中国科学技术大学
University of Science and Technology of China

程序设计II

第9讲 类与对象

计算机学院 黄章进

zhuang@ustc.edu.cn

内容

- 9.1 类
- 9.2 动态内存管理
- 9.3 标准库类型
- 9.4 拷贝控制

类

- 类和对象
- 构造函数和析构函数
- 对象数组
- 组合类
- 类类型
- 友元

C++中的类

类和对象

- 类是具有相同属性和行为的一组对象的集合，它为属于该类的全部对象提供了统一的抽象描述，其内部包括属性和行为两个主要部分。
- 利用类可以实现数据的封装、隐藏、继承与派生。
- 利用类易于编写大型复杂程序，其模块化程度比C中采用函数更高。

类的声明形式

类和对象

类是一种用户自定义类型，声明形式：

```
class 类名称
{
  public:
        公有成员（外部接口）
  private:
        私有成员
  protected:
        保护型成员
};
```

成员类型

- 公有类型：在关键字public后面声明，它们是类与外部的接口，任何外部函数都可以访问公有类型数据和函数
- 私有类型：在关键字private后面声明，只允许本类中的函数访问，而类外部的任何函数都不能访问
  - 如果紧跟在类名称的后面声明私有成员，则关键字private可以省略。
- 保护类型：在关键字protected后面声明与private类似，其差别表现在继承与派生时对派生类的影响不同

使用class或struct

- 可以使用class或struct来定义一个类，区别在于它们的默认访问权限不同
  - 使用struct关键字，则定义在第一个访问说明符之前的成员是public的
    - ✓一般地，若定义的类的所有成员是public时，使用struct
  - 使用class关键字，则定义在第一个访问说明符之前的成员是private的
    - ✓若定义的类有private成员时，使用class

类的成员举例

类和对象

```
class Clock {
  public:
     void setTime(int newH, int newM,
                  int newS);
     void showTime();
  private:
    int hour, minute, second;
};
```

成员函数

成员数据

```
void Clock::setTime(int newH, int newM, int newS) {
   hour = newH;
   minute = newM;
   second = newS;
}

void Clock::showTime() {
   cout << hour << ":" << minute << ":" <<    second;
}
```

成员数据和成员函数

- 成员数据：与一般的变量声明相同，但需要将它放在类的声明体中
  - 通常声明为私有成员
- 成员函数：在类中说明原型，可以在类外给出函数体实现，并在函数名前使用类名加以限定。也可以直接在类中给出函数体，形成内联成员函数。
  - 允许声明重载函数和带默认实参值的函数

内联成员函数

- 为了提高运行时的效率，对于较简单的函数可以声明为内联形式
- 内联函数体中不要有复杂结构（如循环语句和switch语句）
- 在类中声明内联成员函数的两种方式：
  - 将函数体放在类的声明中
  - 使用inline关键字

内联成员函数举例(一)


```
class Point {
  public:
    void init(int initX, int initY) {
      x = initX;
      y = initY;
    }
    int getX() { return x; }
    int getY() { return y; }

  private:
    int x, y;
};
```

内联成员函数举例(二)


```
class Point {
  public:
    void init(int initX, int initY);
    int getX();
    int getY();

  private:
    int x, y;
};
```

```
inline void Point::init(int initX,int initY) {
    x = initX;
    y = initY;
}

inline int Point::getX() {
    return x;
}

inline int Point::GetY() {
    return y;
}
```

对象

类和对象

- 类的对象是该类的某一特定实体，即类类型的变量。
- 声明形式：

  类名 对象名;

- 例：

  Clock myClock;

类中成员的访问方式

类和对象

- 类中成员互访
  - 直接使用成员名
- 类外访问
  - 使用“对象名.成员名”方式访问 public 属性的成员
    - ✓对象名.数据成员名
    - ✓对象名.函数成员名(参数表)

类的应用举例

类和对象

```
#include<iostream>
using namespace std;
class Clock {
      ......//类的声明略
};
//......类的实现略
int main() {
    Clock myClock;
    myClock.setTime(8, 30, 30);
    myClock.showTime();
    return 0;
}
```

const成员函数

- 把const关键字放在成员函数的参数列表之后，声明一个常成员函数：

  类型说明符 函数名(参数表) const;

  - 这里，const是函数类型的一个组成部分，因此在实现部分也要带const关键字。
  - const关键字可以用于参与对重载函数的区分
- 常量成员函数不能改变调用它的对象的数据成员
- 常量对象，以及常量对象的引用或指针都只能调用常量成员函数

常成员函数举例

类和对象

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

class R {
public:
    R(int r1, int r2) : r1(r1), r2(r2) { }
    void print();
    void print() const;
private:
    int r1, r2;
};
```

```
void R::print() {
    cout << r1 << ":" << r2 << endl;
}
void R::print() const {
    cout << r1 << “:" << r2 << endl;
}
int main() {
    R a(5,4);
    a.print(); //调用void print()
    const R b(20,52);
    b.print(); //调用void print() const
    return 0;
}
```

this指针

- 成员函数通过一个额外的隐式形参this来访问调用它的那个对象
  - 定义名为this的形参或变量是不合法的
- 编译器把this指针初始化成调用成员函数的对象的地址
  - 调用myClock.showTime()等价于如下伪代码：

```
// pseudo-code illustration of how a call to a member function is translated
Clock::showTime(&myClock)
```

this指针

类和对象

- 成员函数体内任何对类成员的直接访问都可以看做this指针的隐式引用

```
void Clock::showTime() const{
   cout << this->hour << ":" << this->minute << ":" <<
      this->second;
}
```

- 默认情况下，this的类型是指向类类型非常量对象的常量指针：Clock *const
- 常量成员函数的this是指向常量的常量指针：const Clock *const

静态成员

类和对象

- 有时候，类需要一些成员来表征自身的类属性，而不是其个体对象的属性
  - 实现同一个类的不同对象之间的数据和函数共享
  - 例如，一个银行账户类可能需要一个表示当前基准利率的数据成员

声明静态成员

静态成员

- 通过在成员声明前加关键字static来表示类属性

```
class Account {
public:
    void calculate() { amount += amount * interestRate; }
    static double rate() { return interestRate; }
    static void rate(double);
private:
    std::string owner;
    double amount;
    static double interestRate;
    static double initRate();
};
```

声明静态成员

静态成员

- 类的静态成员存在于任何对象之外
  - 对象中不包含与静态数据成员相关的数据
- 静态成员函数不绑定到任何对象，即不包含this指针
  - 静态成员函数不能声明为const
  - 不能在静态成员函数体内使用this指针
    - ✓限制适用于this的显式使用，以及调用非静态成员的隐式使用
    - ✓必须通过对象来访问非静态成员

使用静态成员

类的静态成员

- 在类外使用类名和作用域运算符访问静态成员

```
double r;
r = Account::rate(); // access a static member using the scope operator
```

- 也可以使用类的对象、引用或指针来访问静态成员

```
Account ac1;
Account *ac2 = &ac1;
// equivalent ways to call the static member rate function
r = ac1.rate();      // through an Account object or reference
r = ac2->rate();     // through a pointer to an Account object
```

使用静态成员

类的静态成员

- 成员函数不用通过作用域运算符就能直接使用本类的静态成员

```
class Account {
public:
    void calculate()
        { amount += amount * interestRate; }
private:
    static double interestRate;
    // remaining members as before
};
```

定义静态成员

类的静态成员

- 可以在类内也可以在类外定义静态成员函数
  - 在类外定义静态成员时，不能重复static
    - ✓static关键字只出现在类内的声明语句中

```
void Account::rate(double newRate)
{
    interestRate = newRate;
}
```

定义静态成员

类的静态成员

- 静态成员函数可以直接访问该类的静态数据和函数成员；访问非静态成员，必须通过对象名

```
class A {
public:
    static void f(A a);
private:
    int x;
};
void A::f(A a) {
    cout << x;    // 对x的引用是错误的
    cout << a.x; // 正确
}
```

定义静态成员

类的静态成员

- 静态数据成员不属于类的任何一个对象，必须在类外定义和初始化每个静态数据成员
  - 静态数据成员定义在任何函数之外，具有静态生存期

```
// define and initialize a static class member
double Account::interestRate = initRate();
```

  - 为确保只定义一次，一般把静态数据成员和类的非内联成员函数定义在同一个文件

具有静态数据、函数成员的 Point类

类的静态成员

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

class Point {      //Point类定义
public:  //外部接口
    Point(int x = 0, int y = 0) : x(x), y(y) { count++; }
    Point(Point &p);
    ~Point() {  count--; }
    int getX() { return x; }
    int getY() { return y; }
    static void showCount() {      //静态函数成员
        cout << "  Object count = " << count << endl;
    }
private: //私有数据成员
    int x, y;
    static int count;          //静态数据成员声明
};
```

```
Point::Point(Point &p) {
    x = p.x;
    y = p.y;
    count++;
}
int Point::count=0;
int main() { //主函数实现
    Point a(4,5);   //声明对象A
    cout << "Point A,“ << a.getX() << ",“ << a.getY();
    Point::showCount();     //输出对象个数
    Point b(a);      //声明对象B
    cout << "Point B,“ << b.GetX() << ",“ << b.GetY();
    b.showCount();          //输出对象个数
    return 0;
}
```

构造函数

构造函数和析构函数

- 构造函数的作用是在对象被创建时使用特定的值构造对象，或者说将对象初始化为一个特定的状态。
- 构造函数的名字和类名相同，没有返回类型
  - 允许为内联函数、重载函数、带默认实参值的函数
- 在对象创建时由系统自动调用。
- 如果程序中未声明任何构造函数时，则系统自动产生出一个隐含的参数列表为空的构造函数—默认构造函数

构造函数举例

构造函数和析构函数

```
class Clock {
public:
    Clock(int newH, int newM, int newS); // 构造函数
    void setTime(int newH, int newM, int newS);
    void showTime();
private:
    int hour, minute, second;
};
```

构造函数的实现：

```
Clock::Clock(int newH, int newM, int newS) {
    hour = newH;
    minute = newM;
    second = newS;
}
```

建立对象时构造函数的作用：

```
int main() {
    Clock c(0,0,0); //隐含调用构造函数，将初始值作为实参。
    c.showTime();
    return 0;
}
```

拷贝构造函数

构造函数和析构函数

- 拷贝构造函数是一种特殊的构造函数，其形参为本类对象的引用。

```
class 类名 {
public :
    类名(形参);   //构造函数
    类名(类名 &对象名); //拷贝构造函数
        ...
};
类名::类名(类名 &对象名) //拷贝构造函数的实现
{    函数体    }
```

拷贝构造函数举例

构造函数和析构函数

```
class Point {
public:
    Point(int xx = 0, int yy = 0) { x = xx; y = yy; }
    Point(Point& p);
    int getX() { return x; }
    int getY() { return y; }
private:
    int x, y;
};
Point::Point (Point& p) {
  x = p.x;
  y = p.y;
  cout << "Calling the copy constructor " << endl;
}
```

拷贝构造函数

- 当用类的一个对象去初始化该类的另一个对象时系统自动调用拷贝构造函数实现拷贝赋值。

```
int main() {
    Point a(1,2);
    Point b = a; //拷贝构造函数被调用
    cout << b.getX() << endl;
}
```

拷贝构造函数

- 若函数的形参为类对象，调用函数时，实参赋值给形参，系统自动调用拷贝构造函数。例如：

```
void fun1(Point p) {
    cout << p.getX() << endl;
}
int main() {
    Point a(1, 2);
    fun1(a); // 调用拷贝构造函数
    return 0;
}
```

值传递才调用拷贝构造函数，引用传递不调用

拷贝构造函数

- 当函数的返回值是类对象时，系统自动调用拷贝构造函数。例如：

```
Point fun2() {
    Point a(1, 2);
    return a; // 调用拷贝构造函数
}
int main() {
    Point b;
    b = fun2();
    return 0;
}
```

取决于编译器的实现，gcc就不调用！

隐含的拷贝构造函数

- 如果没有为类声明拷贝构造函数，则编译器自己生成一个隐含的拷贝构造函数。
- 该构造函数执行的功能：用作为初始值的对象的每个数据成员的值，初始化将要建立的对象的对应数据成员。

析构函数

构造函数和析构函数

- 完成对象被删除前的一些清理工作。
- 函数名由波浪号(~)接类名构成，没有返回值，也不接受参数
  - 不能被重载，只有唯一的析构函数
- 在对象的生存期结束的时刻系统自动调用它，然后再释放此对象所属的空间。
- 如果程序中未声明析构函数，编译器将自动产生一个隐含的析构函数。
  - 该析构函数的函数体为空，即什么也不做

构造函数和析构函数举例

构造函数和析构函数

```
#include <iostream>
using namespace std;
class Point {
public:
    Point(int xx,int yy);
    ~Point();
  //...其他函数原型
private:
    int x, y;
};
```

```
Point::Point(int xx,int yy) {
    x = xx;
    y = yy;
}

Point::~Point() {
}
//...其他函数的实现略
```

对象数组

- 声明：

  类名　数组名[元素个数];

- 访问方法：

  通过下标访问

  数组名[下标].成员名

对象数组初始化

对象数组

- 数组中每一个元素对象被创建时，系统都会调用类构造函数初始化该对象。
- 通过初始化列表赋值：

  例：

  Point a[2] = {Point(1,2), Point(3,4)};

- 如果没有为数组元素指定显式初始值，数组元素便使用默认值初始化（调用默认构造函数）

数组元素所属类的构造函数

对象数组

- 不声明构造函数，则采用默认构造函数。
- 各元素对象的初值要求为相同的值时，可以声明具有默认形参值的构造函数。
- 各元素对象的初值要求为不同的值时，需要声明带形参的构造函数。
- 当数组中每一个对象被删除时，系统都要调用一次析构函数。

对象数组应用举例

对象数组

```
// Point.h
#ifndef _POINT_H
#define _POINT_H

class Point {          //类的定义
public:       //外部接口
    Point();
    Point(int x, int y);
    ~Point();
    void move(int newX,int newY);
    int getX() const { return x; }
    int getY() const { return y; }
private:      //私有数据成员
    int x, y;
};
#endif        //_POINT_H
```

对象数组应用举例


```
// Point.cpp
#include <iostream>
#include "Point.h"
using namespace std;

Point::Point() {
    x = y = 0;
    cout << "Default Constructor called." << endl;
}

Point::Point(int x, int y) : x(x), y(y) {
    cout << "Constructor called." << endl;
}

Point::~Point() {
    cout << "Destructor called." << endl;
}

void Point::move(int newX,int newY) {
    cout << "Moving the point to (" << newX << ", " << newY << ")" << endl;
    x = newX;
    y = newY;
}
```

```
//6-3.cpp
#include "Point.h"
#include <iostream>
using namespace std;

int main() {
   cout << "Entering main..." << endl;
   Point a[2];
   for(int i = 0; i < 2; i++)
      a[i].move(i + 10, i + 20);
   cout << "Exiting main..." << endl;
   return 0;
}
```

请给出运行结果

运行结果


```
Entering main...
Default Constructor called.
Default Constructor called.
Moving the point to (10, 20)
Moving the point to (11, 21)
Exiting main...
Destructor called.
Destructor called.
```

组合类的概念

类的组合

- 类中的成员数据是另一个类的对象。
- 可以在已有抽象的基础上实现更复杂的抽象。

举例

类的组合

```
class Point {
private:
    float x, y; //点的坐标
public:
    Point(float h, float v); //构造函数
    float getX(); //取X坐标
    float getY(); //取Y坐标
    void draw(); //在(x,y)处画点
};
//...函数的实现略
```

```
class Line {
private:
    Point p1, p2; //线段的两个端点
public:
    Line(Point a, Point b); //构造函数
    void draw(void); //画出线段
};
//...函数的实现略
```

组合类的构造函数

类的组合

- 原则：不仅要负责对本类中的基本类型成员数据赋初值，也要对对象成员初始化。
- 构造函数声明形式：

类名::类名(对象成员所需的形参，本类成员形参) : 对象1(参数), 对象2(参数), ……

{ 本类初始化 }

初始化列表

初始化列表

类的组合

- 初始化列表为新创建对象的一个或多个数据成员赋初值
  - 逗号分隔的成员名列表，每个名字后紧跟括号括起来的成员初始值
  - 在初始化列表中，每个成员只能出现一次
- 如果在初始化列表中没有显式地初始化成员，则该成员将在构造函数体执行之前被默认初始化
  - 即使初始化列表是空的，对象的成员仍然会在函数体执行前被初始化
  - 默认初始化的内置类型或复合类型的数据成员的值是未定义的

初始化列表

初始化列表

- 如果成员是const、引用或者属于某种未提供默认构造函数的类类型，必须通过初始化列表来提供初值

```
class ConstRef {
public:
    ConstRef(int ii);
private:
    int i;
    const int ci;
    int &ri;
};
```

初始值有时必不可少

初始化列表

- 常量或引用成员必须被初始化

```
// error: ci and ri must be initialized
ConstRef::ConstRef(int ii)
{               // assignments:
    i = ii;   // ok
    ci = ii;  // error: cannot assign to a const
    ri = i;   // error: ri was never initialized
}
```

  - 只能通过初始化列表来初始化

```
// ok: explicitly initialize reference and const members
ConstRef::ConstRef(int ii): i(ii), ci(ii), ri(i) {  }
```

成员初始化的顺序

初始化列表

- 成员的初始化顺序与它们在类定义中出现的顺序一致
  - 最好令构造函数初始值的顺序与成员声明的顺序一致，且尽量避免使用某些成员初始化其他成员

```
class X {
    int i;
    int j;
public:
    // undefined:  i is initialized before  j
    X(int val): j(val), i(j) { }
};
```

类的组合举例（二）

类的组合

```
class Part { //部件类
public:
    Part();
    Part(int i);
    ~Part();
    void Print();
private:
    int val;
};
```

```
class Whole {
public:
    Whole();
    Whole(int i, int j, int k);
    ~Whole();
    void Print();
private:
    Part one;
    Part two;
    int date;
};
```

```
Whole::Whole() {
    date = 0;
}
Whole::Whole(int i, int j, int k)
        : two(i), one(j), date(k)
{}

//...其他函数的实现略
```


62

组合类的构造函数调用

类的组合

- 编译器自动生成的隐含的拷贝构造函数，自动调用内嵌对象的拷贝构造函数，为各个内嵌对象初始化
- 为组合类编写拷贝构造函数，需要为内嵌成员对象的拷贝构造函数传递参数

例：线段类


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

class Point {            //Point类定义
public:
        Point(int xx = 0, int yy = 0) {
                x = xx;
                y = yy;
        }
        Point(Point &p);
        int getX() { return x; }
        int getY() { return y; }
private:
        int x, y;
};

Point::Point(Point &p) {        //拷贝构造函数的实现
        x = p.x;
        y = p.y;
        cout << "Calling the copy constructor of Point" << endl;
}
```

```
//类的组合
class Line {          //Line类的定义
public:   //外部接口
    Line(Point xp1, Point xp2);
    Line(Line &l);
    double getLen() { return len; }
private:  //私有数据成员
    Point p1, p2;    //Point类的对象p1,p2
    double len;
};
//组合类的构造函数
Line::Line(Point xp1, Point xp2) : p1(xp1), p2(xp2) {
    cout << "Calling constructor of Line" << endl;
    double x = static_cast<double>(p1.getX() - p2.getX());
    double y = static_cast<double>(p1.getY() - p2.getY());
    len = sqrt(x * x + y * y);
}
//组合类的拷贝构造函数
Line::Line (Line &l): p1(l.p1), p2(l.p2) {
    cout << "Calling the copy constructor of Line" << endl;
    len = l.len;
}
```

```
//主函数
int main() {
    Point myp1(1, 1), myp2(4, 5);    //建立Point类的对象
    Line line(myp1, myp2);           //建立Line类的对象
    Line line2(line);          //利用拷贝构造函数建立一个新对象
    cout << "The length of the line is: ";
    cout << line.getLen() << endl;
    cout << "The length of the line2 is: ";
    cout << line2.getLen() << endl;
    return 0;
}
```

类类型

- 每个类定义了唯一的类型
  - 即使两个类定义的成员完全一样，也是不同类型

```
struct First {
    int memi;
    int getMem();
};
struct Second {
    int memi;
    int getMem();
};
First obj1;
Second obj2 = obj1; // error: obj1 and obj2 have different types
```

类类型

- 可以把类名直接作为类类型的名字，也可以把类名跟在struct或class关键字后面（C风格方式）

```
Point pt1;
class  Point pt1; // equivalent declaration
```

类类型

类的声明

类类型

- 可以仅声明类而暂时不定义它：

```
class Line; // declaration of the Line class
```

这种声明被称作前向声明(forward declaration)

- 在声明之后定义之前，类型Line是一个不完全类型(incomplete type)
  - 知道Line是一个类类型，但是不清楚它到底有哪些成员
  - 可以定义指向不完全类型的指针或引用，也可以声明（但不能定义）以不完全类型作为形参或者返回类型的函数

前向声明举例

类类型

```
class B;  // 前向声明
class A {
public:
    void f(B b);
};
class B {
public:
    void g(A a);
};
```

类的声明

类类型

- 创建类的对象之前必须先定义该类
  - 仅通过声明，编译器无法确定对象需要多少存储空间
  - 类也必须先被定义，才能用引用或指针访问其成员
- 类先被定义后，才能声明该类类型的数据成员
  - 一个类的数据成员的类型不能是该类自己
  - 类可以包含指向自身的指针或引用类型的数据成员
    - ✓一旦类名出现后，就认为该类已被声明

```
class Link_point {
    Point pt;
    Link_point *next;
    Link_point *prev;
};
```

前向声明注意事项

类类型

- 使用前向声明虽然可以解决一些问题，但它并不是万能的。即使使用了前向声明，但在提供一个完整的类定义之前，不能声明该类的对象，也不能在内联成员函数中使用该类的对象。

```
class Fred; //前向声明
class Barney {
    Fred x; //错误：类Fred的声明尚不完善
};
class Fred {
    Barney y;
};
```

前向声明注意事项

类类型

```
class Fred;          //前向声明

class Barney {
public:
    ……
    void method() {
        x.yabbaDabbaDo();   //错误：Fred类的对象在定义之前被使用
    }
private:
    Fred &x;   //正确：经过前向声明，可以声明Fred类的对象引用
};

class Fred {
public:
    void yabbaDabbaDo();
private:
    Barney &y;
};
```

记住：当使用前向声明时，只能使用被声明的符号，而不能涉及类的任何细节。

友元

- 友元是C++提供的一种破坏数据封装和数据隐藏的机制。
- 通过将一个模块声明为另一个模块的友元，一个模块能够引用到另一个模块中本是被隐藏的信息。
- 可以使用友元函数和友元类。
- 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。

友元函数

友元

- 友元函数是在类声明中由关键字friend修饰说明的非成员函数，在它的函数体中能够通过对象名访问 private 和 protected成员
- 作用：增加灵活性，使程序员可以在封装和快速性方面做合理选择。
- 访问对象中的成员必须通过对象名。

友元的声明

友元

- 友元声明(friend declaration)只能出现类定义的内部
  - 一般地，最好在类定义开始或结束的位置集中声明友元
- 友元不是类的成员，也不受它所在区域访问控制权限的约束
- 友元声明不是通常意义上的函数声明
  - 为了使友元对类的用户可见，通常把友元的声明(在类外)与类放在同一个头文件

使用友元函数计算两点距离

友元

```
#include <iostream>
#include <cmath>
class Point {    // Point类声明
public:          // 外部接口
    Point(int x=0, int y=0) : x(x), y(y) { }
    int getX() { return x; }
    int getY() { return y; }
    friend float dist(Point &a, Point &b);
private:         // 私有数据成员
    int x, y;
};
```

```
float dist( Point& a, Point& b) {
    double x = a.x - b.x;
    double y = a.y - b.y;
    return static_cast<float>(sqrt(x * x + y * y));
}

int main() {
    Point p1(1, 1), p2(4, 5);
    cout <<"The distance is: ";
    cout << dist(p1, p2) << endl;
    return 0;
}
```

友元类

友元

- 若一个类为另一个类的友元，则此类的所有成员都能访问对方类的私有和保护成员。
- 声明语法：将友元类名在另一个类中使用friend修饰说明。

友元类举例

友元

```
class A {
    friend class B;
public:
    void display() {
    cout << x << endl;
    }
private:
    int x;
};

class B {
public:
    void set(int i);
    void display();
private:
    A a;
};

void B::set(int i) {
   a.x = i;
}

void B::display() {
   a.display();
}
```

成员函数作为友元

友元

- 除了令整个类作为友元外，可以把其他类的成员函数作为友元
  - 当把一个成员函数声明成友元时，必须明确指出该成员函数属于哪个类
- 如果一个类想把一组重载函数声明为它的友元，它必须对这组函数中的每一个分别声明

友元声明和作用域

友元

- 类和非成员函数的声明不是必须在它们的友元声明之前
  - 当一个名字第一次出现在友元声明中时，隐式地假定该名字在当前作用域中可见
    - ✓友元本身不一定真的声明在当前作用域中
- 友元声明只影响访问权限，不是普通意义上的声明
  - 即使在类内定义友元函数，也必须在类外提供相应的声明使得该函数可见
    - ✓就算只被声明友元函数的类的成员来调用

友元声明和作用域

友元

```
struct X {
    friend void f()
    { /* friend function can be defined in the class body */ }
    X() { f(); } // error: no declaration for f
    void g();
    void h();
};
void X::g() { return f(); } // error: f hasn't been declared
void f();                   // declares the function defined inside X
void X::h() { return f(); } // ok: declaration for f is now in
scope
```

友元关系

友元

- 友元关系是不能传递的
  - B类是A类的友元，C类是B类友元，如无声明，C类和A类之间没有任何友元关系
- 友元关系是单向的
  - 如果声明B类是A类的友元，B类的成员函数就可以访问A类的私有和保护数据，但A类的成员函数却不能访问B类的私有、保护数据。
- 友元关系是不被继承的
  - 如果类B是类A的友元，类B的派生类BB并不会自动成为类A的友元

内容

- 9.1 类
- 9.2 动态内存管理
- 9.3 标准库类型
- 9.4 拷贝控制

内存空间

动态内存管理

- C++程序中使用的内存空间
  - 静态内存(static memory)用来保存局部静态对象、类静态数据成员以及定义在任何函数之外的变量（全局变量）
  - 栈内存(stack memory)用于保存定义在函数内的非静态对象
    - ✓分配在静态或栈内存中对象由编译器自动创建和销毁
  - 自由空间(free store)或堆(heap)用来存储动态分配的对象
    - ✓动态对象的生存期由程序来控制
    - ✓new / delete 运算符

内存分配操作符new

new 类型名T (初始化参数列表)

- 功能：在程序执行期间，申请用于存放T类型对象的内存空间，并依初值列表赋以初值。
- 结果值：
  - 成功：T类型的指针，指向新分配的内存
  - 失败：抛出异常。

基本类型变量初始化

分配内存后，用初值来初始化：

```
int *point = new int(2);
```

不初始化：

```
int *point = new int;
```

用0初始化：

```
int *point = new int();
```

类对象初始化

- 分配内存后，调用构造函数初始化：

T *point = new T(初值列表);

- 有用户定义的默认构造函数时，new T和new T()效果相同，都调用默认构造函数
- 若用户未定义默认构造函数：
  - new T 调用系统生成的默认构造函数
  - new T() 执行默认构造函数，并为基本数据类型和指针类型的成员用0赋初值，该过程是递归的

释放内存操作符delete

delete 指针p

- 功能：释放指针p所指向的内存。p必须是new操作的返回值。
- delete零指针是安全的：

```
int *ip = 0;
delete ip;
```

  - 最好在delete指针后，把其值置为0

动态创建对象举例

动态内存管理

```
#include <iostream>
using namespace std;
class Point {
public:
    Point() : x(0), y(0) {
        cout << "Default Constructor called." << endl;
    }
    Point(int x, int y) : x(x), y(y) {
        cout << "Constructor called." << endl;
    }
    ~Point() { cout << "Destructor called." << endl; }
    int getX() const { return x; }
    int getY() const { return y; }
    void move(int newX, int newY) {
        x = newX;
        y = newY;
    }
private:
    int x, y;
};
```

```
int main() {
    cout << "Step one: " << endl;
    Point *ptr1 = new Point;//调用缺省构造函数
    delete ptr1; //删除对象，自动调用析构函数

    cout << "Step two: " << endl;
    ptr1 = new Point(1,2);
    delete ptr1;

    return 0;
}
```

运行结果：

```
Step One:
Default Constructor called.
Destructor called.
Step Two:
Constructor called.
Destructor called.
```

申请和释放动态数组

动态内存管理

- 分配：

  new 类型名T [数组长度]

  - 数组长度可以是任何正整数值表达式，在运行时计算
  - 方括号后可加小括号”()”，但小括号内不能带任何参数。是否加”()”的区别同”new T”和”new T()”初始化时的区别

    int *p = new int[10]();

- 释放：

  delete[] 指针名p

  - 释放指针p所指向的数组。p必须是用new分配得到的数组首地址。

动态创建对象数组举例

动态内存管理

```
#include<iostream>
using namespace std;
class Point {   //类的声明同前例，略 };
int main() {
    Point *ptr = new Point[2]; //创建对象数组
    ptr[0].move(5, 10);//通过指针访问数组元素的成员
    ptr[1].move(15, 20);//通过指针访问数组元素的成员
    cout << "Deleting..." << endl;
    delete[] ptr; //删除整个对象数组
    return 0;
}
```

运行结果：

Default Constructor called.

Default Constructor called.

Deleting...

Destructor called.

Destructor called.

96

内容

- 9.1 类
- 9.2 动态内存管理
- 9.3 标准库类型
- 9.4 拷贝控制

标准库类型string

- 标准库类型string表示可变长字符序列
  - 使用string类型首先要包含string头文件
  - 作为标准库的一部分，string定义在命名空间std中

```
#include <string>
using std::string;
```

定义和初始化string对象

标准库类型string

◆类定义了其对象的初始化方式

`string s1`	Default initialization; `s1` is the empty string.
`string s2(s1)`	`s2` is a copy of `s1`.
`string s2 = s1`	Equivalent to `s2(s1)`, `s2` is a copy of `s1`.
`string s3("value")`	`s3` is a copy of the string literal, not including the null.
`string s3 = "value"`	Equivalent to `s3("value")`, `s3` is a copy of the string literal.
`string s4(n, 'c')`	Initialize `s4` with `n` copies of the character `'c'`.

- 默认初始化，得到一个空的string，即该string对象中没有任何字符
- 如果提供字符串字面值，字面值中除去最后空(null)字符外的其他字符都被拷贝到新创建的string对象

直接和拷贝初始化

- 使用等号(=)初始化变量，执行的是拷贝初始化(copy initialization)
  - 编译器把等号右侧的初始值拷贝到新创建的对象中去
- 如果不使用等号，执行的是直接初始化(direct initialization)

直接和拷贝初始化

- 当初始值只有一个时，使用直接初始化或拷贝初始化都可以
- 如果用多个值初始化一个变量，一般来说只能使用直接初始化方式

```
string s5 = "hiya";  // copy initialization
string s6("hiya");   // direct initialization
string s7(10, 'c');  // direct initialization; s7 is cccccccccc
```

直接和拷贝初始化

- 当用多个值初始化一个变量时，也可通过显式地创建一个临时对象来间接使用拷贝初始化方式

```
string s8 = string(10, 'c'); // copy initialization; s8 is cccccccccc
```

- 等价形式：

```
string temp(10, 'c'); // temp is cccccccccc
string s8 = temp;     // copy temp into s8
```

string对象上的操作

标准库类型string

◆string的一些操作：

`os << s`	Writes `s` onto output stream `os`. Returns `os`.
`is >> s`	Reads whitespace-separated string from `is` into `s`. Returns `is`.
`getline(is, s)`	Reads a line of input from `is` into `s`. Returns `is`.
`s.empty()`	Returns `true` if `s` is empty; otherwise returns `false`.
`s.size()`	Returns the number of characters in `s`.
`s[n]`	Returns a reference to the `char` at position `n` in `s`; positions start at 0.
`s1 + s2`	Returns a `string` that is the concatenation of `s1` and `s2`.
`s1 = s2`	Replaces characters in `s1` with a copy of `s2`.
`s1 == s2`	The `strings` `s1` and `s2` are equal if they contain the same characters.
`s1 != s2`	Equality is case-sensitive.
`<, <=, >, >=`	Comparisons are case-sensitive and use dictionary ordering.

读写string对象

string对象上的操作

- 使用IO操作符cin和cout读写string对象

```
int main()
{
    string s;          // empty string
    cin >> s;          // read a whitespace-separated string into s
    cout << s << endl; // write s to the output
    return 0;
}
```

- string输入运算符(>>)自动忽略开头的空白（即空格、换行、制表符等），读入字符直到遇到下一处空白
- string输入输出操作返回运算符左侧的运算对象作为结果
  - 多个输入或输出可以连写在一起

读取未知数量的string对象

string对象上的操作

```
int main()
{
    string word;
    while (cin >> word)        // read until end-of-file
        cout << word << endl; // write each word followed by a new line
    return 0;
}
```

- 如果流有效，也就是说没有遇到文件结束标记或非法输入，那么执行while语句循环体

使用getline读取一整行

string对象上的操作

- getline函数的参数是一个输入流对象和一个string对象
  - getline函数从输入流中读取内容，直到遇到换行符（换行符也被读入），然后把内容（不包括换行符）存入string对象
    - ✓如果第一个字符就是换行符，getline停止读入并返回，string实参置为空string
  - getline函数返回其流参数

```
int main()
{
   string line;
   // read input a line at a time until end-of-file
   while (getline(cin, line))
       cout << line << endl;
   return 0;
}
```

empty和size操作

string对象上的操作

- empty根据string对象是否为空返回一个bool值

```
string line;
// read input a line at a time and discard blank lines
while (getline(cin, line))
    if (!line.empty())
        cout << line << endl;
```

- size返回string对象的长度(即字符的个数)

```
// read input a line at a time and print lines that are longer than 80 characters
while (getline(cin, line))
    if (line.size() > 80)
        cout << line << endl;
```

string::size_type类型

- size函数返回一个string::size_type类型的值
- string::size_type是无符号整数类型，能存放下任何string对象的大小
- 如果n是一个int型负数，那么s.size()<n的结果几乎肯定是true

比较string对象

string对象上的操作

- 相等运算符(==和!=)检测两个string对象是否相等
  - 相等意味着长度相同，且所含字符全相同
- 关系运算符<、<=、>、>=分别检测一个string对象是否小于、小于等于、大于、大于等于另一个string对象

比较string对象

string对象上的操作

- 这些运算符按照大小写敏感字典序：
  - 如果长度不同，且短string对象的每个字符都与长string对应位置上字符相同，那么短string对象小于长string对象
  - 如果两个string对象的字符在某些对应位置上不一样，那么string对象比较的结果是第一对相异字符比较的结果

```
string str = "Hello";
string phrase = "Hello World";
string slang  = "Hiya";
```

string对象的赋值

string对象上的操作

- 大多数库类型都支持赋值操作

```
string st1(10, 'c'), st2; // st1 is ccccccccccc; st2 is an empty string
st1 = st2; // assignment: replace contents of st1 with a copy of st2
           // both st1 and st2 are now the empty string
```

两个string对象相加

string对象上的操作

- 对string对象使用加法运算符(+)的结果是一个新的string对象，其内容是左侧对象所含字符和右侧对象串接而成
- 复合赋值运算符(+=)把右侧string对象的内容追加到左侧对象的后面

```
string s1  = "hello, ", s2 = "world\n";
string s3 = s1 + s2;   // s3 is hello, world\n
s1 += s2;   // equivalent to s1 = s1 + s2
```

字面值和string对象相加

string对象上的操作

- 标准库允许把字符字面值和字符串字面值转换为string对象

```
string s1 = "hello", s2 = "world";
string s3 = s1 + ", " + s2 + '\n';
```

- 必须确保每个加法运算符(+)的两侧运算对象至少有一个是string

```
string s4 = s1 + ", ";          // ok: adding a string and a literal
string s5 = "hello" + ", ";     // error: no string operand
string s6 = s1 + ", " + "world"; // ok: each + has a string operand
string s7 = "hello" + ", " + s2; // error: can't add string literals
```

处理string对象中的字符

标准库类型string

◆cctype头文件中定义了字符处理函数

`isalnum(c)`	`true` if `c` is a letter or a digit.
`isalpha(c)`	`true` if `c` is a letter.
`iscntrl(c)`	`true` if `c` is a control character.
`isdigit(c)`	`true` if `c` is a digit.
`isgraph(c)`	`true` if `c` is not a space but is printable.
`islower(c)`	`true` if `c` is a lowercase letter.
`isprint(c)`	`true` if `c` is a printable character (i.e., a space or a character that has a visible representation).
`ispunct(c)`	`true` if `c` is a punctuation character (i.e., a character that is not a control character, a digit, a letter, or a printable whitespace).
`isspace(c)`	`true` if `c` is whitespace (i.e., a space, tab, vertical tab, return, newline, or formfeed).
`isupper(c)`	`true` if `c` is an uppercase letter.
`isxdigit(c)`	`true` if `c` is a hexadecimal digit.
`tolower(c)`	If `c` is an uppercase letter, returns its lowercase equivalent; otherwise returns `c` unchanged.
`toupper(c)`	If `c` is a lowercase letter, returns its uppercase equivalent; otherwise returns `c` unchanged.

从string对象获取字符

处理string对象中的字符

- 可以通过下标运算符([])来访问string对象中的单个字符
- 下标运算符接收的输入参数是string::size_type类型的值，返回该值对应位置上字符的引用
- 下标的值称为下标(subscript)或索引(index)
  - 任何整型值表达式都可以作为索引
  - string对象的合法下标范围[0, size()-1]

使用下标进行迭代

处理string对象中的字符

```
string s("some string");
// process characters in s until we run out of characters or we hit a whitespace
for (string::size_type index = 0;
index != s.size() && !isspace(s[index]); ++index)
        s[index] = toupper(s[index]); // capitalize the current character
```

- 程序的输出结果：

```
SOME string
```

使用下标随机访问

处理string对象中的字符

◆ 把0到15间的十进制数转换为十六进制形式

```
const string hexdigits = "0123456789ABCDEF"; // possible hex digits
cout << "Enter a series of numbers between 0 and 15"
     << " separated by spaces. Hit ENTER when finished: "
     << endl;
string result;        // will hold the resulting hexify'd string
string::size_type n;  // hold numbers from the input
while (cin >> n)
    if (n < hexdigits.size())    // ignore invalid input
        result += hexdigits[n];  // fetch the indicated hex digit
cout << "Your hex number is: " << result << endl;
```

◆ 输入：12 0 5 15 8 15

◆ 输出：Your hex number is: C05F8F

练习

◆练习3.9：下面程序是否合法？

```
string s;
cout << s[0] << endl;
```

内容

- 9.1 类
- 9.2 动态内存管理
- 9.3 标准库类型
- 9.4 拷贝控制

拷贝、赋值和销毁

拷贝控制

- 除了定义类的对象如何初始化外，还需要控制拷贝(copy)、赋值(assign)和销毁(destroy)对象时的行为
  - 初始化变量以及以传值方式传递或返回一个对象时，对象被拷贝
  - 使用赋值运算符时，发生对象的赋值
  - 当对象不再存在时，执行销毁的操作
- 如果不定义这些操作，编译将替我们合成它们
  - 编译器生成的版本对对象的每个成员执行拷贝、赋值和销毁操作

拷贝构造函数

- 拷贝构造函数是指只有单个形参，且该形参是自身类类型的引用

```
class Foo {
public:
   Foo();              // default constructor
   Foo(const Foo&);    // copy constructor
   // ...
};
```

  - 虽然可以定义接受非const引用形参的拷贝构造函数，但几乎总是const的引用
  - 拷贝构造函数通常不应是explicit的

合成拷贝构造函数

拷贝构造函数

- 如果没有为类定义拷贝构造函数，编译器会定义一个合成拷贝构造函数
  - 不同于默认构造函数，即使有其他的构造函数，编译器也会合成一个拷贝构造函数
  - 一般地，合成拷贝构造函数会将实参对象的非静态成员逐成员地拷贝到正在创建的对象中
    - ✓对类类型成员，使用其拷贝构造函数来拷贝
    - ✓内置类型的成员被直接拷贝
    - ✓对数组类型的成员，逐元素的拷贝；如果元素是类类型，使用元素的拷贝构造函数来拷贝

合成拷贝构造函数

拷贝构造函数

```
class Sales_data {
public:
    // other members and constructors as before
    // declaration equivalent to the synthesized copy constructor
    Sales_data(const Sales_data&);
private:
    std::string bookNo;
    int units_sold;
    double revenue;
};
// equivalent to the copy constructor that would be synthesized for Sales_data
Sales_data::Sales_data(const Sales_data &orig):
    bookNo(orig.bookNo),        // uses the string copy constructor
    units_sold(orig.units_sold), // copies orig.units_sold
    revenue(orig.revenue)       // copies orig.revenue
    {    }                      // empty body
```

拷贝初始化

拷贝构造函数

- 直接初始化时，编译器使用普通的函数匹配来选择与提供的实参最匹配的构造函数
- 拷贝初始化要求编译器将右侧运算对象拷贝到正在创建中的对象，如果需要还有进行类型转换
  - 拷贝初始化通常使用拷贝构造函数来完成

```
string dots(10, '.');          // direct initialization
string s(dots);                // direct initialization
string s2 = dots;              // copy initialization
string null_book = "9-999-99999-9"; // copy initialization
string nines = string(100, '9');    // copy initialization
```

拷贝初始化

拷贝构造函数

- 拷贝初始化不仅在用=号定义变量时发生，在下列情况下也会发生：
  - 将对象作为实参传递给非引用类型的形参
    - ✓拷贝构造函数的形参必须是引用类型
  - 从返回类型为非引用类型的函数返回对象

拷贝赋值运算符

- 类可以控制其对象如何赋值：

```
Sales_data trans, accum;
trans = accum; // uses the Sales_data copy-assignment operator
```

  - 如果类未定义自己的拷贝赋值运算符，编译器会为它合成一个

重载赋值运算符

拷贝赋值运算符

- 重载运算符(overloaded operator)本质上是函数，函数名由operator关键字后跟表示要定义的运算符的符号组成
  - 运算符函数也有返回类型和形参列表
  - 重载运算符的形参表示运算符的运算对象
- 赋值运算符operator=必须定义为成员函数：a=b等价于a.operator=(b)
  - 左侧运算对象绑定到隐式的this形参
  - 右侧运算对象作为显式形参传递

重载赋值运算符

拷贝赋值运算符

- 拷贝赋值运算符接受一个其所在类类型的参数
  - 通常返回一个指向其左侧运算对象的引用

```
class Foo {
public:
    Foo& operator=(const Foo&); // assignment operator
  // ...
};
```

合成拷贝赋值运算符

拷贝赋值运算符

- 如果类未定义拷贝赋值运算符，编译器会生成一个合成拷贝赋值运算符(synthesized copy-assignment operator)
  - 将右侧运算对象的每个非静态成员赋给左侧运算对象的对应成员
    - ✓类类型成员通过类的拷贝赋值运算符完成
    - ✓对于数组类型的成员，逐个元素进行赋值
  - 返回左侧运算对象的引用

合成拷贝赋值运算符

拷贝赋值运算符

◆ 合成拷贝赋值运算符的等价实现如下：

```
// equivalent to the synthesized copy-assignment operator
Sales_data& Sales_data::operator=(const Sales_data &rhs)
{
    bookNo = rhs.bookNo;       // calls the string::operator=
    units_sold = rhs.units_sold;  // uses the built-in int assignment
    revenue = rhs.revenue;       // uses the built-in double assignment
    return *this;                // return a reference to this object
}
```

析构函数

- 析构函数(destructor)释放对象使用的资源，并销毁对象的非static数据成员
  - 析构函数是类的成员函数
    - ✓函数名由波浪号(~)接类名构成
    - ✓没有返回值，也不接受参数
    - ✓不能被重载，只有唯一的析构函数

```
class Foo {
public:
   ~Foo();    // destructor
  // ...
};
```

析构函数完成的工作

析构函数

- 析构函数首先执行函数体，然后销毁成员
  - 析构函数体通常释放对象在生存期分配的所有资源
  - 成员按初始化顺序的逆序销毁
    - ✓销毁类类型的成员需要执行成员的析构函数
    - ✓销毁内置类型成员什么也不需要做
    - ✓隐式地销毁内置指针类型的成员不会delete它所指向的对象

何时调用析构函数

析构函数

- 无论何时对象被销毁，都会自动调用其析构函数
  - 变量在离开其作用域时被销毁
  - 当一个对象被销毁时，其成员被销毁
  - 容器（无论是标准库容器还是数组）被销毁时，其元素被销毁
  - 对于动态分配的对象，当delete指向它的指针时被销毁
  - 对临时对象，当创建它的完整表达式结束时被销毁

合成析构函数

析构函数

- 当类未定义析构函数时，编译器会为它定义一个合成析构函数(synthesized destructor)
  - 合成析构函数的函数体为空
  - 成员是在析构函数体之后隐式的析构阶段中被销毁

```
class Sales_data {
public:
  // no work to do other than destroying the members, which happens automatically
   ~Sales_data() { } // equivalent synthesized destructor
  // other members as before
};
```