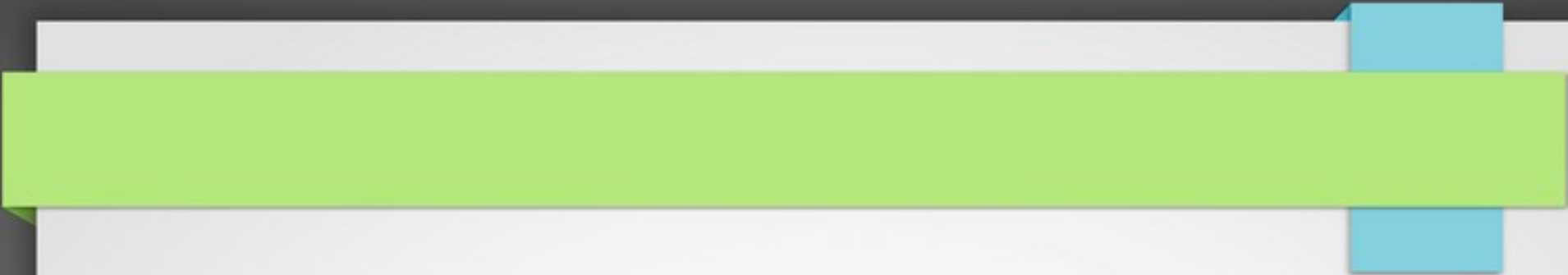




GLS UNIVERSITY

ADVANCED OBJECT ORIENTED PROGRAMMING

- Prof. Disha Shah



Module-1

Inheritance & Polymorphism

Points to be Covered

- Introduction to Inheritance
- Types of Inheritance
 - Single Level Inheritance
 - Multiple Inheritance
 - Multilevel Inheritance
 - Hierarchical Inheritance
 - Hybrid Inheritance
- Virtual Base Class
- Pointers to Object
- Pointers to Derived Classes and Virtual Functions
- Pure Virtual Functions and Abstract Class
- Constructors and Destructor in Derived Classes
- Constructors and Destructor in Multiple Inheritance
- Virtual Destructor
- Private Inheritance
- Protected Inheritance
- Containership

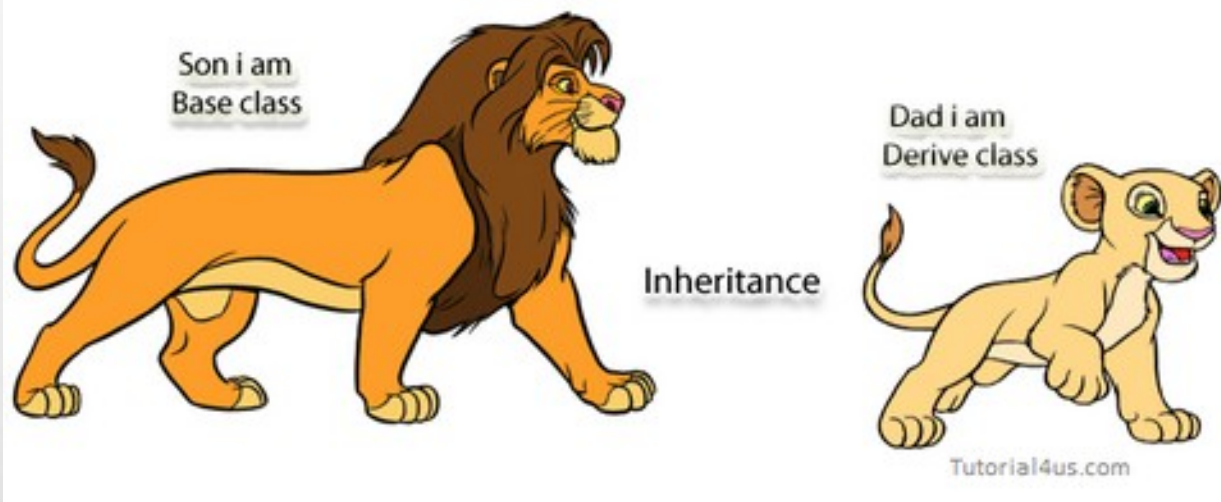
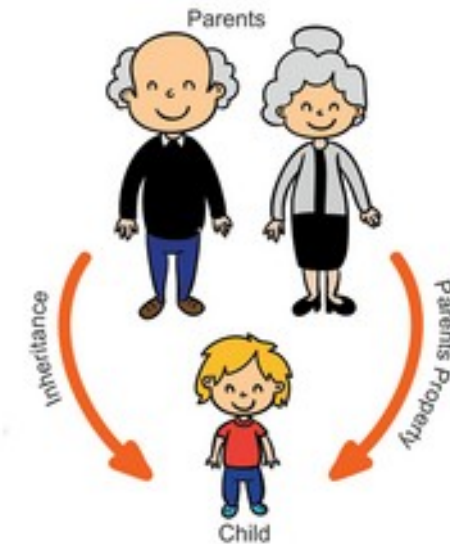
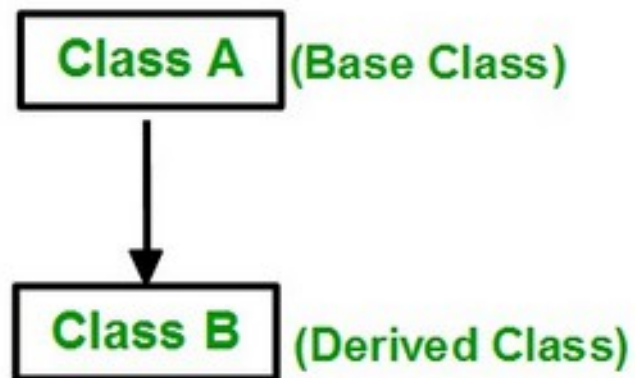
Why Inheritance?

- Concept of **reusability**
- Reducing the time required for program development
- **Deriving a new class from an old one is called inheritance**
- **Old class** – Base class / Parent class
- **New class** – Derived class / Child class

Introduction to Inheritance & Polymorphism

- ♦ The ability of object of one class to acquire the characteristics of objects of another class is known as **inheritance**.
- ♦ The existing class is called the **base class**, and the new class is referred to as the derived class.
- ♦ An existing class that is "parent" of a new class is called a **base class or parent class**. New class that inherits properties of the base class is called a **derived class or child class**.
- ♦ The basic syntax of inheritance is:
class DerivedClass : accessSpecifier BaseClass

Introduction to Inheritance & Polymorphism



Introduction to Inheritance & Polymorphism

Polymorphism:-

- The ability of an operator and function to take multiple forms is known as polymorphism.
- The word polymorphism means **having many forms**.
- Typically, polymorphism occurs when there is a hierarchy of classes and they are related by inheritance.



Introduction to Inheritance & Polymorphism

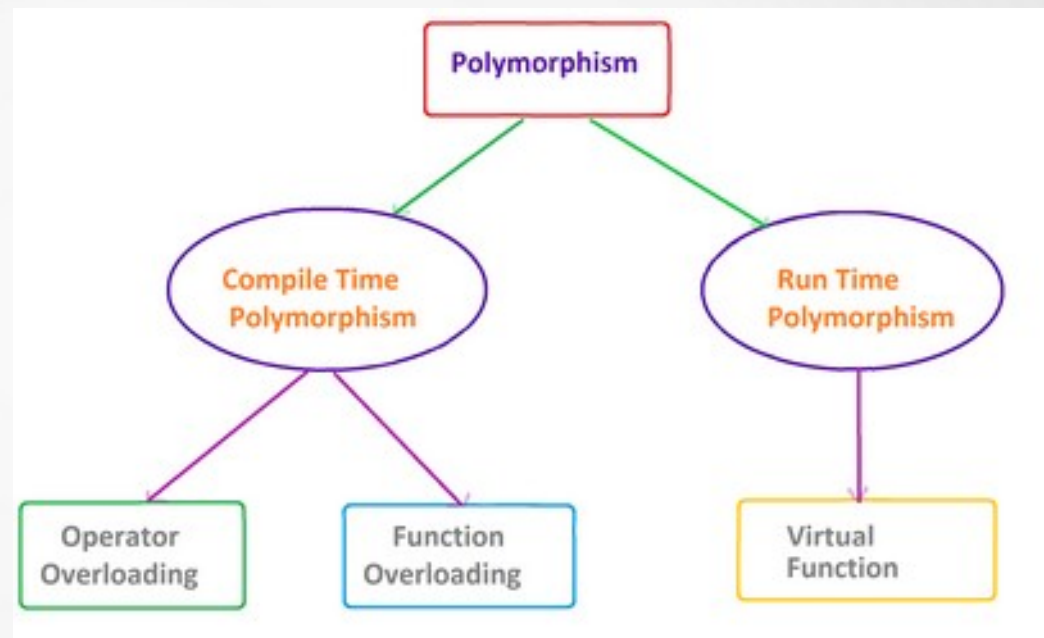
Polymorphism:-

• Ex:-

```
int fun(int a)
{
    return(sqrt(a));
}
```

```
int fun(int a,int b)
{
    return(a*b);
}
```

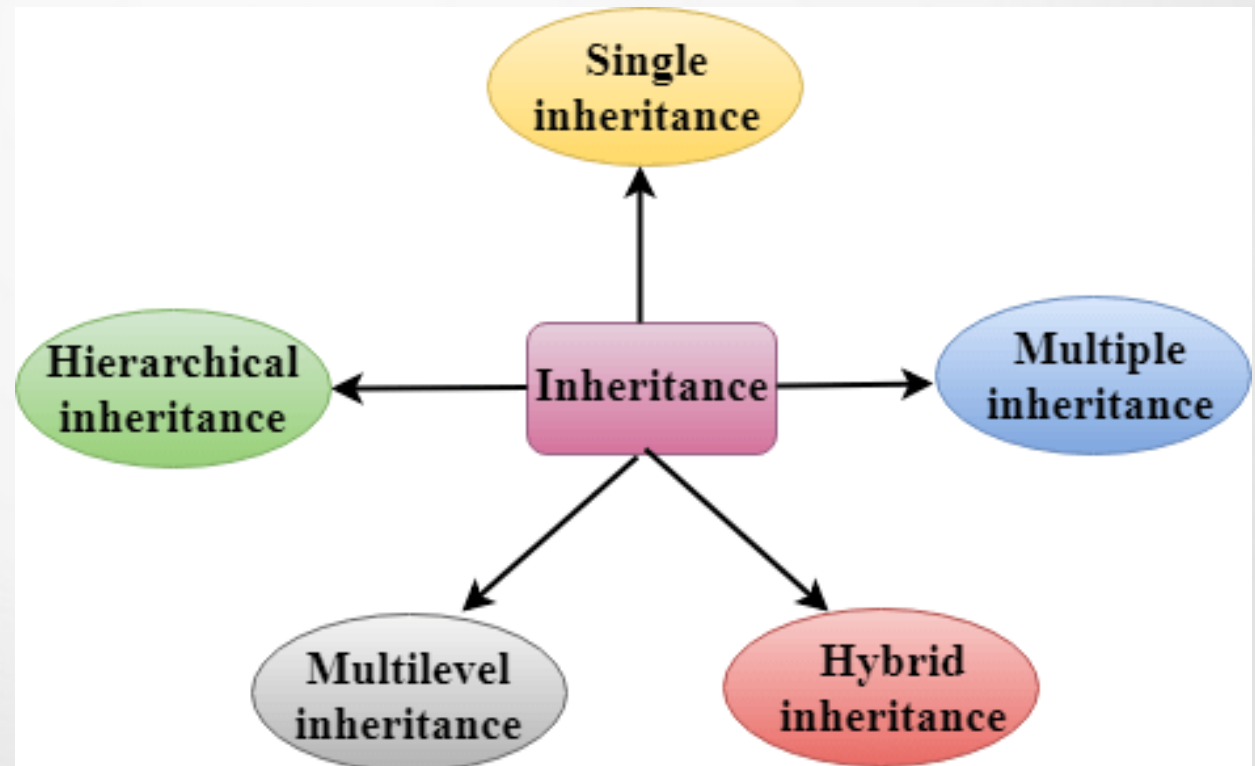
```
int fun(int a,int b,int c)
{
    int s=(a+b+c)/2;
    return s;
}
```



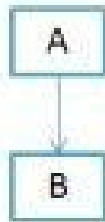
Types of Inheritance

• In C++, we have 5 different types of Inheritance. Namely,

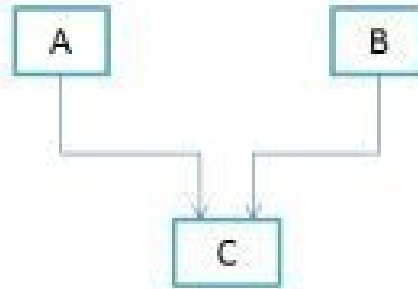
- 1) Single Inheritance
- 2) Multiple Inheritance
- 3) Hierarchical Inheritance
- 4) Multilevel Inheritance
- 5) Hybrid Inheritance



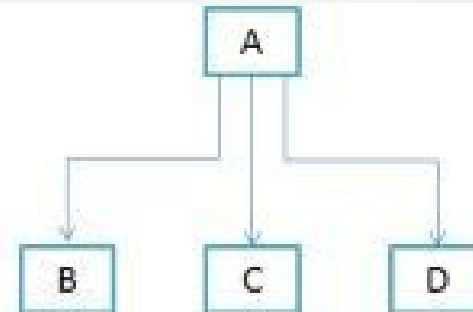
Types of Inheritance



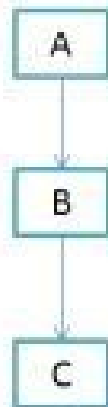
(a) Single Inheritance



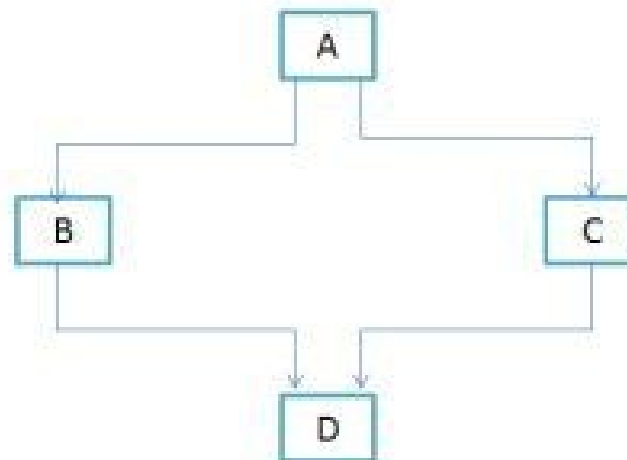
(b) Multiple Inheritance



(c) Hierarchical Inheritance



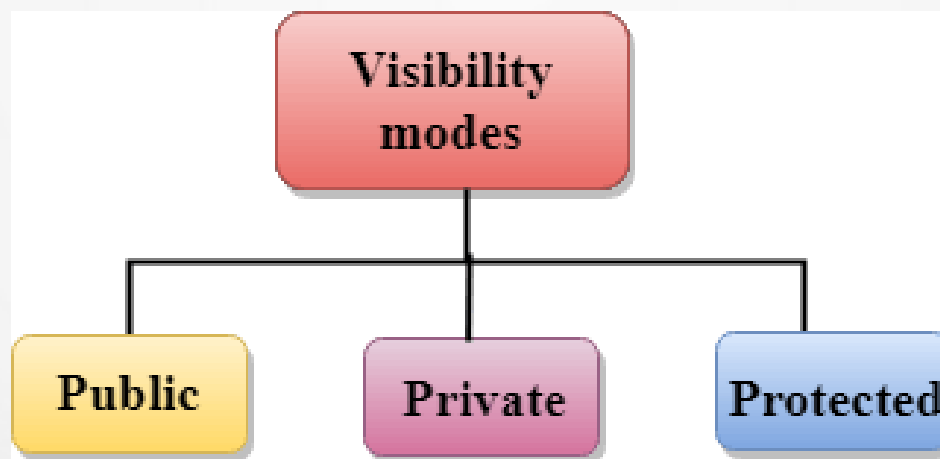
(d) Multilevel Inheritance



(e) Hybrid Inheritance

Access Control and Inheritance

- A derived class can access all the non-private members of its base class.
- Thus base-class members that should not be accessible to the member functions of derived classes should be declared private in the base class.



Access Control and Inheritance

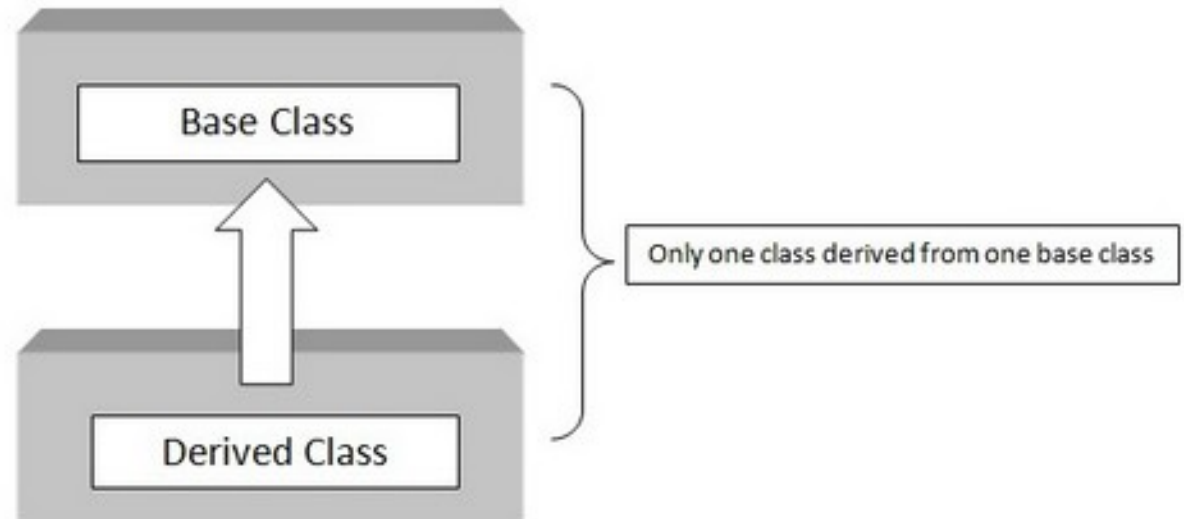
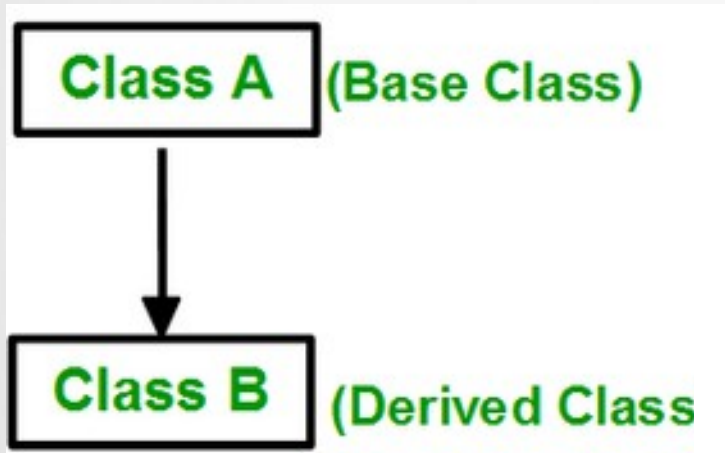
We can summarize the different access types according to who access them in the following way:

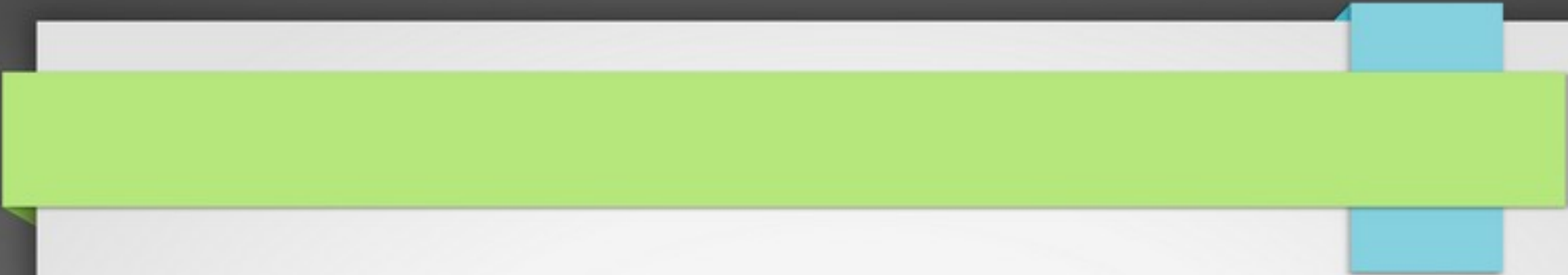
| Access | public | protected | private |
|-----------------|--------|-----------|---------|
| Same class | yes | yes | yes |
| Derived classes | yes | yes | no |
| Outside classes | yes | no | no |

Types of Inheritance

(1). Single Inheritance

- In this type of inheritance one derived class inherits from only one base class.
- It is the most simplest form of Inheritance.





```
class A // base class
```

```
{
```

```
.....
```

```
};
```

```
class B : access_specifier A // derived class
```

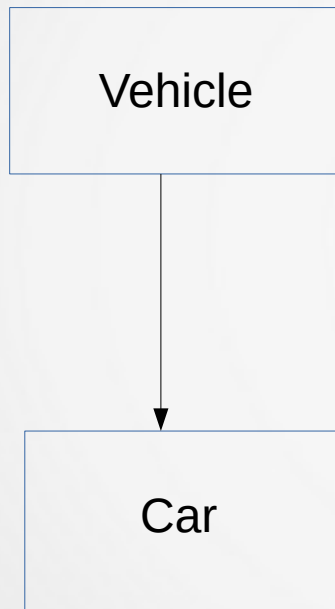
```
{
```

```
.....
```

```
};
```

Single Inheritance

Syntax:-



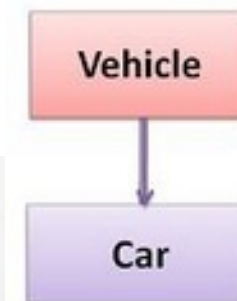
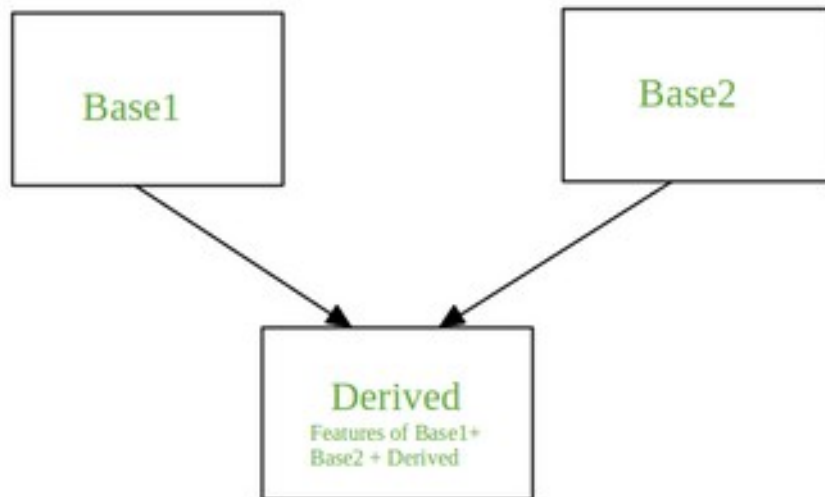
```
Class Vehicle  
{  
    .....  
    .....  
}
```

```
Class Car : public Vehicle  
{  
    .....  
    .....  
}
```

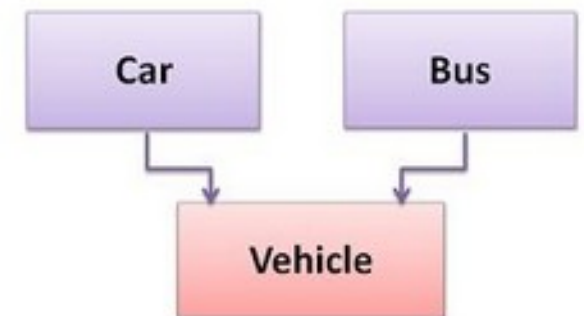
Types of Inheritance

(2). Multiple Inheritance

- In this type of inheritance a single derived class may inherit from two or more than two base classes.



Single Inheritance



Multiple Inheritance

Multiple Inheritance

SYNTAX :-

```
Class Area
{
    Area = width*length;
}
```

```
Class Perimeter
{
    Perimeter = 2 (width + length);
}
```

```
Class Rectangle : public Area, public Perimeter
{
    Int width;
    Int length;
}
```

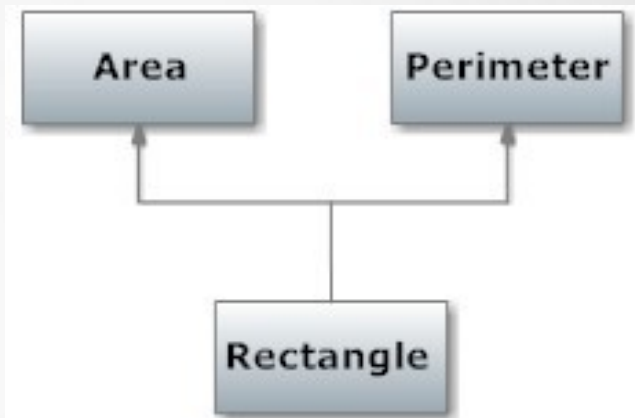
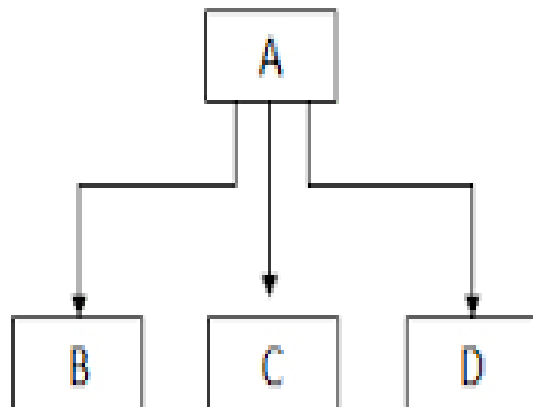


Figure: Multiple Inheritance Example

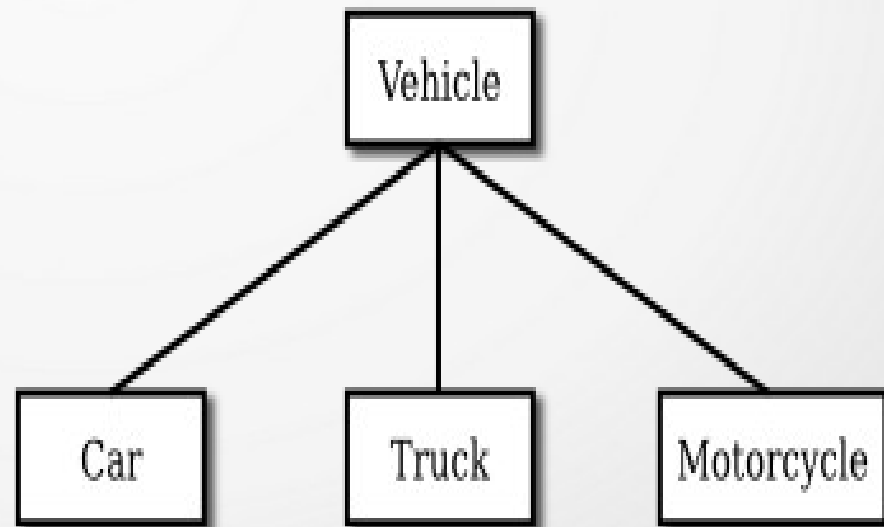
Types of Inheritance

(3). Hierarchical Inheritance

- In this type of inheritance, multiple derived classes inherit from a single base class.



Hierarchical Inheritance



Hierarchical Inheritance

SYNTAX :-

```
Class Student  
{  
    }  
}
```

```
Class Arts : public Student  
{  
    }  
}
```

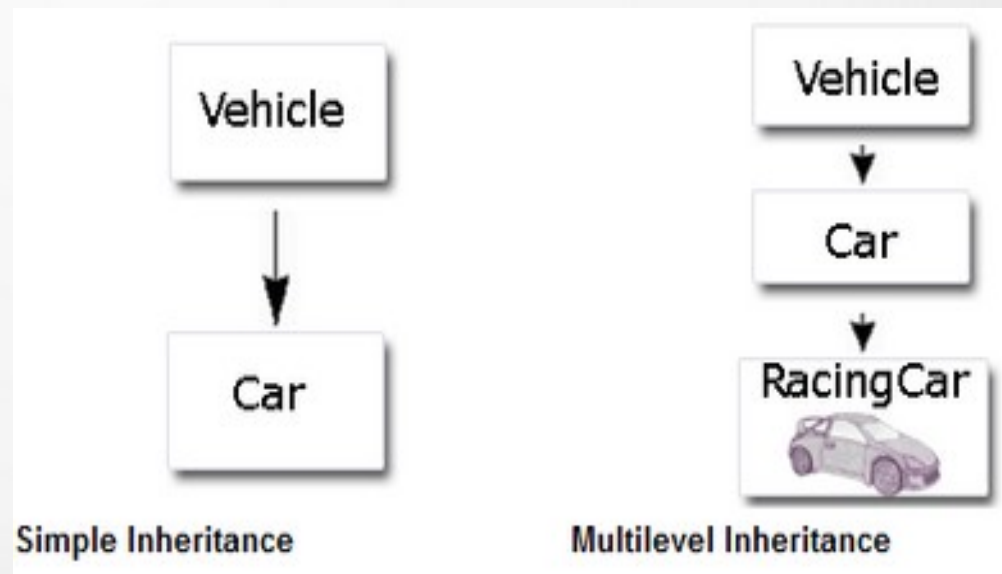
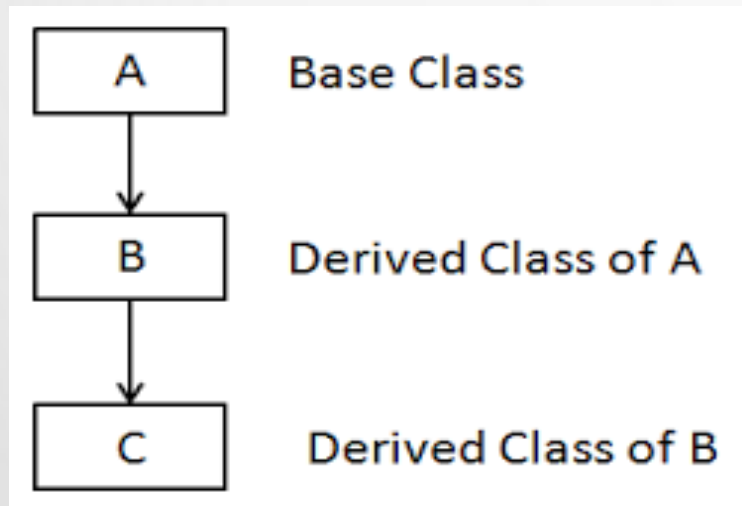
```
Class Science : public Student  
{  
    }  
}
```

```
Class Commerce : public Student  
{  
    }  
}
```

Types of Inheritance

(4). Multilevel Inheritance

- In this type of inheritance the derived class inherits from a class, which in turn inherits from some other class. The Super class for one, is sub class for the other.



Multilevel Inheritance

SYNTAX:-

```
Class Vehicle  
{
```

```
}
```

```
Class Car : public Vehicle  
{
```

```
}
```

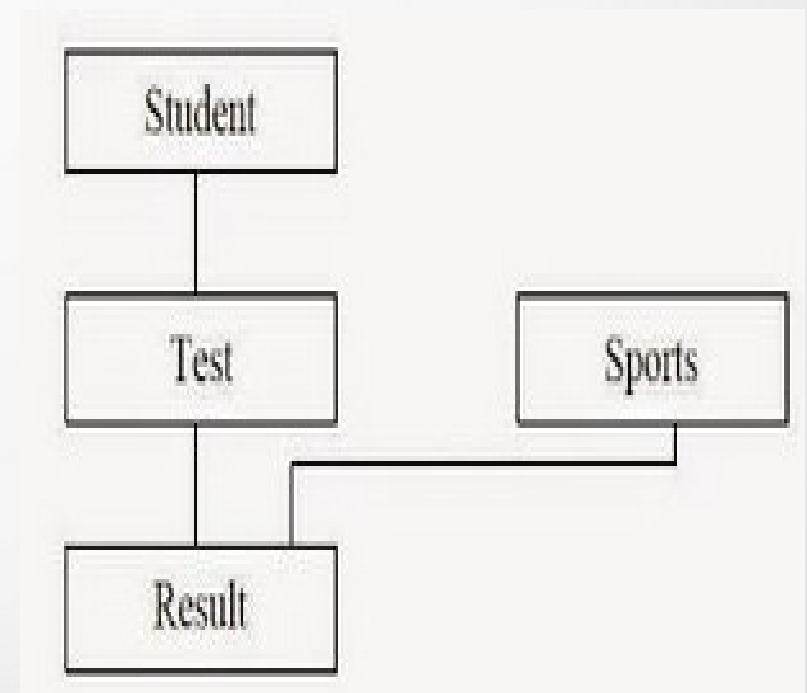
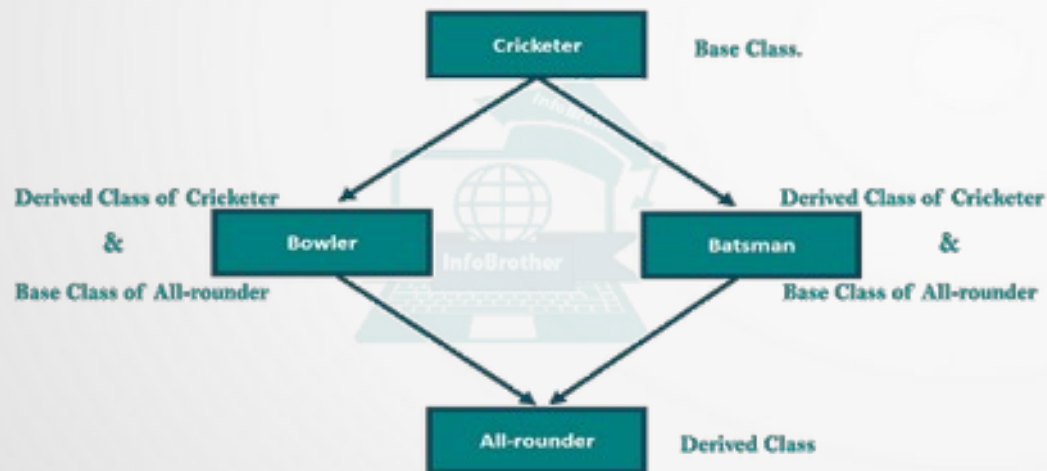
```
Class RacingCar : public Car  
{
```

```
}
```

Types of Inheritance

(5). Hybrid Inheritance

- Hybrid Inheritance is combination of more than one type of Inheritance.



Hybrid Inheritance

SYNTAX :-

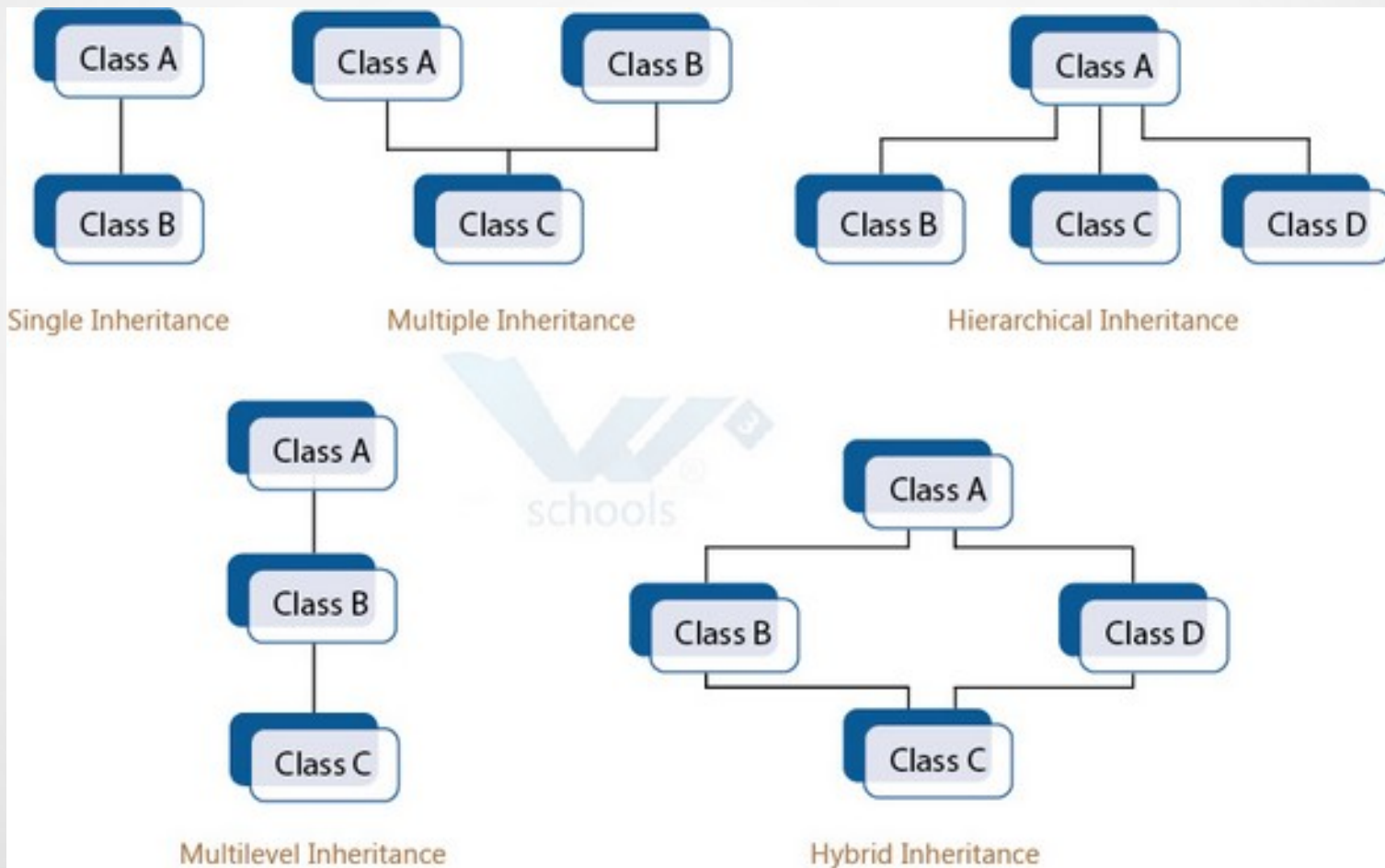
```
Class Student  
{  
    }  
}
```

```
Class Test : public Student  
{  
    }  
}
```

```
Class Sports  
{  
    }  
}
```

```
Class Result : public Test , public Sports  
{  
    }  
}
```

Types of Inheritance



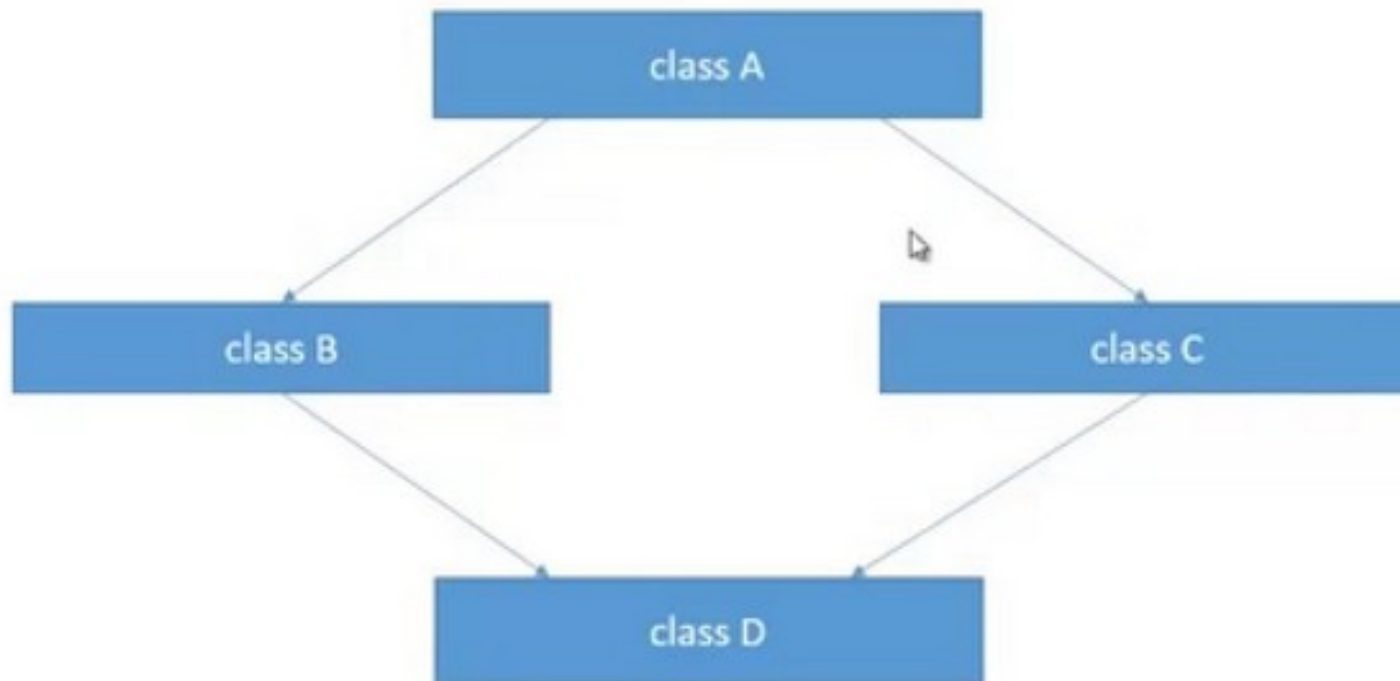
Question - Answer

1. What is Inheritance?
2. Which are the types of Inheritance?
3. What is Single Inheritance?
4. What is Multiple Inheritance?
5. Define Hierarchical Inheritance.
6. Give an example of Multi-level Inheritance.
7. What is Hybrid Inheritance?
8. Which are the different types of access specifiers?

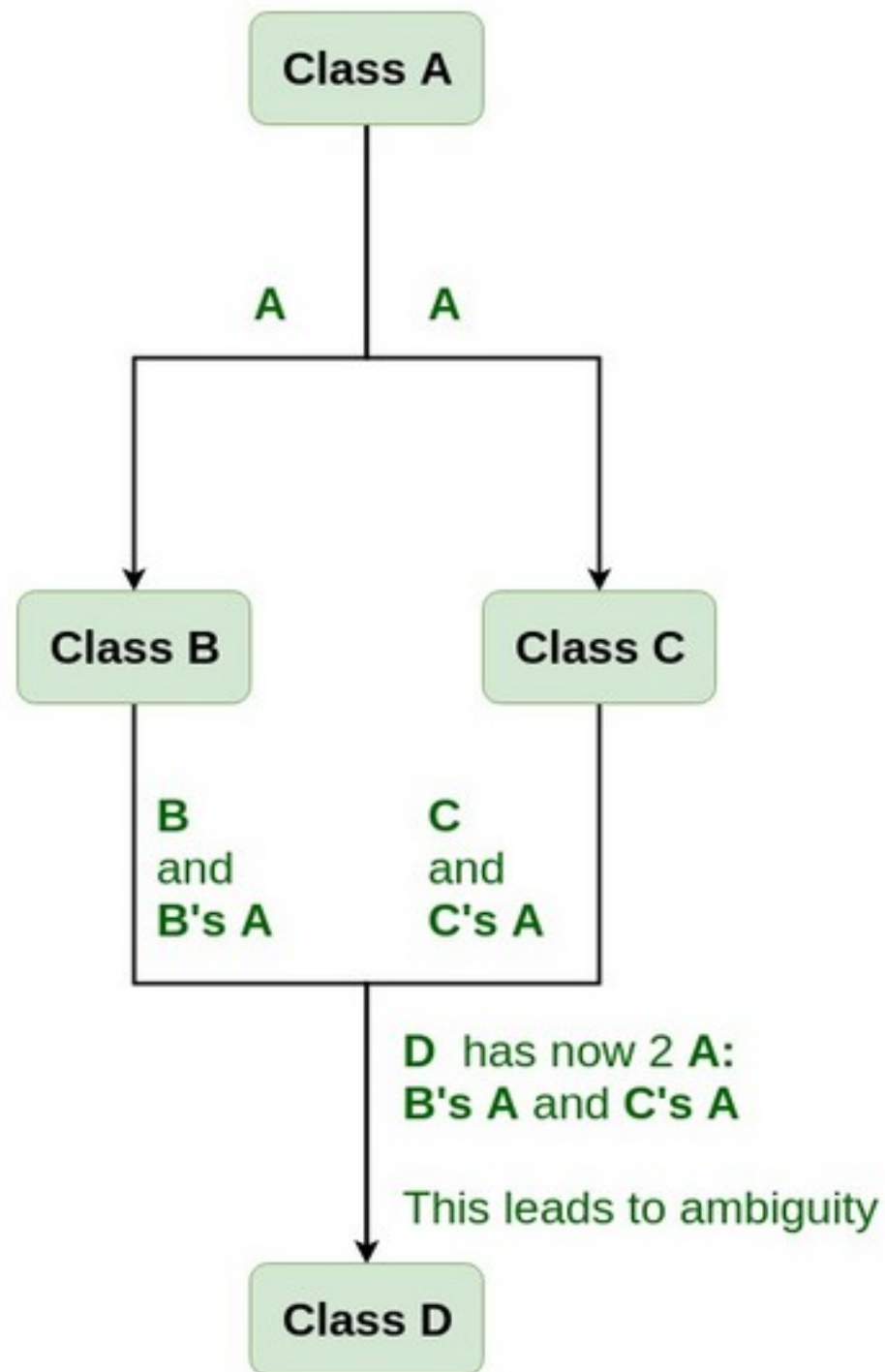
Virtual Base Class

Why we need Virtual Base Class ?

Virtual Base Class



Diamond Shaped Class Diagram



Virtual Base Class

- An **ambiguity** can arise when several paths exist to a class from the same base class.
- This means that a child class could have duplicate sets of members inherited from a single base class.
- C++ solves this issue by introducing a **virtual base class**.
- When two or more objects are derived from a common base class, we can prevent multiple copies of the base class being present in an object derived from those objects by declaring the base class as virtual when it is being inherited.
- Such a base class is known as **virtual base class**.

Virtual Base Class

- This can be achieved by preceding the base class' name with the word virtual.
- Virtual classes are primarily used during multiple inheritance.
- To avoid, multiple instances of the same class being taken to the same class which later causes ambiguity, virtual classes are used.

Virtual Base Class

```
class A {  
    public:  
    int a;  
    A(){  
        a = 10; }  
};  
class B : public virtual A {  
};  
class C : public virtual A {  
};  
class D : public B, public C {  
};  
int main(){  
    D object;  
    cout << "a = " << object.a << endl;  
    return 0; }
```

Pointers to Object

- ♦ Pointer can point to objects as well as to simple data types and arrays.
- ♦ Sometimes we don't know, at the time that we write the program, how many objects we want to create.
- ♦ When this is the case we can use new to create objects while the program is running.
- ♦ new operator returns a pointer to an unnamed object.

Pointers to Derived Classes

- Suppose **A** is the base class for the class **B**.
- Any pointer to A type(base class) can be assigned the address of an object of the class B(derived class).
- A base class pointer can point to derived class object but the reverse is not true.

Pointers to Derived Classes

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

```
class B:public A
{public:
void show()
{ cout<<endl<<"B class..";
}};
```

```
int main()
{
A a1;
B b1;
A *ap;

ap = &b1; //base class pointer is pointing to
//derived class object
ap->show(); //class B
ap = &a1;
ap->show(); //class A
return 0; }
```

Virtual Function

- Main objective of polymorphism is the ability to refer to objects without any regard to their classes.
- Achieve run-time polymorphism
- Base class “virtual”
- When a function is virtual, the decision as to which function to be invoked is done at the run-time based on the type of the object pointed to by the pointer, rather than the type of the pointer.

virtual <return_type> function_name (parameters);

or

<return_type> virtual function_name (parameters);

Virtual Function - Rules

- Virtual functions **cannot be static** and also **cannot be a friend** function of another class.
- Virtual functions should be **accessed using pointer or reference** of base class type to achieve run time polymorphism.
- The **prototype** of virtual functions **should be same** in base as well as derived class.
- They are always **defined in base class** and overridden in derived class.
- It is **not mandatory for derived class to override** (or re-define the virtual function), in that case base class version of function is used.
- A class may have **virtual destructor** but it **cannot have a virtual constructor**.

```
class base {
public:
    void fun_1() { cout << "base-1\n"; }
    virtual void fun_2() { cout << "base-2\n"; }
    virtual void fun_3() { cout << "base-3\n"; }
    virtual void fun_4() { cout << "base-4\n"; }
};
```

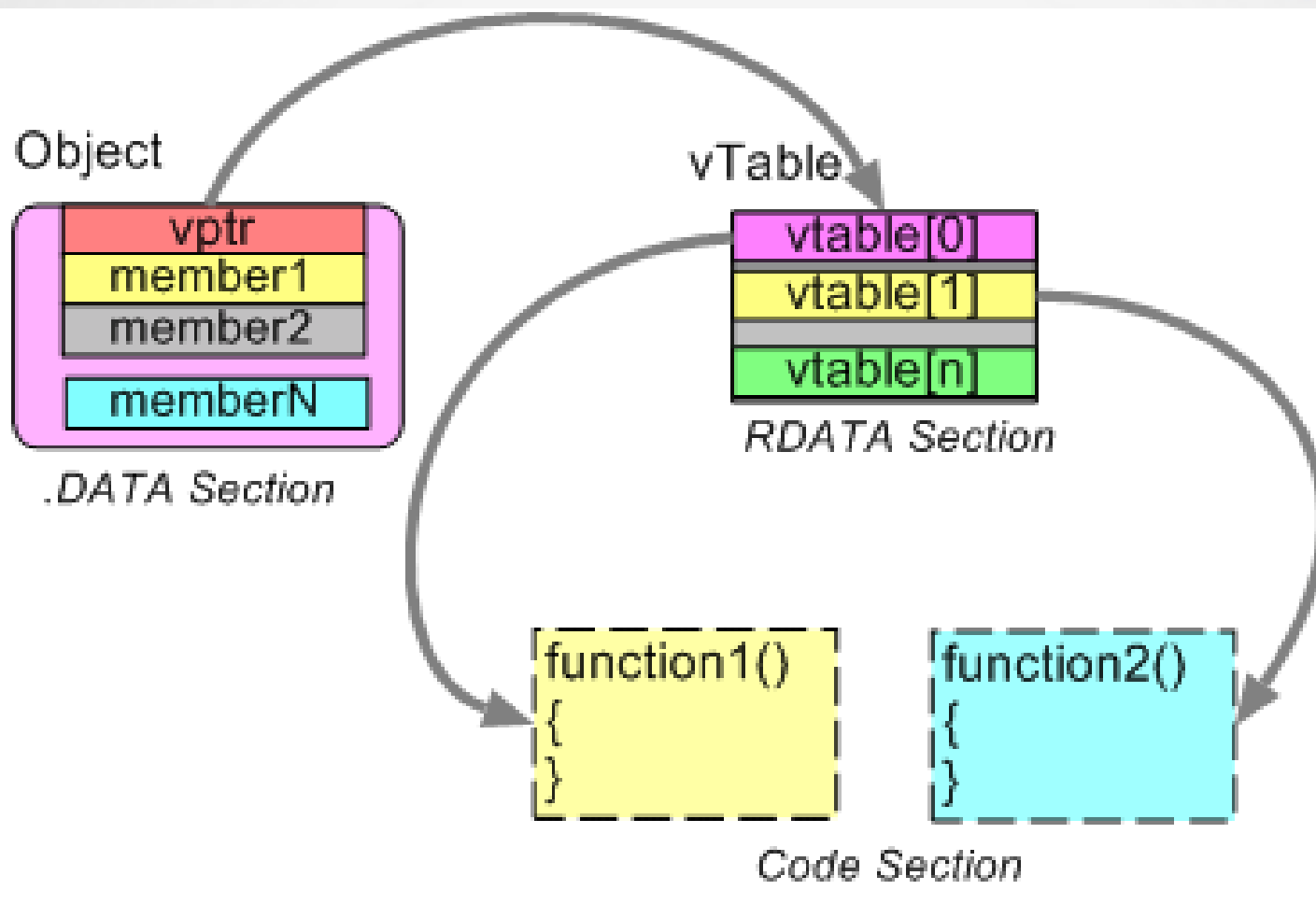
```
class derived : public base {
public:
    void fun_1() { cout << "derived-1\n"; }
    void fun_2() { cout << "derived-2\n"; }
    void fun_4(int x) { cout << "derived-4\n"; }
};
```

```
int main()
{
    base* p;
    derived obj1;
    p = &obj1;

    p->fun_1();
    p->fun_2();
    p->fun_3();
    p->fun_4();
}
```

Working on Virtual Functions

- How does the compiler perform runtime resolution?
- The compiler maintains two things to serve this purpose:
 - vtable: A table of function pointers, maintained per class.
 - vptr: A pointer to vtable, maintained per object instance



Virtual Table

- To implement virtual functions, C++ uses a special form of **late binding** known as the virtual table.
- The virtual table is a lookup table of functions used to resolve function calls in a dynamic/late binding manner.
- The virtual table sometimes goes by other names, such as “**vtable**”, “**virtual function table**”, “**virtual method table**”, or “**dispatch table**”.
- This table is simply a **static array** that the compiler sets up at compile time.
- A virtual table contains **one entry for each virtual function** that can be called by objects of the class.
- Each entry in this table is simply **a function pointer that points to the most-derived function accessible by that class**.

Virtual Pointer

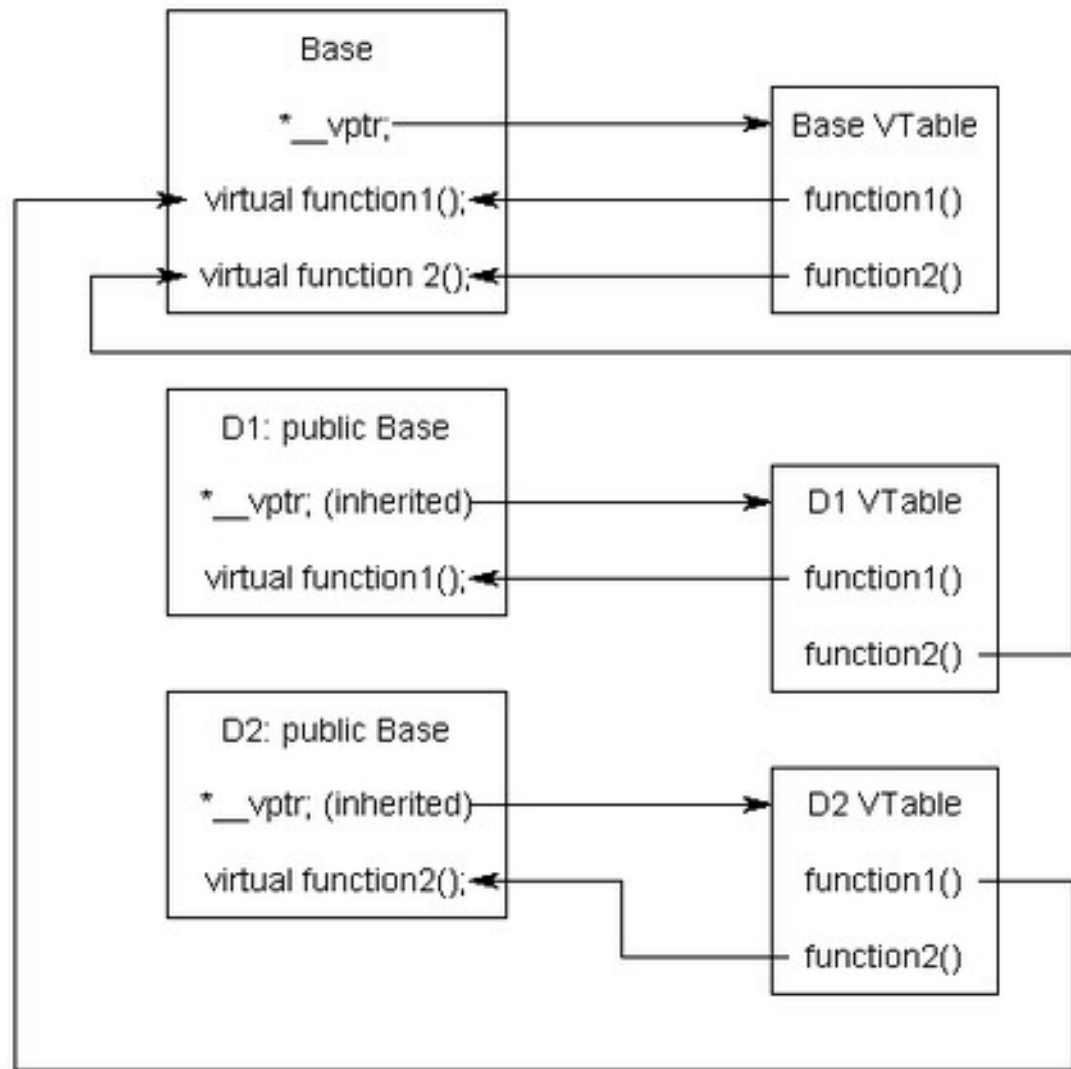
- The compiler also adds a hidden pointer to the base class, which we will call `*__vptr`.
- `*__vptr` is set (automatically) when a class instance is created so that it points to the virtual table for that class.
- Consequently, it makes each class object allocated bigger by the size of one pointer.
- It also means that `*__vptr` is inherited by derived classes

Working on Virtual Functions

```
class Base
{
public:
    virtual void function1() {};
    virtual void function2() {};
};
```

```
class D1: public Base
{
public:
    virtual void function1() {};
};
```

```
class D2: public Base
{
public:
    virtual void function2() {};
};
```



Virtual Function

```
class A
{
public:
virtual void show()
{ cout<<endl<<"A class.."; }
};
```

```
class B:public A
{
public:
void show()
{ cout<<endl<<"B class.."; }
};
```

```
int main()
{
A a1;
B b1;
A *ap;
ap = &b1;
ap->show(); //B class...
ap = &a1;
ap->show();//A class...
return 0; }
```

Pure Virtual Functions and Abstract Class

- A **pure virtual function** or **pure virtual method** is a virtual function that is **required to be implemented by a derived class** if the derived class is not abstract.
- Classes containing pure virtual methods are termed **"abstract"** and they **cannot be instantiated** directly.
- A pure virtual function is a function that has the **notation "= 0"** in the declaration of that function.
- The notation "= 0" is just there to indicate that the virtual function is a pure virtual function, and that the **function has no body or definition.**
- The pure virtual functions are also called **"do-nothing" functions.**

Ex:-

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

Pure Virtual Functions and Abstract Class

```
class Base
```

```
{
```

```
    int x;
```

```
public:
```

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

```
    int getX() { return x; }
```

```
};
```

```
class Derived: public Base
```

```
{
```

```
    int y;
```

```
public:
```

```
    void fun() { cout << "fun() called"; }
```

```
};
```

```
int main(void)
```

```
{
```

```
    Derived d;
```

```
    d.fun();
```

```
    return 0; }
```

Some facts of Pure Virtual Function

1) A class is abstract if it has at least one pure virtual function.

using namespace std;

```
class Test
{
    int x;
public:
    virtual void show() = 0;
    int getX() { return x; }
};
```

```
int main(void)
{
    Test t;
    return 0;
}
```

Some facts of Pure Virtual Function

2) We can have pointers and references of abstract class type.

using namespace std;

```
class Base
{
public:
    virtual void show() = 0;
};

class Derived: public Base
{
public:
    void show() { cout << "In Derived \n"; }
};

int main(void)
{
    Base *bp = new Derived();
    bp->show();
    return 0;
}
```

Some facts of Pure Virtual Function

3) If we do not override the pure virtual function in derived class, then derived class also becomes abstract class.

```
#include<iostream>
using namespace std;
class Base
{
public:
    virtual void show() = 0;
};

class Derived : public Base { };

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


Some facts of Pure Virtual Function

4)An abstract class can have constructors.

```
class Base
```

```
{
```

```
protected:
```

```
    int x;
```

```
public:
```

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

```
    Base(int i) { x = i; }
```

```
};
```

```
class Derived: public Base
```

```
{
```

```
    int y;
```

```
public:
```

```
    Derived(int i, int j):Base(i) { y = j; }
```

```
    void fun() { cout << "x = " << x << ", y = " << y; }
```

```
};
```

```
int main(void)
```

```
{
```

```
    Derived d(4, 5);
```

```
    d.fun();
```

```
    return 0; }
```

Constructors and Destructors in Derived Classes

- Constructor and destructor for a class are special member functions for the class.
- Constructor gets invoked implicitly and helps in creating objects(i.e allocating the required amount of memory for the member data and setting values for them).
- Destructor for a class is called implicitly when an object of the class goes out of scope and destroys the object thereby releasing the block of memory occupied by the object.
- When an object of derived class is declared, constructor of the base class gets executed first and then the constructor of the derived class gets executed.
- But, the order execution of destructor of base class and derived class is reverse

Constructors and Destructor in Multiple Inheritance

Constructors in Multiple Inheritance

- Base class constructors are always called in the derived class constructors.
- Whenever you create derived class object, first the base class default constructor is executed and then the derived class's constructor finishes execution.
- **Points to Remember**
 - Whether derived class's default constructor is called or parameterised is called, base class's default constructor is always called inside them.
 - To call base class's parameterised constructor inside derived class's parameterised constructor, we must mention it explicitly while declaring derived class's parameterized constructor.

Constructors and Destructor in Multiple Inheritance

Destructor in Multiple Inheritance

- If you create an instance for the derived class then base class constructor will also be invoked and when derived instance is destroyed then base destructor will also be invoked and the order of execution of constructors will be in the same order as their derivation and order of execution of destructors will be in reverse order of their derivation.

Points to Remember:

- Constructors from all base class are invoked first and then the derived class constructor is called.
- Order of constructor invocation depends on the order of how the base is inherited.
- 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.

Virtual Destructor

- Suppose we dynamically allocate space for derived type object and assign it to the base pointer and after using the object if deallocate the space by the use of delete operator, the destructor in the base class only gets executed.
- The normal rule of the order of execution of destructors can be reinforced with what is known as **virtual destructor**.
- The destructor in the base class is made virtual by preceding its header with the keyword virtual.

Access Specifier

| | Derived Class | | |
|------------|---------------|---------------|---------------|
| Base Class | Private | Protected | Public |
| Private | Not inherited | Not inherited | Not inherited |
| Protected | Private | Protected | Protected |
| Public | Private | Protected | Public |

Private Inheritance

- When deriving from a **private** base class, **public** and **protected** members of the base class become **private** members of the derived class.
- When a class **B** is derived from the class **A** publicly(i.e., with the line `class B : class A` in the definition of **B**), here, the protected and the public members of the class **A** are accessible in the class **B** and also an object of derived class **B** can access all the public members of the base class.
- `class B : private A`
`{`
`};`
- Here, the class **B** is said to be privately derived from the class **A**. All the protected and public members of the class **A** are accessible in the derived class **B**. But an object of the class **B** can not access even the public members of the base class **A** also.
- The protected and public members of the base class will become the private members of the derived class.

Private Inheritance

- The functionality of the base class **A** is hidden in the derived class **B**.
- But there is a way out for making only some public member functions of the base class accessible to objects of derived class, i.e., by using the base class name followed by `::` followed by the function name in the public section of the derived class.

Protected Inheritance

- When deriving from a **protected** base class, **public** and **protected** members of the base class become **protected** members of the derived class.

- Ex:-**

```
class B : protected A
{
};
```

- Here, the class **B** is said to be protectedly derived from the class **A**. All the protected and public members of the base class **A** would now become protected members of the derived class **B** and hence they can be accessed within the class **B** and in any other class which is derived from **B**.
- No members of the class **A** can be accessed by an object of class **B**.

Containership

- The containership is another way of reusing existing code.
- In this case objects of one class are made the members of another class.
- Suppose **A** is a class and **a** is an object of the class. In another class, say **B**, the object **a** can be made a member.
- As a result of this, the code in the class **A** is reused in the class **B** as well. Since the class **B** contains an instance of class **A**, it is said to exhibit “**Has a Relationship**”.
- It is also said that the class **A** has delegated some responsibility to the object **b** and, thus, exhibits delegation phenomenon.