第 3 章　类

在像 C++这样的面向对象程序设计语言中，**类**（class）是包括数据成员和函数的用户定义的类型。类是用于对象的数据类型。

3.1　Point　类

本章中的前三个示例演示了一个简单的类，其对象代表笛卡儿平面中的点。

示例 3.1　用于 Point 类的接口

```
class Point
{   public:
        Point(double=0.0,double=0.0);       // default constructor
        Point(const Point&);                // copy constructor
        ~Point();                           // destructor
        Point& operator=(const Point&);     // assignment operator
        double x() const;                   // accessor function
        double y() const;                   // accessor function
        string toString() const;
protected:
        double _x, _y;
};
```

这段代码将保存在名为 Point 的**头文件**（header file）中。它被称为类的**接口**（interface），因为它是程序员为了在他们的客户程序中使用类所需的类的一部分。它也是编译器编译使用类的语句所需的类的一部分。

这个类声明具有 7 个成员函数和两个数据成员。成员函数分别被命名为：Point(double,double)、Point(const Point&)、operator=()、~Point()、x()、y()和 toString()。数据成员分别被命名为：_x 和_y。注意：成员函数被声明为 public，数据成员则被声明为 protected。这意味着客户程序可以使用成员函数，但是它们不能直接访问数据成员（它们将只能间接地通过访问器函数 x()和 y()以只读方式访问数据）。

前两个成员函数是**构造函数**（constructor）。第一个构造函数是**默认构造函数**（default constructor），无论何时通过如下声明构造一个对象，默认都会调用它：

```
Point p2(7.2,4.9);      // constructs an object for the point (7.2,4.9)
```

由于它的两个实参都具有默认值，因此可以像下面这样调用它：

```
Point p1(7.2);          // constructs an object for the point (7.2,0.0)
Point p0;               // constructs an object for the point (0.0,0.0)
```

在构造匿名的对象时也会调用默认构造函数，如下所示：

```
p1 = Point(7.2);        // constructs an object for the point (7.2,0.0)
```

第二个构造函数是**复制构造函数**（copy constructor）。当通过构造现有对象的一个副本来复制它时，就会调用这个构造函数。

第三个成员函数是类的**析构函数**（destructor）。当对象超出作用域时，就会自动调用它。

第四个成员函数是重载的**赋值运算符**（assignment operator），其调用方式如下：

```
p0 = p1;
```

第五个和第六个成员函数是**访问器函数**（accessor function）。它们允许间接访问对象的protected数据，如下所示：

```
double x1 = p1.x();             // assigns the value of p1._x to x1
```

最后一个成员函数返回一个代表对象的字符串，可以像下面这样使用它：

```
string s1 = p1.toString();      // initializes s1 to "(7.2,0.0)"
```

在每个类中都包括toString()函数是有用的（在Java中这是自动完成的）。

注意：在只读的成员函数x()、y()和toString()的末尾使用了const关键字。如果函数并不打算更改它们绑定到的对象的状态，那么把它们声明为常量函数就是一种良好的“防御性编程”实践。通过这样做，程序员可以获得来自编译器的额外帮助，它将因此而禁止在函数中添加将更改对象状态的代码。

在C++中，类X的复制构造函数、析构函数和赋值运算符的声明必须具有以下形式：

```
X(const X&);                    // copy constructor
~X();                           // destructor
X& operator=(const X&);         // assignment operator
```

如果没有在类定义中显式声明它们或默认构造函数，编译器将自动为类创建它们。

示例3.2 用于Point类的测试驱动程序

下面是用于示例3.1中定义的Point类的简单的测试驱动程序：

```
int main()
{   Point p0;                   // invokes default constructor
    cout << "p0 = " << p0.toString() << "\n";
    Point p1(5,-2);             // invokes default constructor
```

```
    cout << "p1 = " << p1.toString() << "\n";
    Point p2=p1;                // invokes copy constructor
    cout << "p2 = " << p2.toString() << "\n";
    p0 = p1;                    // invokes assignment operator
    cout << "p0 = " << p0.toString() << "\n";
    cout << "p0.x() = " << p0.x() << "\n";
    cout << "p0.y() = " << p0.y() << "\n";
}
```

其输出如下：

```
p0 = (0,0)
p1 = (5,-2)
p2 = (5,-2)
p0 = (5,-2)
p0.x() = 5
p0.y() = -2
```

示例 3.3　Point 类的实现

下面是示例 3.1 中定义的类的实现：

```
Point::Point(double x, double y) : _x(x), _y(y) { }
Point::Point(const Point& p) : _x(p._x), _y(p._y) { }
Point::~Point() { }
Point& Point::operator=(const Point& p)
{   _x = p._x;
    _y = p._y;
    return *this;
}
double Point::x() const { return _x; }
double Point::y() const { return _y; }
string Point::toString() const
{   ostringstream output;
    output << "(" << _x << "," << _y << ")";
    return output.str();
}
```

两个构造函数和析构函数的主体都是空的。两个构造函数通过它们的**初始化列表**（initialization list）来执行它们的工作。例如，下面以粗体显示了复制构造函数的初始化列表：

```
Point::Point(const Point& p) : _x(p._x), _y(p._y) { }
```

这个定义等价于：

```
Point::Point(const Point& p)
{   _x = p._x;
    _y = p._y;
}
```

注意：这个主体与赋值运算符的主体几乎相同。复制构造函数与赋值运算符之间的主要区别在于前者在创建对象后再给它赋值。

只有构造函数可以具有初始化列表。

另请注意：实参的默认值是在接口而不是在实现中给出的。

toString()函数使用**字符串流**（string stream）来累积其输出。这是 ostringstream 类型的名为 output 的对象。使用该类需要以下指令：

```
#include <sstream>                    // defines the ostringstream class
```

通过输出运算符<<以与 cout 流对象相同的方式把输出存放在 ostringstream 对象中。流对象的优点是：将输出保存在一个缓冲区字符串中，可以利用 str()成员函数访问它，如下所示：

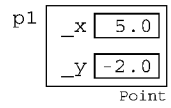
```
return output.str();
```

3.2 实例、隐式形参和 this 指针

下面的声明：

```
Point p1(5,-2);
```

创建一个名为 p1 的 Point 对象，并将其数据成员初始化为 5.0 和–2.0。我们可以形象地表示得到的对象，如下所示。



构造对象的过程称为**实例化**（instantiation），得到的对象称为类的**实例**（instance）。对象 p1 是 Point 类的实例。

在以下语句中：

```
cout << "p1 = " << p1.toString() << "\n";
```

表达式 p1.toString()调用 toString()函数。作为类成员函数的调用，它必须绑定到其类的实例上。这个调用绑定到实例 p1。调用绑定到的实例称为调用的**隐式形参**（implicit argument）。对象 p1 是调用 p1.toString()的隐式形参。在下面的语句中：

```
output << "(" << _x << "," << _y << ")";
```

名称_x 和_y 指代隐式形参的数据。因此，在这个示例中，_x 意指 p1._x，其值是 5.0。

在任何类成员函数中，关键字 this 都可用于指代隐式形参。变量 this 总是指向类的那个实例的指针。因此，*this 总是指向隐式形参的引用。重载的赋值运算符（它是返回指向类的实例的引用所必需的）总是返回引用*this。这使得有可能像下面这样串接赋值：

```
p4 = p3 = p2 = p1 = Point(2.9,6.1);
```

每次调用赋值运算符都会返回一个指向当前隐式形参的引用，它然后将变成链中的下一次调用的隐式形参。

3.3 编译类及其客户程序

可以用多种方式编译类及其客户程序。最简单的方式是把类的接口、客户程序以及类实现全都一起放在同一个文件中，如下图所示。在这种情况下，应该以如下顺序放置这三个部分：接口、程序、实现。

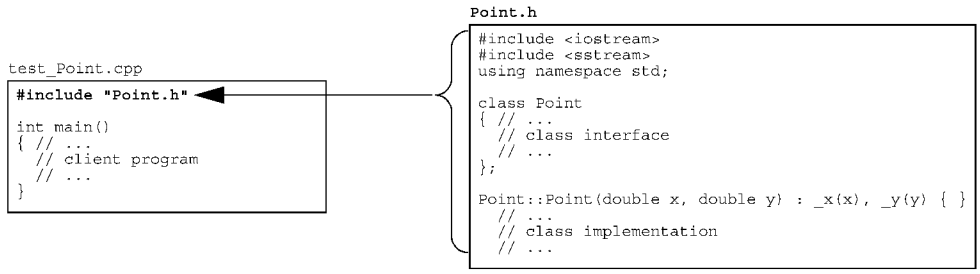
test_Point.cpp

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

class Point
{ // ...
  // class interface
  // ...
};

int main()
{ // ...
  // client program
  // ...
}

Point::Point(double x, double y) : _x(x), _y(y) { }
  // ...
  // class implementation
  // ...
```

为了获得更大的灵活性，可以把类的接口和实现存放在单独的**头文件**（header file）中，如下图所示。注意，必须在下面的指令中指定头文件的名称 Point.h：

```
#include "Point.h"
```

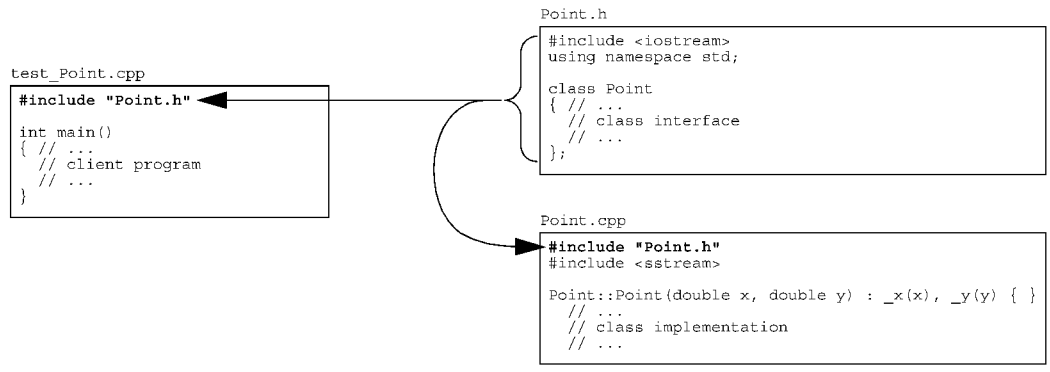
它将放置在使用类的每个客户程序的开始处。



编译器将在#include "Point.h"指令的位置把 Point.h 文件的完整内容插入到 test_Point.cpp 文件中。注意：在 test_Point.cpp 文件中不需要另外两个指令和 using 语句，因为将从 Point.h 文件插入它们。

把类定义与其客户程序分隔开的优点是：只需单独编译它一次即可。然后只需把编译代码与使用它的单独编译的程序链接起来即可。

编译类定义的第三种方式是把它的实现放在单独的文件中。在下图中演示了这一点。这里，Point 类的接口位于头文件 Point.h 中（“h”代表“头文件（header）”），它的实现位于 Point.cpp 文件中。这允许更好的“信息隐藏”。



可以单独编译实现文件，并且可以把得到的（不可读的）二进制代码与头文件一起交付给客户，从而使实现的源代码保密。客户在使用类时，所需的只是由头文件提供的接口信息以及将链接到编译的客户程序的实际机器代码。

注意：两个.cpp 文件都必须包括#include Point.h 头文件。因此，从编译器的角度讲，三个文件的配置等价于下图所示的这种两个文件的替代方案：

test_Point.cpp

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

class Point
{ // ...
  // class interface
  // ...
};

int main()
{ // ...
  // client program
  // ...
}
```

Point.cpp

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

class Point
{ // ...
  // class interface
  // ...
};

Point::Point(double x, double y) : _x(x), _y(y) { }
  // ...
  // class implementation
  // ...
```

有可能把类的成员的定义放在类定义自身内，而不是把实现分隔开。下面的示例演示了这一点。

示例 3.4　把类实现放在其定义内

这是一个完整的、自含式程序，它等价于前三个示例中的组合式代码：

```
#include <iostream>
#include <sstream>                  // defines the ostringstream class
using namespace std;

class Point
{   public:
        Point(double x=0.0, double y=0.0) : _x(x), _y(y) { }
        Point(const Point& p) : _x(p._x), _y(p._y) { }
        ~Point() { }
        Point& operator=(const Point& p)
        {   _x = p._x;
            _y = p._y;
            return *this;
        }
        double x() const { return _x; }
        double y() const { return _y; }
        string toString() const
        {   ostringstream output;
            output << "(" << _x << "," << _y << ")";
            return output.str();
        }
    protected:
        double _x, _y;
};

int main()
{   Point p0;                       // invokes default constructor
    cout << "p0 = " << p0.toString() << "\n";
    Point p1(5,-2);                 // invokes default constructor
    cout << "p1 = " << p1.toString() << "\n";
    Point p2=p1;                    // invokes copy constructor
    cout << "p2 = " << p2.toString() << "\n";
    p0 = p1;                        // invokes assignment operator
    cout << "p0 = " << p0.toString() << "\n";
    cout << "p0.x() = " << p0.x() << "\n";
    cout << "p0.y() = " << p0.y() << "\n";
}
```

显然，这种方法比以前示例中使用的方法简洁得多。

仅当单独一位程序员想要简洁、完整的自含式程序时，像示例 3.4 中那样把类的实现

放在其定义内一般才是可取的。“现实世界”中的更大的项目通常遵守以前的示例中三个单独的文件演示过的信息隐藏原则。

当把类定义放在单独的头文件中时，把类定义“包装”在#ifndef 指令内是一种良好的防御性编程实践，如下所示：

```
#ifndef POINT_H
#define POINT_H
// …
// class definition
// …
#endif // POINT_H
```

这确保如果在一个程序中多次间接地包括用于类定义的代码，编译器将不会多次尝试编译它（重复的尝试将引发错误）。下面的指令：

```
#ifndef POINT_H
```

告诉编译器如果已经定义了符号 POINT_H，就忽略该行与下一次出现的#endif 指令之间的所有代码。下面的指令：

```
#define POINT_H
```

定义了这个符号。因此，编译器在第一次遇到指令的这个“包装器”时，它将定义 POINT_H 符号，以后无论何时遇到它，编译器都将忽略它所包装的所有代码。

以后使用#ifndef 指令包装器时，下面给出了用于 Point 类及其测试驱动程序的首选配置：

test_Point.cpp

```
#include "Point.h"

int main()
{ // ...
  // client program
  // ...
}
```

Point.h

```
#ifndef POINT_H
#define POINT_H

#include <iostream>
using namespace std;

class Point
{ // ...
  // class interface
  // ...
};

#endif  //POINT_H
```

Point.cpp

```
#include "Point.h"
#include <sstream>
using namespace std;

Point::Point(double x, double y) : _x(x), _y(y) { }
  // ...
  // class implementation
  // ...
```

3.4 友 元 函 数

类中的**友元函数**（friend function）并非成员函数，但它可以访问类的 protected（和 private）成员。使用 friend 关键字声明友元函数。当程序员希望隐式实参变成显式的时，通常声明友元函数来代替成员函数。重载的输出运算符就是最常见的示例。

示例 3.5 Point 类中重载的输出运算符

```
class Point
{       friend ostream& operator<<(ostream&, const Point&);
    public:
        Point(double=0.0,double=0.0);       // default constructor
        Point(const Point&);                // copy constructor
        Point& operator=(const Point&);     // assignment operator
        double x() const;
        double y() const;
        string toString() const;
    protected:
        double _x, _y;
};
ostream& operator<<(ostream& ostr, const Point& point)
{   return ostr << point.toString();
}
```

利用这个额外的函数，客户可以像下面这样输出 Point 对象：

```
cout << "p0=" << p0 << ", p1=" << p1 << ", and p2=" << p2 << "\n";
```

获得的输出如下：

```
p0=(0,0), p1=(1,0), and p2=(5,-2)
```

这允许以与我们输出 int 和 string 相同的方式输出 Point 对象。

3.5 Line 类

下一个示例演示了其实例代表笛卡儿平面中的直线的类。

示例 3.6 Line 类的接口

```
class Line
{   public:
```

```
        Line(double=1.0,double=1.0);        // two-intercept form of a line
        Line(const Point&, double=0.0);     // point-slope form of a line
        Line(const Point&, const Point&);   // two-point form of a line
        Line(double,double,double);         // general form of a line
        Line(const Line&);                  // copy constructor
        ~Line();                            // destructor
        Line& operator=(const Line&);       // assignment operator
        double xCoef() const;
        double yCoef() const;
        double cTerm() const;
        string toString() const;
    protected:
        double _a, _b, _c;      // coefficients in general form: ax+by+c=0
};
```

这个类有 5 个构造函数，其中两个使用了 Point 类。

第一个构造函数是默认构造函数，可以像下面这样使用它：

```
Line line1;                                 // x + y = 1
Line line2(7);                              // x + 7y = 7
Line line3(7,5);                            // 5x + 7y = 35
```

用于这个构造函数的两个实参代表直线的两个截距。因此 line3 具有 x 截距(7, 0)和 y 截距(0, 5)，它的方程是 $7x + 5y = 35$。默认的截距是(1, 0)和(0, 1)。

第二个构造函数使用直线的点斜式实参，可以像下面这样使用它：

```
Line line4(Point(1,6),-2);                  // 2x + y = 8
```

这将构造穿过点(1, 6)并且斜率为–2 的直线，它的方程是 $2x + y = 8$。

第三个构造函数使用直线的两点式实参，可以像下面这样使用它：

```
Line line5(Point(1,6),Point(-1,5));         // x - 2y = -11
```

这将构造穿过点(1, 6)并且斜率为 1/2 的直线，它的方程是 $x - 2y = -11$。

第四个构造函数使用直线的 3 个参数形式的通用方程：$Ax + By + C = 0$。可以像下面这样使用它：

```
Line line6(2,3,6);                          // 2x + 3y = -6
```

这将构造其通用方程为 $2x + 3y + 6 = 0$ 的直线。

类具有三个访问器函数：xCoef()、yCoef()和 cTerm()，它们分别用于三个数据成员_a、_b 和_c。

注意，由于这个接口使用了 Point 类，因此应该在它前面放置以下指令：