BIT

14. Inheritance & Composition

Hu Sikang

skhu@163.com

BIT

Content

- Composition
- Inheritance
- Single Inheritance
- Accessing Control
- Constructor and Destructor in inheritance
- Multiple Inheritance

14.1 Composition syntax

Composition is to embed an object of a class as an object in a new class. It implements a “*has-a*” relationship with each other.

```
#include "CMyString.h"
#include <iostream>
using namespace std;
enum PRIORITY
        { LOWER, EQUALITY, HIGHER};
class CExpresstion
{
public:
            CExpression(string s = "");
            double Value();
            void SetExpression(string s);
            void Print();
private:
            string m_strExpr;
            CMyStack  stackOperator;
            CMyStack  stackOperand;
            PRIORITY Precede(char first, char second);
            bool isNumber(char ch);
            double Compute(double x, double y, char ch);
};
```

BIT

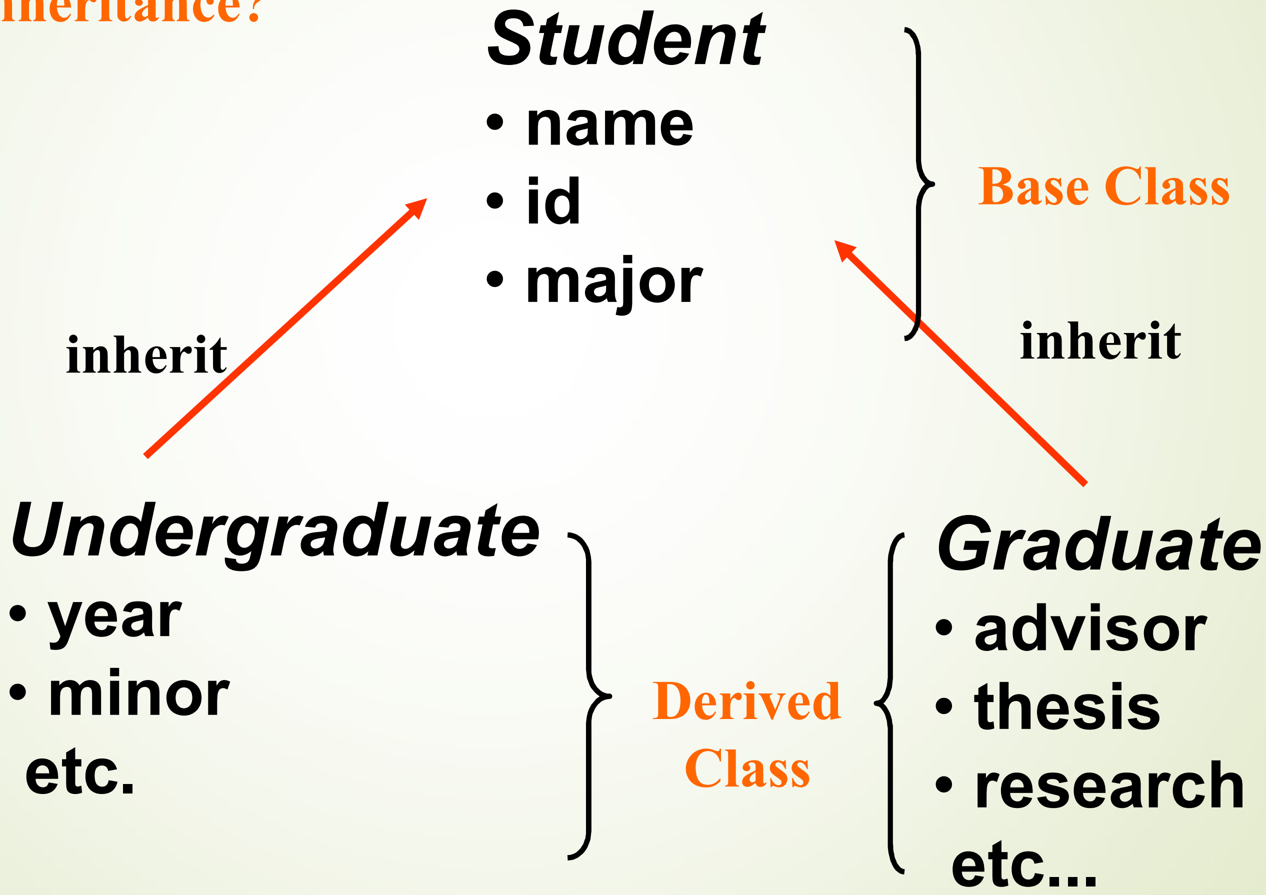
14.1 Composition syntax

Composition is to embed an object of a class as an object in a new class. It implements a “*has-a*” relationship with each other.

```
#include "CMyString.h"
#include <iostream>
using namespace std;
enum PRIORITY
	{ LOWER, EQUALITY, HIGHER};
class CExpresstion : public CMyStack
{
public:
		CExpression(string s = "");
		double Value();
		void SetExpression(string s);
		void Print();
private:
		string m_strExpr;
		PRIORITY Precede(char first, char second);
		bool isNumber(char ch);
		double Compute(double x, double y, char ch);
};
```

14.2 Inheritance syntax

Why use inheritance?



Student

- name
- id
- major

Base Class

inherit

inherit

Undergraduate

- year
- minor

etc.

Derived Class

Graduate

- advisor
- thesis
- research

etc...

BIT

14.3 Base and Derived Classes

- A *base class* is a previously defined class that is used to define new classes.
- *Base class* is also called *super class* or *father class* or *ancestor class*.

- A *derived class* inherits all(*exceptions*) the data and member functions of a base class. The object of derived class can call on the member functions and member data of base class.
- Derived class is also called *subclass* or *posterity*.

14.4 Inheritance

The *single inheritance* is that the derived class only has one base class. It implements an “is-a” relationship with each other.

The *multiple inheritance* is that the derived class has more than one base class.

BIT

14.4.1 Single Inheritance

Syntax:

```
class derived_class_name : accessing_control  base_class
{
        // define data member and function member
}
```

Here the *accessing_control* may be as:

public, private and protected.

14.4.1 Single Inheritance

```
class employee
{
private:
   string name;
   short department;
public:
   void print();
};

class manager : public employee
{
    short level;
public:
    void meeting(int num);
};
```

manager() : employee() { }

```
int main( )
{
        employee E;
        manager  M;
        E.print();          //ok
        E.meeting(2);       //error
        M.print();          //ok
        M.meeting();        //ok
        return 0;
}
```

The member function, meeting(), doesn’t belong to base class.

14.4.2 Accessing Control: public

class manager : public employee;

If a derived class, *manager*, has a *public* base class *employee,* then:

[1] the object of *manager* can access the member functions and member data of *employee*'s *public*.

[2] the member functions of *manager* can access the member functions and member data of *employee*'s *public* and *protected*.

[3] the member functions and the object of *manager* *CANNOT* access member functions and data of *employee*'s *private*.

14.4.2 Accessing Control: public

```
#include <string>
using namespace std;
class employee     {
private:
    string name;
    short department;
public:
    void print();      };

class manager : public employee  {
            short level;
public:
            void meeting(int Num)
            { department = Num; }    //error
};
```

```
int main( ) {
      employee E;
      manager  M;
      E.print();                  //ok
      E.meeting(2);           //error
      M.name = "John"; //error
      M.print();                //ok
      M.meeting();           //ok
      return 0;
}
```

If it's certain to assign to *department* in the *meeting, what shall* we do?

14.4.3 Accessing Control: protected

```
class  class_name
{
protected:
        // define member data and functions
};
```

The keywords, *protected*, is used to define a part of class where the object of class can’t access member functions and data, but the member functions of derived class of this class can access.

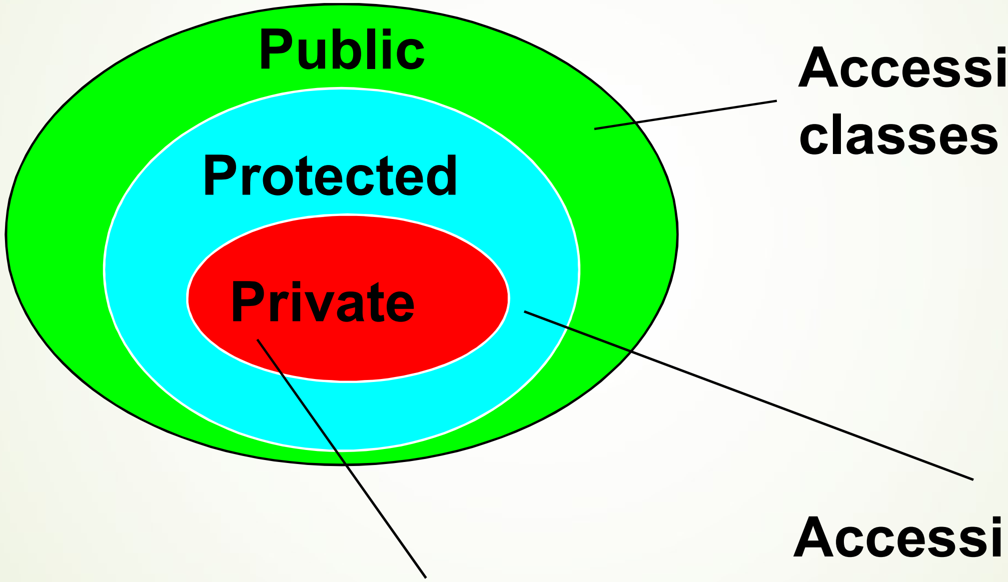
14.4.3 Accessing Control: protected

```
class employee {
pirvate:
    string name;
protected:
    short department;
public:
    void print();
};

class manager : public employee  {
            short level;
public:
            void meeting(int Num)
            { department = Num; }    //ok
};
```

```
int main( )
{
      employee E;
      manager  M;
      E.print();                //ok
      E.meeting(2);             //error
      M.department = 2; //error
      M.print();                //ok
      M.meeting(2);             //ok
      return 0;
}
```

14.4.3 Accessing Control: protected



Public

Protected

Private

Accessible to derived classes and the instances

Accessible to derived classes only

Not accessible

14.5 Functions that don’t automatically inherit

Constructors and Destructors

[1] Constructors and destructors cannot be inherited.

[2] If a base class has constructors, then a constructor must be invoked by derived class.

[3] Default constructors can be invoked implicitly.

[4]However, if all constructors for a base require arguments, then a constructor for that base must be explicitly called.

[5] Arguments for the base class’ constructor are specified in the definition of a derived class’ constructor.

[6] The member function, operator =(const classType& obj), isn’t inherit yet because its action looks like *the copy-constructor*.

BIT

Overloading assignment in a inheritance

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

class Base {
protected:          int value;
public:
            Base(int x) { value = x; }
            void operator=(const Base& bb)
            {    this->value = bb.value;     }
};
class Derived : public Base {
private: int der;
public:
            Derived(int x, int y) : Base(x) { der = y; }
            void operator=(const Derived& dd)
            {    this->der = dd.der;     }
            friend ostream& operator << (ostream& os, const Derived dd)
            {    return os << dd.value << ", " << dd.der << endl;   }
};
```

```
int main()
{
      Derived d1(11, 22);
      Derived d2(33, 44);

      d1 = d2;

      cout << d1 << endl;

      return 0;
}
```

What’s the output?

BIT

14.5 Order of constructor& destructor called

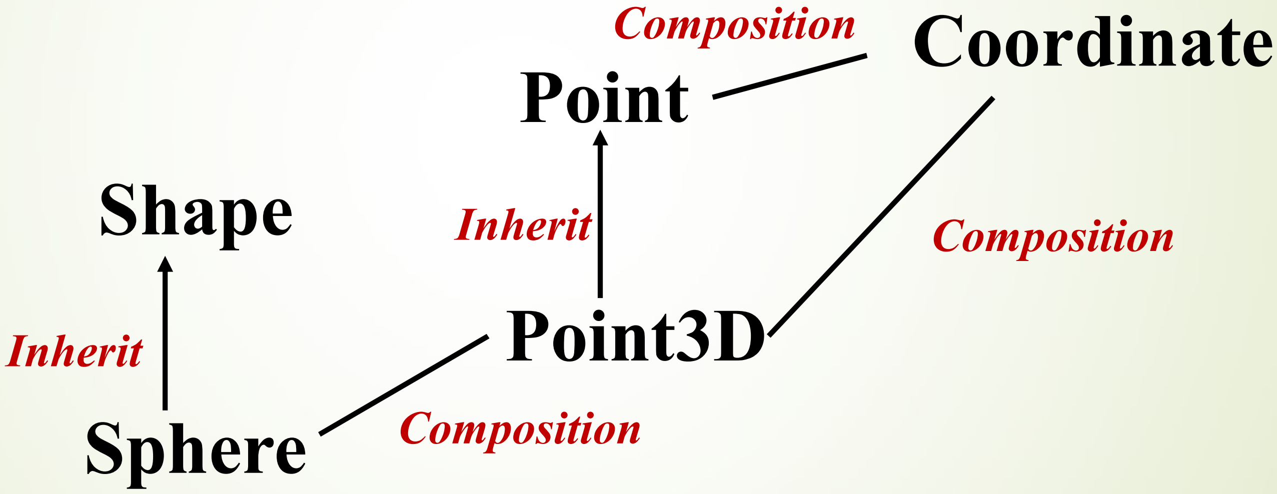
Class objects are constructed from the bottom to up:

[1] first the base, then the members, and then the derived class itself.

They are destroyed in the opposite order:

[2] first the derived class itself, then the members, and then the base.

Exercise


Composition
Coordinate
Point
Inherit
Composition
Shape
Point3D
Inherit
Sphere
Composition

BIT

Constructors and Destructors

```
#include <iostream>
using namespace std;
class Coordinate {
public:
      Coordinate() { cout << "Coordinate," << endl; }
      ~Coordinate() { cout << "~Coordinate,"<< endl; }
};
class Point {
public:
      Point() { cout << "Point," << endl; }
      ~Point() { cout << "~Point," << endl; }
private:
      Coordinate x;
};
```

BIT

Constructors and Destructors

```
class Point3D :public Point {
public:
    Point3D() { cout << "Point3D," << endl; }
    ~Point3D() { cout << "~Point3D," << endl; }
private:
    Coordinate z;
};
class Shape {
public:
     Shape() { cout << "Shape," << endl; }
     ~Shape() { cout << "~Shape," << endl; }
};
```

Constructors and Destructors

```
class Sphere :public Shape {
public:
    Sphere() { cout << "Sphere" << endl; }
    ~Sphere() { cout << "~Sphere" << endl; }
private:
    Point3D center;
    unsigned radius;
};
int main()
{
     Sphere S;
     return 0;
}
```

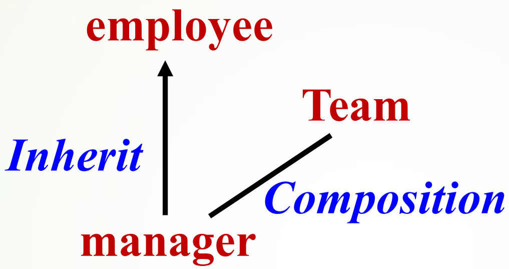
BIT

14.6 Combining composition & inheritance

Of course, we can use composition & inheritance together. The following example shows the creation of a more complex class using both of them.

```
class employee   {
private:
   string name;
protected:
   short department;
public:
   employee(string s = ″″);
   void print();
};

class Team {
private:
      string m_name;
public:
      Team(string s)  {  m_name = s; }
      string  TeamName( );
};
```



```
class manager : public employee
{
private:
            Team T;
public:
            string GetTeamName( )
            {  return T.TeamName(); }
};

int main( ) {
     employee E;
     manager  M;
     E.print();              //ok
     M.print();              //ok
     M.meeting(2);           //ok
     M.GetTeamName();        // ok
     return 0;
}
```

manager(string s) : T(s)

manager M(“GroupInC++”);

BIT

14.7 Upcasting

The most important aspect of inheritance is not that it provides member functions form the new class, however. It’s the relationship expressed between the new class and the base class.

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

class Instrument {
private:  int a;
public:    void play( ) const;
};
class Wind : public Instrument
{ private:  int b;     };
```

```
void tune(const Instrument& i)
{    i.play( );    }

int main( )   {
    Wind flute;
    tune(flute);  // Upcasting
    return 0;
}
```

14.7 Upcasting

- **static_cast<new type> (expression)**: It's mainly used for mutual conversion between built-in data types, and type safety checks.

  double b = 3.14; int a = static_cast<int>(b);

- **const_cast<new_type* / &> (expression)**: It's ONLY used to add / remove pointer / reference of variable.

```
void fun(Shape& cs);
int main( )  // Remove const characteristic
{
    const Shape s;
    fun(const_cast<Shape&>(s));
    return 0;
}
```

```
int main( ) // Add const characteristic
{
        const int a =10;
        int *p = const_cast<int*>(&a);
        return 0;
}
```

14.7 Upcasting

- **dynamic_cast<new_type * / &>**): It’s mainly used for mutual conversion between pointers or references of base class an derived class. Especially Conversion is from base class to derive class.

```
int main( )
{
        Shape shape;
        Shpere *ps = dynamic_cast<Sphere*>(&shape);
        return 0;
}
```

- ● The class, Shape, must contain virtual function.
- ● The dynamic_cast is used for type safety checks.

14.7 Upcasting

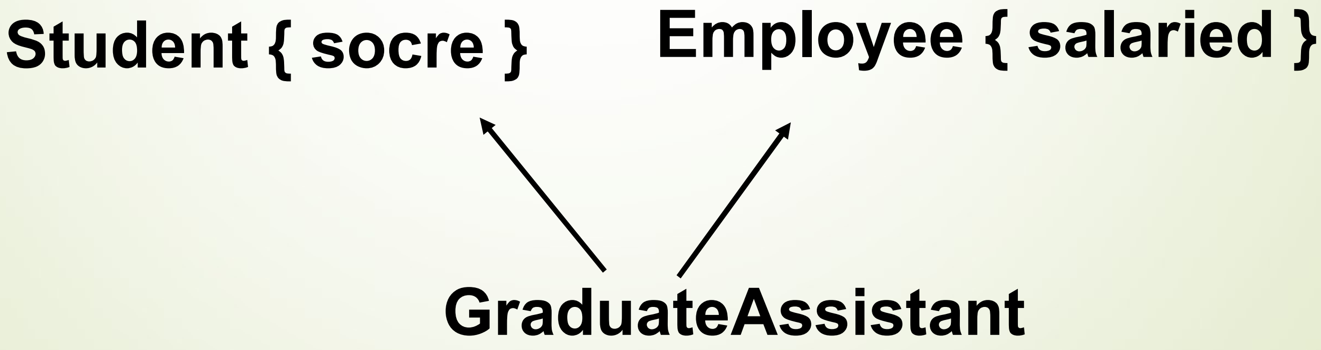
- **reinterpret_cast<new_type>:** It’s mainly used for mutual casting between different types.

```
int main()
{
        int p = 0xff44;
        int *pb = static_cast<int*>(p);         // ERROR
        int *pc = reinterpret_cast<int*>(p);  // RIGHT
}
```

14.8 Multiple Inheritance

A class can have more than one direct base class, that is, more than one class specified after the **:** in the class definition.

The use of more than one immediate base class is usually called *multiple inheritance*.



Student { socre }

Employee { salaried }

GraduateAssistant

14.8.1 Multiple Inheritance

Syntax:

class *derived_class_name* : *accessing_control* base_class1, …… , *accessing_control* base_classN,

{

//define data member and function member

}

Thereinto the *accessing_control* may be as: public, private and protected.

14.8.1 Multiple Inheritance

Example:

```
#include <iostream>
using namespace std;
class A {
     public:
          void setA(int x)  {  a = x;  }
     private:
          int a;
};

class B {
public:
    void setB(int x)  {  b = x;  }
private:
     int b;
};
```

```
class C : public A, public B {
public:
     void setC(int x)  {  c = x;  }
private:
      int c;
};

int main( )
{
     C obj;
     obj.setA(5);
     obj.setB(6);
     obj.setC(7);
     return 0;
}
```

C() : A(), B();

14.8.2 Multiple Inheritance

Problem 1: If there is a same name function, ***fun()***, in the base class A and the base class B, and the object of derived class C calls fun(), then which fun() you want to call?

```
class A
{
public:
    void fun();
};
```

```
class B
{
 public:
     void fun();
 };
```

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

int main()
{
    C obj;
    obj.fun();    // ambiguous
    return 0;
}
```

14.8.2 Multiple Inheritance

Problem 1: If there is a same name function, ***fun()***, in the base class A and the base class B, and the object of derived class C calls fun(), then which fun() you want to call?

```
class A
{
public:
    void fun();
};

class B
{
public:
    void fun();
};
```

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

int main()
{
    C obj;
    obj.fun();    //ambiguous
    return 0;
}
```

Solution 1: **Explicit declaration is added to member function.**

```
int main() {
    C obj;
    obj.A::fun();  //call A' fun()
    obj.B::fun();  //call B' fun()
    return 0;
}
```

BIT

14.8.2 Multiple Inheritance

Problem 1: If there is a same name function, fun(), in the base class A and the base class B, and the object of derived class C calls fun(), then which fun() you want to call?

```
class A
{
public:
    void fun();
};

class B
{
public:
    void fun();
};
```

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

int main()
{
    C obj;
    obj.fun();     //ambiguous
    return 0;
}
```

Solution 2: Defining a new function in the derived class C.

```
class C : public A, public B
{
 public:
    void fun()  //Name Hiding
    {   A::fun();     B::fun();   }
};
```

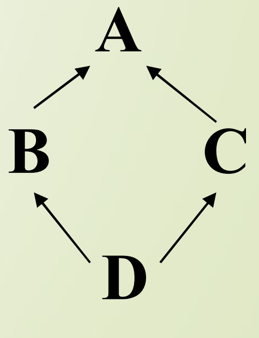
14.8.2 Multiple Inheritance

Problem 2: A derived class, *class D*, has two base classes, and the two base classes have same base class A. When the object of class D calls the member function of class A, there will be a problem.

```
class A {
public:
            void fun( );
};
class B : public A {
public:
            void FB( );
};
class C : public A {
public:
            void FC( );
};
```

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

int main( )  {
    D obj;
    obj.FB( );     //ok
    obj.FC( );     //ok
    obj.fun( );     //ambiguous
    return 0;
}
```



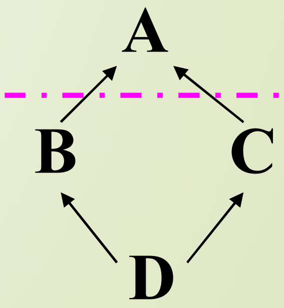
14.9 Virtual Base Classes

Solution: Defining base class as virtual base class.

```
class A {
public:
        void fun();
};
class B : virtual public A
{
public:
        void FB();
};
class C : virtual public A
{
public:
        void FC();
};
```

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

int main()
{
        D obj;
        obj.FB();     //ok
        obj.FC();     //ok
        obj.fun();    //ok
        return 0;
}
```



virtual base class