

C/C++ Program Design

LAB 13

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
- ❑ Learn to use dynamic memory allocation in derived class

2 Knowledge Points

2.1 Inheritance

2.2 Virtual functions

2.3 Polymorphism

2.4 Static and Dynamic binding

2.5 Inheritance Dynamic Memory Allocation

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
```

```
{
```

Subclass, Derived class,
Child class

public, protected, private

Base class, Super class,
Parent class

```
// body of subclass
```

```
};
```

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)

`class Employee` ← base class

`protected:`
`char* name;`
`char ssn[20];` ← base class data members

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

`Employee& operator=(const Employee& e);`

`void show();`
`};`

base class member functions

derived class member functions

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

`class SalariedEmployee : public Employee` ← derived class

`private:`
`double salary;` ← derived class data member(s)

`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;`
`}`

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

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

`double earning() { getSalary(); }`

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

```
Employee::Employee(const char* n, const char* s)
{
    name = new char[strlen(n) + 1];
    strcpy(name, n);
    strcpy(ssn, s);
}
```

base class constructor

```
Employee::Employee(const Employee& e)
{
    name = new char[strlen(e.name) + 1];
    strcpy(name, e.name);
    strcpy(ssn, e.ssn);
}
```

base class copy constructor

```
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

```
Employee::~~Employee()
{
    delete[] name;
}
```

base class destructor

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

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.


```

testemployee.cpp > main()
1  #include <iostream>
2  #include <string>
3  #include "employee.h"
4
5  using namespace std;
6
7  int main()
8  {
9      Employee e1("Liming", "1000");
10     Employee e2("Xutong", "1001");
11
12     SalariedEmployee se1("Wangfang", "2000", 3000);
13     SalariedEmployee se2("zhangxiao", "2001", 2800);
14
15     cout << "\nEmployee:e1,e2:" << endl;
16     e1.show();
17     e2.show();
18
19     cout << "\nSalaried Employee:se1,se2:" << endl;
20     se1.show();
21     se2.show();
22
23     Employee e3(e1);
24     SalariedEmployee se3(se1);
25
26     cout << "\nEmployee:e3(created by e1), Salaried Employee:se3(create by se1):" << endl;
27     e3.show();
28     se3.show();
29
30     e3 = e2;
31     se3 = se2;
32     cout << "\nAfter assigned e2 and se2 to e3 and se3:" << endl;
33     e3.show();
34     se3.show();
35
36     cout << endl;
37     return 0;
38 }

```

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

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 constructor an object.**

C employee2.h > ...

This time the two attributes are defined as string type.

Using member initialization list to initialize the data members

```
3  #include <iostream>
4  using namespace std;
5
6  class Employee
7  {
8  private:
9      string name;
10     string ssn;
11
12 public:
13     Employee(const string& n, const string& s) :name(n), ssn(s)
14     {
15         cout << "The base class constructor is invoked." << endl;
16     }
17
18     virtual ~Employee()
19     {
20         cout << "The base class destructor is invoked." << endl;
21     }
22
23     string getName() const { return name; }
24     string getSSN() const { return ssn; }
25
26     void setName(const string& n) { name = n; }
27     void setSSN(const string& s) { ssn = s; }
28
29     virtual void show()
30     {
31         cout << "Name is:" << name << ",SSN number is: " << ssn << endl;
32     }
33
34 };
```

```
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 has the effect of 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:

- This form(member initialization list) can be used **only with constructors**.
- You must (at least, in pre-C++11) use this form to initialize a **nonstatic 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**.

```
class SalariedEmployee :public Employee ← derived class
{
private:
    double salary; ← new data in derived class
public:
    SalariedEmployee(const string& n, const string& s, double sa):Employee(n,s),salary(sa){}
    ← passing arguments from the derived-class constructor to the base-class constructor

    ~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;
    }
};
```

```

testemployee2.cpp > ...
1  #include <iostream>
2  #include <string>
3  #include "employee2.h"
4
5  using namespace std;
6
7  int main()
8  {
9      Employee e1("Liming", "1000");
10     Employee e2("Xutong", "1001");
11
12     SalariedEmployee se1("Wangfang", "2000", 3000);
13     SalariedEmployee se2("zhangxiao", "2001", 2800);
14
15     cout << "\nEmployee:e1,e2:" << endl;
16     e1.show();
17     e2.show();
18
19     cout << "\nSalaried Employee:se1,se2:" << endl;
20     se1.show();
21     se2.show();
22
23     Employee e3(e1);
24     SalariedEmployee se3(se1);
25
26     cout << "\nEmployee:e3(created by e1), Salaried Employee:se3(create by se1):" << endl;
27     e3.show();
28     se3.show();
29
30     e3 = e2;
31     se3 = se2;
32     cout << "\nAfter assigned e2 and se2 to e3 and se3:" << endl;
33     e3.show();
34     se3.show();
35
36     cout << endl;
37     return 0;
38 }

```

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 memberwise copying. 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 cast and a base-class reference can refer to a derived-class object without an explicit type cast.
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 empolyee1("BaiXue", "2003");
SalariedEmployee salaryemployee1("Hu Zhixing", "3210", 1500);
```

base-class object as the argument

derived-class object as the argument

```
Show(empolyee1);
Show(salaryemployee1);
```

Name:BaiXue, SSN:2003

Name:Hu Zhixing, SSN:3210

Note: there is no salary value.

base-class object address as the argument

derived-class object address as the argument

```
Show(&empolyee1);
Show(&salaryemployee1);
```


An is-a Relationship

A derived class instance inherits all the properties of the base class, in the case of public-inheritance. It can do whatever a base class instance can do. **This is known as a “is-a” relationship.** Hence, you can substitute a subclass instance to a superclass reference.


The below table summarizes the above 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 Inheritance		
	Public	Protected	Private
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Not accessible (Hidden)	Not accessible (Hidden)	Not accessible (Hidden)

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.

```

virtualfunction.cpp > base
1  // CPP program to illustrate concept of Virtual Functions
2
3  #include<iostream>
4  using namespace std;
5
6  class base
7  {
8  public:
9      virtual void print()
10     {
11         cout << "print base class\n";
12     }
13
14     void show()
15     {
16         cout << "show base class\n";
17     }
18 };
19
20 class derived : public base
21 {
22 public:
23     void print()
24     {
25         cout << "print derived class\n";
26     }
27
28     void show()
29     {
30         cout << "show derived class\n";
31     }
32 };

```

virtual function defined in base-class

virtual function redefined in derived-class

```

34 int main()
35 {
36     base *bptr;
37     derived d;
38     bptr = &d;
39
40     // Virtual function, binded at runtime
41     bptr->print();
42
43     // Non-virtual function, binded at compile time
44     bptr->show();
45
46     return 0;
47 }

```

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.

```
34 int main()
35 {
36     base *bptr;
37     derived *dptr;
38     derived d;
39     bptr = &d;
40     dptr = &d;
41
42     // invoke base show function
43     bptr->show();
44
45     // invoke derived show function
46     dptr->show();
47
48
49     // Virtual function, binded at runtime
50     // bptr->print();
51
52     // Non-virtual function, binded at compile time
53     // bptr->show();
54
55     return 0;
56 }
```

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 public inheritance**:

1. **Redefining base-class methods in a derived class**
2. **Using virtual methods**


```

// shape.h -- Shape class

#ifndef SHAPE_SHAPE_H
#define SHAPE_SHAPE_H

#include <iostream>
// formatting stuff
struct Formatting
{
    std::ios_base::fmtflags flag;
    std::streamsize pr;
};

class Shape
{
private:
    static int numberOfObjects;

protected:
    //methods for formatting
    Formatting SetFormat() const;
    void Restore(Formatting& f) const;

public:
    Shape() { numberOfObjects++; }
    static int GetNumOfObj() { return numberOfObjects; }
    virtual void Show() { }
};

#endif //SHAPE_SHAPE_H

```

base class

```

// Shape.cpp -- Shape class methods

#include <iostream>
#include "shape.h"
using namespace std;

int Shape::numberOfObjects = 0;

//protected methods for formatting
Formatting Shape::SetFormat() const
{
    // set up ###.## format
    Formatting f;
    f.flag = cout.setf(ios_base::fixed, ios_base::floatfield);
    f.pr = cout.precision(3);
    return f;
}

void Shape::Restore(Formatting& f) const
{
    cout.setf(f.flag, ios_base::floatfield);
    cout.precision(3);
}

```

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 "shape.h"

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

public:
    Rectangle(double width, double height);
    Rectangle(Rectangle& rec);
    Rectangle()
    {
        width = 1;
        height = 1;
    }
    double GetArea() const;
    void Show();
};

#endif //SHAPE_RECTANGLE_H

```

derived class

redefine the function **Show()** in Rectangle

```

#include "rectangle.h"

Rectangle::Rectangle(double width, double height)
{
    this->width = width;
    this->height = height;
}

Rectangle::Rectangle(Rectangle& rec)
{
    width = rec.width;
    height = rec.height;
}

double Rectangle::GetArea() const
{
    return width * height;
}

void Rectangle::Show()
{
    // set up ###.## format
    Formatting flag = SetFormat();
    std::cout << "width: " << width
        << "\theight: " << height
        << "\tthe area: " << GetArea() << std::endl;

    //Restore original format
    Restore(flag);
}

```

```

// circle.h --- Circle class

#ifndef SHAPE_CIRCLE_H
#define SHAPE_CIRCLE_H

#define PI 3.1415
#include "shape.h"

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

public:
    Circle(double radius);
    Circle(Circle& C);
    ~Circle();
    double GetRadius();
    double GetArea() const;
    void Show();
};
#endif //SHAPE_CIRCLE_H

```

redefine the function **Show()** in Circle

```

// Circle.cpp --- Circle class methods
#include <iostream>
#include "circle.h"

Circle::Circle(double radius) : radius(radius) {}
Circle::Circle(Circle& C)
{
    radius = C.radius;
}
Circle::~~Circle() {}

double Circle::GetRadius()
{
    return radius;
}

double Circle::GetArea() const
{
    return PI * radius * radius;
}

void Circle::Show()
{
    // set up ###.## format
    Formatting flag = SetFormat();

    std::cout << "radius:" << radius
              << "\tthe area: " << GetArea() << std::endl;

    // Restore original format
    Restore(flag);
}

```

```
// main.cpp--- the main program
```

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

```
int main()
```

```
{
    Circle circle(3);
    Shape& c_ref = circle;
    c_ref.Show();    //use circle.Show()
```

```
    Rectangle rectangle(4, 4);
    Shape& r_ref = rectangle;
    r_ref.Show();    // use rectangle.Show()
```

```
    cout << "This program generates " << Shape::GetNumOfObj() << " objects";
```

```
    return 0;
}
```

```
radius:3.000    the area: 28.273
width: 4.000    height: 4.000    the area: 16.000
This program generates 2 objects
```

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


```
// main.cpp--- the main program
```

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

```
int main()
{
    Shape* p;

    Circle circle(5);
    Rectangle rectangle(2, 6);

    p = &circle;
    p->Show();

    p = &rectangle;
    p->Show();

    cout << "This program generates " << Shape::GetNumOfObj() << " objects";

    return 0;
}
```

```
radius:5.000    the area: 78.538
width: 2.000    height: 6.000    the area: 12.000
This program generates 2 objects
```

The pointer type of **P** 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 one type of object with a single array.

```
// main.cpp--- the main program
```

```
#include <iostream>
#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();
```

polymorphism

```
    cout << "This program generates " << Shape::GetNumOfObj() << " objects";
```

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

```

```
    return 0;
}
```

p

Circle 2.5

Circle 10.3

Rectangle 4 6

Rectangle 8.5 3.7

```
radius:2.500    the area: 19.634
radius:10.300   the area: 333.282
width: 4.000    height: 6.000    the area: 24.000
width: 8.500    height: 3.700    the area: 31.450
This program generates 4 objects
```


2.4 Static Binding vs Dynamic Binding

For non-virtual function, the compiler selects the function that will be invoked at compile-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()
```

```
    fb(bp);   // uses BrassPlus::ViewAcct()
```

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

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

```
    ...
```

```
}
```

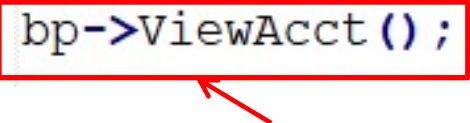
base-class object

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 **Brass::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 Inheritance and Dynamic Memory Allocation

If a **base class** uses dynamic memory allocation and redefines assignment and a copy constructor, how does that affect the implementation of the **derived class**? The answer depends on the nature of the derived class.

If the **derived class does not itself use dynamic memory allocation**, you needn't take any special steps.

If the **derived class does use `new`**, you do have to define an explicit destructor, copy constructor, and assignment operator for the derived class.


```
// Base class using DMA
class baseDMA
{
private:
    char * label;
    int rating;

public:
    baseDMA(const char * la = "null", int r = 0);
    baseDMA(const baseDMA & rs);
    virtual ~baseDMA();
    baseDMA & operator=(const baseDMA & rs);

...
};
```

base class

derived class

```
// Derived class with DMA
class hasDMA : public baseDMA
{
private:
    char * style; // use new in constructors

public:
    hasDMA(const char * s = "none", const char * la = "null", int r = 0);
    hasDMA(const char * s, const baseDMA & rs);
    hasDMA(const hasDMA & rs);
    virtual ~hasDMA();
    hasDMA & operator=(const hasDMA & rs);

...
};
```

```
baseDMA::~~baseDMA()
{
    delete [] label; // takes care of baseDMA stuff
}

hasDMA::~~hasDMA()
{
    delete [] style; // takes care of hasDMA stuff
}
```

A derived class destructor automatically calls the base-class destructor, so its own responsibility is to clean up after what the derived-class destructors do.

The data fields both in the base class and in the derived class hold pointers, which indicate they would use dynamic memory allocation.

Consider constructor:

the base-class **baseDMA** constructor

```
// baseDMA constructor
baseDMA::baseDMA(const char * la, int r)
{
    label = new char[std::strlen(la) + 1];
    std::strcpy(label, la);
    rating = r;
}
```

invoke **baseDMA** constructor

```
// hasDMA constructor
hasDMA::hasDMA(const char * s, const char * la, int r) : baseDMA(la, r)
```

```
{
    style = new char[std::strlen(s) + 1];
    std::strcpy(style, s);
}
```

invoke **baseDMA** copy constructor

```
hasDMA::hasDMA(const char * s, const baseDMA & rs) : baseDMA(rs)
{
    style = new char[std::strlen(s) + 1];
    std::strcpy(style, s);
}
```

the derived-class **hasDMA** constructor

Consider copy constructor:

```
// baseDMA copy constructor
baseDMA::baseDMA(const baseDMA & rs)
{
    label = new char[std::strlen(rs.label) + 1];
    std::strcpy(label, rs.label);
    rating = rs.rating;
}
```

the base-class **baseDMA** copy constructor

```
// hasDMA copy constructor
hasDMA::hasDMA(const hasDMA & hs) : baseDMA(hs)
{
    style = new char[std::strlen(hs.style) + 1];
    std::strcpy(style, hs.style);
}
```

the derived class **hasDMA** copy constructor

The derived class **hasDMA** copy constructor only has access to **hasDMA** data, so it must invoke the **baseDMA** copy constructor to handle the **baseDMA** share of the data.

The member initialization list passes a **hasDMA** reference to a **baseDMA** constructor. The **baseDMA** copy constructor has a **baseDMA** reference parameter, and a base class reference can refer to a derived type. Thus, the **baseDMA** copy constructor uses the **baseDMA** portion of the **hasDMA** argument to constructor the **baseDMA** portion of the new object.

Consider copy assignment operators:

```
// baseDMA copy assignment operator
baseDMA & baseDMA::operator=(const baseDMA & rs)
{
    if (this == &rs)
        return *this;

    delete [] label;
    label = new char[std::strlen(rs.label) + 1];
    std::strcpy(label, rs.label);
    rating = rs.rating;

    return *this;
}
```

the base-class **baseDMA** copy assignment operator

```
// hasDMA copy assignment operator
hasDMA & hasDMA::operator=(const hasDMA & hs)
{
    if (this == &hs)
        return *this;

    baseDMA::operator=(hs); // copy base portion
    delete [] style; // prepare for new style
    style = new char[std::strlen(hs.style) + 1];
    std::strcpy(style, hs.style);

    return *this;
}
```

the derived-class **hasDMA** copy assignment operator
Because **hasDMA** uses dynamic memory allocation, it needs an explicit copy assignment operator. Being a **hasDMA** method, it only has direct access to **hasDMA** data.

An explicit assignment operator for a derived class also has to take care of copy assignment for the inherited base class **baseDMA** object. You can accomplish this by explicitly calling the base class copy assignment operator.

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 lenght:4    width:5 height:6
Cube area:108    volume:120
Sphere radius:7.9    area:783.87    volume:2064.19
2 objects are created.
```