Ch-09

类和对象的使用

主要内容

- 构造函数
- 析构函数
- 调用构造函数和析构函数的顺序
- 对象数组
- 对象指针
- 共用数据的保护
- 对象的动态建立和释放
- 对象的赋值和复制
- 静态成员
- 友元
- 类模板

9.1 构造函数与对象的初始化

9.1.1 什么是构造函数

构造函数在创建对象时，由系统自动调用，分配对象存储空间并使用设定的值来初始化对象。

构造函数的特点：

1. 构造函数是成员函数，函数体可写在类体内，也可写在类体外；
2. 构造函数的函数名必须与类名相同；
3. 构造函数没有函数类型，也不能有返回值；
4. 构造函数可以没有参数，也可以有一个或多个参数；
5. 构造函数可以重载；
6. 程序中不能直接调用构造函数，系统会在创建对象时自动调用构造函数。
7. 如果用户没有定义构造函数，编译系统会自动生成一个无参默认构造函数且函数体为空。默认构造函数是指不使用实参即可调用的构造函数，一个类中只能有一个默认构造函数。

9.1.2 对象的初始化

对象的初始化是有由构造函数完成的，主要包括对象存储空间的分配和对象成员数据的初始化（按照成员数据在类中的声明顺序初始化）。

□ 用自动生成的默认构造函数初始化对象

自动生成的默认构造函数没有参数且函数体为空，因此只能完成对象存储空间的分配，而没有对成员数据的初始化。

```
class Student {
private:
        int   num;
        string  name;
        char sex;
public:
        void  setdata();
        void  display();
};
Student std;  //对象定义
```

□ 用定义的构造函数的初始化对象

1、定义无参构造函数初始化对象

- 分配对象存储空间
- 对象成员数据的初始化在函数体中完成

```
class Student {
private:
        int   num;
        string  name;
        char sex;
public:
        Student();
        void  setdata();
        void  display();
};
```

```
Student::Student() {
        num = 0;
        name = "\0";
        sex = '\0';
}

Student std;  //对象定义
```

2、定义带参数构造函数的初始化对象

- 分配对象存储空间
- 使用参数值初始化对象成员数据
  - ① 可以在函数体中赋值
  - ② 可以使用初始化列表

②

```
class Student {
private:
        int   num;
        string  name;
        char sex;
public:
        Student(int a, string b, char c): num(a), name(b), sex(c) {};
        void   setdata();
        void  display();
};
```

①

```
class Student {
private:
        int   num;
        string  name;
        char sex;
public:
        Student(int a, string b, char c);
        void  setdata();
        void  display();
};

Student::Student (int a, string b, char c)
 {
        num = a;
        name = b;
        sex = c;
}
```

Student std(1, “Li Ming”, ‘M’); //对象定义

3、定义有默认参数值的构造函数初始化对象

- 分配对象存储空间
- 如果用户指定实参则用指定的实参初始化对象成员数据；否则，则使用默认参数值

  注：默认参数值应写在函数声明中，且实参对形参的赋值必须从左向右依次对应。

```
class Student {
private:
        int   num;
        string  name;
        char sex;
public:
        //①所有参数都有默认值
        Student(int a=0, string b="Wang", char c='M'): num(a), name(b), sex(c) {};
        // ②部分参数有默认值，必须从右向左依次指定
        //Student(int a, string b, char c='M'): num(a), name(b), sex(c) {};
        void   setdata();
        void  display();
};
```

```
//对象定义
Student std(1, "Li Ming", 'M');    //①②
Student std(1, "Li Ming");    //①②
Student std;    //①
```

9.1.3 构造函数的重载

一个类可以有多个同名的构造函数，但要求参数的个数、参数的类型各不相同。

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

class Student {
private:
    int    num;
    string  name;
    char sex;
public:
    Student();
    Student(int a, string b, char c): num(a), name(b), sex(c) {};
    void  setdata();
    void  display();
};

Student::Student()
{
    num = 0;
    name = "A";
    sex = 'M';
}
```

```cpp
void Student::display()
{
    cout << num << endl;
    cout << name << endl;
    cout << sex << endl;
}

int main()
{
    Student std1;
    std1.display();

    Student std2(2, "Ming", 'F');
    std2.display();

    return 0;
}
```

◆ **为了避免歧义**

1、若定义了全部形参带默认值的构造函数后，不能再定义重载构造函数；反之亦然。

2、不要同时使用重载构造函数和带默认值的构造函数。

① Student();

② Student(int a, string b);

③ Student(int a=0, string b="Wang", char c='M');

Student std(1, “Hu”); //调用②还是③？

Student std; //调用①还是③？

④ Student(int a);

⑤ Student(int a, string b ="Wang",);

⑥ Student(int a, string b="Wang", char c='M');

Student std(1); //调用④或⑤？

Student std(1, “Li”); //调用⑤或⑥？

9.2 析构函数

9.2.1 什么是析构函数

析构函数在对象的生命周期结束时，由系统自动调用，先执行函数体，然后收回对象占用的存储空间（即按照成员数据初始化顺序依次进行销毁）。

析构函数的特点：

1. 析构函数是成员函数，函数体可写在类体内，也可写在类体外；
2. 析构函数的名字在类名前加 ~ 字符；
3. 析构函数没有函数类型，也不能有返回值；
4. 析构函数没有参数；
5. 一个类中只能定义一个析构函数；
7. 析构函数在对象存在的函数体结束时或使用delete运算符释放new运算符创建的对象时被系统自动调用；
8. 如果用户没有定义析构函数，编译系统会自动生成一个默认析构函数且函数体为空。

```
class Student {
private:
        int   num;
        string  name;
        char sex;
public:
        Student(int a, string b, char c): num(a), name(b), sex(c) {};
        ~Student();
        void   setdata();
        void  display();
};

Student::~Student()
{
        cout << "destrunctor is called" << endl;
}
```

9.2.2 调用构造函数和析构函数的顺序

- 对于一个对象，创建时调用构造函数，生命期终止时调用析构函数；
- 对于多个对象，相同存储类别情况下是先构造的后析构，后构造的先析构；但若存储类别不同，则需要根据生命期确定何时进行析构。

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

class Student {
private:
    int    num;
    string  name;
    char sex;
public:
    Student(int a, string b, char c): num(a), name(b), sex(c) {
        cout << this->name << "'s constructor" << endl;
    };
    ~Student();
    void  setdata();
    void  display();
};
```

```
Li's constructor
1
Li
M
Ming's constructor
2
Ming
F
Ming's destrunctor
Li's destrunctor
```

```cpp
Student::~Student()
{
    cout << this->name << "'s destrunctor" << endl;
}

void Student::display()
{
    cout << num << endl;
    cout << name << endl;
    cout << sex << endl;
}

int main()
{
    Student std1(1, "Li", 'M');;
    std1.display();

    Student std2(2, "Ming", 'F');
    std2.display();

    return 0;
}
```

9.3 对象数组

```
Student  std[3] = {
    Student(1001, “Li”, ‘M’),
    Student(1002, “Wang”, ‘F’),
    Student(1003, “Hu”, ‘F’)
}; //对于每个数组元素，分别用实参调用构造函数进行初始化
```

9.4 对象与指针

9.4.1 指向对象的指针

- 对象在内存中的存储空间首地址称为对象指针。
- 对于具体的对象，可以通过&运算符获得对象指针.

&对象名

- 要用与对象类型相同的指针变量保存对象指针。

```
Student std;

Student *pStd = &std;
```

9.4.2 指向对象成员的指针

分为指向成员数据的指针和指向成员函数的指针。

- 指向成员数据的指针

```
int *p = &std.num;    //num是对象std的int型成员数据
cout << *p << endl;    //输出std.num的值
```

- 指向成员函数的指针

```
数据类型 (类名::*变量名 )(形参表);
```

其中数据类型、类名和形参表都要与成员函数相同。

例如：//display成员函数的定义是void Student::display() { … }

```
void (Student::*p)() = &Student::display();
```

或 void (Student::*p)(); p = &Student::display();

```
(std.*p)();    //通过成员函数的指针调用display函数
```

9.4.3 this指针

this是一个指向当前对象的常量指针，它的值是当前对象的地址。

this指针是成员函数的一个隐含参数，只能在成员函数中使用。当调用成员函数时，编译系统把对象的地址作为实参传给this，这样在成员函数内部就可以通过它使用当前对象。

通过this实现了用同一个成员函数访问不同对象的数据成员。

```
Student::Student (int a, string b, char c)
{
        this->num = a;
        this-> name = b;
        this-> sex = c;
}
```

```
void Student::display()
{
        cout << this-> num << endl;
        cout << this-> name << endl;
        cout << this-> sex << endl;
}
```

9.5 共用数据的保护

如果既希望数据在一定范围内共享，又不愿它被随意修改，从技术上可以把数据指定为只读型的。C++提供const手段，将数据、对象、成员函数指定为常量，从而实现了只读要求，达到保护数据的目的。

9.5.1 常对象

口 定义格式：

const 类名 对象名(实参表)；　　或　　类名 const 对象名(实参表)；

若把对象定义为常对象，则对象中的成员数据默认就是常变量（但成员函数并不一定是const的），因此在定义常对象时必须带实参作为成员数据的初值，在程序中不允许修改常对象的成员数据值。

说明：（1）常对象只能调用常成员函数，而不能调用该对象的普通成员函数（除了构造函数和析构函数外），常成员函数是常对象唯一的对外接口，因此在常对象中一定要有常成员函数；（2）常成员函数可以访问常对象中的数据成员，但不允许修改。

□ 可变数据成员

如果修改常对象中的某个数据成员，C++提供了指定可变数据成员的方法：

mutable 类型 数据成员；

在定义数据成员时加mutable后，将数据成员声明为可变的数据成员，就可以用const成员函数修改它的值。

9.5.2 对象的常成员

口 常成员数据

常成员数据是用const声明的成员数据，但是常成员数据只能使用构造函数的参数初始化列表进行初始化。

```
const   int  num;
Student::Student(int n) { num = n; }  //错误，不能对常成员数据赋值
```

应该写成：

```
Student::Student(int n):num(n) { }  //正确，通过参数初始化列表进行初始化
```

口 常成员函数

定义格式：

类型　函数名(形参表)　const

const是函数类型的一部分，在声明函数原型和定义函数时都要使用const关键字。

常成员函数不能修改对象的成员数据，也不能调用该类的非const成员函数，从而保证了在常成员函数中不会修改成员数据的值。

如果一个对象被说明为常对象，则通过该对象只能调用它的常成员函数。

数据成员	非 const 成员函数	const 成员函数
非 const 的数据成员	可以引用,也可以改变值	可以引用,但不可以改变值
const 数据成员	可以引用,但不可以改变值	可以引用,但不可以改变值
const 对象的数据成员	不允许引用和改变值	可以引用,但不可以改变值

9.5.3 指向对象的常指针

口 定义格式：

类名 *const 指针变量名 = 对象地址

例：Student std1(1, "Ming", 'F'), std2;

Student *const p1 = &std1;

p1 = &std2; // 错误，p1不能修改

说明：（1）指向对象的常指针，在程序运行中始终指向一个对象，但它所指对象的成员数据值可以修改；（2）往往用常指针作为函数的形参，目的是不允许在函数中修改指针变量的值，让它始终指向原来的对象。

9.5.4 指向常对象的指针变量

□ 指向常变量的指针变量

const 类型 *指针变量名;

例如：const int *ptr;

说明：（1）常变量只能用指向常变量的指针变量指向它；（2）指向常变量的指针变量也可以指向非const变量，但不能通过指针变量修改它的值；（3）函数形参是指向非const变量的指针，则实参也必须是指向非const变量的指针。

形参	实参	是否合法	可否改变指针指向的变量的值
指向非const型变量的指针	非const变量的地址	合法	可以
指向非const型变量的指针	const变量的地址	非法	—
指向const型变量的指针	const变量的地址	合法	不可以
指向const型变量的指针	非const变量的地址	合法	不可以

□ 指向常对象的指针变量

- 常对象只能用指向常对象的指针变量指向它；
- 如果用指向常对象的指针指向一个非const对象，则不可以通过该指针修改这个对象；

  例如：Student std(1, "Ming", 'F'); const Student *p = &std;

  std.num = 2; //合法

  (*p).num = 2; //非法，不可以通过p修改对象的值

- 指向常对象的指针最常用于函数的形参，目的是保护形参指针指向的对象不被修改；

  例如：void set(const Student *);

- 不能通过指向常对象的指针修改对象的值，但是指针本身是可以改变的。

  例如： Student std1, std2; const Student *p = &std1; p = &std2;

9.5.5 对象的常引用

用引用形参时，形参变量与实参变量是同一个变量，在函数内修改引用形参也就是修改实参变量。

如果用引用形参又不想让函数修改实参，可以使用常引用机制。

□ 定义格式

const 类名 &对象名

例如：set(std);

```
void set(Student &t);    //对t的修改会改变std的值
void set(const Student &t);    //不允许通过t修改std的值
```

9.5.6 const数据的小结

形式	含义
const Student std; 或 Student const std;	std是常对象，其值在任何情况下都不能改变
void Student::set() const;	Set是常成员函数，可以引用，但不能修改成员数据
Student *const p;	P是指向Student类对象的常指针变量，p的值不能改变
const Student *p;	P是指向Student类常对象的指针变量，不可以通过p修改对象的值
const Student &t = std;	Std是Student类对象std的引用，二者指向同一存储空间，std的值不能改变

9.6 对象的动态建立和释放

C++提供了new和delete运算符，实现内存的动态分配和回收。它们也可以用来动态建立对象和释放对象。

□ 动态建立对象

类名 *指针变量 = new (相同)类名；

功能：在堆里分配存储空间，存储指定类的一个对象，并调用构造函数进行对象初始化。如果分配成功，将返回分配的存储空间的首地址；如不成功，返回0。

例如：Student *pt; pt = new Student; //若内存分配成功，就可通过指针变量pt访问对象成员。

```
cout << pt->display() << endl;
```

说明：通过new动态建立的对象是一个无名对象，无法通过对象名进行访问，因此pt成为访问该对象的唯一途径，若pt的值被改变则该对象将无法访问和释放。

口 动态建立的对象的释放

delete 指针变量;

功能：释放指针变量指向的存储空间，该存储空间必须是由new动态建立的。

例如：delete pt;

说明：由于该指针变量指向的是一个对象，因此在释放存储空间之前，系统会自动调用对象的析构函数，进行对象的清理。

9.7 对象的赋值和复制

9.7.1 对象的赋值

同类对象之间可以相互赋值。

例如： Student std1(1, "Ming", 'F'), std2;

std2 = std1;

说明：（1）对象间的赋值是通过赋值运算符的重载实现的；（2）对象赋值其实是成员数据（不包括成员函数）的对应复制；（3）参与赋值的对象的成员数据不可以是动态建立的数据。

9.7.2 对象的复制

用一个已有的对象复制出多个相同的对象。

形式1：Student std2(std1);　//用对象std1复制出std2

形式2：Student std2 = std1;

◆ 复制对象产生新对象时，由系统自动调用拷贝构造函数进行新对象的初始化。

□ 拷贝构造函数

```
Student::Student(const Student &t) {
        num = t.num;
        name = t.name;
        sex = t.sex;
}
```

说明：

（1）拷贝构造函数只有一个参数，这个参数是const的同类对象的引用；（2）在拷贝构造函数的函数体中用实参对象的成员数据值给新对象的对应成员数据进行赋值；（3）如果程序中未定义拷贝构造函数，编译系统将提供默认的拷贝构造函数，完成对象的初始化。

口 使用拷贝构造函数的三种情况：

- **需要产生一个对象在某个状态时的多个副本**
- **对象作为函数的参数**

  void set(Student t) {…}

  int main() { Student std(1, "Ming", 'F'); set(std); return 0; }
- **对象作为函数的返回值**

  Student f() { Student std(1, “Ming”, ‘F’); return std; }

  int main() { Student std2; std2=f(); return 0; }

  说明：return std时会调用拷贝构造函数产生一个新的临时对象，然后将它赋值给std2。

9.8 静态成员

静态成员属于类，而不属于某一个对象，可以实现在同类的多个对象之间的数据共享。

9.8.1 静态成员数据

```
class Student {
private:
        …
        static string position;    //position is student
public:
        Student(int a, string b, char c): num(a), name(b), sex(c) {};
        ~Student();
        …
};
```

说明：

（1）由于一个类的所有对象共享静态成员数据，所以静态成员数据的初始化既不能用构造函数，也不能用参数初始化列表，只能在类外专门对其初始化（不需要static修饰），格式如下：

数据类型 类名::静态数据成员名 = 初值;

如果程序未对静态数据成员赋初值，则编译系统自动赋初值0。

（2）即可以用对象名引用静态成员，也可以用类名引用静态成员。

（3）静态成员数据在对象外单独开辟内存空间，只要在类中定义了静态成员，即使不定义对象，系统也为静态成员分配内存空间，可以被引用。

（4）在程序开始时为静态成员分配内存空间，直到程序结束才释放内存空间。

9.8.2 静态成员函数

C++提供静态成员函数，是为了用来访问静态数据成员，而不是为了实现数据共享。

类中的非静态成员函数可以访问类中所有成员数据；而静态成员函数可以直接访问类的静态成员数据，不能直接访问非静态成员数据，因为静态成员函数没有this指针，只能通过对象名来访问。

- 静态成员函数定义格式：

static 类型 成员函数(形参表) {...}

- 调用公有静态成员函数格式：

类名::成员函数(实参表);

引用方式	静态成员数据	非静态成员数据
静态成员函数	成员名	对象名.成员名
非静态成员函数	成员名	成员名

例9.1 静态成员

```
#include <iostream>
using namespace std;
class Student
{
        public:
                Student(int n, int a, float s):num(n), age(a), score(s) {}
                void total();
                static float average();
        private:
                int num;
                int age;
                float score;
                static float sum;
                static int count;
};

float Student::sum = 0;
int Student::count = 0;

void Student::total()
{
        sum += score;
        count++;
}
float Student::average()
{
        return sum/count;
}
```

```
int main()
{
        Student std[2]= {
                Student(1001, 18, 70),
                Student(1002, 20, 96)
        };
        for(int i=0; i<2; i++) {
                std[i].total();
        }
        cout << "The average score is " << Student::average() << endl;
        return 0;
}
```

9.9 友元（friend）

C++通过友元可以在类的外部访问类的私有成员。

友元可以是不属于任何类的普通函数，也可以是另一个类的成员函数，还可以是另一个类（这个类中的所有成员函数都成为友元函数）。

友元是一种破坏数据封装和数据隐藏的机制。

为了确保数据的完整性，及数据封装与隐藏的原则，建议尽量不使用或少使用友元。

9.9.1 友元函数

□ 友元函数的声明

friend 类型 类A::成员函数x(类B &对象); //在B类中声明A类的成员函数x为友元

friend 类型 函数y(类B &对象); //在B类中声明普通函数y为友元

功能：友元函数x和y用B类对象的引用作为形参即可访问B类中的私有成员。

□ 友元函数访问对象成员

对象名. 成员名

因为友元不是成员函数，它不属于类，所以它访问成员时必须冠以对象名。

定义友元函数时形参通常定义为对象引用，这样在友元函数内就能访问实参对象了。

□ 将普通函数声明为友元，如例9.2所示

```
#include <iostream>
using namespace std;
class Time
{
        public:
                Time(int a, int b, int c):
                        hour(a), minute(b), second(c) {}
                friend void display(Time &);
        private:
                int hour;
                int minute;
                int second;
};
void display(Time &t)
{
        cout << t.hour << ":" <<
        t.minute <<  ":" << t.second
        << endl;
}

int main()
{
        Time t(10, 13, 56);
        display(t);
        return 0;
}
```

□ 将成员函数声明为友元，如例9.3所示

```
#include <iostream>
#include "9.3.h"
using namespace std;
class Date;   //提前引用声明
class Time {
        public:
                Time(int, int, int);
                void display(Date &);
        private:
                int hour;
                int minute;
                int second;
};

class Date {
        public:
                Date(int, int, int);
                friend void Time::display(Date &);
        private:
                int month;
                int day;
                int year;
};
```

```
Time::Time(int h, int m, int s)
{
        hour = h;
        minute = m;
        second = s;
}

void Time::display(Date &d)
{
        cout << d.month << "/" << d.day
                << "/" << d.year << endl;
        cout << hour << ":" << minute
                << ":" << second << endl;
}

Date::Date(int m, int d, int y)
{
        month = m;
        day = d;
        year = y;
}
```

```
int main()
{
        Time t(10, 13, 56);
        Date d(12, 25, 2016);
        t.display(d);
        return 0;
}
```

9.9.2 友元类

C++允许将一个类声明为另一个类的友元。假定A类是B类的友元类，A类中所有的成员函数都是B类的友元函数。

在B类中声明A类为友元的格式：

```
friend A;
```

实际中一般并不把整个类声明友元，而只是将确有需要的成员函数声明为友元函数。

对于友元类：

- 友元关系是单向的，不是双向的；
- 友元关系不能传递。

9.10 类模板

对于功能相同而只是数据类型不同的函数，不必须定义出所有函数，我们定义一个可对任何类型变量操作的函数模板。对于功能相同的类而数据类型不同，不必定义出所有类，只要定义一个可对任何类进行操作的类模板。

□ 类模板的定义

template <class 类型参数名>

class 类模板名

{ … … }

说明：

- 类型参数名：按标识符取名。如有多个类型参数，每个类型参数都要以class为前导，两个类型参数之间用逗号分隔，如template <class T1, class T2>。
- 类模板名：按标识符取名。
- 类模板{ …}内定义成员数据和成员函数的规则：用类型参数作为数据类型，用类模板名作为类。

例9.4

```
template<class numtype>
class Compare {
  public:
      Compare(numtype a, numtype b) {x=a; y=b;}
      numtype max() { return (x>y)?x:y; }
      numtype min() { return (x<y)?x:y; }
  private:
      numtype x,y;
};
```

◆ **如果在类模板外定义成员函数，则应使用以下定义形式：**

```
template  <class  类型参数名>
函数类型  类模板名<类型参数名>::成员函数名(形参表) { … }
```

例如：

```
template<class numtype>
numtype Compare<numtype>::max() {  return (x>y)?x:y; }
```

- **使用类模板时，对象定义的格式：**

类模板名 <实际类型名> 对象名;

类模板名 <实际类型名> 对象名(实参表);

例如用类模板Compare定义对象：

```
Compare <int> cmp(4, 7);
```

在编译时，编译系统用 int 取代类模板中的类型参数numtype，就把类模板具体化了。

CH09 End