Amritsar College of Engineering & Technology, Amritsar, Punjab, INDIA
NAAC - A grade, NBA accredited courses(2009-12, 2016-18), UGC Autonomous College

SUBJECT: OOPS (ACCS-16302)
UNIT-III

Er. Neha Chadha
Assistant Professor
Department of Computer Science and Engineering

Class

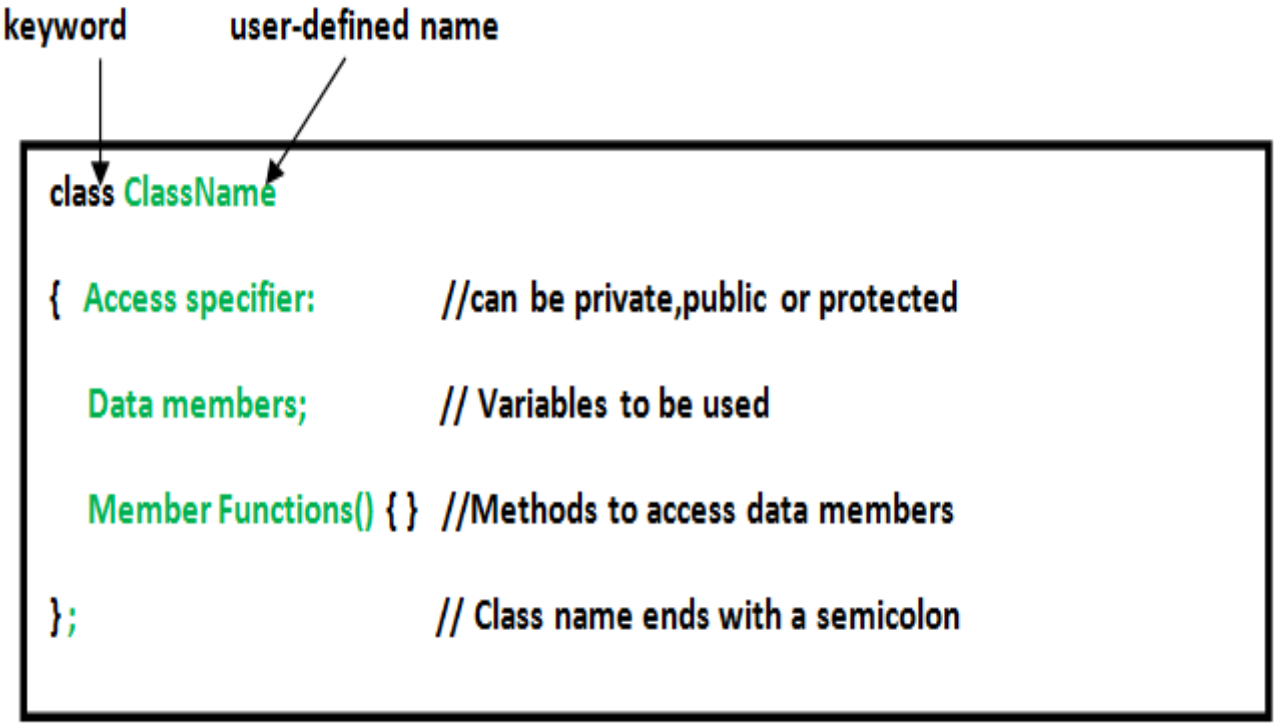
- A **class in C++** is the building block, that leads to Object-Oriented programming. It is a user-defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that **class**

Data members
Member Functions

- Data members are the data variables

- Member functions are the functions used to manipulate these variables and together these data members and member functions defines the properties and behavior of the objects in a Class. Member Functions can be defined inside the class as well as outside the class

Syntax of Class



Object

- When a class is defined, only the specification for the object is defined; no memory or storage is allocated. To use the data and access functions defined in the class, you need to create objects
- Syntax

  ClassName ObjectName;

Accessing data members and member functions:

- The data members and member functions of class can be accessed using the dot('.') operator with the object.

  For example

✓ name of object is **obj**

✓ member function **Showname()**

obj.Showname()

- Accessing a data member depends solely on the access control of that data member.

Access Specifier

Specifier	Within Same Class	In Derived Class	Outside the Class
Private	Yes	No	No
Protected	Yes	Yes	No
Public	Yes	Yes	Yes

Empty Class

- **Empty class** means, a **class** that does not contain any data members

- minimum memory available at object address location is **1 byte**

Array of object

- Like array of other user-defined data types, an array of type class can also be created.
- The array of type class contains the objects of the class as its individual elements.
- Thus, an array of a class type is also known as an array of objects.
- An array of objects is declared in the same way as an array of any built-in data type.

Array of Objects Program

```
#include <iostream.h>
 #include<conio.h>
 class stu
 {
 int rn;
 public:
 void getdata()
 {cin>>rn;}
 void showdata();
 };
 void stu::showdata()
 {
 cout<<rn;
 }
 void main ()
 {
 clrscr();
  stu s[10];
  int i;
  for(i=0;i<3;i++)
  {
  s[i].getdata();
  }
  for(i=0;i<3;i++)
  {
  s[i].showdata();
  }
getch();}
```

Nested Class

- A nested class is a class that is declared in another class. The nested class is also a member variable of the enclosing class and has the same access rights as the other members.

Nested Class Program

```
#include<iostream.h>
#include<conio.h>
class stu
{
public:
      class dob         //Member var of Class stu
      {
      int date;
      int mon;
      public:
      void getinner()
      {
      cin>>date>>mon;
      }
      void showinner()
      {
      cout<<date<<" "<<mon;
      }};
      void print()
      {
      cout<<"helo"; }
};
```

```
void main()
{
clrscr();
stu s;
stu::dob s1;      //if dob s1(give
   warning only)
s1.getinner();
s1.showinner();
s.print();
getch();
}
```

Pointer to object

```
#include <iostream.h>
 #include<conio.h>
 class stu
 {
 int rn;
 public:
 void getdata()
 {cin>>rn;}
 void showdata();
 };
 void stu::showdata()
 {
 cout<<rn;
 }
 void main ()
 {
 clrscr();
  stu s1;
  stu *p;
  p=&s1;
  p->getdata();
  p->showdata();
getch();
}
```

Const keyword(qualifier)

- **'const' keyword** stands for constant. In **C++** it is used to make some values constant throughout the program.
- If we make an artifact of a **C++** program as constant then its value cannot be changed during the program execution.

- Note: take value in const variable otherwise it will be fix garbage value

Const keyword contd..

- const member function

returntype funcname(argu..)const

returntype classname:: funcname(argu..)const

- const object

const classname objname;

Const keyword example

```
#include<iostream.h>
#include<conio.h>
void main()
{
clrscr();
const int a=90;
a=80;
cout<<a;
getch();
}
```

Const member function program

```
#include<iostream.h>
#include<conio.h>
class stu
{
int a;
public:
void get(int b)
{
a=b;
}
int incr() const
{
a++;
return a;}
};
void main()
{
clrscr();
stu s1;
s1.get(10);
s1.incr();
getch();
}
```

Const object program

```
#include<iostream.h>
#include<conio.h>
const int a=100;
class stu
{
public:
stu(int b)
{
a=b;
cout<<a;
}};
void main()
{
clrscr();
const stu s1(10);
getch();
}
```

Constructor

- It is special member function of the class.
- In C++, **Constructor** is automatically called when object(instance of class) create.
- A **constructor** is a member function of a class which initializes objects of a class.

Characteristics of Constructors

- Used to initialize data members
- They should be declared in the public section.
- They are invoked automatically when the objects are created.
- They do not have return types, not even void and they cannot return values.
- They have the same name as that of class
- They cannot be inherited.
- Like other C++ functions, Constructors can have default arguments.

Types of Constructors in C++

Constructors are of three types:

- Default Constructor
- Parameterized Constructor
- Copy Constructor

Destructor

- Destructor is a special class function which destroys the object as soon as the scope of object ends.
- The destructor is called automatically by the compiler when the object goes out of scope.
- The syntax for destructor is same as that for the constructor, the class name is used for the name of destructor, with a **tilde** ~ sign as prefix to it.

Constructor Overloading

- **Overloaded constructors** essentially have the same name (name of the class) and different number of arguments. A **constructor** is called depending upon the number and type of arguments passed.

Constructor Overloading

```
#include<iostream.h>
#include<conio.h>
class stu
{
int a;
public:
stu()                //default
{cout<<"default";}
stu(int x);          //parameterized
/*{a=x;
cout<<a;} */
stu(stu &p)          //copy
{
a=p.a;}
void show()
{
cout<<a;}
~stu()               //destructor
{cout<<"destruct";}
};
stu::stu(int x)      //you can also declare const o/s class
{
a=x;
}
void main()
{
clrscr();
stu s1(10),s2=s1;    //copy obj
stu s3(s1);          //copy obj
s1.show();
s2.show();
s3.show();
getch();
}
```

Dynamic Constructor

- When allocation of memory is done dynamically using dynamic memory allocator new in a constructor, it is known as dynamic constructor. By using this, we can dynamically initialize the objects.

Dynamic Constructor Program

```
#include<iostream.h>
#include<conio.h>
class stu
{ int *rn;
public:
stu()
{}
stu(int x)
{
rn=new int;
*rn=x;}
void show()
{cout<<*rn;}
};
void main()
{clrscr();
stu s(901);
s.show();
```

Order of Execution of constructor and destructor

```
#include<iostream.h>
#include<conio.h>
class A
{
public:
A()
{cout<<"class A";}
~A(){"class A destr";}
};
class B :public A
{
public:
B()
{
cout<<"class B";}
~B(){"class B destr";}};
void main()
{
clrscr();
B ob;
getch();
}
```

Output:

class A

class B

class B destr

class A destr

Container class

- When a class contains objects of another class or its members, this kind of relationship is called **containership.** That class is known as container class.

Container Class Program

```
#include<iostream.h>
#include<conio.h>
class A
{
public:
void get1()
{cout<<"A";}
};
class B
{A x;
public:
void get2()
{cout<<"B";
}};
void main()
{
clrscr();
B o2;
o2.get2();
```

Friend Function

- If a function is defined as a friend function in C++, then the protected and private data of a class can be accessed using the function.
- By using the keyword friend compiler knows the given function is a friend function.
- For accessing the data, the declaration of a friend function should be done inside the body of a class starting with the keyword friend.

Syntax of friend function

```
class class_name
{
   friend data_type function_name(class_name
   object(optional));
};
```

- The function definition does not use either the keyword **friend or scope resolution operator**.

Characteristics of a Friend function

- The function is not in the scope of the class to which it has been declared as a friend.
- It cannot be called using the object as it is not in the scope of that class.
- It can be invoked like a normal function without using the object.
- It cannot access the member names directly and has to use an object name and dot membership operator with the member name.
- It can be declared either in the private or the public part.

Friend function using single classes

```
#include<iostream.h>
#include<conio.h>
class A
{
int a, b;
public:
void getA()
{cin>>a>>b;}
friend void add(A o1);    //friend void
    add(A ,B );
};
void add(A o1)
{
cout<<o1.a+o1.b;}
void main()
{
clrscr();
A o1;
o1.getA();
add(o1);
getch();
}
```

Friend function using multiple classes

```
#include<iostream.h>
#include<conio.h>
class B;
class A
{int a;
public:
void getA()
{cin>>a;}
friend void add(A o1,B o2);    //friend void add(A
     ,B );
};
class B
{
int b;
public:
void getB()
{cin>>b;}
friend void add(A o1,B o2); //  friend void add(A
     B ); };
void add(A o1,B o2)
{
cout<<o1.a+o2.b;
}
void main()
{
clrscr();
A o1;B o2;
o1.getA();
o2.getB();
add(o1,o2);
getch();
}
```

What is Inheritance ?

- Inheritance is a relationship between two or more classes where derived class inherits properties of pre-existing (base) classes.

Base Class: It is the class whose properties are inherited by another class. It is also called Super Class or Parent Class.

Derived Class: It is the class that inherit properties from base class(es). It is also called Sub Class or Child Class.

Defining Derived Classes

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

```
Class derived_class_name : visibility_mode base_class_name
{
……..// members of derived class
};
```

- The colon indicates that the derived_class_name is derived from the base_class_name.
- The visibility mode (access specifier) is optional, if present, may be private, public, or protected.

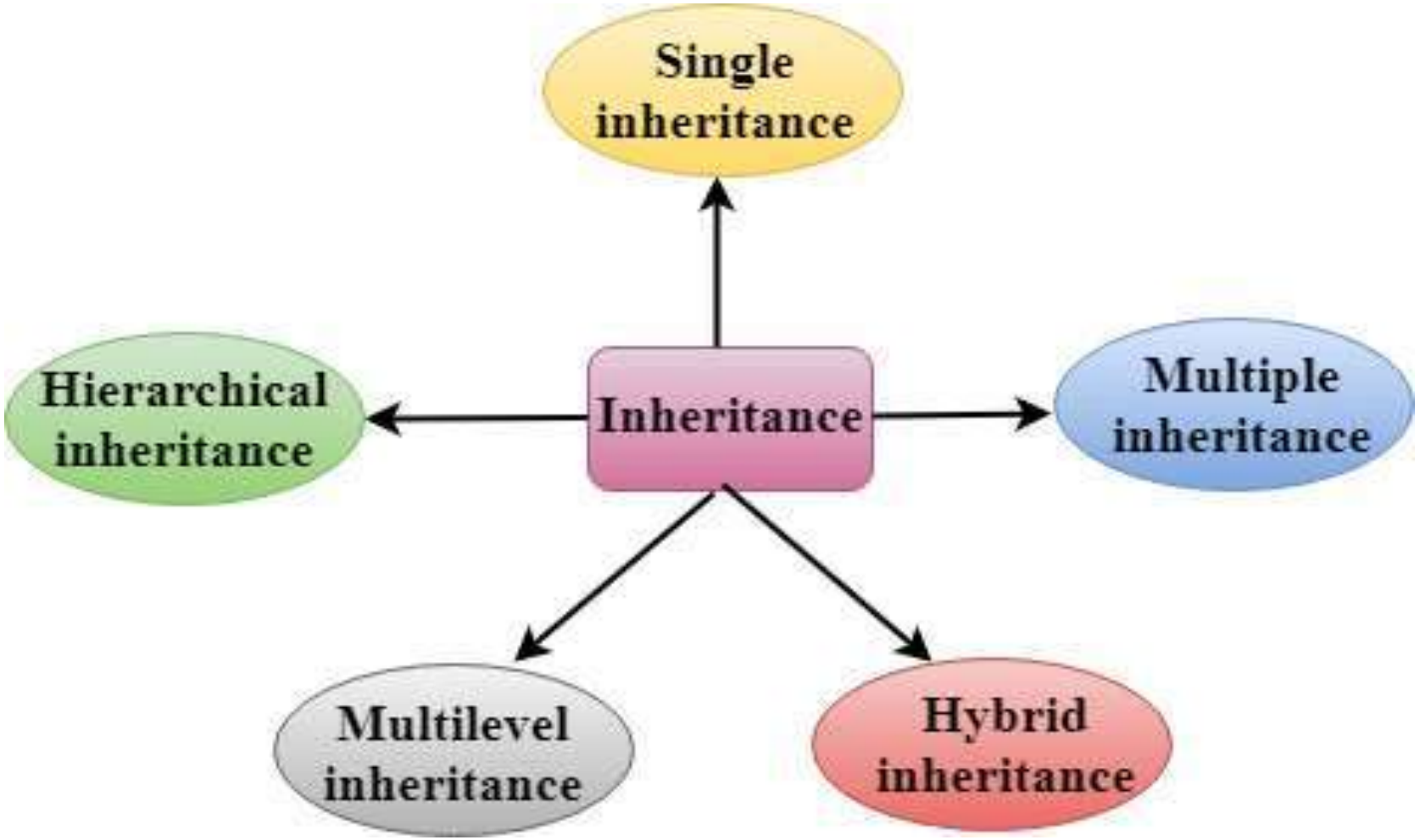
Visibility Mode

The visibility mode specifies how the features of the base class are visible to the derived class.

Private : When a derived class privately inherits a base class, the protected and public members of base class become private members of the derived class.

Public : In Public mode, the protected and public members of base class become protected and public members of derived class respectively.

Protected : In Protected mode, the protected and public members of base class become protected members of the derived class.

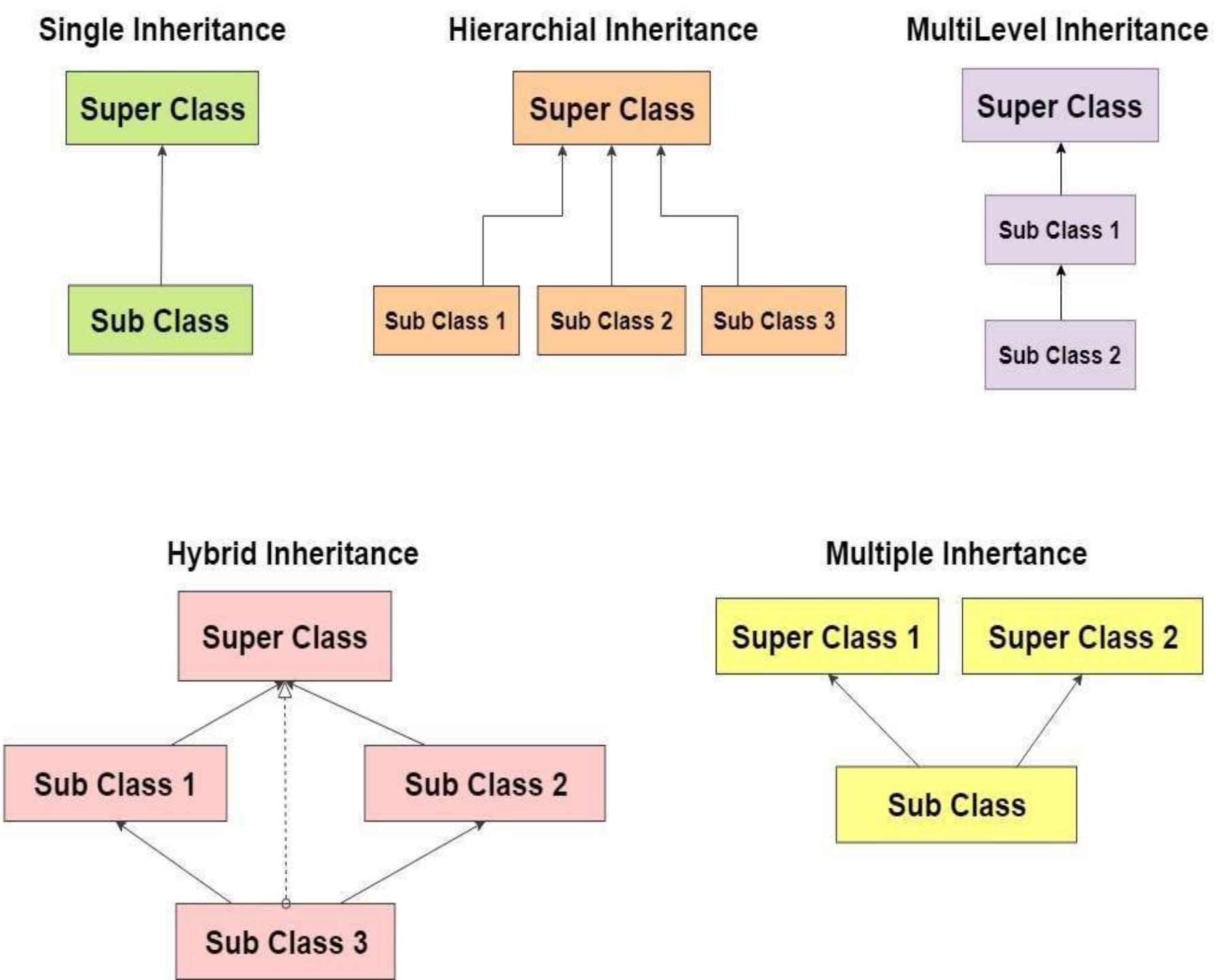

Single inheritance
Hierarchical inheritance
Inheritance
Multiple inheritance
Multilevel inheritance
Hybrid inheritance

BASE CLASS IN C++

VERSUS

DERIVED CLASS IN C++

BASE CLASS IN C++	DERIVED CLASS IN C++
A class that helps to derive or create new classes	A class created or derived from an already existing class
Also called parent class or superclass	Also called child class or subclass
Cannot inherit properties and methods of the derived class	Can inherit the properties and methods of the base class


Single Inheritance
Super Class
Sub Class
Hierarchial Inheritance
Super Class
Sub Class 1
Sub Class 2
Sub Class 3
MultiLevel Inheritance
Super Class
Sub Class 1
Sub Class 2
Hybrid Inheritance
Super Class
Sub Class 1
Sub Class 2
Sub Class 3
Multiple Inhertance
Super Class 1
Super Class 2
Sub Class

Single Inheritance

- In Single Inheritance, one derived class inherits from one base class

Single Inheritance

Single Inheritance is declared as follows :

```
class A
{
        ..... // Members of base class
};
class B : public A
{
        ..... // Members of derived class
};
```

Multilevel Inheritance

- In Multiple Inheritance, a class (Inter-mediate base class) is derived from base class, which is again derived into some other class
- The class A serves as a base class for the derived class B, which in turn serves as a base class for the derived class C.

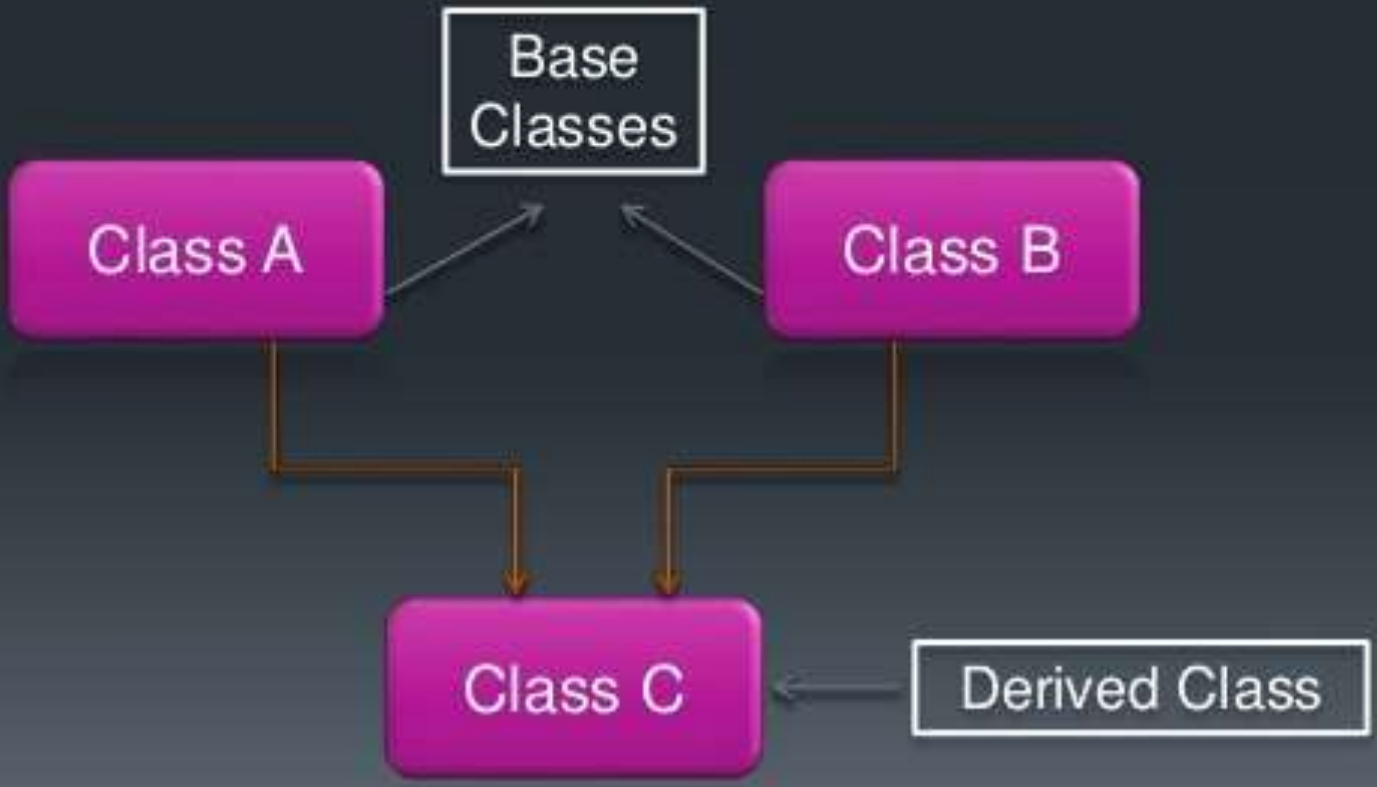
Multilevel Inheritance

Multilevel Inheritance is declared as follows :

```
class A
{
        ..... //Members of base class
{;
class B : public A
{
        ..... //Members of derived class, B derived from A
};
class C : public B
        ..... //Members of derived class, C derived from B
};
```

Multiple Inheritance

- In Multiple Inheritance, a class is derived from more than one base classes



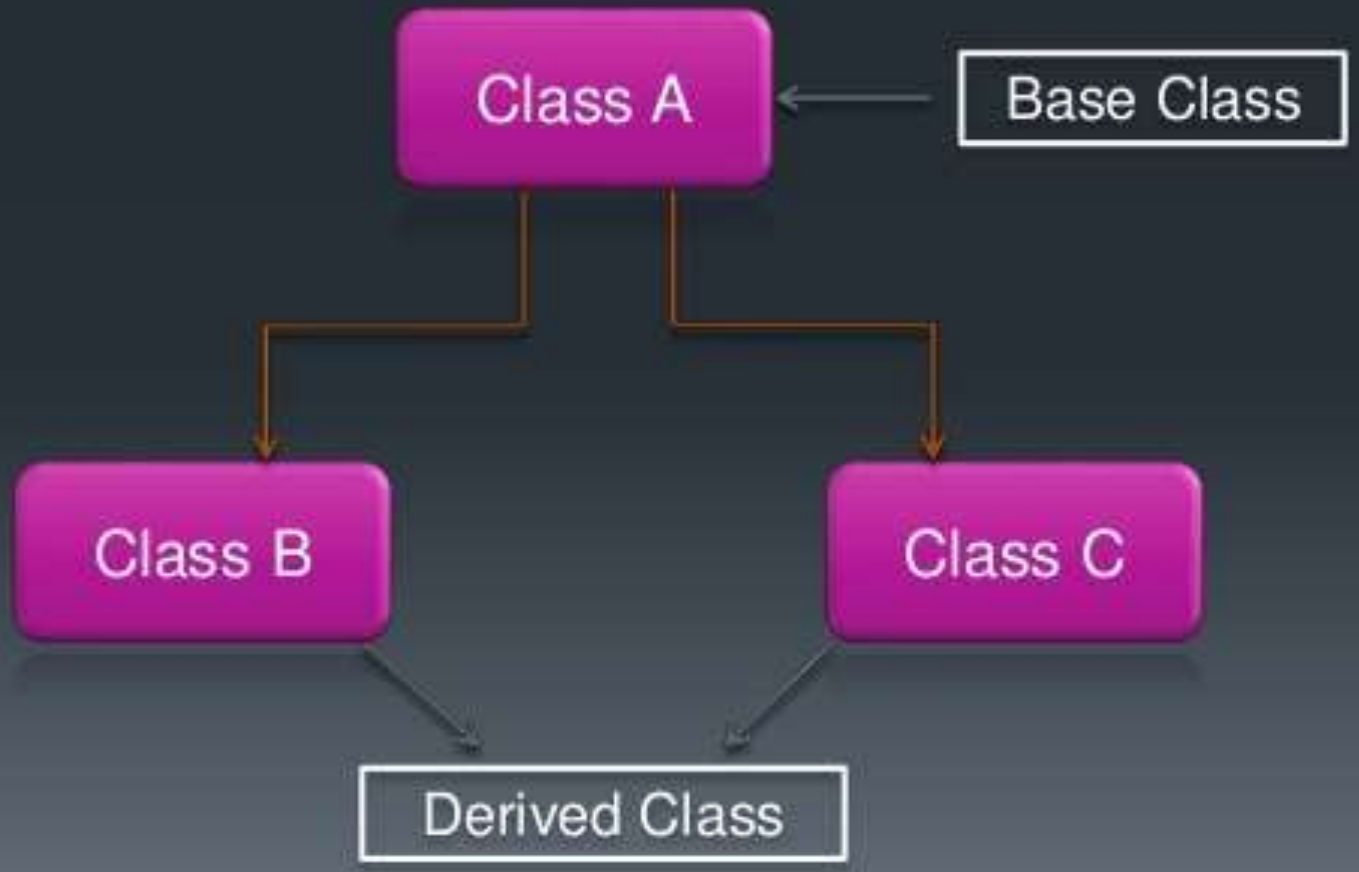
Multiple Inheritance

Multiple Inheritance is defined as follows :

```
class A
{
        ..... //Members of base class A
{;
class B
{
        ..... //Members of base class B
};
class C : public A, public B
{
        ..... //Members of derived class
};
```

Hierarchical Inheritance

- In Hierarchical Inheritance, one base class is inherited into two or more Derived Classes



Hierarchical Inheritance

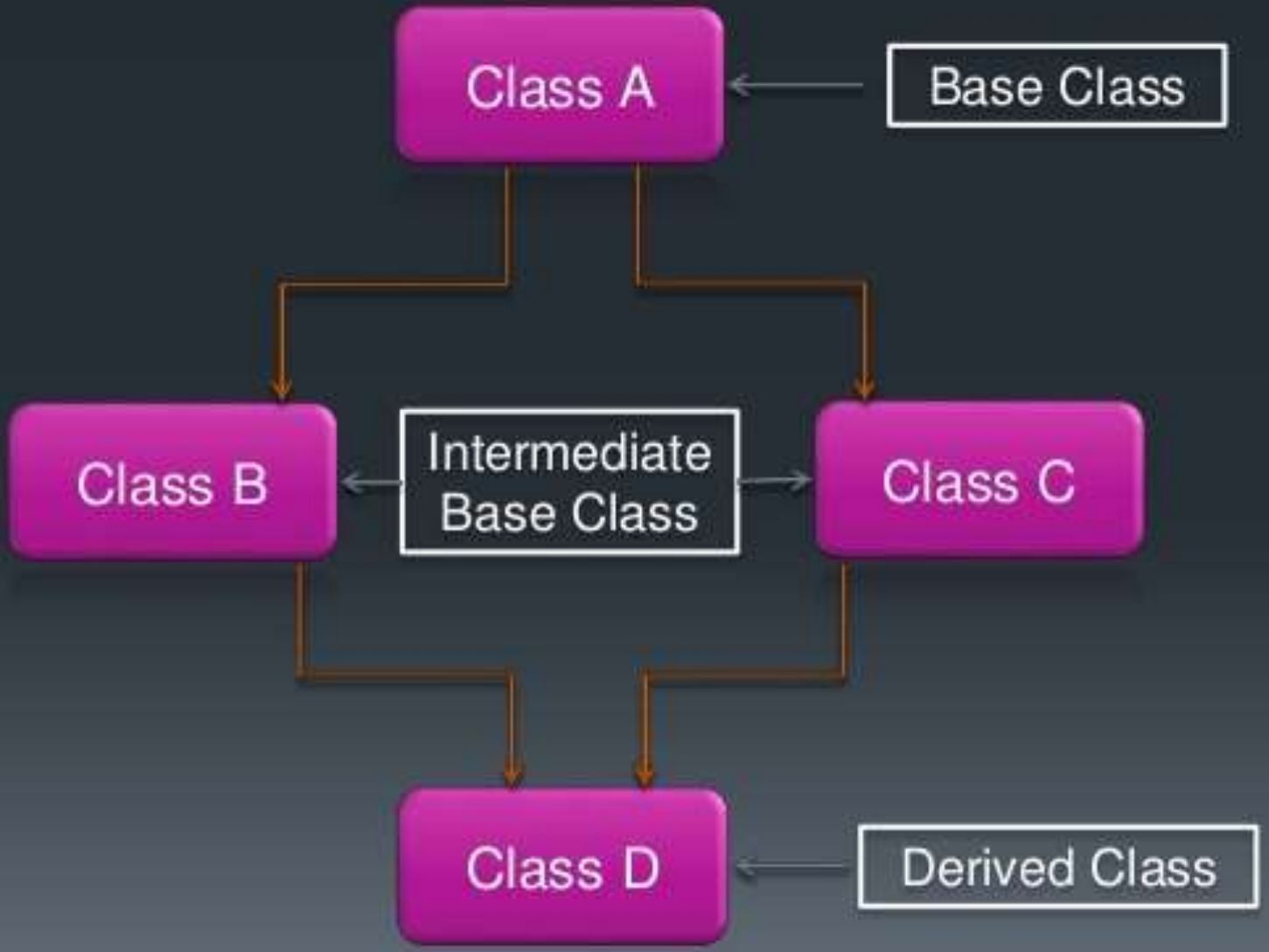
Hierarchical Inheritance is declared as follows

```
class A
{
        ..... //Members of base class A
{;
class B : public A
{
        ..... //Members of Derived class , B derived from A
};
class C : public A
{
        ..... //Members of derived class , C derived from A
};
```

Hybrid Inheritance

- In Hybrid Inheritance, more than one type of inheritances are used to derive a new Sub Class.
- For example, inheriting a class from two different classes, which in turn have been derived from the same base class.
- Any legal combination of other four types of inheritance comes under Hybrid Inheritance

Hybrid Inheritance



Hybrid Inheritance

A basic form of Hybrid Inheritance :

```
class A
{
        ..........
        ..........
};
class B : public A
{
        ..........
};
class C : public A
{
        ..........
};
class D : public B, public C
{
        .........
};
```

Virtual Base Class

- Virtual base class is a way of preventing multiple instances(duplicate copies) of a given class appearing in an inheritance hierarchy when using multiple/hybrid inheritance.
- Suppose we have a base class called ‘A’ and two classes derived from it ‘B’ and ‘C’ and we derived class ‘D’ from class ‘B’ and ‘C’.
- That means all the members of class ‘A’ are inherited into ‘D’ twice, first via ‘B’ and again via ‘C’ This means, class ‘D’ would have duplicate sets of members inherited from ‘A’. This is an ambiguous situation

Virtual Base Class

This is how we create Virtual Base Class

```
class A
{
        ..........
        ..........
};
class B : virtual public A
{
        ..........
};
```

```
class C : virtual public A
{
        ..........
};
class D : public B, public C
{
        .........
};
```

Advantages of Inheritance

- Reusability : Inheritance helps the code to be reused in many situations. The base class is defines and once it is compiled, it need not be reworked. Using the concept of Inheritance, the programmer can create as many derived classes from the base class as needed while adding specific features to each derived class as needed
- Saves Time and Effort : The above concept of reusability achieved by inheritance saves the programmer's time and effort. Since the main code written can be reused in various situations as needed

Single Inheritance

```
#include<iostream.h>
#include<conio.h>
class A
{
int a;
public:
void getdataA()
{cin>>a; }
void putdataA()
{cout<<"a"<<a;}
};
class B:public A
{
int b;
public:
void getdataB()
{cin>>b; }
void putdataB()
{cout<<b;}
};
void main()
{clrscr();
B obj1;
obj1.getdataA();
obj1.putdataA();
obj1.getdataB();
obj1.putdataB();
getch();
}
```

Multilevel Inheritance

```
#include<iostream.h>
#include<conio.h>
class A
{
int a;
public:
void getdataA()
{cin>>a; }
void putdataA()
{cout<<"a "<<a;}
};
class B:public A
{
int b;
public:
void getdataB()
{cin>>b; }
void putdataB()
{cout<<"b "<<b;}
};
class C:public B
{
int b;
public:
void getdataB()
{cin>>b; }
void putdataB()
{cout<<"b "<<b;}
};
void main()
{clrscr();
B obj1;
obj1.getdataA();
obj1.putdataA();
obj1.getdataB();
obj1.putdataB();
getch();
}
```

Multiple Inheritance

```
#include<iostream.h>
#include<conio.h>
class A
{ int a;
public:
void getdataA()
{cin>>a; }
void putdataA()
{cout<<"a "<<a;}
};
class B
{
int b;
public:
void getdataB()
{cin>>b; }
void putdataB()
{cout<<"b "<<b;}
```

```
class C:public A,public B
{
int c;
public:
void getdataC()
{cin>>c; }
void putdataC()
{cout<<"c "<<c;}
};
void main()
{clrscr();
C obj1;
obj1.getdataA();
obj1.putdataA();
obj1.getdataB();
obj1.putdataB();
obj1.getdataC();
obj1.putdataC();
getch();
}
```

Hierarchical Inheritance

```
#include<iostream.h>
#include<conio.h>
class A
{
int a;
public:
void getdataA()
{cin>>a; }
void putdataA()
{cout<<"a "<<a;}
};
class B:public A
{
int b;
public:
void getdataB()
{cin>>b; }
void putdataB()
{cout<<"b "<<b;}
};
```

```
class C:public A
{
int c;
public:
void getdataC()
{cin>>c; }
void putdataC()
{cout<<"c "<<c;}
};
void main()
{clrscr();
C obj1;  B obj2;
obj1.getdataA();
obj1.putdataA();
obj1.getdataC();
obj1.putdataC();
obj2.getdataA();
obj2.putdataA();
obj2.getdataB();
obj2.putdataB();
getch();
}
```

Hybrid Inheritance

```
#include<iostream.h>
#include<conio.h>
class A
{
int a;
public:
void getdataA()
{cin>>a; }
void putdataA()
{cout<<"a "<<a;}
};
class B:virtual public A
{
int b;
public:
void getdataB()
{cin>>b; }
void putdataB()
{cout<<"b "<<b;}
};
class C:virtual public A
{
int c;
public:
void getdataC()
{cin>>c; }
void putdataC()
{cout<<"c "<<c;}
};
class D:public B,public C
{
int d;
public:
void getdataD()
{cin>>d; }
void putdataD()
{cout<<"d "<<d;}
};
void main()
{clrscr();
D obj3;
obj3.getdataB();
obj3.putdataB();
obj3.getdataC();
obj3.putdataC();
obj3.getdataA();
obj3.putdataA();
getch();
}
```

Virtual base Class

```
#include<iostream.h>
#include<conio.h>
class A
{
int a;
public:
void getdataA()
{cin>>a; }
void putdataA()
{cout<<"a "<<a;}
};
class B:virtual public A
{
int b;
public:
void getdataB()
{cin>>b; }
void putdataB()
{cout<<"b "<<b;}
};
class C:virtual public A
{
int c;
public:
void getdataC()
{cin>>c; }
void putdataC()
{cout<<"c "<<c;}
};
class D:public B,public C
{
int d;
public:
void getdataD()
{cin>>d; }
void putdataD()
{cout<<"d "<<d;}
};
void main()
{clrscr();
D obj3;
obj3.getdataB();
obj3.putdataB();
obj3.getdataC();
obj3.putdataC();
obj3.getdataA();
obj3.putdataA();
getch();
}
```

Operator overloading

C++ provides a special function to change the current functionality of some operators within its class which is often called as operator overloading.

Operator Overloading is the method by which we can change the function of some specific operators to do some different task.

rules of operator overloading

- The first and basic rule of operator overloading is: we can overload unary operator as only unary operator, it cannot be overload as binary operator and vice versa.
- We cannot overload those operators that are not a part of C++ language like '$'.
- We can perform operator overloading in only user defined classes. We cannot change the operators existing functionality.
- Using operator overloading we cannot change the presidency and associatively of operators.

operators cannot be overloaded

- :: Scope resolution operator.
- . Class membership operator.
- ?: ternary or conditional operator.
- .* pointer to member operator.
- ->* pointer to member operator.
- sizeof()

Syntax for C++ Operator Overloading

```
class className
{ … .. …
 public
returnType operator symbol (arguments) { … .. … }
… .. …
 };
```

- Here,
- returnType is the return type of the function.
- operator is a keyword.
- symbol is the operator we want to overload. Like: +, <, -, ++, etc.
- arguments is the arguments passed to the function.

Overloading Unary Operator

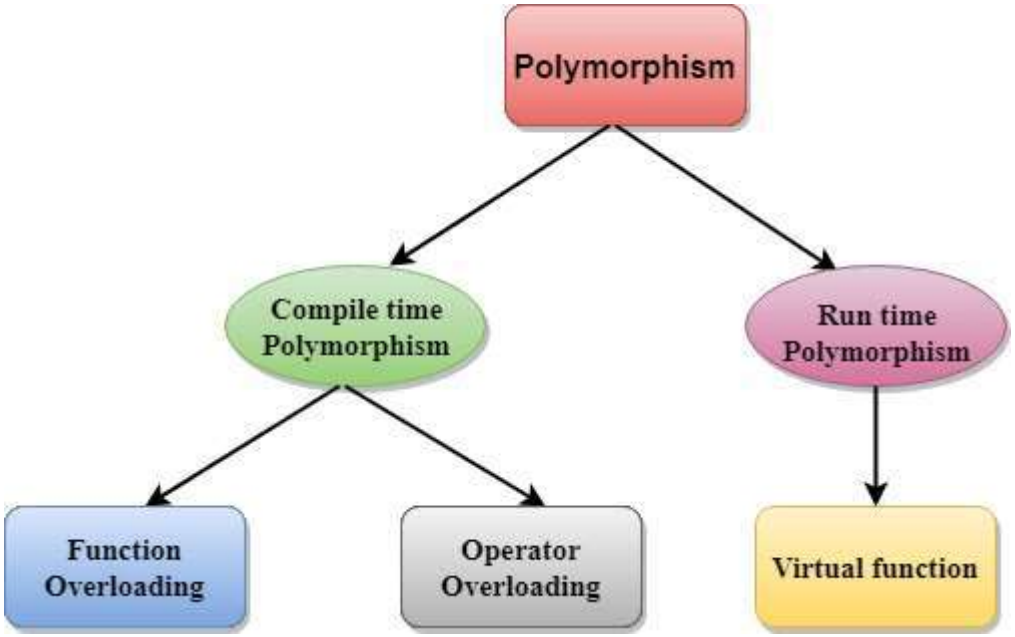
- Unary operators operate on only one operand. The increment operator ++ and decrement operator -- are examples of unary operators.

Overloading Binary Operator

- In **binary operator overloading** function, there should be one argument to be passed. It is **overloading** of an **operator** operating on two operands.

Polymorphism

- The term "Polymorphism" is the combination of "poly" + "morphs" which means many forms



Early binding

- The overloaded functions are invoked by matching the type and number of arguments. This information is available at the compile time and, therefore, compiler selects the appropriate function at the compile time. It is achieved by function overloading and operator overloading which is also known as static binding or early binding

Late binding

- Run time polymorphism is achieved when the object's method is invoked at the run time instead of compile time. It is achieved by method overriding which is also known as dynamic binding or late binding.

Function overriding

- If derived class defines same function as defined in its base class, it is known as function overriding in C++. It is used to achieve runtime polymorphism. It enables you to provide specific implementation of the function which is already provided by its base class.

Virtual Function

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

Pure Virtual Function

- A virtual function is not used for performing any task. It only serves as a placeholder.
- When the function has no definition, such function is known as "**do-nothing**" function.
- The "**do-nothing**" function is known as a **pure virtual function**. A pure virtual function is a function declared in the base class that has no definition relative to the base class.
- A class containing the pure virtual function cannot be used to declare the objects of its own, such classes are known as abstract base classes.
- The main objective of the base class is to provide the traits to the derived classes and to create the base pointer used for achieving the runtime polymorphism.
- Syntax

```
virtual void display() = 0;
```

Abstract Class

- An abstract class is a class that is designed to be specifically used as a base class. An abstract class contains **at least one pure virtual function**. You declare a pure virtual function by using a pure specifier (= 0) in the declaration of a virtual member function in the class declaration.

Virtual destructor

- A virtual destructor is **used to free up the memory space allocated by the derived class object or instance** while deleting instances of the derived class using a base class pointer object.

Type Conversions

- Basic to Class
- Class to Basic
- Class to Class

Basic to Class

```
//pre to user //premitive to class use constructor concept
#include<iostream.h>
#include<conio.h>
class A
{int a;
public:
A(){cout<<"hi";}
A(int y){a=y;}
void show()
{cout<<a;}
};
void main()
{
clrscr();
A obj1;
int x=5;
obj1=x;  //constru only call wen diff type of value assigned in object
//obj1.A(x) this is nt syntax
obj1.show();
getch();
}
```

Class to Basic

```
//pre to class to premitive use casting operator concept
#include<iostream.h>
#include<conio.h>
class A
{int a;
public:
void get(int x){a=x;}
void show()
{cout<<a<<endl;}
operator int()
{    return a;
}
};
void main()
{
clrscr();
A obj1;
obj1.get(1);
obj1.show();
int x;
x=obj1;  //x=obj1.operator int()
cout<<"value of x is "<<x;
getch();
}
```

Class to Class

```
//class to class either use casting operator concept or constructor
//using constructor
#include<iostream.h>
#include<conio.h>
class B
{int m;
public:
void set(int y){m=y;}
int get(){return m;}
};
class A
{int b;
public:
void show(){cout<<b;}
A(){}
A(B o1)
{b=o1.get();}
};
void main()
{
clrscr();
A obj1;
B obj2;
obj2.set(1);
obj1=obj2;
obj1.show();
getch();
}
```