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 Polymorphism

2.3 Static and Dynamic binding

2.4 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.

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

```cpp
//  tabletenn1.h   -- a table-tennis base class

#ifndef TABTENN1_H_
#define TABTENN1_H_
#include <string>

using std::string;

// simple base class
class TableTennisPlayer
{
private:
    string firstname;
    string lastname;
    bool hasTable;

public:
    TableTennisPlayer(const string& fn = "none", const string& ln = "none", bool ht = false);
    void Name() const;
    bool HasTable() const { return hasTable; }
    void ResetTable(bool v) { hasTable = v; }

};

// simple derived class
class RatePlayer : public TableTennisPlayer
{
private:
   unsigned int rating;

public:
   RatePlayer(unsigned int r = 0, const string& fn = "none", const string& ln = "none", bool ht = false);
   RatePlayer(unsigned int r, const TableTennisPlayer& tp);
   unsigned int Rating() const { return rating; }
   void ResetRating(unsigned int r) { rating = r; }

};

#endif
```

base class

data in base class

derived class

new data in derived class

```cpp
// tabbletenn1.cpp -- simple base-class methods

#include <iostream>
#include "tabletenn1.h"
```

base-class constructor

```cpp
TableTennisPlayer::TableTennisPlayer(const string& fn, const string& ln, bool ht) : firstname(fn),lastname(ln), hasTable(ht)  { }
```

member initializer list syntax

```cpp
TableTennisPlayer::TableTennisPlayer(const string& fn, const string& ln, bool ht)
{
    firstname = fn;
    lastname = ln;
    hasTable = ht;
}
```

Both constructors do the same things, but the latter approach has the effect of first calling the default string constructor for firstname and then invoking the string assignment operator to reset firstname to fn. Whereas the member initializer list syntax saves a step by just using the string copy constructor to initialize firstname to fn.

NOTE：

- This form(member initializer list syntax) 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.

```cpp
// tabbletenn1.cpp -- simple base-class methods

#include <iostream>
#include "tabletenn1.h"

TableTennisPlayer::TableTennisPlayer(const string& fn, const string& ln, bool ht) :
    firstname(fn),lastname(ln), hasTable(ht) {     }

void TableTennisPlayer::Name() const
{
    std::cout << lastname << ", " << firstname;
}

//derived-class methods
//RatedPlayer methods
RatePlayer::RatePlayer(unsigned int r, const string& fn, const string& ln, bool ht) : TableTennisPlayer(fn, ln, ht)
{
    rating = r;
}

RatePlayer::RatePlayer(unsigned int r, const TableTennisPlayer& tp) : TableTennisPlayer(tp), rating(r)
{
}
```

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

derived-class constructor

base-class constructor

derived-class copy constructor

invoke the base-class copy constructor

The base class didn't define a copy constructor, but the compiler automatically generates a copy constructor which does memberwise copying, is fine because the class doesn't directly use dynamic memory allocation.

```cpp
// usett1.cpp -- using base class and derived class
#include <iostream>
#include "tabletenn1.h"

int main()
{
    using std::cout;
    using std::endl;

    TableTennisPlayer player1("Tara", "Boomdea", false);
    RatePlayer rplayer1(1140, "Mallory", "Duck", true);

    rplayer1.Name();      //derived object uses base method

    if (rplayer1.HasTable())
    {
        cout << ": has a table.\n";
    }
    else
        cout << ": hasn't a table.\n";

    player1.Name();      //base object uses base method

    if (player1.HasTable())
        cout << ": has a table";
    else
        cout << ": hasn't a table.\n";

    cout << "Name:";
    rplayer1.Name();
    cout << "; Rating: " << rplayer1.Rating() << endl;

    // initialize RatedPlayer using TableTennisPlayer object
    RatePlayer rplayer2(1212, player1);
    cout << "Name:";
    rplayer2.Name();
    cout << "; Rating: " << rplayer2.Rating() << endl;

    return 0;
}
```

create a base object

create a derived object, invoking base class constructor and then invoking derived class constructor

```
Duck, Mallory: has a table.
Boomdea, Tara: hasn't a table.
Name:Duck, Mallory; Rating: 1140
Name:Boomdea, Tara; Rating: 1212
```

create a derived object, invoking base class copy constructor and then invoking derived class copy constructor

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**.

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(const TableTennisPlayer & rt)
{
    using std::cout;
    cout << "Name: ";
    rt.Name();
    cout << "\nTable: ";
    if (rt.HasTable())
        cout << "yes\n";
    else
        cout << "no\n";
}
```

base-class object as argument

derived-class object as argument

```
TableTennisPlayer player1("Tara", "Boomdea", false);
RatedPlayer rplayer1(1140, "Mallory", "Duck", true);
Show(player1);   // works with TableTennisPlayer argument
Show(rplayer1);  // works with RatedPlayer argument
```

```
Name:Boomdea, Tara
Table:no
Name:Duck, Mallory
Table:yes
```

4. A similar relationship would hold for a function with a pointer-to-base-class formal parameter; it could be used with either the address of a base-class object or the address of a derived-class object as an argument.

base-class pointer

```
void Wohs(const TableTennisPlayer * pt);  // function with pointer parameter
...
TableTennisPlayer player1("Tara", "Boomdea", false);
RatedPlayer rplayer1(1140, "Mallory", "Duck", true);
Wohs(&player1);    // works with TableTennisPlayer * argument
Wohs(&rplayer1);   // works with RatedPlayer * argument
```

base-class object as argument

derived-class object as argument

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 Inheritence		
	Public	Protected	Private
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Not accessible (Hidden)	Not accessible (Hidden)	Not accessible (Hidden)

2.2 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

```cpp
// 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

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.

```cpp
// 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);
}
```

```cpp
//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

```cpp
#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);
}
```

```cpp
// 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

```cpp
// 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);
}
```

```cpp
// 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.

```cpp
// 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;
}
```

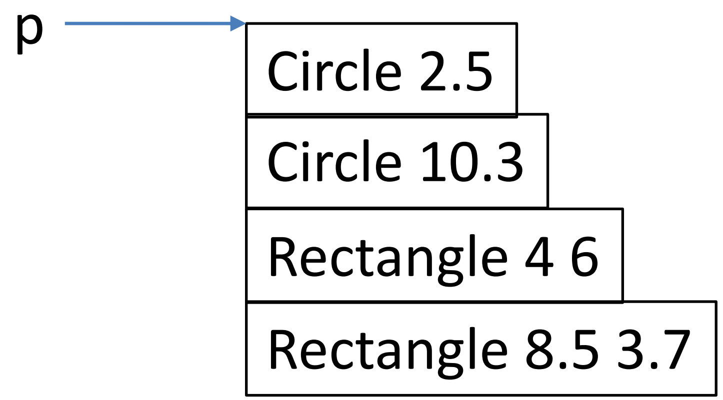
```
radius:5.000    the area: 78.538
width: 2.000    height: 6.000   the area: 12.000
This program generates 2 objects
```

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 on type of object with a single array.

```cpp
// 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();

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

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

    return 0;
}
```

polymorphism




```
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
```

2.3 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**.

```
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, 10000.0);
    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**.

Destructors

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

For example, suppose **Employee** is a base class and **Singer** is a derived class that adds a **char *** member that points to memory allocation by **new**. Then, when a **Singer** expires, it's vital that the **~Singer()** destructor be called to free that memory. Consider the following code:

```
Employee * pe = new Singer; // legal because Employee is base for Singer
...
delete pe;                  // ~Employee() or ~Singer()?
```

If the destructors are not virtual, the delete statement invokes the ***~Employee()*** destructor. This frees memory pointed to by the ***Employee*** components of the ***Singer*** object but not memory pointed to by the new class members. However, if the destructors are virtual, the same code invokes the ***~Singer()*** destructor, which frees memory pointed to by the ***Singer*** component, and then calls the ***~Employee()*** destructor to free memory pointed to by the ***Employee*** component.

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**.

The example of Method Overloading

```
Class Add
{
    int sum(int a, int b)
    {
      return a + b;
    }
   int sum(int a)
   {
     return a + 10;
    }
}
```

The example of Method Overriding

```
Class A  // Super Class
{
  void display(int num)
   {
      print num ;
    }
}
//Class B inherits Class A
Class B //Sub Class
{
  void display(int num)
   {
      print num ;
    }
}
```


Polymorphism
Compile Time
Overloading
Run Time
Overriding

2.4 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 * l = "null", int r = 0);
    baseDMA(const baseDMA & rs);
    virtual ~baseDMA();
    baseDMA & operator=(const baseDMA & rs);
...
};
```

base class

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

derived class

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

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

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 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::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 only has accesse to **hasDMA** data, so it must invoke the **baseDMA** copy constructor to handle the **baseDMA** share of the data.

The member initializer 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 assignment operators:

```
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** 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;
}
```

Because **hasDMA** uses dynamic memory allocation, it needs an explicit 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 assignment for the inherited base class **baseDMA** object. You can accomplish this by explicitly calling the base class assignment operator.