Advanced Programming

Lab 14

CONTENTS

- Learn to define and use class inheritance relationships
- Learn how to derive one class from another
- Learn polymorphism
- Learn the difference between overloading and overriding
- Learn Static and Dynamic binding
- Access control

2 Knowledge Points

2.1 Inheritance

2.2 Virtual functions

2.3 Polymorphism

2.4 Static and Dynamic binding

2.5 Access control

2.1 Inheritance

Inheritance is one of the most important feature of Object-Oriented Programming. **Inheritance** allows us to **define a class in terms of another class**, which makes it easier to maintain an application. This also provides an opportunity to **reuse the code** functionality and fast implementation time.

The existing class is called the **base class**, and the new class is called the **derived class**.

Inheritance syntax:

```
class derived_class_name : access_mode   base_class_name
{
    // body of subclass
};
```

Subclass, Derived class, Child class

public, protected, private

Base class, Super class, Parent class

The derived class consists of **two parts**:

- The subobject of its base class (consisting of the non-static base class data members)
- The derived class portion (consisting of the non-static derived class data members)

```cpp
class Employee
{
protected:
    char* name;
    char ssn[20];

public:
    Employee(const char* n, const char* s);
    Employee(const Employee& e);
    virtual ~Employee();


    Employee& operator=(const Employee& e);


    void show();
};
```

base class

base class data members

base class member functions

```cpp
Employee::Employee(const char* n, const char* s)
{
    name = new char[strlen(n) + 1];
    strcpy(name, n);
    strcpy(ssn, s);
    cout << "The base class constructor is invoked." << endl;
}
```

base class constructor

```cpp
Employee::Employee(const Employee& e)
{
    name = new char[strlen(e.name) + 1];
    strcpy(name, e.name);
    strcpy(ssn, e.ssn);
    cout << "The base class constructor is invoked." << endl;
}
```

base class copy constructor

```cpp
Employee& Employee::operator=(const Employee& e)
{
    if (this == &e)
        return *this;
    delete[] name;
    name = new char[strlen(e.name) + 1];
    strcpy(name, e.name);
    strcpy(ssn, e.ssn);
    return *this;
}
```

base class copy assignment operator

```cpp
Employee::~Employee()
{
    delete[] name;
    cout << "The base class destructor is invoked." << endl;
}
```

base class destructor

```cpp
void Employee::show()
{
    cout << "Name is: " << name << ", SSN number is: " << ssn << endl;
}
```

public derivation, it means every derived-class object ***is an*** object of its base class, represents an ***is-a*** relationship.

```
class SalariedEmployee : public Employee
{
private:
    double salary;

public:
    SalariedEmployee(const char* name, const char* ssn, double s) : Employee(name, ssn), salary(s)
    {
        cout << "The derived class constructor is invoked." << endl;
    }
    virtual ~SalariedEmployee()
    {
        cout << "The derived class destructor is invoked." << endl;
    }
```

当执行派生类的析构函数时，会自动调用基类的析构函数

```
    SalariedEmployee(const Employee& e, double s) : Employee(e), salary(s) {}

    double getSalary() const { return salary; }
    void setSalary(double s) { salary = s; }

    double earning() { return getSalary(); }

    void show()
    {
        cout << "Name is: " << name << ", SSN number is: " << ssn << ", Salary is: " << salary << endl;
    }
};
```

derived class

derived class data member(s)

derived class member functions

The base-class constructor can not be inherited in the derived class, so derived-class constructor must use the base-class constructor.

passing arguments from the derived-class constructor to the base-class constructor

```
SalariedEmployee(const char* name, const char* ssn, double s) : Employee(name, ssn), salary(s)
{
    cout << "The derived class constructor is invoked." << endl;
}
```

Use member initialization list to invoke the base-class constructor to initialize the base-class data members.

Use member initialization list to initialize the derived-class data member.

```
SalariedEmployee(const Employee& e, double s) : Employee(e), salary(s) {}
```

Use member initialization list to invoke the base-class copy constructor to create the base-class object.

```
virtual ~SalariedEmployee()
{
    cout << "The derived class destructor is invoked." << endl;
}
```

If no destructor in the derived-class, the compiler will provide one without doing anything.

```cpp
#include <iostream>
#include <string>
#include "Employee.h"
using namespace std;

int main()
{
    Employee e1("Liming", "1000");
    Employee e2("Xutong", "1001");

    SalariedEmployee se1("Wangfang", "2000", 3000);
    SalariedEmployee se2("Zhangxiao", "2001", 2800);

    cout << "\nEmployee:e1,e2:" << endl;
    e1.show();
    e2.show();

    cout << "\nSalaried Employee:se1,se2:" << endl;
    se1.show();
    se2.show();

    Employee e3(e1);
    SalariedEmployee se3(se1);

    cout << "\nEmployee:e3(created by e1), Salaried Employee:se3(create by
se1):" << endl;
    e3.show();
    se3.show();

    e3 = e2;
    se3 = se2;
    cout << "\nAfter assigned e2 and se2 to e3 and se3:" << endl;
    e3.show();
    se3.show();

    cout << endl;
    return 0;
}
```

create two base-class objects

create two derived-class objects

create a base-class object by another base-class object

create a derived-class object by another derived-class object

assignment

```
The base class constructor is invoked.
The base class constructor is invoked.
The base class constructor is invoked.
The derived class constructor is invoked.
The base class constructor is invoked.
The derived class constructor is invoked.

Employee:e1,e2:
Name is: Liming, SSN number is: 1000
Name is: Xutong, SSN number is: 1001

Salaried Employee:se1,se2:
Name is: Wangfang, SSN number is: 2000, Salary is: 3000
Name is: Zhangxiao, SSN number is: 2001, Salary is: 2800
The base class constructor is invoked.
The base class constructor is invoked.

Employee:e3(created by e1), Salaried Employee:se3(create by se1):
Name is: Liming, SSN number is: 1000
Name is: Wangfang, SSN number is: 2000, Salary is: 3000

After assigned e2 and se2 to e3 and se3:
Name is: Xutong, SSN number is: 1001
Name is: Zhangxiao, SSN number is: 2001, Salary is: 2800
The derived class destructor is invoked.
The base class destructor is invoked.
The base class destructor is invoked.
The derived class destructor is invoked.
The base class destructor is invoked.
The derived class destructor is invoked.
The base class destructor is invoked.
The base class destructor is invoked.
The base class destructor is invoked.
```

destroy se3

destroy e3

destroy se2,se1

destroy e2,e1

Note:

When creating an object of a derived class, a program first calls the base-class constructor and then calls the derived-class constructor. The base-class constructor is responsible for initializing the inherited data member. The derived-class constructor is responsible for initializing any added data members. A derived-class constructor always calls a base-class constructor.

When an object of a derived class expires, the program first calls the derived-class destructor and then calls the base-class destructor. That is, **destroying an object occurs in the opposite order used to construct an object**.

```cpp
class Employee
{
private:
    string name;
    string ssn;

public:
    Employee(const string &n, const string &s) : name(n), ssn(s)
    {
        cout << "The base class constructor is invoked." << endl;
    }

    virtual ~Employee()
    {
        cout << "The base class destructor is invoked." << endl;
    }


    string getName() const { return name; }
    string getSSN() const { return ssn; }
    void setName(const string &n) { name = n; }
    void setSSN(const string &s) { ssn = s; }


    virtual void show()
    {
        cout << "Name is:" << name << ", SSN number is:
" << ssn << endl;
    }
};
```

This time the two attributes are defined as string type.

Using member initialization list to initialize the data members

- 使用初始化列表，name 和 ssn 会直接调用其构造函数（例如 std::string 的拷贝构造函数）进行初始化。
- 避免了额外的赋值操作，因此性能更高，特别是当成员变量是复杂类型时（如 std::string 或自定义类）。

```cpp
Employee(const string &n, const string &s)
{
    name = n;
    ssn = s;
    cout << "The base class constructor is invoked." << endl;
}
```

Using assignment to initialize the data members

Both constructors do the same things, but the latter approach indicates two steps: first, calling the default string **constructor** for **name** and then invoking the string **assignment operator** to reset **name** to **n**. Whereas the member initialization list saves a step by just using the string **copy constructor** to initialize **name** to **n**.

NOTE：

- Member initialization list can be used **only with constructors**.
- You must (at least, in pre-C++11) use this form to initialize a **non-static const** data member.
- You must use this form to initialize a **reference data member**.
- Data members are initialized in the order in which they appear in the class declaration, not in the order in which initializers are listed.
- It's more efficient to use the member initializer list for members that are themselves **class objects**.

derived class

new data in derived class

passing arguments from the derived-class constructor to the base-class constructor

```
class SalariedEmployee : public Employee
{
private:
    double salary;

public:
    SalariedEmployee(const string &n, const string &s, double sa) : Employee(n, s), salary(sa) {}

    ~SalariedEmployee()
    {
        cout << "The derived class destructor is invoked." << endl;
    }

    double getSalary() const { return salary; }
    void setSalary(double sa) { salary = sa; }

    void show()
    {
        cout << "Name is: " << getName() << ", SSN number is: " << getSSN() << ", Salary is: " << salary << endl;
    }
};
```

```cpp
#include <iostream>
#include <string>
#include "Employee.h"
using namespace std;

int main()
{
    Employee e1("Liming", "1000");
    Employee e2("Xutong", "1001");


    SalariedEmployee se1("Wangfang", "2000", 3000);
    SalariedEmployee se2("Zhangxiao", "2001", 2800);


    cout << "\nEmployee:e1,e2:" << endl;
    e1.show();
    e2.show();


    cout << "\nSalaried Employee:se1,se2:" << endl;
    se1.show();
    se2.show();


    Employee e3(e1);
    SalariedEmployee se3(se1);


    cout << "\nEmployee:e3(created by e1), Salaried Employee:se3(create by se1):" << endl;
    e3.show();
    se3.show();


    e3 = e2;
    se3 = se2;
    cout << "\nAfter assigned e2 and se2 to e3 and se3:" << endl;
    e3.show();
    se3.show();


    cout << endl;
    return 0;
}
```

Neither the base class nor the derived class didn't define the copy constructor, but the compiler automatically generates two copy constructors for base class and derived class respectively which do the memberwise copy. These default copy constructors are fine because both base class and derived class do not directly use dynamic memory allocation.

Special relationships between derived and base classes

1. A derived-class object can use base-class methods, provided that the methods are not private.

2. A base-class pointer can point to a derived-class object without an explicit type casting and a base-class reference can refer to a derived-class object without an explicit type casting.

3. Functions defined with base-class reference or pointer arguments can be used with either base-class or derived-class object.

base-class reference

```
void Show(Employee &em)
{
    cout << "Name:" << em.getName() << ", SSN:" << em.getSSN() << endl;
}
```

base-class pointer

```
void Show(Employee *pem)
{
    cout << "Name:" << pem->getName() << ", SSN:" << pem->getSSN() << endl;
}
```

```
Employee employee1("BaiXue", "2003");
SalariedEmployee salaryemployee1("Hu Zhixing", "3210", 1500);

Show(employee1);
Show(salaryemployee1);
```

base-class object as the argument

derived-class object as the argument

```
Name:BaiXue, SSN:2003
Name:Hu Zhixing, SSN:3210
```

Note: there is no salary value.

```
Show(&employee1);
Show(&salaryemployee1);
```

base-class object address as the argument

derived-class object address as the argument

2.2 Virtual Functions

A virtual function is a **member function** which is **declared within a base class** and is **re-defined (overridden) by a derived class**. When you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class's version of the function.

virtual return_type function_name(parameter list);

keyword

- Virtual functions ensure that the correct function is called for an object, regardless of the type of reference (or pointer) used for function call.
- They are mainly used to achieve runtime **polymorphism**.
- Functions are declared with a **virtual** keyword in base class.
- The resolving of function call is done at runtime.

```cpp
// CPP program to illustrate concept of Virtual Functions
#include<iostream>
using namespace std;

class base
{
public:
    virtual void print()
    {
        cout << "print base class\n";
    }

    void show()
    {
        cout << "show base class\n";
    }
};

class derived : public base
{
public:
    void print()
    {
        cout << "print derived class\n";
    }

    void show()
    {
        cout << "show derived class\n";
    }
};
```

virtual function defined in base-class

non- virtual function defined in base-class

virtual function redefined in derived-class

non- virtual function defined in base-class

```cpp
int main()
{
    base *bptr;
    derived d;
    bptr = &d;

    // Virtual function, bound at runtime
    bptr->print();

    // Non-virtual function, bound at compile time
    bptr->show();

    return 0;
}
```

```
print derived class
show base class
```

A base-class pointer or reference can point(refer) to a derived-class object. When you use such pointer or reference to invoke a **virtual function**, which one will be invoked, base version or derived version? It depends on the actual object rather than the pointer or reference type.

```cpp
int main()
{
    base *bptr;
    derived *dptr;
    derived d;
    bptr = &d;
    dptr = &d;

    // invoke base show function
    bptr->show();

    // invoke derived show function
    dptr->show();

    // Virtual function, bound at runtime
    // bptr->print();

    // Non-virtual function, bound at compile time
    // bptr->show();

    return 0;
}
```

both base class pointer and derived class pointer point to the derived object

```
show base class
show derived class
```

A base-class pointer or reference can point(refer) to a derived-class object. When you use such pointer or reference to invoke a **non virtual function**, which one will be invoked, base version or derived version? It depends on the pointer or reference type.

In derived class, redefine a non virtual function of base class is not recommended.

Destructors

Destructors should be virtual unless a class isn't to be used as a base class.

```
int main()
{
    Employee* pe = new SalariedEmployee("Wangfang", "1001", 2000);

    pe->show();

    delete pe;

    return 0;
}
```

```
The base class constructor is invoked.
The derived class constructor is invoked.
Name is: Wangfang, SSN number is: 1001, Salary is: 2000
The base class destructor is invoked.
```

If the destructors is **not virtual**, the delete statement invokes the ***~Employee()*** destructor. This frees memory pointed to by the ***Employee*** component of the ***SalariedEmployee*** object not memory pointed to by **SalariedEmployee** component.

If the destructor is **virtual**, the same code invokes the ***~SalariedEmployee()*** destructor, which frees memory pointed to by the **SalariedEmployee** component, and then calls the ***~Employee()*** destructor to free memory pointed to by the ***Employee*** component.

```
The base class constructor is invoked.
The derived class constructor is invoked.
Name is: Wangfang, SSN number is: 1001, Salary is: 2000
The derived class destructor is invoked.
The base class destructor is invoked.
```

2.3 Polymorphism

Polymorphism is one of the most important feature of object-oriented programming. Polymorphism works on object **pointers** and **references** using so-called **dynamic binding** at run-time. It does not work on regular objects, which uses static binding during the compile-time.

There are **two key mechanisms for implementing polymorphic in public inheritance:**

1. Redefining base-class virtual methods in a derived class

2. Invoking virtual methods by pointer or reference of the base-class

```
// shape.h -- Shape class

#ifndef SHAPE_SHAPE_H
#define SHAPE_SHAPE_H

class Shape
{
public:
    virtual double Area() const { return 0; }
    virtual void Show() const {}
};

#endif // SHAPE_SHAPE_H
```

base class

const 写在函数声明后修饰的是当前对象（this 指针）

If you use the keyword **virtual**, the program choose a method based on the type of object the reference or pointer refers to rather than based on the reference type or pointer type.

```
// rectangle.h -- Rectangle class
#ifndef SHAPE_RECTANGLE_H
#define SHAPE_RECTANGLE_H

#include <iostream>
#include "Shape.h"

class Rectangle : public Shape // public inheritance
{
private:
    double width, height;

public:
    Rectangle() : width(1), height(1) { }
    Rectangle(double w, double h) : width(w), height(h) { }

    double Area() const override
    {
        return width * height;
    }

    void Show() const override
    {
        std::cout << "Rectangle:" << std::endl;
        std::cout << "width: " << width << ", height: " << height << ", area: " << Area() << std::endl;
    }
};

#endif //SHAPE_RECTANGLE_H
```

derived class

redefine the virtual functions in Rectangle class. **override** keyword indicates that the virtual function is redefined.

```
// circle.h -- Circle class
#ifndef SHAPE_CIRCLE_H
#define SHAPE_CIRCLE_H

#include <iostream>
#include "Shape.h"
#define PI 3.1415

class Circle : public Shape // public inheritance
{
private:
    double radius;


public:
    Circle() : radius(1) { }
    Circle(double r) : radius(r) { }

    double Area() const override
    {
        return PI * radius * radius;
    }

    void Show() const override
    {
        std::cout << "Circle:" << std::endl;
        std::cout << "radius: " << radius << ", area: " << Area() << std::endl;
    }
};


#endif //SHAPE_CIRCLE_H
```

derived class

redefine the virtual functions in Circle class

```cpp
#include <iostream>
#include "Shape.h"
#include "circle.h"
#include "rectangle.h"
using namespace std;

int main()
{
    Circle circle1(3);
    Circle circle2(4.5);
    Rectangle rectangle1(3.5, 4);
    Rectangle rectangle2(1.6, 5.3);

    Shape& c_ref = circle1;
    c_ref.Show();

    Shape& r_ref = rectangle1;
    r_ref.Show();

    Shape *ps = &circle2;
    ps->Show();
    ps = &rectangle2;
    ps->Show();

    return 0;
}
```

```
Circle:
radius: 3, area: 28.2735
Rectangle:
width: 3.5, height: 4, area: 14
Circle:
radius: 4.5, area: 63.6154
Rectangle:
width: 1.6, height: 5.3, area: 8.48
```

Both reference types are **Shape**, but they refer to different objects.They invoke different objects' **Show()** functions. This is polymorphism.

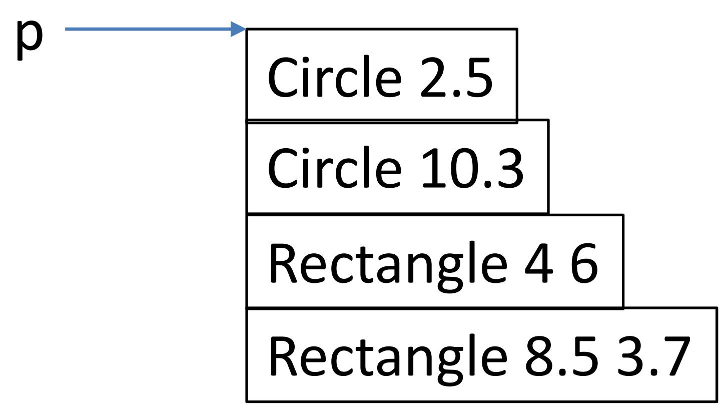
The pointer type of **ps** is **Shape**, it points to a different object respectively, and invokes different objects' **Show()** functions. This is polymorphism.

Suppose you would like to manage a mixture of **Circle** and **Rectangle**. It would be nice if you could have a single array that holds a mixture of Circle and Rectangle objects, but that’s not possible. Every item in an array has to be of the same type, but Circle and Rectangle are two separate types. However, you can create an **array of pointers-to-Shape**. In that case, every element is of the same type, but because of the public inheritance mode, a pointer-to-Shape can point to either a Circle or a Rectangle object. Thus, in effect, you have a way of representing a collection of more than on type of object with a single array.

```cpp
#include "Shape.h"
#include "circle.h"
#include "rectangle.h"
using namespace std;


const int AMOUNT = 4;


int main()
{
    Shape* p[AMOUNT] = {
        new Circle(2.5),
        new Circle(10.3),
        new Rectangle(4, 6),
        new Rectangle(8.5, 3.7)
    };

    for (int i = 0; i < AMOUNT; i++)
        p[i]->Show();

    for (int i = 0; i < AMOUNT; i++)
        delete p[i];

    return 0;
}
```

polymorphism

free the memory




```
Circle:
radius: 2.5, area: 19.6344
Circle:
radius: 10.3, area: 333.282
Rectangle:
width: 4, height: 6, area: 24
Rectangle:
width: 8.5, height: 3.7, area: 31.45
```

What's the problem of the program?

The destructor of Shape should be virtual.

```cpp
#pragma once
#include <iostream>

class Pet
{
public:
    virtual void eat() const = 0;
    ~Pet()
    {
        std::cout << "Pet's destructor is invoked." << std::endl;
    }
};

class Dog : public Pet
{
public:
    void eat() const override
    {
        std::cout << "A dog is eating." << std::endl;
    }
    virtual ~Dog()
    {
        std::cout << "Dog's destructor is invoked." << std::endl;
    }
};

class Cat : public Pet
{
public:
    void eat() const override
    {
        std::cout << "A cat is eating." << std::endl;
    }
    virtual ~Cat()
    {
        std::cout << "Cat's destructor is invoked." << std::endl;
    }
};
```

The destructor of Pet is non-virtual.

```cpp
#include <iostream>
#include "animal.h"

void feed(Pet *);

int main()
{
    Pet* p[5] = {
        new Cat,
        new Dog,
        new Cat,
        new Dog,
        new Cat
    };

    for(int i = 0; i < 5; i++)
        feed(p[i]);

    for(int i = 0; i < 5; i++)
        delete p[i];

    return 0;
}

void feed(Pet *p)
{
    p->eat();
}
```

```
A cat is eating.
A dog is eating.
A cat is eating.
A dog is eating.
A cat is eating.
Pet's destructor is invoked.
Pet's destructor is invoked.
Pet's destructor is invoked.
Pet's destructor is invoked.
Pet's destructor is invoked.
```

The destructors of derived class are not invoked.

polymorphism

You can add as many new types as you want to the system without changing the **feed() method**. The method that manipulate the superclass will not need to changed at all to accommodate the new classes.

2.4 Static Binding vs Dynamic Binding

For non-virtual function, the compiler selects the function that will be invoked at compiled-time(known as **static binding**).

The function selected depends on the actual type that invokes the function(known as **dynamic binding** or **late binding**).

Dynamic binding in C++ is associated with methods invoked by **pointers** and **references**, and this is governed, in part, **by the inheritance process**.

Suppose **Brass** is a base class and **BrassPlus** is a derived class. **ViewAcct()** is a virtual function in two classes.

```
void fr(Brass & rb);    // uses rb.ViewAcct()
void fp(Brass * pb);    // uses pb->ViewAcct()
void fv(Brass b);      // uses b.ViewAcct()

int main()
{
    Brass b("Billy Bee", 123432, 1000.0);
    BrassPlus bp ("Betty Beep", 232313, 12345.0);

    fr(b);     // uses Brass::ViewAcct()
    fr(bp);   // uses BrassPlus::ViewAcct()
    fp(&b);    // uses Brass::ViewAcct()
    fp(&bp);   // uses BrassPlus::ViewAcct()

    fv(b);    // uses Brass::ViewAcct()
    fv(bp);   // uses Brass::ViewAcct()

    ...
}
```

base-class object

derived-class object

The implicit upcasting that occurs with references and pointers causes the **fr()** and **fp()** functions to use **Brass::ViewAcct()** for **Brass** objects and **BrassPlus::ViewAcct()** for **BrassPlus** objects.

Passing by value causes only the **Brass** component of a **BrassPlus** object to be passed to the **fv()** function.

```
BrassPlus ophelia;    // derived-class object
Brass * bp;           // base-class pointer
bp = &ophelia;        // Brass pointer to BrassPlus object
bp->ViewAcct();       // Which version?
```

If **ViewAcct()** is not declared as virtual in the base class, **bp->ViewAcct()** goes by the pointer type(Brass *) and invokes **Brass::ViewAcct().** The pointer type is known at compile time, so the compiler can bind **ViewAcct()** to **Barass::ViewAcct()** at compile time. In short, the compiler uses **static binding for non-virtual method**.

If **ViewAcct()** is declared as virtual in the base class, **bp->ViewAcct()** goes by the object type(BrassPlus) and invokes **BrassPlus::ViewAcct()**. The object type might only be determined when the program is running. Therefore, the compiler generates code that binds **ViewAcct()** to **Brass::ViewAcct()** or **BrassPlus::ViewAcct()**, depending on the object type, while the program executes. In short, the compiler uses **dynamic binding for virtual methods**.

Overloading vs Overriding

	Method Overloading	Method Overriding
Definition	In Method Overloading, Methods of the same class shares the same name but each method must have different number of parameters or parameters having different types and order.	In Method Overriding, sub class have the same method with same name and exactly the same number and type of parameters and same return type as a super class.
Meaning	Method Overloading means more than one method shares the same name in the class but having different signature.	Method Overriding means method of base class is re-defined in the derived class having same signature.
Behavior	Method Overloading is to "add" or "extend" more to method's behavior.	Method Overriding is to "Change" existing behavior of method.

Overloading and Overriding is a kind of polymorphism means “one name, many forms”.

	Method Overloading	Method Overriding
Polymorphism	It is a **compile time polymorphism**.	It is a **run time polymorphism**.
Inheritance	It may or may not need **inheritance** in Method Overloading.	It always requires inheritance in Method Overriding.
Signature	In Method Overloading, methods must have **different signature**.	In Method Overriding, methods must have **same signature**.

2.5 Access control

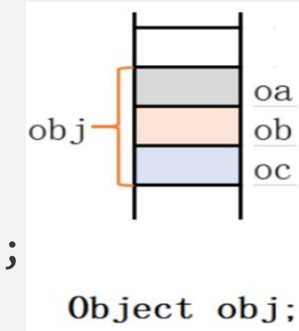
The below table summarizes the three modes and shows the access specifier of the members of base class in the sub class when derived in public, protected and private modes:

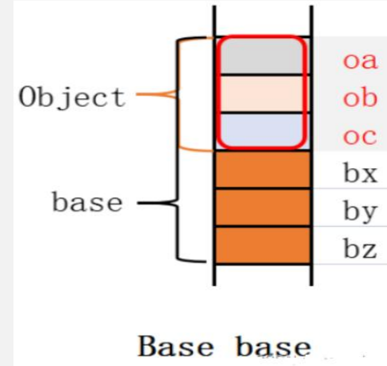
Base class member access specifier	Type of Inheritence		
	Public	Protected	Private
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Not accessible (Hidden)	Not accessible (Hidden)	Not accessible (Hidden)

```cpp
class Object
{
private:
    int oa;
protected:
    int ob;
public:
    int oc;
};

class Base : public Object
{
private:
    int bx;
protected:
    int by;
public:
    int bz;
    void fun()
    {
        bx = by = bz = 0; // ok
        oa = 1; // error
        ob = 2; // ok
        oc = 3; // ok
    }
};
```

The following results are the same whether the inheritance is public, private or protected.

```cpp
int main()
{
    Object obj;
    Base base;
    cout << "obj size: " << sizeof(obj) << endl;
    cout << "base size: " << sizeof(base) << endl;

    // Access the parent data member outside the class
    base.Object::oc = 2;

    return 0;
}
```





oc is the public member data in the Object class, which is accessible to all outsiders. (using scope resolution operator)

The derived class method can access all attributes of itself(including the base attribute), whether the inheritance is public, private or protected. Whether the attributes of the base class can be accessed depends on access control of them. Derived class methods can only access its public and protected attribute, but not its private ones.

```cpp
class Object
{
private:
    int oa;
protected:
    int ob;
public:
    int oc;
};

class Base : public Object
{
private:
    int bx;
protected:
    int by;
public:
    int bz;
    Object obja;
    void fun()
    {
        oc = 1; // ok
        ob = 2; // ok
        obja.oc = 1; // ok
        obja.ob = 2; // error
    }
};
```

```cpp
int main()
{
    Base base;
    // Access the public attributes of the parent class by parent class
    base.Object::oc = 1; // ok

    // Access the public attributes of the parent class by parent object
    base.obja.oc = 2; // ok

    return 0;
}
```

When the base class object is a member of the derived class, it cannot access the protected attributes of the parent class.

3 Exercises

1. Design a stereo graphic class (**CStereoShape** class), and meet the following requirements:

- A virtual function **GetArea**, which can get the surface area of the stereo graphic. Here we let it print out **CStereoShape::GetArea()** and return a value of 0.0, which means that CStereoShape's GetArea is called.
- A virtual function **GetVolume**, which can get the volume of the stereo graphic. Here we let it print out **CStereoShape::GetVolume()** and return a value of 0.0, which means that CStereoShape's GetVolume is called.
- A virtual function **Show**, which print out the description of the stereo graphics. But here we let it print out **CStereoShape::Show()**, which means that Show of CStereoShape is invoked.
- A static private integer variable named **numberOfObject**, whose initial value is 0, which denotes the number of Stereo graphics generated by our program.
- A method named **GetNumOfObject()** that returns the value of numberOfObject.
- Add constructor functions based on requirement.

2. Design a cube class (**CCube** class), which inherits the **CStereoShape** and meets the following requirements:

- A no-arg constructor that creates a default Cube.
- A constructor with parameters whose parameters correspond to the length, width, and height of the cube, respectively.
- A copy constructor that creates a Cube object with the specified object of Cube.
- Override **GetArea**, **GetVolume** of the **CStereoShape** class to complete the calculation of the surface area and volume of the cube, respectively.
- Override **Show()** of the **CStereoShape** class to print out the description (includes length, width, height, the surface area and volume) for the **Cube** object.

3. Design a sphere class (**CSphere** class), which inherits the **CStereoShape** and meets the following requirements:

- A no-arg constructor that creates a default Sphere.
- A constructor with parameters whose parameters correspond to the radius of the Sphere.
- A copy constructor that creates a **Sphere** object with the specified object of Sphere.
- Override **GetArea**, **GetVolume** of the **CStereoShape** class to complete the calculation of the surface area and volume of the sphere, respectively.
- Override **Show()** of the CStereoShape class to print out the description (includes radius, the surface area and volume) for the **Sphere** object.

4. Write a test program and complete at least the following tasks in the main functions:

- Create a **Ccube** object named **a_cube**, which the length, width and height are 4.0, 5.0, 6.0 respectively.
- Create a **CSphere** object named **c_sphere**, which radius is 7.9.
- Define the **CStereoShape** pointer **p**, point **p** to **a_cube**, and then print the information of **a_cube** to the terminal by **p**.
- Point **p** to **c_sphere**, then print the information of **c_sphere** to the terminal by p.
- Points out the **number** of Stereo graphics created by the test program.

Note that you may need to use the “setf( )” and “precision( )” formatting methods to set output mode.

Output sample:

```
Cube: Length = 4.00, Width = 5.00, Height = 6.00
Area = 148.00, Volume = 120.00
Sphere: Radius = 7.90
Area = 784.27, Volume = 2065.24
2 objects are created.
```