Programming Paradigm

Note : These slides are not study material, rather just a guide to study these topics. Students are suggested to go through books and refer class notes to understand these topics in detail.

Inheritance

Inheritance is used to set-up a class hierarchy. It is a method of reuse, but this is not it's main purpose.

When do we use inheritance ?

A derived class should pass the litmus test of "**is a**"

- 2 wheeler "is an" automobile. OK
- 4 wheeler "is an" automobile. OK
- Car "is a" 4 wheeler. OK
- Steering wheel "is a" Car. Not OK
- Event "is a" Date. Not OK

The last two are example of **Composition** or "**has a**" hierarchy, hence using Inheritance mechanism here is a misuse

- Car "has a" steering wheel. OK
- Event "has a" date. OK

Inheritance

Example

```
class Person {
    private :
        char * name;
        int age;
    public :
        Person(); //default
        Person(char *, int);
        Person(const Person &);
        Person & operator=
                (const Person &);
        ~Person();
        void Read();
        void Write() const;
}
```

```
class Employee : public Person {
    private :
        int empId;
        Int salary
    public :
        Employee(...);
        Employee(const Employee &);
        Employee & operator=
                (const Employee &);
        ~Employee();
        void ReadEmp();
        void WriteEmp() const;
}
```

Inheritance

Example

Person class is called the base class and Employee class is called the derived class

Employee class inherits all the methods of base class Person except the Constructor, Destructors, Copy constructor and Assignment Operator

Any of the inherited functions can be directly invoked using an object of class Employee

Constructor & Destructor

When an object of the derived class is created, the control is transferred to the constructor of derived class

The base class sub-objects are initialized in the initialization list by invoking the constructor of the base class

If the base class constructor is not explicitly invoked in the initialization list, then the default constructor of the base class is invoked

After the base class sub-object are initialized by executing the constructors of the corresponding base classes, the data members of the derived class are initialized

Destructor are executed in the reverse order of constructor. The derived class destructors will be executed before that of base class destructor

Constructor & Destructor

Example

```
class CBase {
    public :
        CBase() { cout << "In base class
   constructor" << endl; }
        ~CBase() { cout << "In base class
   destructor" << endl; }
};


class CDerived : public CBase {
    public :
        CDerived() { cout << "In derived class
   constructor" << endl; }
        ~CDerived() { cout << "In derived class
   destructor" << endl; }
};
```

```
int main() {
    CDerived d;
}
```

Output :

In base class constructor

In derived class constructor

In derived class destructor

In base class destructor

Constructor & Destructor

Example II

```
class CBase {
        int a;
    public :    ...
        CBase(int z) : a(z) { }
        void Display() { cout << "a : " << a <<
   endl; }
};


class CDerived : public CBase {
        int b;
    public :    ...
        CDerived(int x, int y) : b(y), CBase(x) { }
        void Display() {
            CBase::Display();
            cout << "b : " << b << endl;
        }
};
```

```
int main() {
    CDerived d(1,2);
    d.Display();
}
```

Copy Constructor

Example

```
class CBase {
        int a;
    public :    ...
        CBase(int z) : a(z) {}
        CBase(const CBase & rhs): a(rhs.a){}
};
class CDerived : public CBase {
        int b;
    public :    ...
        CDerived(int x, int y) : b(y), CBase(x) {}
        CDerived(const CDerived & rhs):b(rhs.b),
   Cbase(rhs)      {}
        // note how the base class constructor is
   invoked - derived class object to a base class
   reference
};
```

```
int main() {
    CDerived
   d1(1,2);
    d1.Display();
    CDerived
   d2(d1);
    d2.Display();
}
```

Assignment Operator

Example

Do by your own

Overloading member functions from base and derived classes

```
class CBase
{    ...
    public :     ...
        void fn() { cout << "In
base class" << endl; }

};

class CDerived : public CBase
{    ...
    public :     ...
        void fn() { cout << "In
derived class" << endl; }

};
```

Note : Base class pointer or reference can point or refer to a derived classs object but not vice versa

```
int main() {
    CBase oB, *pB = NULL;
    CDerived oD, *pD = NULL;

oB.fn(); // calls fn of Base class

    oD.fn(); // calls fn of Derived
class
    oD.CBase::fn(); // calls fn of
Base class

    // pD = &oB; // will not compile
    // pD->fn();


    pB = &oD;
    pB->fn(); // calls fn of the
base class
}
```

Understanding

Base class pointer or reference can point or refer to a derived class object

Derived class pointer can not automatically be made to point to the base class object

- Generally derived class may have additional members compared to the base class. If the derived class pointer were allowed to point to the base class object, and if it tries to access the additional members using this pointer, the compiler will not be able to give an error.
  - If this is allowed, this may cause **dangling reference** at run time.

Function Overriding or Dynamic Polymorphism

Even though base class pointer can point to an object of the derived class, the base class function is invoked as the pointer by type is a pointer to the base class.

- This happens as function calls are generally resolved at compile time

Many a times, it is desirable to postpone the resolution of the call until run times and the function should be invoked based on the object to which the pointer points to rather than the type of the pointer.

- This is achieved by declaring the function to be **virtual**

This concept, where the function calls are resolved at run time based on the object to which they point, is called **Dynamic Polymorphism**.

- In C++, it is implemented using **virtual function**

Function Overriding

Example

```
class CBase
{   ...
    public :    ...
        virtual void fn() {
        cout << "In base class" <<
endl; }

};

class CDerived : public CBase
{   ...
    public :    ...
        void fn() {
        cout << "In derived class"
<< endl; }

};
```

```
int main() {
    CBase oB, *pB = NULL;
    CDerived oD, *pD = NULL;

    oB.fn(); // calls fn of Base
class

    oD.fn(); // calls fn of Derived
class

    pB = &oD;
    pB->fn(); // calls fn of the
derived class
}
```

Function Overriding

Few points

Few points :

- Should be member function of the class
- Only in inheritance
- Function should be virtual
- Signature of the functions should be exactly same
- Can extends the functionality of the base class function
- *Extra overhead at run time of a per-class table and a per-object pointer*
- *Extra overhead at run time of a de-referencing of pointer*

Last 2 points are discussed in the next section

Virtual Table / VTBL

Virtual Table is a lookup table of function pointers used to dynamically bind the virtual functions to objects at runtime.

Every class that uses virtual functions (or is derived from a class that uses virtual functions) is given it's own virtual table as a secret data member.

It is not intended to be used directly by the program, and as such there is no standardized way to access it. This table is set up by the compiler at compile time.

A virtual table contains one entry as a function pointer for each virtual function that can be called by objects of the class.

Virtual Pointer / VPTR

This vtable pointer or `_vptr`, is a hidden pointer added by the Compiler to the base class. And this pointer is pointing to the virtual table of that particular class.

This `_vptr` is inherited to all the derived classes.

Each object of a class with virtual functions transparently stores this `_vptr`.

Call to a virtual function by an object is resolved by following this hidden `_vptr`.

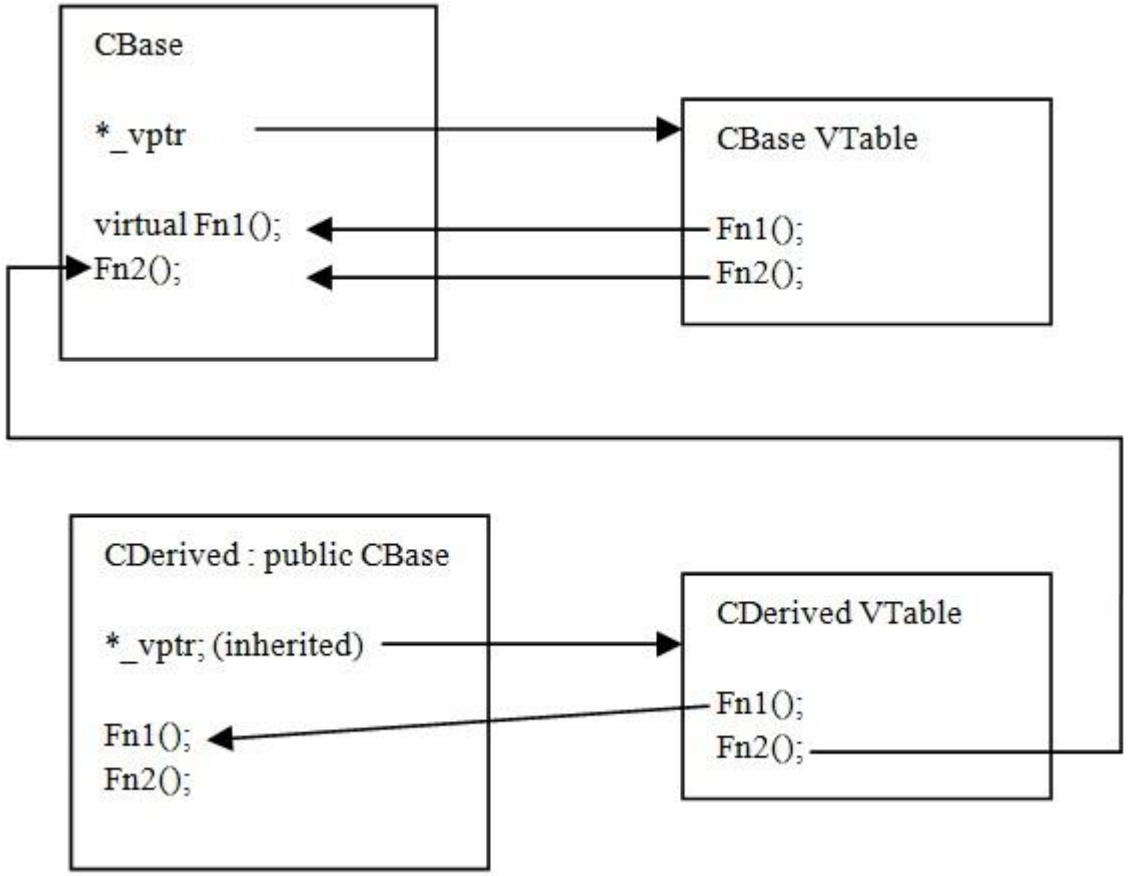
Understanding VTBL and VPTR

```
class CBase
{    ...
    public :    ...
        virtual void fn1() { cout
<< "Fn1 in base class" << endl; }

        void fn2() { cout << "Fn2
in base class" << endl; }
};

class CDerived : public CBase
{    ...
    public :    ...
        void fn1() { cout << "Fn1
in derived class" << endl; }

        void fn2() { cout << "Fn2
in derived class" << endl; }
};
```

```
int main() {
    CBase oB, *pB = NULL;
    CDerived oD;

    pB = &oD;
    pB->fn1(); // calls fn1 of the
derived class

    pB->fn2(); // calls fn2 of the
base class - as fn2 in not a virtual
function in base class

}
```

Understanding VTBL and VPTR

Contd...



Pure Virtual Functions and Abstract Base Class

If the base class provides only the interface of the methods, without implementation, then the **virtual functions** of the base class are made **pure virtual** – no definition is provided for the functions and the pointer to the function is grounded.

Such a Class is called an **Abstract Class**.

- Abstract class can not be instantiated
- A derived class of a Abstract class becomes a **concrete** class only if all the pure virtual functions of the base class are overridden by this class or any of its ancestors

A Class with only pure virtual function provides interface only

- Each of the derived classes will inherit the signature of the methods and override them in their classes

Virtual functions supporting dynamic dispatch provides flexibility to the programmer, the client code becomes more robust and does not break down when new classes are derived.

Pure Virtual Functions and Abstract Base Class

Example

```
class Shape {
    public :
        virtual void Read() = 0;
        virtual double Area() = 0;
};

class Rectangle : public Shape {
        double length, breadth;
    public :
        void Read(){
            cin >> length >>
breadth;
        }
        double Area() {
            return length*breadth;
        }
};
```

```
class Triangle : public Shape {
        double base, height;
    public :
        void Read(){
            cin >> base >> height;
        }
        double Area() {
            return 0.5*base*height;
        }
};
int main() {
    Double totalArea = 0;
    Shape *p[2];
    p[0] = new Rectangle();
    p[1] = new Triangle();
    for(int i = 0; i < 2; i++)
p[i]->Read();

    for(int i = 0; i < 2; i++)
        totalArea += p[i]->Area();
}
```

Virtual Destructor

Whenever the constructor of the derived class is non-trivial, the base class destructor should be **virtual** to force the execution of the derived class destructor before that of the base class

Virtual Destructors

Example

```
class CBase {
    public :
        CBase() {
            Cout << "Constructor of
the base class" << endl;
        }

        ~CBase() {
            Cout << "Destructor of
the base class" << endl;
        } // Incorrect way
};


int main() {
    CBase * ptr = new CDerived();
    Delete ptr;
}
```

```
class CDerived : public CBase {
    int * p;
    public :
        CDerived () {
            cout << "Constructor of
the derived class" << endl;
            p = new int;
        }
        ~CDerived() {
            cout << "Destructor of
the derived class" << endl;
            delete p;
        }
}
```

Outputs :

```
Constructor of the base class
Constructor of the derived class
Destructor of the base class
```

Virtual Destructors

Example contd

```
class CBase {
    public :
        CBase() {
            Cout << "Constructor of
the base class" << endl;
        }

        virtual ~CBase() {
            Cout << "Destructor of
the base class" << endl;
        }
};


int main() {
    CBase * ptr = new CDerived();
    Delete ptr;
}
```

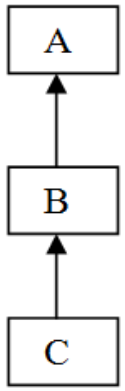
```
class CDerived : public CBase {
    int * p;
    public :
        CDerived () {
            cout << "Constructor of
the derived class" << endl;
            p = new int;
        }
        ~CDerived() {
            cout << "Destructor of
the derived class" << endl;
            delete p;
        }
}
```

Outputs :

Constructor of the base class
Constructor of the derived class
Destructor of the derived class
Destructor of the base class

Multi-level Inheritance

Example




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

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

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

}
```

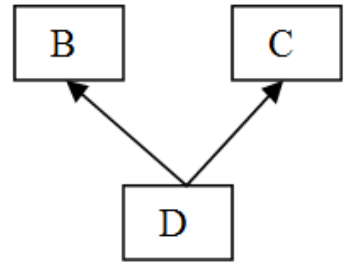
```
int main() {
    C oC;
    oC.fn(); // fn of class C
    oC.B::fn(); // fn of class B
    // oC.A::fn(); // ERROR
}
```

Class A is not direct base of Class C, hence function of same name of class A can not be called.

But other functions can be called.

Multiple Inheritance

Examp




```
class B {
    public :
        void fn() { cout << "In B"
<< endl; }
}

class C {
    public :
        void fn() { cout << "In C"
<< endl; }
}

class D : public B, public C {
    public :
}
```

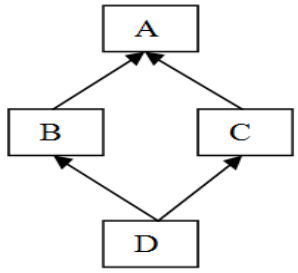
```
int main() {
    D oD;
    oD.fn(); // ERROR
    oD.B::fn(); // fn of class B
    oD.C::fn(); // fn of class C
}
```

When a class inherits from more than one class, then it is called multiple inheritance.

It can lead to ambiguity as the base classes may have the members with the same name.

In that case reference to functon can be resolved upon the class name

Hybrid Inheritance




```
class A {
        int a;
    public :
        A(int w) : a(w) {}
}

class B : public A {
        int b;
    public :
        B(int w, int x) : A(w),
b(x) {}
}

class C : public A {
        int c;
    public :
        C(int w, int y) : A(w),
c(y) {}
}
```

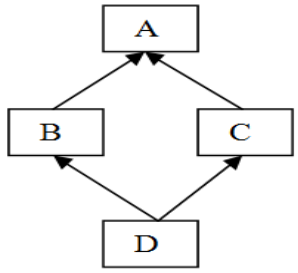
```
class D : public B, public C{
        int d;
    public :
        D(int w, int x, int y, int
z) : B(w, x), C(w, y), d(z) {}
}
```

Note : An object of classe D will have following structure

Base class subobject of class B Also has base class sub object of class A	a b
Base class subobject of class C Also has base class sub object of class A	a c
Members of clas D	d

Observe that the base class subobject of class A appears twice in object of class D

HI & Virtual Base Class




```
class A {
        int a;
    public :
        A(int w) : a(w) {}
}

class B : virtual public A {
        int b;
    public :
        B(int w, int x) : A(w),
b(x) {}
}

class C : public virtual A {
        int c;
    public :
        C(int w, int y) : A(w),
c(y) {}
}
```

```
class D : public B, public C{
        int d;
    public :
        D(int w, int x, int y, int
z) : B(w, x), C(w, y), A(w), d(z) {}
}
```

Note : An object of classe D will have following structure

Base class subobject of class A	a
Base class subobject of class B With a referenct to base class sub object of class A	&a b
Base class subobject of class C With a referenct to base class sub object of class A	&a c
Members of clas D	d

Observe that here Class D will have to explicitely invoke the constructor of class A. But class B and class C will **not** invoke the constructor of class A