Programming Paradigm

Inheritance and Polymorphism using C++

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

Protected Access Specifier

Protected : can be access from member function of the same class or any of it's publicly derived class – can not be access by the client function.

Private members would always be hard-and-fast **private**, but in real projects there are times when you want to make something hidden from the world at large and yet allow access for members of derived classes. The **protected** keyword is used for this reason.

"This is **private** as far as the class user is concerned, but available to anyone who inherits from this class."

Protected example

The best approach is to leave the data members **private** – you should always preserve your right to change the underlying implementation. You can then allow controlled access to inheritors of your class through **protected** member functions:

```
class Base {
  int i;
protected:
  int read() const { return i; }
  void set(int ii) { i = ii; }
public:
  Base(int ii = 0) : i(ii) {}
  int calcValue(int m) const {
    return m*i;
  }
};
```

```
class Derived : public Base {
  int j;
public:
  Derived(int jj = 0) : j(jj) {}
  void change(int x) { set(x); }
};

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

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();
}
```

Initialization List

Few points

- Necessary to invoke the constructor of the base class
- Base class constructors are invoked in the order of derivation and not in the order of occurrence in the initialization list
- If a base class constructor is not specified, default base class constructor will be invoked
- Base class constructor are invoked before the members of the class are initialized

Assignment Operator

Example

Do on 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; }

};
```

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

    oB.fn();

    oD.fn();

    oD.CBase::fn();


    pD = &oB;
    pD->fn();


    pB = &oD;
    pB->fn();

}
```

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();

    oD.fn();


    pB = &oD;
    pB->fn();
}
```

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 :

- 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

}
```

How C++ implements late binding

To accomplish this, the typical compiler creates a single table (called the VTABLE) for each class that contains **virtual** functions. The compiler places the addresses of the virtual functions for that particular class in the VTABLE. In each class with virtual functions, it secretly places a pointer, called the *vpointer* (abbreviated as VPTR), which points to the VTABLE for that object. When you make a virtual function call through a base-class pointer (that is, when you make a polymorphic call), the compiler quietly inserts code to fetch the VPTR and look up the function address in the VTABLE, thus calling the correct function and causing late binding to take place.

All of this – setting up the VTABLE for each class, initializing the VPTR, inserting the code for the virtual function call – happens automatically, so you don't have to worry about it. With virtual functions, the proper function gets called for an object, even if the compiler cannot know the specific type of the object.