

# CS-1004 Object Oriented programming

## Week 7

---

**Instructor:**

ATIYA

LECTURER

(COMPUTER SCIENCE DEPARTMENT)

NATIONAL UNIVERSITY- FAST (KHI CAMPUS)

EMAIL: ATIYA.JOKHIO@NU.EDU.PK

ROOM: (LECTURER ROOM#19, CS DEPT BASEMENT-2 )

# Inheritance

---

A form of software reusability where a class *inherits* an existing class' behavior and enhances it by adding more functionalities

The existing class is called base class (or sometimes super class) and the new class is referred to as derived class (or sometimes sub class)

# Inheritance : Is-a Relationship

---

Sometimes, one class is an extension of another class.

inheritance is an **is-a relationship**. We use inheritance only if an **is-a relationship** is present between the two classes.

It is just like saying that “A is type of B”.

- For example
- is “Apple is a fruit”, “Ferrari is a car”.
- A car is a vehicle.
- Orange is a fruit.
- A surgeon is a doctor.
- A dog is an animal.

# Inheritance cont.

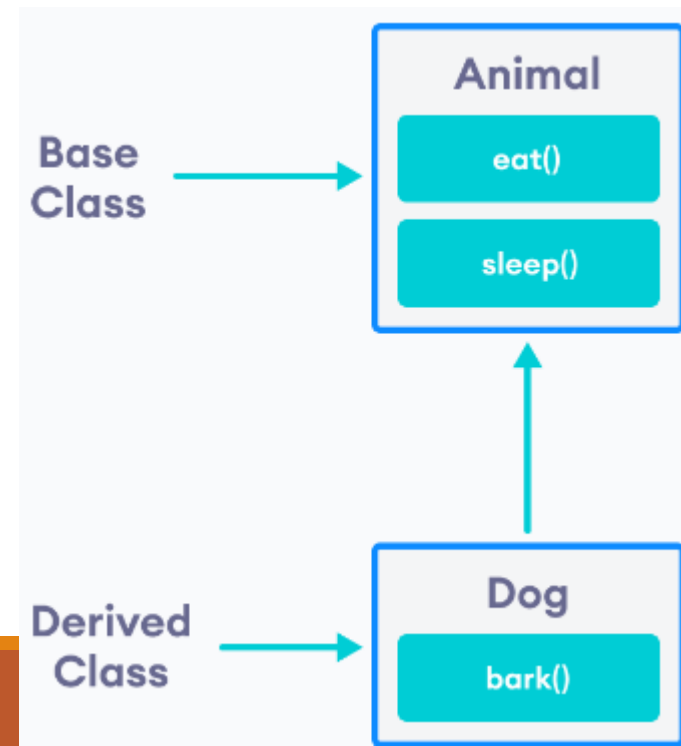
---

**Sub Class:** The class that inherits properties from another class is called Sub class or Derived Class.

**Super Class:** The class whose properties are inherited by sub class is called Base Class or Super class.

**For example,**

```
class Animal {  
    // eat() function  
    // sleep() function  
};  
  
class Dog : public Animal {  
    // bark() function  
};
```



# Implementing inheritance in C++

---

Syntax:

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

# Base & Derived Classes

---

Every derived-class object is also an object of its base class, and one base class can have many derived classes

A derived class can access all non-private members of its base class

# Why and when to use inheritance?

---

Consider a group of vehicles. You need to create classes for Bus, Car and Truck. The methods `fuelAmount()`, `capacity()`, `applyBrakes()` will be same for all of the three classes.

## **Class Bus**

`fuelAmount()`  
`capacity()`  
`applyBrakes()`

## **Class Car**

`fuelAmount()`  
`capacity()`  
`applyBrakes()`

## **Class Truck**

`fuelAmount()`  
`capacity()`  
`applyBrakes()`

# Why and when to use inheritance?

---

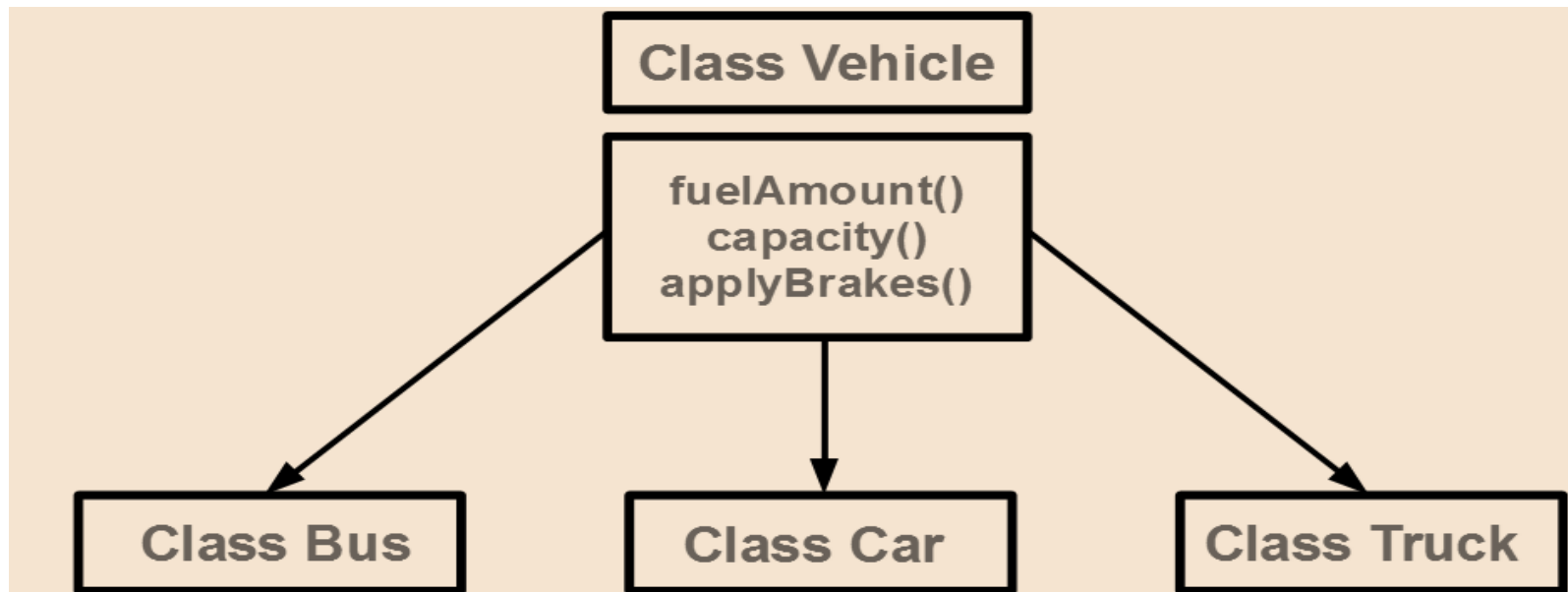
You can clearly see that above process results in duplication of same code 3 times. This increases the chances of error and data redundancy. To avoid this type of situation, inheritance is used.

If we create a class Vehicle and write these three functions in it and inherit the rest of the classes from the vehicle class, then we can simply avoid the duplication of data and increase re-usability.



# Why and when to use inheritance?

---



# Example

```
// 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 bark()
    { cout << "I can bark! Woof woof!!" << endl; }
};
```

```
int main() {
// Create object of the Dog class Dog
dog1;
// Calling members of the base class
dog1.eat();
dog1.sleep();
// Calling member of the derived class
dog1.bark();
return 0;
}
```

Output

```
I can eat!
I can sleep!
I can bark! Woof woof!!
```

# Visibility of Base Class Members

---

A derived class can use the access modifiers public, protected or private to restrict access to its base class members

In all situations, a derived class can never access private members of its base class

# Access Modes in C++ Inheritance

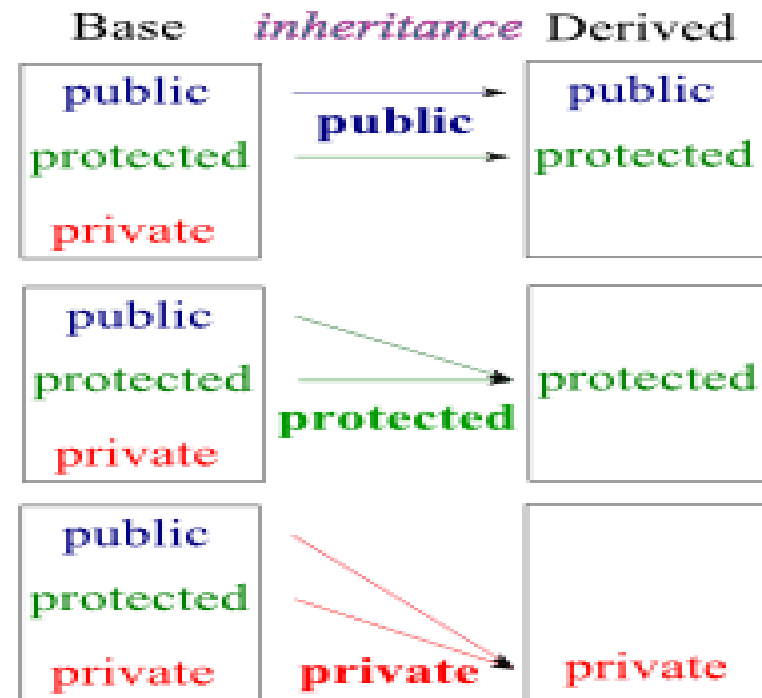
---

In C++ Inheritance, we can derive a child class from the base class in different access modes.

```
class Derived : Access Specifier Base {  
  
    ....  
  
};
```

# Access Modes in C++ Inheritance

---



# Public Inheritance

---

The use of access modifier public in derived class header

**Example:**

```
class myDerived: public myBase
{
    // derived class members
}
```

# Public Inheritance

---

In public inheritance:

- The **public** members of a base class are treated as **public** members of the derived class by other classes further down the hierarchy
- The **protected** members of a base class are treated as **protected** members of the derived class by other classes further down the hierarchy

# Public Inheritance

```
class Parent
{
    private: int a;
    public:  int b;

    protected:      int c;
}

class Child: public Parent
{
    // can never access a directly
    // can access b & c directly
}
```

```
class GrandChild: public Child
{
    // can never access a directly
    // can access b directly
    // can access c directly
}
```



# Protected Inheritance

---

The use of access modifier protected in derived class header

**Example:**

```
class myDerived: protected myBase
{
    // derived class members
}
```

# Protected Inheritance

---

In protected inheritance:

- The **public** members of a base class are treated as **protected** members of the derived class by other classes further down the hierarchy
- The **protected** members of a base class are treated as **protected** members of the derived class by other classes further down the hierarchy

# Protected Inheritance

```
class Parent
{
    private: int a;
    public:  int b;

    protected:      int c;
}

class Child: protected Parent
{
    // can never access a directly
    // can access b & c directly
}
```

```
class GrandChild: public Child
{
    // can never access a directly
    // can access b directly
    // can access c directly
}
```

# Private Inheritance

---

The use of access modifier private in derived class header

**Example:**

```
class myDerived: private myBase
{
    // derived class members
}
```

# Private Inheritance

---

In private inheritance:

- All **public** & **protected** members of a base class are treated as **private** members of the derived class by other classes further down the hierarchy
- In other words, these members can be seen as *locked* and cannot be accessed further down the hierarchy

# Private Inheritance

```
class Parent
{
    private: int a;
    public:  int b;

    protected:      int c;
}

class Child: private Parent
{
    // can never access a directly
    // can access b & c directly
}
```

```
class GrandChild: public Child
{
    // can never access a directly
    // cannot access b directly
    // cannot access c directly
}
```

# Mode of Inheritance (Summary)

---

```
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
};
```

# Types of Inheritance

---



# Types of Inheritance

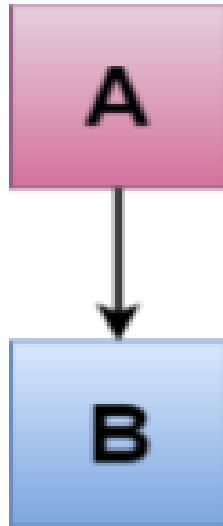
---

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

# Single Inheritance

---

Single inheritance is defined as the inheritance in which a derived class is inherited from the only one base class. i.e. one sub class is inherited by one base class only.



# Syntax:

---

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

# Single Level Inheritance Example: Inheriting Fields

---

```
#include <iostream>

using namespace std;

class Account {

    public:

    float salary = 60000;    };

    class Programmer: public Account {

    public:

    float bonus = 5000;    };

int main(void) {

    Programmer p1;

    cout<<"Salary: "<<p1.salary<<endl;

    cout<<"Bonus : "<<p1.bonus<<endl;

    return 0; }
```

# Output

---

```
Salary: 60000  
Bonus : 5000  
  
-----  
Process exited after 0.1252 se  
Press any key to continue . .
```

# OUTPUT?

---

```
#include <iostream>

using namespace std;

class Account {

    public:

    float salary = 60000;  };

    class Programmer: public Account {

    public:

    float bonus = 5000;  };

int main(void) {

    Account a1;

    cout<<"Salary: "<<a1.salary<<endl;

    cout<<"Bonus : "<<a1.bonus<<endl;

    return 0; }
```

# ERROR

---

```
#include <iostream>

using namespace std;

class Account {

    public:

    float salary = 60000;  };

    class Programmer: public Account {

    public:

    float bonus = 5000;  };

int main(void) {

    Account a1;

    cout<<"Salary: "<<a1.salary<<endl;

    cout<<"Bonus : "<<a1.bonus<<endl;  //[Error] 'class Account' has no member named 'bonus'

    return 0; }
```

# C++ Single Level Inheritance Example: Inheriting Methods

---

```
#include <iostream>

using namespace std;

class Animal {
public:
void eat() {
cout<<"Eating..."<<endl;
}
};
```



# C++ Single Level Inheritance Example: Inheriting Methods

---

```
class Dog: public Animal {  
public:  
void bark(){  
cout<<"Barking..."; } };  
int main(void) {  
Dog d1;  
d1.eat();  
d1.bark();  
return 0;  
}
```

# Output

---

```
Eating...  
Barking...  
-----  
Process exited after 0.129 seconds  
Press any key to continue . . .
```

# TASK

---

**Write a C++ program demonstrating Single Level Inheritance in a sports scenario. Create a base class named Sport that stores the name of the sport. Then, create a derived class named Athlete that inherits from Sport and includes additional attributes such as athlete name, age, and country. Implement functions to set and display details.**

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

// Base Class: Sport
class Sport {
protected:
    string sportName;

public:
    void setSport(string name) {
        sportName = name;
    }

    void displaySport() {
        cout << "Sport: " << sportName << endl;
    }
};

// Derived Class: Athlete (inherits from Sport)
class Athlete : public Sport {
private:
    string athleteName;
    int age;
    string country;

public:
    void setAthleteDetails(string name, int a, string c) {
        athleteName = name;
        age = a;
        country = c;
    }
};
```

```
void displayAthlete() {
    cout << "Athlete Name: " << athleteName << endl;
    cout << "Age: " << age << " years" << endl;
    cout << "Country: " << country << endl;
}

};

int main() {
    // Creating an object of Athlete class
    Athlete athlete1;

    // Setting values
    athlete1.setSport("100m Sprint");
    athlete1.setAthleteDetails("Bolt", 37, "Africa");

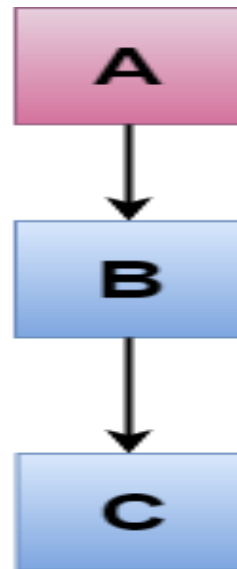
    // Displaying details
    athlete1.displaySport();
    athlete1.displayAthlete();

    return 0;
}
```

# Multilevel Inheritance

---

**Multilevel inheritance** is a process of deriving a class from another derived class.



# Multilevel Inheritance

---

When one class inherits another class which is further inherited by another class, it is known as multi level inheritance in C++. Inheritance is transitive so the last derived class acquires all the members of all its base classes.

## Multi Level Inheritance Example

---

```
#include <iostream>
using namespace std;
class Animal {
    public:
    void eat() {
        cout<<"Eating..."<<endl;    }    };
    class Dog: public Animal    {
        public:
        void bark(){
            cout<<"Barking..."<<endl;    }    };
```

# Multi Level Inheritance Example

---

```
class BabyDog: public Dog {
    public:
    void weep() {
        cout<<"Weeping...";    }    };
int main(void) {
    BabyDog d1;
    d1.eat();
    d1.bark();
    d1.weep();
    return 0; }
```



# Multi Level Inheritance TASK

---

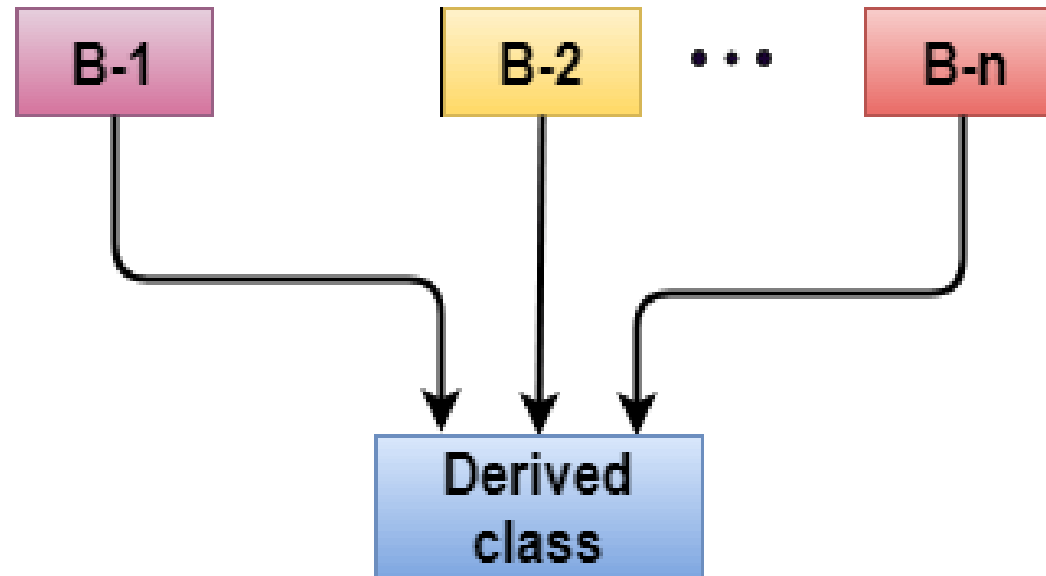
- Design a **Library Management System**. In which
- Create a **base class** Library that stores the name of the library.
- Create a **derived class** Book that inherits from Library and stores book details like title and author.
- Create another **derived class** IssuedBook that inherits from Book and adds details about the issued book, such as the borrower's name and issue date.
- Implement functions to set and display all details.

Solution is available in DEV file

# Multiple Inheritance

---

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



## Syntax of the Derived class:

---

```
class D : visibility B-1, visibility B-2
{
    // Body of the class;
}
```

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

# Multiple Inheritance Example

---

```
class A {  
    protected:  
    int a;  
    public:  
    void get_a(int n) {  
        a = n; } };  
class B {  
    protected:  
    int b;  
    public:  
    void get_b(int n) {  
        b = n; } };
```

# Multiple Inheritance Example

---

```
class C : public A,public B {
    public:
    void display() {
        std::cout << "The value of a is : " <<a<< std::endl;
        std::cout << "The value of b is : " <<b<< std::endl;
        cout<<"Addition of a and b is : "<<a+b; } };
int main() {
    C c;
    c.get_a(10);
    c.get_b(20);
    c.display();
    return 0;
}
```

# Multiple Inheritance Example

---

```
#include <iostream>
using namespace std;
class stud {
protected:
    int roll, m1, m2;
public:
    void get()
    { cout << "Enter the Roll No.: ";
      cin >> roll;
      cout << "Enter the two highest marks: ";
      cin >> m1 >> m2; } };

class extracurriculam {
protected:
    int xm;
public:
    void getsam()
    {
        cout << "\nEnter the mark for Extra Curriculam Activities: "; cin >> xm; } };

class output : public stud, public extracurriculam {
int tot, avg;
public:
    void display() {
        tot = (m1 + m2 + xm);
        avg = tot / 3;
        cout << "\n\n\tRoll No : " << roll << " \n\tTotal : " << tot;
        cout << "\n\tAverage : " << avg; } };

int main() {
    output O;
    O.get();
    O.getsam();
    O.display(); }
```

## **Ambiguity Resolution in Inheritance**

---

Ambiguity can be occurred in using the multiple inheritance when a function with the same name occurs in more than one base class.

# Ambiguity Resolution in Inheritance

---

```
#include <iostream>

using namespace std;

class A {
    public:
    void display() {
        std::cout << "Class A" << std::endl; } };

class B {
    public:
    void display() {
        std::cout << "Class B" << std::endl; } };
```



# Ambiguity Resolution in Inheritance

---

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

```
public:
```

```
    void view() {  
        display(); } };
```

```
int main() {
```

```
    C c;
```

```
    c.view();
```

```
    return 0; }
```

# OUTPUT

---

[Error] reference to 'display' is ambiguous

The above issue can be resolved by using the class resolution operator with the function. In the above example, the derived class code can be rewritten as:

---

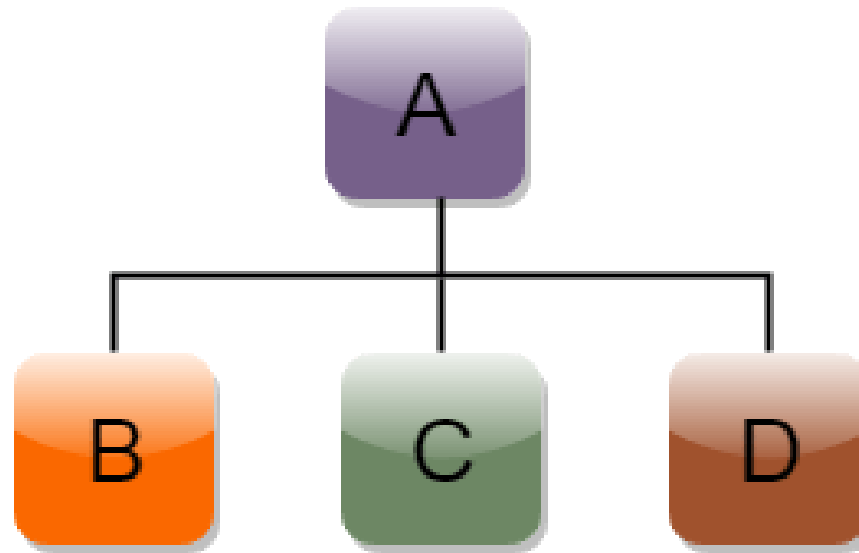
```
class C : public A, public B
{
    public: void view()
    {
        A :: display();    // Calling the display() function of class A.
        B :: display();    // Calling the display() function of class B.

    }
};
```

# Hierarchical Inheritance

---

Hierarchical inheritance is defined as the process of deriving more than one class from a base class.



# Syntax of Hierarchical inheritance:

---

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

# Example

---

```
#include <iostream>

using namespace std;

class Shape          // Declaration of base class.
{   public:
    int a;
    int b;
    void get_data(int n,int m)  {
        a= n;
        b = m;  }
};
```

# Example

---

```
class Rectangle : public Shape{ // inheriting Shape class
public:
    int rect_area() {
        int result = a*b;
        return result; } };

class Triangle : public Shape { // inheriting Shape class
public:
    int triangle_area() {
        float result = 0.5*a*b;
        return result; } };
```

# Example

---

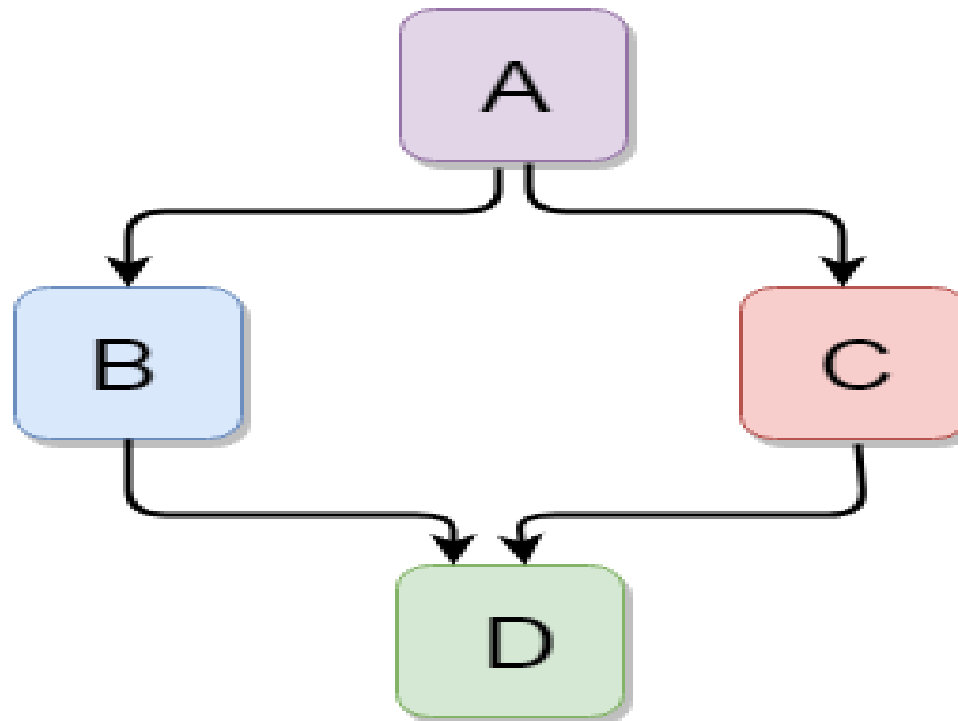
```
int main() {  
    Rectangle r;  
    Triangle t;  
    int length,breadth,base,height;  
    cin>>length>>breadth;  
    r.get_data(length,breadth);  
    int m = r.rect_area();
```

```
    std::cout << "Area of the rectangle is : " <<m<<  
    std::endl;  
    cin>>base>>height;  
    t.get_data(base,height);  
    float n = t.triangle_area();  
    std::cout <<"Area of the triangle is : " <<  
    n<<std::endl;  
    return 0; }
```



# C++ Hybrid Inheritance

Hybrid inheritance is a combination of more than one type of inheritance.



# Example

---

Consider a **transport system** where:

1. **Vehicle** is the base class.
2. **LandTransport** and **WaterTransport** inherit from **Vehicle** (Hierarchical Inheritance).
3. **AmphibiousVehicle** inherits from both **LandTransport** and **WaterTransport** (Multiple Inheritance).

# Example

---

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

// Base class
class Vehicle {
public:
    void showType() {
        cout << "This is a vehicle." << endl;
    }
};

// Derived class 1 (inherits from Vehicle)
class LandTransport : public Vehicle {
public:
    void drive() {
        cout << "Land transport: Drives on roads." << endl;
    }
};
```

# Example

---

*// Derived class 2 (inherits from Vehicle)*

```
class WaterTransport : public Vehicle {  
public:  
    void sail() {  
        cout << "Water transport: Sails in water." << endl;  
    }  
};
```

*// Derived class 3 (inherits from LandTransport & WaterTransport)*

```
class AmphibiousVehicle : public LandTransport, public WaterTransport {  
public:  
    void amphibiousFeature() {  
        cout << "Amphibious vehicle: Can move on both land and water!" << endl;  
    }  
};
```

# Example

---

```
int main() {  
    AmphibiousVehicle myAmphibian;  
  
    cout << "Amphibious Vehicle Features:\n";  
    myAmphibian.showType(); // creates ambiguity so can't use this  
    //myAmphibian.LandTransport::showType();  
    //myAmphibian.WaterTransport::showType();  
    myAmphibian.drive();      // From LandTransport  
    myAmphibian.sail();       // From WaterTransport  
    myAmphibian.amphibiousFeature(); // Specific to AmphibiousVehicle  
  
    return 0;  
}
```

---

When static members are inherited, are they static for the entire hierarchy, or just that class?

```

#include <iostream>
#include <string.h>
using namespace std;
class SomeClass{
public:
    SomeClass() {total++;}
    static int total;

    void Print(string n)
    { cout << n << ".total = " << total << endl; }

};

int SomeClass::total = 0;
class SomeDerivedClass: public SomeClass{
public:
    SomeDerivedClass() {
        total++;
    }
};

int main(){
    SomeClass A;
    SomeClass B;
    SomeDerivedClass C;
    A.Print("A");
    B.Print("B");
    C.Print("C");
    return 0;}

```

```

A.total = 4
B.total = 4
C.total = 4

-----
Process exited after 0.08119 seconds with return value 0
Press any key to continue . . .

```

// The answer is actually four in all cases, since the construction of SomeDerivedClass will cause the total to be incremented twice.

# Order of constructor and Destructor call

---

Whenever we create an object of a class, the default constructor of that class is invoked automatically to initialize the members of the class.

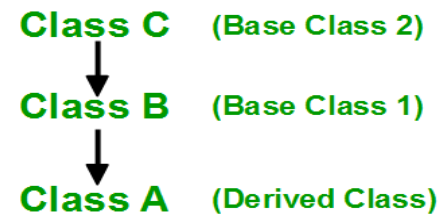
If we inherit a class from another class and create an object of the derived class, it is clear that the default constructor of the derived class will be invoked but before that the default constructor of all of the base classes will be invoke, i.e the order of invocation is that the base class's default constructor will be invoked first and then the derived class's default constructor will be invoked.



# Order of constructor and Destructor call

---

## Order of Inheritance



## Order of Constructor Call

1. **C()** (Class C's Constructor)
2. **B()** (Class B's Constructor)
3. **A()** (Class A's Constructor)

## Order of Destructor Call

1. **~A()** (Class A's Destructor)
2. **~B()** (Class B's Destructor)
3. **~C()** (Class C's Destructor)

# Example

---

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

// base class
class Parent
{
    public:

    // base class constructor
    Parent()
    {
        cout << "Inside base class" << endl;
    }
};
```

```
// sub class
class Child : public Parent
{
    public:

    //sub class constructor
    Child()
    {
        cout << "Inside sub class" << endl;
    }
};
```

```
// main function
int main() {

    // creating object of sub class
    Child obj;

    return 0;
}
```

## Output

Inside base class  
Inside sub class

# Concept: Base class Default Constructor in Derived class Constructors!

---

## Example

```
class Base
{
    int x;
    public:
    // default constructor
    Base()
    {
        cout << "Base default constructor\n";
    }
};
```

```
class Derived : public Base
{
    int y;
    public:
    // default constructor
    Derived()
    {
        cout << "Derived default
constructor\n";
    }
    // parameterized constructor
    Derived(int i)
    {
        cout << "Derived parameterized
constructor\n";
    }
};
```

```
int main()
{
    Base b;
    Derived d1;
    Derived d2(10);
}
```

## Output

```
Base default constructor
Base default constructor
Derived default constructor
Base default constructor
Derived parameterized constructor
```

**Concept:** Calling parameterized constructor of base class in derived class constructor!

---

To call the parameterized constructor of base class when derived class's parameterized constructor is called, you have to explicitly specify the base class's parameterized constructor in derived class

# Concept: Calling parameterized constructor of base class in derived class constructor!

---

## Example

```
class Base
{
    int x;
    public:
    // parameterized constructor
    Base(int i)
    {
        x = i;
        cout << "Base Parameterized
Constructor\n";
    }
};
```

```
class Derived : public Base
{
    int y;
    public:
    // parameterized constructor
    Derived(int j):Base(j)
    {
        y = j;
        cout << "Derived Parameterized
Constructor\n";
    }
};
```

```
int main()
{
    Derived d(10);
}
```

## Output

```
Base Parameterized Constructor
Derived Parameterized Constructor
```

# Important Points

---

Whenever the derived class's default constructor is called, the base class's default constructor is called automatically.

To call the parameterized constructor of base class inside the parameterized constructor of sub class, we have to mention it explicitly.

# Constructor and destructor in Hierarchical inheritance

---

```
#include <iostream>
using namespace std;
class A{
public:
    A() { cout << "A()" << endl; };
class B : public A{
public:
    B() { cout << "B()" << endl; } };
class C : public A{
public:
    C() { cout << "C()" << endl; } };
int main(){
    B b;
    C c;
    return 0;}
```

# Constructor and destructor in Hierarchical inheritance

---

```
A()
B()
A()
C()

-----
Process exited after 0.2444 seconds
Press any key to continue . . .
```



# Constructor and destructor in multiple inheritance

---

- Constructors from all base class are invoked first and the derived class constructor is called.
- Order of constructor invocation depends on the order of how the base is inherited.
- For example:
- `class C : public A , public B`

# Constructor and destructor in multiple inheritance

---

Here, A is inherited first, so the constructor of class A is called first and then constructor of class B is called next.

- However, the destructor of derived class is called first and then destructor of the base class which is mentioned in the derived class declaration is called from last towards first in sequentially.

# Syntax

---

```
Derived class constructor(arg1,arg2,arg3) : base1(arg1) , base2(arg2){  
derived= arg3;}
```

OR

```
Derived class constructor(arg1,arg2,arg3) : base1(arg1) , base2(arg2), derived(arg3){}
```

# Constructor and destructor in multiple inheritance

---

```
class A{
public: A(int a) { cout << "A()" << a << endl; };
class B{
public: B(int b ){ cout << "B()" << b << endl; } };
class C :public A , public B{
public:
    int m;
    C(int x, int y, int z): A(x) , B(y) ,m(z){
        cout<< "C()" << m << endl;};
    int main(){
        C c(2,4,6);
        return 0;}
```

# Constructor and destructor in multiple inheritance

---

```
A()2  
B()4  
C()6  
  
-----  
Process exited after 0.1935 seconds  
Press any key to continue . . .
```

# Constructor and destructor in multi level inheritance

---

```
class A{
public:
    A() { cout << "A()" << endl; };

class B : public A{
public:
    B() { cout << "B()" << endl; } };

class C : public B{
public:
    C() { cout << "C()" << endl; } };

int main(){
    C c;
    return 0;}
```

# Constructor and destructor in multi level inheritance

---

```
A()  
B()  
C()  
  
-----  
Process exited after 0.1995 s  
Press any key to continue . .
```

# You can not directly call indirect parent(A) constructor in C

---

```
class A{
public:
    A(int a) { cout << "A()" << endl; };
}

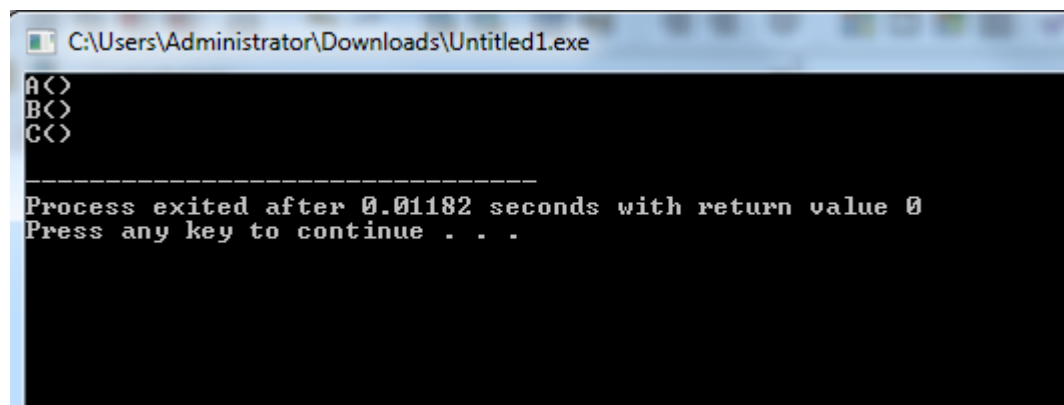
class B : public A{
public:
    B(int b) :A(b){ cout << "B()" << endl; };
}

class C : public B{
public:
    C():B(3) { cout << "C()" << endl; };
}

int main(){
    C c;
    return 0;}
```



---



```
C:\Users\Administrator\Downloads\Untitled1.exe
A<>
B<>
C<>

-----
Process exited after 0.01182 seconds with return value 0
Press any key to continue . . .
```

---

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

class A{
public:
    A(int a) { cout << "A()" << endl; };
class B : public A{
public:
    B(){ cout << "B()" << endl; } };
class C : public B{
public:
    C():A(3) { cout << "C()" << endl; } }; //error
int main(){
    C c;
    return 0;}
```

# Case study

---

Create an Investment class that contains fields to hold the initial value of an investment, the current value, the profit (calculated as the difference between current value and initial value), and the percent profit (the profit divided by the initial value). Include a constructor that requires initial and current values and a display function. Create a House class that includes fields for street address and square feet, a constructor that requires values for both fields, and a display function. Create a HouseThatIsAnInvestment class that inherits from Investment and House. It includes a constructor and a display function that calls the display functions of the parents. Write a main()function that declares a HouseThatIsAnInvestment and displays its values.