Inheritance

Inheritance

- Inheritance is the ability of one class to inherit the properties of another class.

- A new class can be created from an existing class.

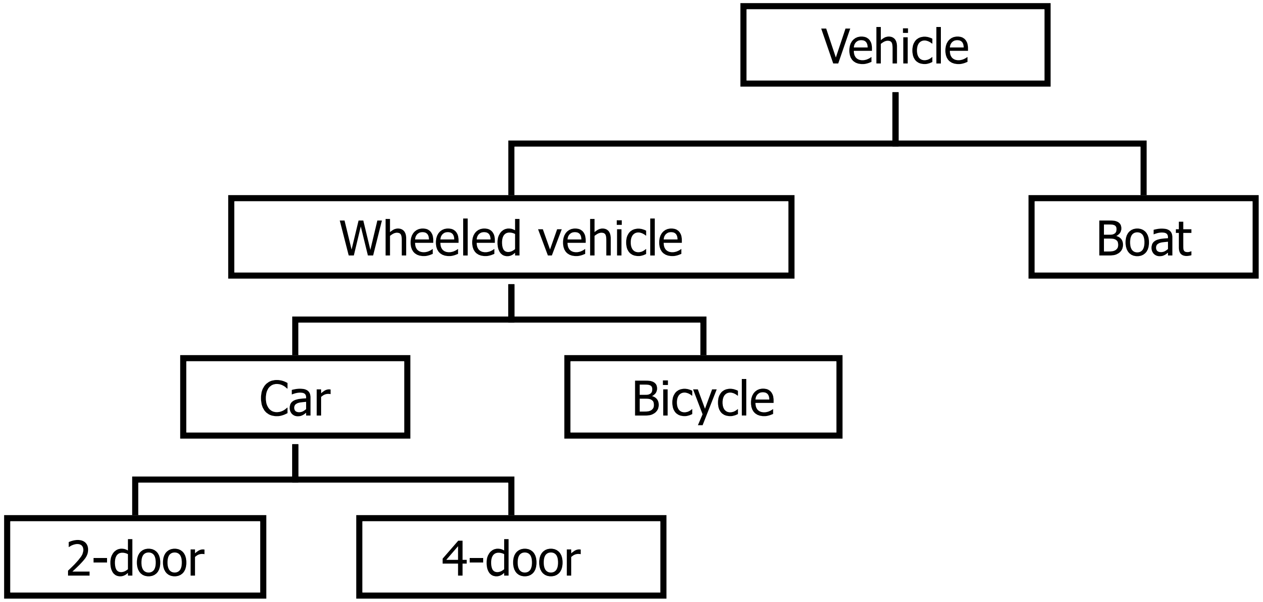
- The existing class is called the Base class or Super class

- The new class is called the Derived class or Sub-class

- Car inherits from another class auto-mobile.

Arrange concepts into an inheritance hierarchy

- Concepts at higher levels are more general
- Concepts at lower levels are more specific (inherit properties of concepts at higher levels)



Advantages of inheritance

(1) You can reuse the methods and data of the existing class

(2) You can extend the existing class by adding new data and new methods

(3) You can modify the existing class by overloading its methods with your own implementations

(4) Size of the code is reduced

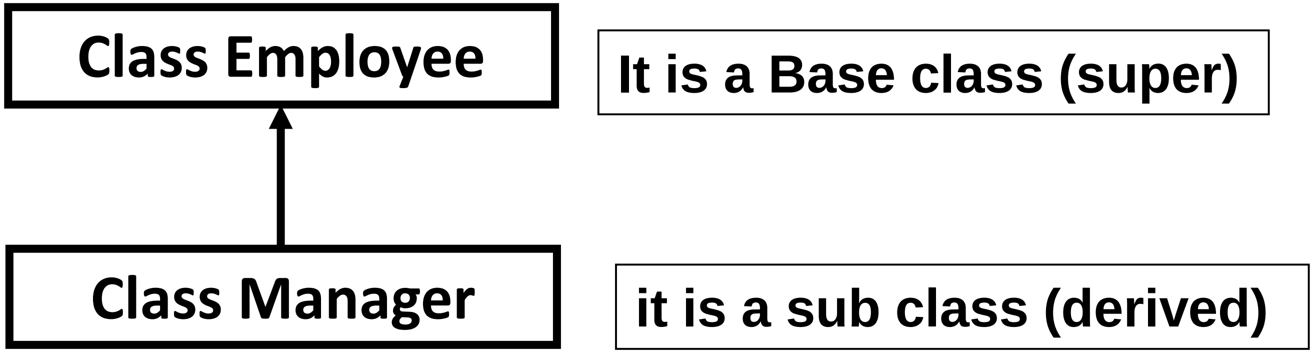
(5) Transitivity: If B is derived from A and C is derived from B then C is also derived from A.

If class B is derived from class A

- Class B is a derived class of class A
- Class B is a child of class A
- Class A is the parent of class B
- Class B inherits the member functions and variables of class A

1. Single class Inheritance:

Single inheritance is the one where you have a single base class and a single derived class.



Class Employee

It is a Base class (super)

Class Manager

it is a sub class (derived)

When a sub class inherits from one base class

General Format for implementing the concept of Inheritance:

class derived_classname: access specifier baseclassname

For example, if the *base* class is *MyClass* and the derived class is sample it is specified as:

class sample: public MyClass

The above makes sample have access to both *public* and *protected* variables of base class *MyClass*

Use of access specifier is optional

- It is private by default if the derived class is a class
- It is public by default if the derived class is a struct

We hardly use **protected** or **private** inheritance, but **public** inheritance is commonly used.

- When a class (derived) inherits from another (base) class, the visibility of the members of the base class in the derived class is as follows.

Member access specifier in base class	Member visibility in derived class		
	Type of Inheritance		
	Private	Protected	Public
Private	Not Inherited	Not Inherited	Not Inherited
Protected	Private	Protected	Protected
Public	Private	Protected	Public

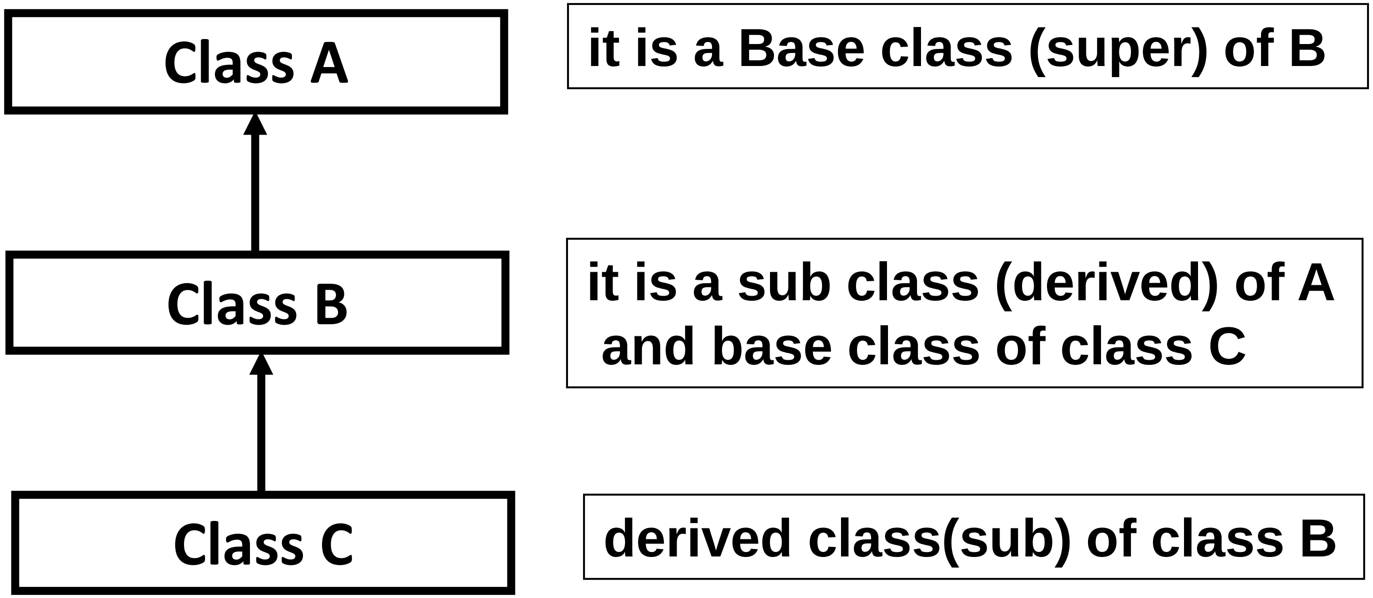
The inherited *public* members of base class
Appear as *private* members of derived class
when we are using private inheritance

```
using namespace std;
class Vehicle {
  public:
    Vehicle()
    {
      cout << "This is a Vehicle" << endl;
    }
};

// sub class derived from two base classes
class Car: public Vehicle
{

};

int main()
{
    // creating object of sub class will
     Car obj;
    return 0;
}
```

```
This is a Vehicle
```

2. Multilevel Inheritance:

In Multi level inheritance, a class inherits its properties from another derived class.



Class A

it is a Base class (super) of B

Class B

it is a sub class (derived) of A and base class of class C

Class C

derived class(sub) of class B

When a sub class inherits from a class that itself inherits from another class

```
using namespace std;
  // base class
class Vehicle
{
  public:
    Vehicle()
    {
      cout << "This is a Vehicle" << endl;
    }
};
class fourWheeler: public Vehicle
{
 public:
    fourWheeler()
    {
      cout<<"Objects with 4 wheels are 
vehicles"<<endl;
    }
};
```

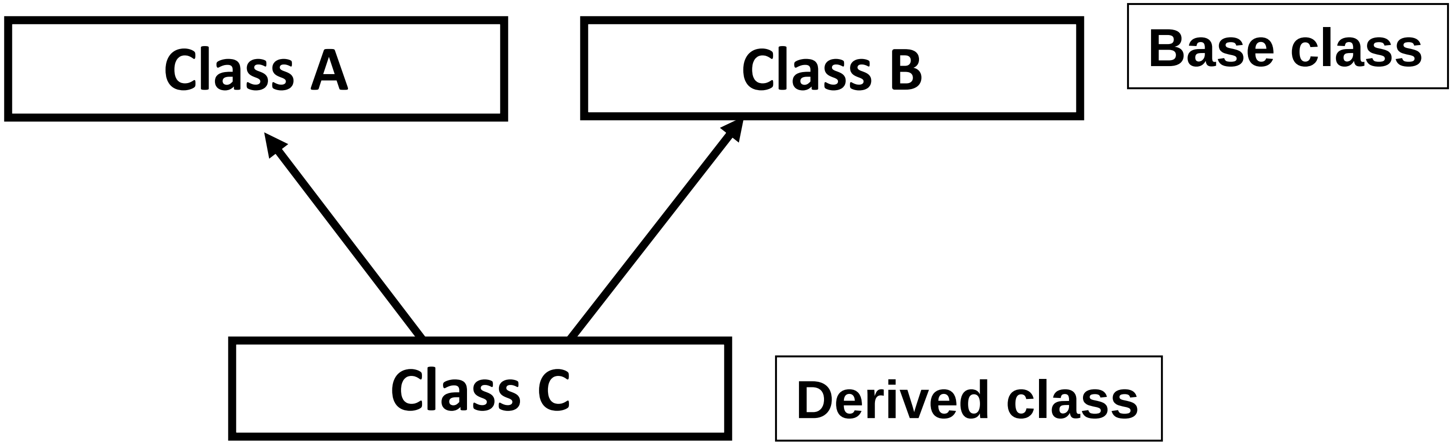
```
// sub class derived from two base classes
class Car: public fourWheeler{
   public:
     car()
     {
       cout<<"Car has 4 Wheels"<<endl;
     }
};
```

```
int main()
{
    //creating object of sub class will

    Car obj;
    return 0;
}
```

```
This is a Vehicle
Objects with 4 wheels are vehicles
Car has 4 Wheels
```

3. Multiple Inheritances:

In Multiple inheritances, a derived class inherits from multiple base classes. It has properties of both the base classes.



Class A

Class B

Base class

Class C

Derived class

```
class subclass_name : access_mode base_class1, access_mode base_class2, ....

{
          //body of subclass
 };
```

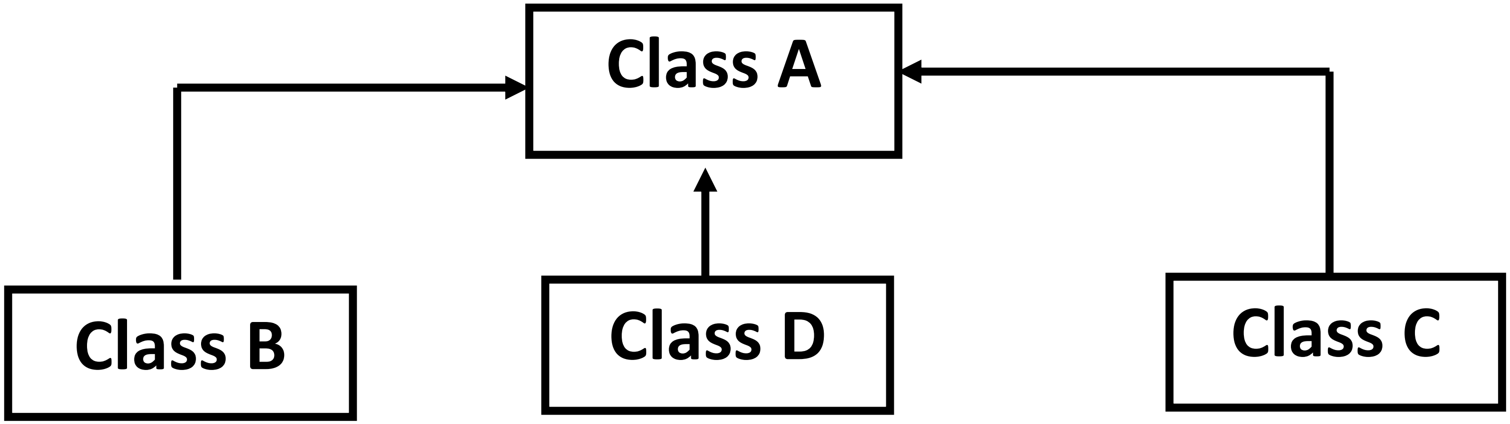
```
using namespace std;

// first base class
class Vehicle {
 public:
   Vehicle()
   {
     cout << "This is a Vehicle" << endl;
   }
};

// second base class
class FourWheeler {
 public:
   FourWheeler()
   {
     cout << "This is a 4 wheeler Vehicle" <<
endl;
   }

};
```

```
class Car: public Vehicle, public FourWheeler
 {


};


// main function
int main()
{
    // creating object of sub class will

    Car obj;
    return 0;

}
```

```
This is a Vehicle

This is a 4 wheeler Vehicle
```

4. Hierarchical Inheritance:

In hierarchical Inheritance, it's like an inverted tree. So multiple classes inherit from a single base class. It's quite analogous to the File system in a unix based system.



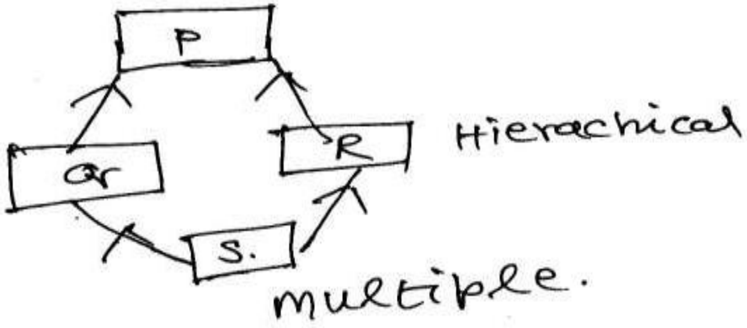
```
using namespace std;
 // base class
class Vehicle
{
 public:
   Vehicle()
   {
     cout << "This is a Vehicle" << endl;
   }
};

// first sub class
class Car: public Vehicle
{


};

// second sub class
class Bus: public Vehicle
{

};
```

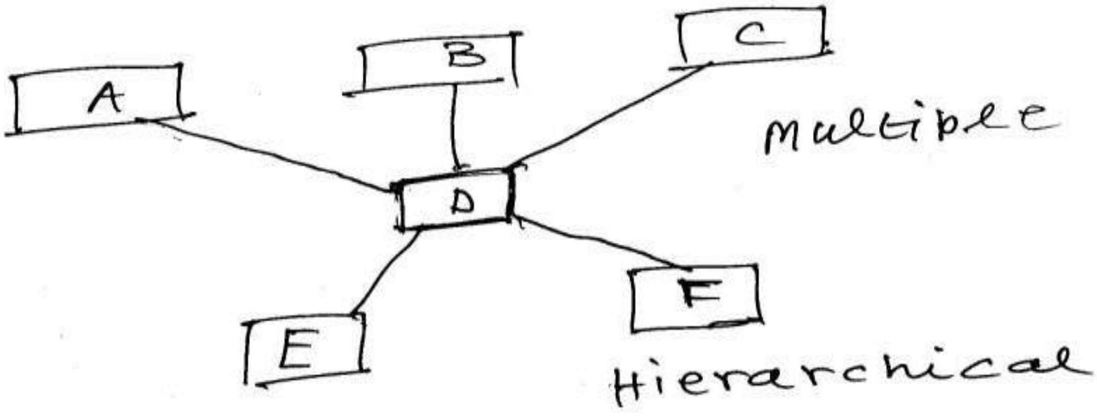
```
int main()
{
    // creating object of sub class will
    Car obj1;
    Bus obj2;
    return 0;

}
```

```
This is a Vehicle
This is a Vehicle
```

Hybrid Inheritance:

Contains two or more forms of inheritance.





```
using namespace std;
 // base class
class Vehicle
{
  public:
    Vehicle()
    {
      cout << "This is a Vehicle" << endl;
    }
};
//base class
class Fare
{
    public:
    Fare()
    {
        cout<<"Fare of Vehicle\n";
    }
};
// first sub class
class Car: public Vehicle
{

}
```

```
class Bus: public Vehicle, public Fare
{


};

// main function
int main()
{
    // creating object of sub class will
     Bus obj2;
    return 0;

}
```

```
This is a Vehicle
Fare of Vehicle
```

```
// C++ Implementation to show that a derived class
// doesn’t inherit access to private data members.
// However, it does inherit a full parent object
class A
{
public:
            int x;
protected:
            int y;
private:
            int z;
};
class B : public A
{
            // x is public
            // y is protected
            // z is not accessible from B
};
class C : protected A
{
            // x is protected
            // y is protected
            // z is not accessible from C
};
```

```
class D : private A // 'private' is default for
classes
{
            // x is private
            // y is private
            // z is not accessible from D
};
```

Constructors, Destructors, and Inheritance

- Both base class and derived class can have constructors and destructors.
- Constructor functions are executed in the order of derivation.
- Destructor functions are executed in the reverse order of derivation.
- While working with an object of a derived class, the base class constructor and destructor are always executed no matter how the inheritance was done (private, protected or public).

C++ Function Overriding

```
class Base {
   public:
    void print() {
        cout << "Base Function"
}
};

class Derived : public Base {
   public:
    void print() {
        cout << "Derived Function";
    }
};

int main() {
    Derived derived1;
    derived1.print();
    Base b;
    b.print();
    return 0;
}
```

Output:

```
Derived Function
Base Function
```

Or

Derived class Print function with the help of base class call the base class print function

```
Base :: print();
```

Output:

```
Derived Function
Base Function
```

Or

With derived class object:

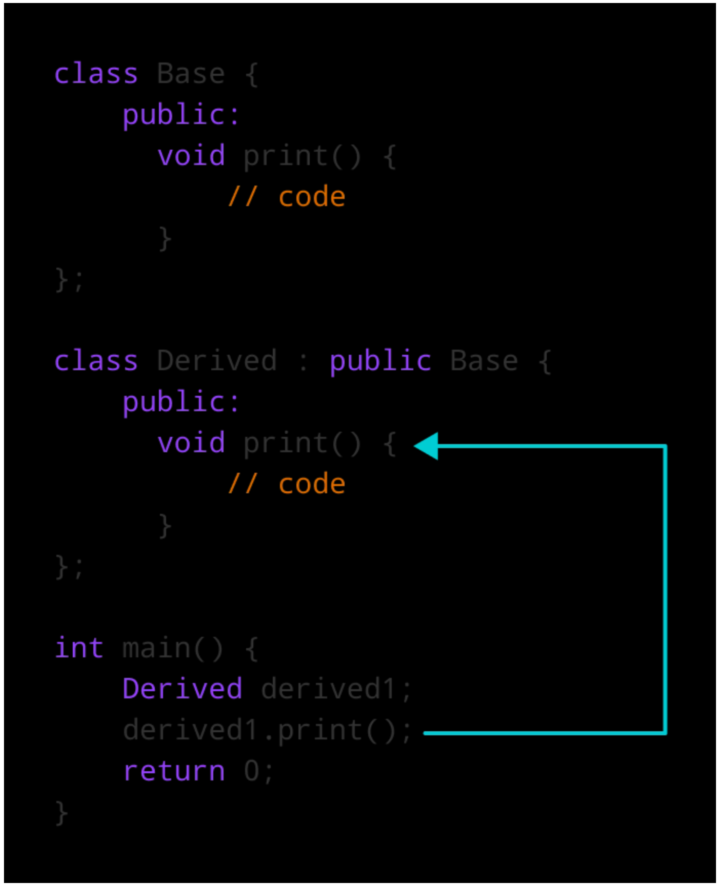
```
derived1.Base:: print();
```

Output:

```
Derived Function
Base Function
```

Or

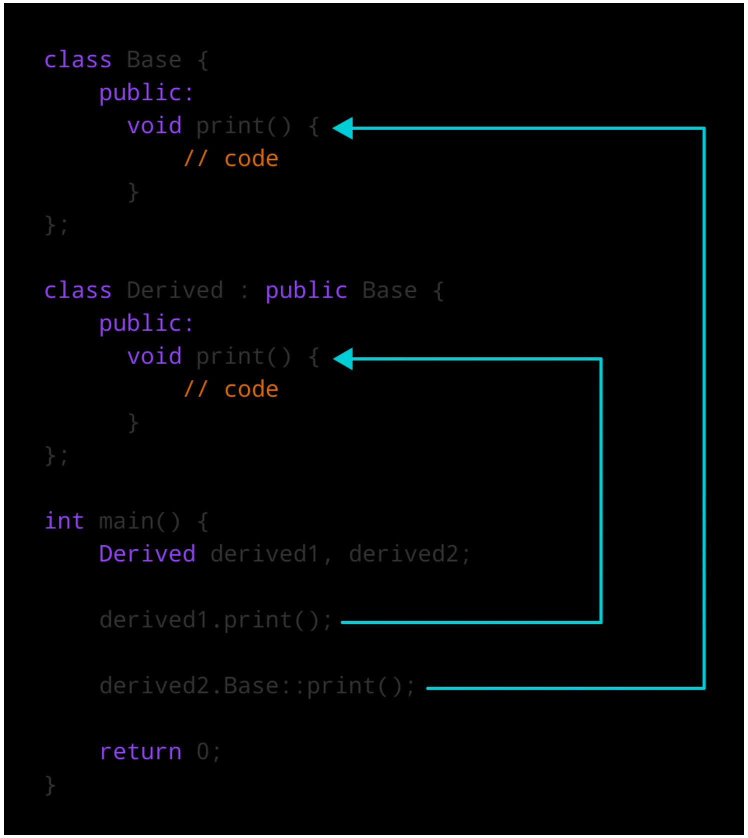
With base class object direct call base class function



```
class Base {
    public:
      void print() {
          // code
      }
};

class Derived : public Base {
    public:
      void print() {
          // code
      }
};

int main() {
    Derived derived1;
    derived1.print();
    return 0;
}
```

Access Overridden Function to the Base Class

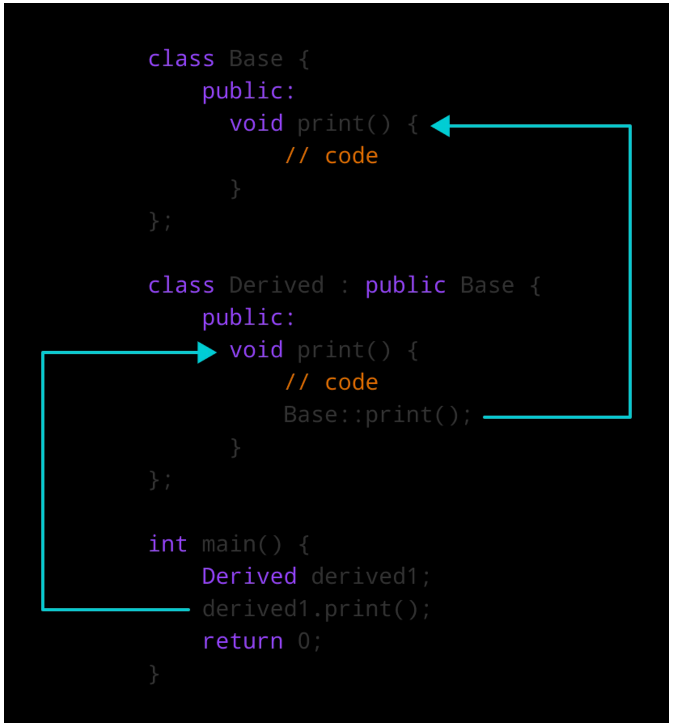
```
class Base {
   public:
    void print() {
        cout << "Base Function" << endl;
    }
};

class Derived : public Base {
   public:
    void print() {
        cout << "Derived Function" << endl;
    }
};

int main() {
    Derived derived1, derived2;
    derived1.print();
    // access print() function of the Base class
    derived2.Base::print();

    return 0;
}
```



```
class Base {
    public:
      void print() {
          // code
      }
};

class Derived : public Base {
    public:
      void print() {
          // code
      }
};

int main() {
    Derived derived1, derived2;

    derived1.print();

    derived2.Base::print();

    return 0;
}
```

Call Overridden Function From Derived Class

```
class Base {
   public:
    void print() {
        cout << "Base Function" << endl;
    }
};

class Derived : public Base {
   public:
    void print() {
        cout << "Derived Function" << endl;

        // call overridden function
        Base::print();
    }
};
int main() {
    Derived derived1;
    derived1.print();
    return 0;
}
```



```
class Base {
    public:
      void print() {
          // code
      }
};

class Derived : public Base {
    public:
      void print() {
          // code
          Base::print();
      }
};

int main() {
    Derived derived1;
    derived1.print();
    return 0;
}
```

Call Overridden Function Using Pointer

```
// C++ program to access overridden function using pointer
// of Base type that points to an object of Derived class

class Base {
   public:
    void print() {
        cout << "Base Function" << endl;
    }
};
class Derived : public Base {
   public:
    void print() {
        cout << "Derived Function" << endl;
    }
};
int main() {
    Derived derived1;
    // pointer of Base type that points to derived1
    Base* ptr = &derived1;
    // call function of Base class using ptr
    ptr->print();
    return 0;
}
```

Output:

```
Base Function
```

Constructors, Destructors, and Inheritance

```
class base {
public:
  base() {
    cout << “Constructing base class\n”;
  }
  ~base() {
    cout << “Destructing base class\n”;
  }
};
class derived : public base {
public:
  derived() {
    cout << “Constructing derived
class\n”;
  }
  ~derived() {
    cout << “Destructing derived
class\n”;
  }
};
```

```
void main() {
  derived obj;
}
```

Output:

```
      Constructing base class
      Constructing derived class
      Destructing derived class
      Destructing base class
```

Identify the type of inheritance:

```
class FacetoFace
{
char CenterCode[10];
public:
void Input();
void Output()
};
class Online
{
char website[50];
public:
void SiteIn();
void SiteOut();
};
```

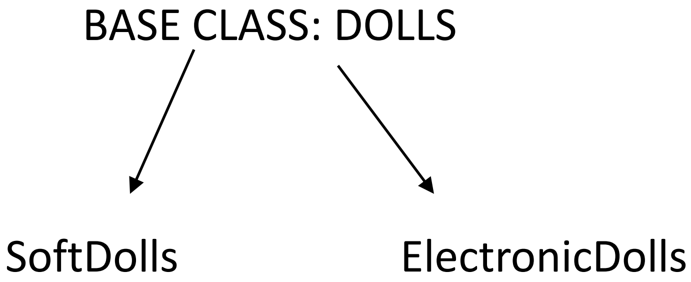
```
class Training : public FacetoFace,
private Online
{
long Tcode;
float Charge;
int Period;
public:
void Register();
void Show();
};
```

Base Classes:
FacetoFace Online
Derived Class:
Training

Multiple base classes so multiple inheritance

```
Class Dolls
{
char Dcode[5];
protected:
float price;
void CalcPrice(float);
Public:
Dolls();
void Dinput();
void Dshow();
};
```

```
class SoftDolls: public Dolls
{
char SDName[20];
float Weight;
public:
SoftDolls();
void SDInput();
void SDShow();
};
class ElectronicDolls: public Dolls
{
char EDName[20];
char BatteryType[10];
int Batteries;
public:
ElectronicDolls();
void EDInput();
void EDShow();
};
```



BASE CLASS: DOLLS

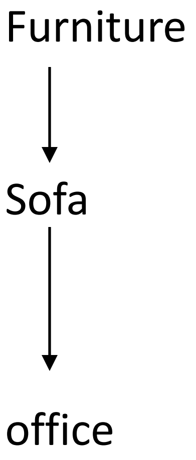
SoftDolls

ElectronicDolls

HIERARCHICAL INHERITANCE

```
class furniture
{
char Type;
char Model[10];
public:
furniture();
void Read_fur_Details();
void Disp_fur_Details();
};
class Sofa : public furniture
{
int no_of_seats;
float cost_of_sofa;
public:
void Read_sofa_details();
void Disp_sofa_details();
};
```

```
class office : private Sofa
{
int no_of_pieces;
char Delivery_date[10];
public:
void Read_office_details();
void Disp_office_details();
};
Void main()
{
office MyFurniture;
}
```



Sofa is derived from furniture
Office is derived from sofa.
Multi-level Inheritance

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

                                                    // x is public
class B : public A                                  // y is protected
{                                                   // z is not accessible from B

};

                                                          // x is protected
class C : protected A                                     // y is protected
{                                                         // z is not accessible from C

};

class D : private A                                // x is private
{                                                  // y is private
};                                                 // z is not accessible from D
```

Storage Classes

A storage class defines the scope (visibility) and life-time of variables and/or functions within a Program

Storage of Memory location (where variable get memory allocation)
scope, default value and life time of the variable

- auto
- register
- static
- extern

The auto Storage Class

The **auto** storage class is the default storage class for all **local variables**.

```
{
            int mount;
            auto int month;
 }
```

The example above defines two variables with the same storage class, auto can only be used within functions, i.e., local variables.

Memory : RAM
Default: garbage value

Life time or scope: declaration with in the function or main
                    can be access with in the block or method

```
auto int a=10;  ✗
Void main()
{

  auto int a=10;      →      Method scope

  {

       int a;         →      block scope
       cout<<a;
}

Cout<<a;

}
```

The register Storage Class

The **register** storage class is used to define local variables that should be stored in a register instead of RAM.

The variable has a maximum size equal to the register size

The register should only be used for variables that require quick access such as counters (repeatedly).

```
{
            register int miles;
 }
```

Memory : cpu register   (faster than auto)
Default: garbage value

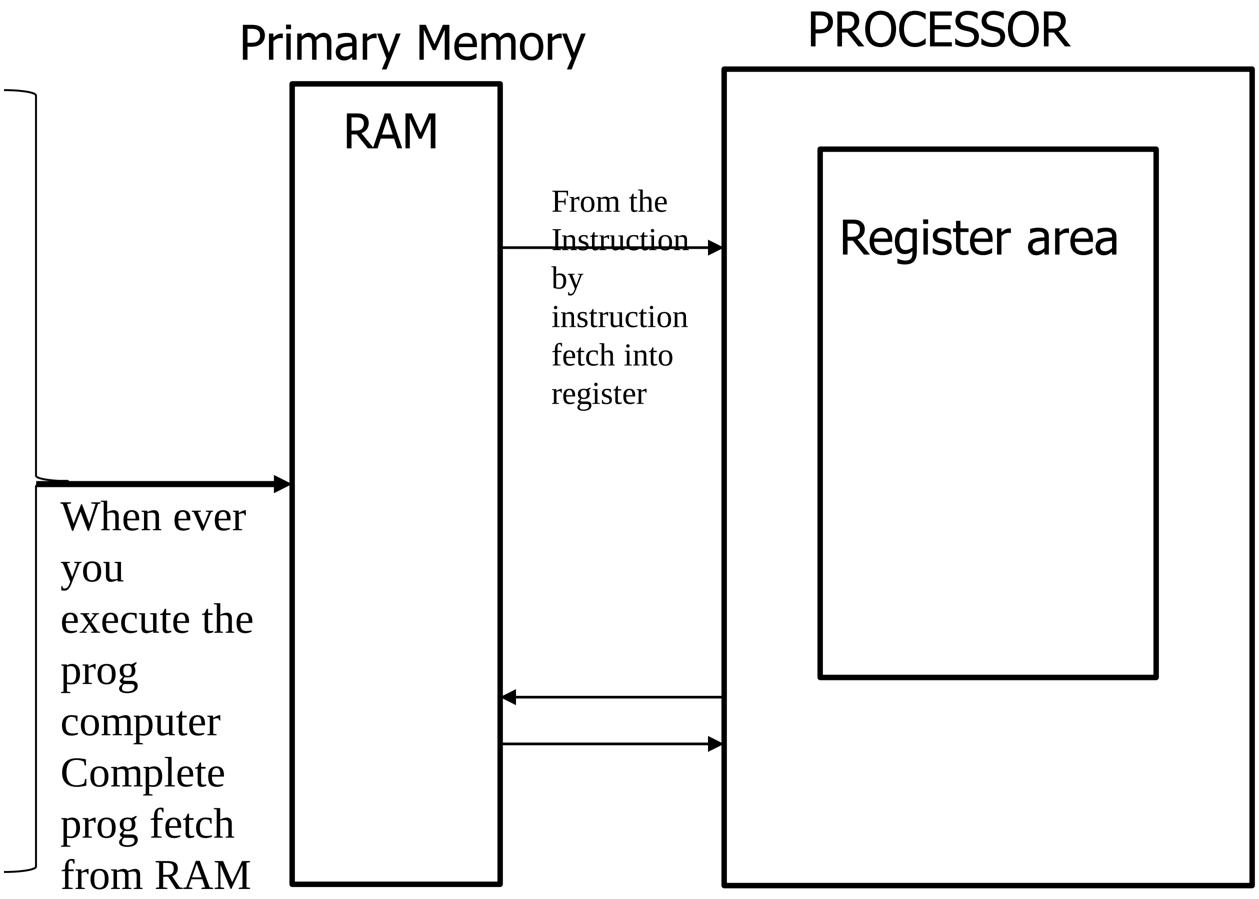
Scope: Local to the function in which it is declared.
Lifetime: Till the end of function/method block, in which the variable is defined.

Applications: Loops

Secondary memory

Sum of first n values

```
Void main()
{

  int n, sum=0, i;
  cin<<n;
  for(i=1; i<=n; i++)
  {
    sum=sum+1;
  }
  cout<<sum;
}
```



When ever you execute the prog computer Complete prog fetch from RAM

From the Instruction by instruction fetch into register

auto int n, sum=0, i; these all are stored in RAM because auto. Processor collect the i values from RAM for each iteration.

So instead of declaring all the variables in RAM better to declare in Register.

Why we not use register to all the variables: register memory is very small. So we can use for execute the instructions not to store the all variables. Its not global (permanent)

The static Storage Class

keep a local variable in existence during the life-time of the program instead of creating and destroying it each time it comes into and goes out of scope.

The static modifier may also be applied to global variable

Scope: the same block where the variable is declared
Life: entire program
Storage: memory
Default : zero

It is a local variable which is capable of returning a value even when control is transferred to the function call.

```
Main()
{
   increment();
   increment();
   increment();
}

increment();
{
  auto int i=1;
  cout<<i;
   i=1+1;

}
```

```
1
1
1
```

```
Main()
{
   increment();
   increment();
   increment();
}

increment();
{
  static int i=1;
  cout<<i;
   i=1+1;

}
```

```
1
2
3
```

The extern Storage Class

The **extern** storage class is used to give a reference of a global variable that is visible to ALL the program files.

The extern modifier is most commonly used when there are two or more files sharing the same global variables or functions

First File: main.cpp

```
#include <iostream>
int count ;
 extern void write_extern();
 main()
{
        count = 5;
        write_extern();
}
```

Second File: support.cpp

```
#include <iostream>
extern int count;
 void write_extern(void)
 {
std::cout << "Count is " << count << std::endl;
}
```

Storage Class	Keyword	Lifetime	Visibility	Initial Value
Automatic	auto	Function Block	Local	Garbage
External	extern	Whole Program	Global	Zero
Static	static	Whole Program	Local	Zero
Register	register	Function Block	Local	Garbage

```
using namespace std;
int g; //global variable, initially holds 0
void test_function()
{
static int s; //static variable, initially holds 0 register  int
r; //register variable
r=5;
s=s+r*2;
cout<<"Inside test_function"<<endl;
 cout<<"g = "<<g<<endl;
cout<<"s = "<<s<<endl;
cout<<"r = "<<r<<endl;
}
```

```
int main()
{
int a; //automatic variable
 g=25;
a=17;
 test_function();
cout<<"Inside main"<<endl; cout<<"a =
"<<a<<endl;
 cout<<"g = "<<g<<endl;
test_function();
return 0;
 }
```

```
class Base1 {
public:
        Base1()
        { cout << " Base1's constructor called" << endl; }
};

class Base2 {
public:
        Base2()
        { cout << "Base2's constructor called" << endl; }
};

class Derived: public Base1, public Base2 {
public:
        Derived()
        { cout << "Derived's constructor called" << endl; }
};

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

```
Base1's constructor called
Base2's constructor called
Derived's constructor called
```

```
class Base1 {
public:
        ~Base1() { cout << " Base1's destructor" << endl; }
};

class Base2 {
public:
        ~Base2() { cout << " Base2's destructor" << endl; }
};

class Derived: public Base1, public Base2 {
public:
        ~Derived() { cout << " Derived's destructor" << endl;
}
};

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

```
Derived's destructor
Base2's destructor
Base1's destructor
```

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

class base {
        int arr[10];
};

class b1: public base { };

class b2: public base { };

class derived: public b1, public b2 {};

int main(void)
{
cout << sizeof(derived);
return 0;
}
```

Assume that an integer takes 4 bytes

80

```
#include<iostream>

using namespace std;
class P {
public:
void print() { cout <<" Inside P"; }
};

class Q : public P {
public:
void print() { cout <<" Inside Q"; }
};

class R: public Q { };

int main(void)
{
R r;
r.print();
return 0;
}
```

```
Inside Q
```

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

class Base {
private:
        int i, j;
public:
        Base(int _i = 0, int _j = 0): i(_i), j(_j) { }
};
class Derived: public Base {
public:
        void show(){
                cout<<" i = "<<i<<" j = "<<j;
        }
};
int main(void) {
Derived d;
d.show();
return 0;
}
```