes and methods

Structure of Hierarchical inheritance



Example:
```
#include <iostream>
using namespace std;
Class A
{
 public:
  void a()
  {
   cout << " class A function called ";
  }
};
Class B: public A
{public:
  void b()
  {
   cout<<" class B function called ";
  }};
```

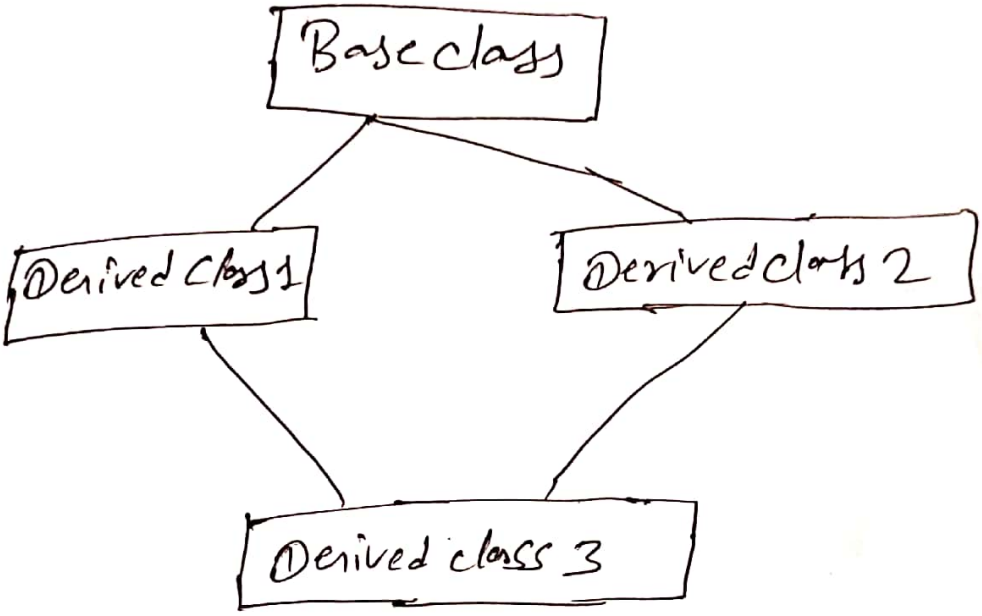
```
class C : public A{
  public:
    void c()
    {
      cout<<"class c function called";
};
  int main()
  {
    B obj1;
    obj1.b();
    obj1.a();

    C obj2;
    obj2.c();
    obj2.a();
  return 0;
}
```

output: class b function called
class a function called
class c function called
class a function called.

5) Hybrid inheritance :-

Hybrid inheritance is a combination of one or more inheritance types.
for example



In this structure :

'Base class' might be inherited by 'Derived Class1' and 'Derived class 2' → (hierarchical inheritance)

'Derived class 3' might inherit from both 'Derived -class 1' and 'Derived class 2' → (Multiple inheritance).

4.2 Base and Derived class

⇒ object oriented Programming feature inheritance allows a new class (derived class/sub class) to inherit properties and behaviours (data members and member functions) from an existing class (base class / parent class).

Base class : The Base class is also known as the parent or superclass whose features are inherited by another class.

A base class contains properties and methods that can be used by the derived class.

Example of Base class:

```
class person {
public:
    void eat()
    { cout << "eating ---" << endl;
};
```

Derived class :- The Derived class is also known as the child or subclass that inherits from the base class.

A derived class can use the members of the base class and also add its own unique members.

A derived class can inherit multiple base classes.

Example of Derived class

```
class child : public person{
public:
  void play()
  {
    cout<<" playing "<<endl;
};}
```

Syntax of Inheritance: Syntax for creating a derived class

```
class DerivedclassName : accessspecifer BaseclassName
{
   // Additional members of Derived class
};
```

Important keypoints

↳ members of the base class can be access directly in the derived class if they are inherited publicly or protected.

↳ The 'protected' members of the base class are accessible in the derived class but are not accessible outside the derived class.

Simple Example of Inheritance in C++:

```
#include <iostream>
using namespace std;
class Animal{
public:
  void eat()
  {cout<<"Animal is eating---";
};}
```

```
class Dog : public Animal{
public:
void bark(){
 cout<<"Dog is barking";
}
};
  int main(){
  Dog d;
  d.eat();
  d.bark();
return 0;}
```

4.3 private, public and protected specifier

Access specifier	Class members	Derived class	outside class
public	yes	yes	yes
protected	yes	yes	No
private	yes	No	No

In C++, Access specifiers are keywords that are used to define the access control of members (attributes and methods) within a class. Access specifiers determine how and where the members of a class can be accessed.

There are three types of Access specifiers :-

1. public
2. ~~protected~~ private
3. ~~protected~~ protected

(1) public Access specifier :- public members are accessible from anywhere in the program. members can be access from within the class, derived classes, and from outside of the class.

Example :-

```
Class Person {
public:
String name;
void displayName() {
 Cout << "Name of the person is: " << name;
}};
int main()
Person p;
P.name = "John";
P.displayName();
return 0;
}
```

2. private Access Specifier :- private members are accessible only from within the class itself. Neither derived classes nor external functions can directly access these members.

Private Access specifier is key for encapsulation ensuring that the internal workings of a class are not exposed.

Example:

```
class person {
private:
  string name;
public:
void setname(string n)
{
name=n; // private member accessed
}       within the class
void displayName()
 {cout<<"Name of the person is: "<<
        name<<endl;
};
int main() {
 person p;
 p.setName("Rajesh");
 p.displayName();
 return 0;
}
```

output: Name of the person is Rajesh

3. Protected Access Specifier :- protected members are accessible within the class itself, and by derived classes but they are not accessible from outside of the class.

Example:-

```
class person{
protected:
  string name;
public:
  void setName(string n)
  {
    name=n;
  }
};
class student : public person{
public:
  void display()
  {cout<<"Name : "<< name; //protected member accessible in child class.

  }
};
int main(){
  student s;
  s.setName("Rajesh");
  s.display();
  return 0;
}
```

output: Name : Rajesh

Summary:

public: Accessible everywhere
private: Accessible only within the class
protected: Accessible within the class and in derived classes.

4.4 Derived class Declaration

A derived class is a class that inherits properties and behaviors (data members and member functions) from base class. Derived class reuse code from the base class while adding its own unique features.

Syntax of Derived class Declaration

```
class DerivedClassName : accessSpecifier BaseClassName
{
  // Additional members and functions of the derived class
};
```

Example:

```
#include <iostream>
using namespace std;
// Base Class
class Animal {
public:
  void eat()
  {
    cout<< "Eating..." <<endl;
  }
};
// Derived class
class Dog : public Animal {
public:
  void bark()
  {
    cout << "Barking..." <<endl;
  }
};
int main(){
  Dog d;
  d.eat(); // access base class functi
  d.bark() // access derived class functi
  return 0;
}
```

4.5 Member function overriding / method overriding

The process of redefining the base class function into the derived class using same signatures (function_name, return_type and parameters).

function overriding is a ~~function~~ where a function in a derived class has the same name and parameters as a function in its base class. It allows the derived class to extend the behavior of the base class's function.

"If derived class defines same function as defined in its base class, then it is called as function overriding.

function overriding is used to achieve Runtime polymorphism [Late Binding / Dynamic function overriding]

function overriding is a feature of object oriented programming where a function in a derived class has the same name, return type, and parameters as a function in its base class but with a different implementation. The overridden function in the derived class replaces the base class function when called through a pointer or reference to the base class.

Syntax:

```
class Baseclass
{
public:
  virtual void functionName()
  {
    //Base class implementation
};
```

```
class Derivedclass : public Baseclass
{
public:
  void functionname() override
  {
    // overriding the base class function
    // Derived class implementation
  }
};

  int main()
  {
   Baseclass *bptr; // Baseclass reference pointer
   Derived class derivedobj; // creating Derived class obje
    bptr = &derivedobj; // Point baseclass pointer to derived
                        class object.
    bptr->functionName(); // calling derived class
                             functionName

    return 0;
  }
```

Program Example of method overriding

```
#include <iostream>
using namespace std;
class Base
{
public:
  virtual void display()
  {
  cout << "Baseclass method";
  }
};
class Derived : public Base{
public:
  void display() override
  {
  cout << "Derived class
       method";
  }
};
  int main()
Base* baseptr;
Derived derivedobj;
baseptr=&derivedobj;
baseptr->display();
return 0;
}
output:
  Derived class method
```

Syntax for calling overridden function

→ when you call the function using a base class pointer or reference, the derived class's overridden function is invoked.

```
int main()
{
    Baseclass *ref = new Derived class();
    ref -> functionName(); // Call Derived class function
    return 0;
}
```

Key terms to know / Key points for function overriding :-

(1). Virtual :- Virtual keyword is used in the class to allow a function to be overridden.

(2). Same signature :- The function in the derived class must have the same name, return-type, and parameters as the base class.

(3). override :- override keyword is optional but use to ensure that the function is overriding a baseclass method.

(4). Base class pointer/Reference : use baseclass reference to achieve runtime polymorphism and call the overridden function.

If you want to get output of Base Class method, you should not declared overriding function in derived class.

4.7 Ambiguity problems in inheritance

Ambiguity in inheritance occurs when a derived class inherits from two or more base classes that contain functions or members with the same name. This creates confusion for the compiler, as it can't determine which function or member to use leading to an ambiguity error.

problem

Ambiguity error: The compiler cannot decite which function to call. (Base1 function or Base2 function) Because of the same name of the function.

Example of Ambiguity in multiple inheritance

```
#include <iostream>
class A1{
public:
 void show()
  {cout<<"Base1class function";
  }
};
class A2{
public:
 void show()
  {cout<<"Base2 class function";
  }
};
class B : public A1, Public A2{
public:
};
  int main()
{
  B obj;
  obj.show(); // Ambiguity - Compiler confused
 return 0;
```

Solution to Ambiguity:

To solve this ambiguity scope resolution operator is used denoted by '::'

Syntax:-

```
object_Name.ClassName :: functionName();
```

```
#include <iostream>
class A1
{
public:
void show()
{cout<<"Base1 Class function";
}
};
class A2
{ public:
  void show()
  {cout<<"Base2 class function";
  }
};
class B: public A1, public A2
{
};
  int main()
  {
    B obj;
  //obj.show(); // ambiguity problem
    return 0;
    obj.A1::show(); // Base1 class function
    obj.A2::show(); // Base2. Class function
    return 0;
  }
```

4.8 Constructor in Derived Class

→ When an object of a derived class is created, the base class constructor is called first, followed by the derived class constructor.

→ This ensures that the base class part of the derived object is initialized before the derived class adds its own initialization.

→ Constructors are called in the order of inheritance, from base class to derived class.

→ Destructor are called in the reverse order, from derived class to base class.

Example:

```
#include <iostream>
using namespace std;
class Base
{
public:
  int a;
  Base(int x)
  {
    a=x;
    cout<<"Base class constructor called";
    cout<<"\n the value of a is: "<<a;
  };
class Derived : public Base
{
public:
  int b;
  Derived(int x, int y):Base(x)
  {
    a=x; // No need to write because inherit from Base class
    b=y;
    cout<<"\nDerived class constructor called";
```

```
    cout<<"the value of b is: "<<b;
};
int main()
{
    Derived obj(10,20);
    return 0;
}
```

Output:

```
base class constructor called
the value of a is: 10
derived class constructor called
the value of b is: 20
```

Syntax :-

```
class Base {
public:
// parameterized constructor in the base class
Base(int base_param)
{ //Initialization code for base class
};
class Derived : public Base
{
public:
// parameterized constructor in the derived class
Derived(int base_param, int derived_param): Base
  (Base_param){
    //Initialization code for derived class
};
```

Example 2

```
#include <iostream>
  using namespace std;
  class Base {
    public:
    Base()
    {cout<<" Base class constructor called ";
  };}
  class Derived : public Base{
   public:
   Derived()
   {cout<<"in derived class constructor called";
  };}
    int main()
   {
     Derived obj;
     return 0;
  }
```

Output:-

Base class constructor called
derived class constructor called.