3

类域、友元、运算符重载

本章主要内容：

类的作用域范围以及它的基本特征。如何使用友元从类外访问类的私有成员。如何使用运算符重载让自定义的类型与基本数据类型的数据操作形式上一致。

3.1 类域

类的作用域简称类域，它是指类的定义中由一对花括号所括起来的部分。每一个类都有自己的类域，该类的成员局限在该类所属的类域中。从类的定义可知，在类域中可以定义变量，也可以定义函数。从这点来看类域与文件域很相似。但是，类域又不同于文件域，在类域中定义的变量不能使用 auto、register 和 extern 等修饰符，可用 const、static 修饰符，而定义的函数也不能用 extern 修饰符。

文件域可以包含类域，显然类域小于文件域。一般而言，类域可包含成员函数的作用域。由于类中成员的特殊访问规则，使得类中成员的作用域变得比较复杂。具体地讲，某个类 C 中的某个成员 M 在以下情况下具有类 C 的作用域：

（1）该成员（M）出现在该类的某个成员函数中，并且该成员函数没有定义同名标识符。

（2）该成员（M）出现在该类（C）的某个对象的表达式中。例如，a 是 C 的对象，即在表达式 a.M 中。

（3）该成员（M）出现在该类（C）的某个指向对象指针的表达式中。例如，Pa 是一个指向 C 类对象的指针，即在表达式 Pa->M 中。

（4）使用作用域运算符所限定的表达式中。例如，在表达式 C::M 中。

一般说来，类域介于文件域和函数域之间，类域问题比较复杂，只能根据具体问题具体分析。

3.1.1 类成员作用域

类中定义的成员变量和成员函数的作用域是整个类，这些成员只有在类中（包含类的定义部分和类外函数实现部分）是可见的，在类外是不可见的，因此，可以在不同类中使用相同的成员名。另外，类作用域意味着不能从外部直接访问类的任何成员，即使该成员的访问权限是public，也要通过该类的对象名来调用，对于 static 成员，要指定类名来调用。如果发生同名覆盖现象，类成员的可见域将小于作用域，但此时可借助 this 指针或“类名::”形式指明所访问的是类成员，这有点类似于使用“::”访问全局变量。如例 3.1 所示。

例 3.1　变量作用域。

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

int x = 30; //定义性声明，全局 int 型变量 x
int z = 40; //定义性声明，全局 int 型变量 z

class CTest //CTest 类定义
{
    //默认为 private 的成员列表
    int x;
    int y;
public:
    CTest(int x = 0, int y = 0) //构造函数
    {
        this->x = x;
        this->y = y;
    }
    void print(int x) //成员函数 print，形参为 x
    {
        //形参 x 覆盖了成员 x 和全局变量 x
        cout << "传递来的参数:  " << x << endl;
        //此处的 y 指的是成员 y，如果要访问成员 x，需使用 this 指针
        cout << "成员 x:  " << (this->x) << ", 成员 y:  " << y << endl;
        //或使用类名加作用域运算符的形式指明要访问成员 x
        cout << "除了 this 指针外，还可以使用类名::的形式：" << endl;
        cout << "成员 x:  " << CTest::x << ", 成员 y:  " << y << endl;
        cout << "全局 x:  " << (::x) << endl; //访问全局变量 x
        //没有形参、成员变量对全局变量 z 构成同名覆盖，直接访问 z 即可
        cout << "全局 z:  " << z << endl;
```

```
    }
};

int main()
{
    CTest t1; //声明一个 CTest 类的对象 t1
    t1.print(6); //调用成员函数 print()
    return 0;
}
```

程序运行结果如下：

```
传递来的参数:  6
成员 x:  0, 成员 y:  0
除了 this 指针外，还可以使用类名::的形式：
成员 x:  0, 成员 y:  0
全局 x:  30
全局 z:  40
Press any key to continue
```

注意：在成员函数 print()中可以看到函数的局部变量覆盖了类的成员变量。如函数的形参 x 覆盖了类的成员变量 x 和全局的变量 x，如果在该成员函数中要引用类的成员变量，可以用 this 指针或者类名::的形式。如果要使用全局变量，可以用::的形式来引用。

3.1.2 类定义的作用域与可见域

与函数一样，类的定义没有生存期的概念，但类定义有作用域和可见域。

使用类名创建对象时，首要的前提是类名可见，类名是否可见取决于类定义的可见域，该可见域同样包含在其作用域中，类本身可被定义在三种作用域内，这也是类定义的作用域：

1. 全局作用域

在函数和其他类定义的外部定义的类称为全局类，绝大多数的 C++类是定义在全局作用域中，本书前面定义的所有类都是在全局作用域中，全局类具有全局作用域。

2. 类作用域（类中类、嵌套类）

一个类可以在另一类中定义，这就是所谓的嵌套类。举例来说，如果类 A 定义在类 B 中，如果 A 的访问权限是 public，则 A 的作用域可认为和 B 的作用域相同，不同之处在于必须使用 B::A 的形式访问 A。当然，如果 A 的访问权限是 private，则只能在 B 类内使用类名创建该类的对象，无法在外部创建 A 类的对象，如例 3.2 所示。

例 3.2 嵌套类（类中类）。

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

```
class CLine
{
public:
    class CPoint //CPoint 类定义在 CLine 类内，且为 public 属性，外部可访问
    {
    private:
        int m_iX;
        int m_iy;

    public:
        CPoint(int x = 0, int y = 0)
        {
            m_iX = x;
            m_iY = y;
        }

        void printpoint(); //CPoint 类成员函数原型声明，在外部实现函数功能
    };

private:
    CPoint p1, p2; //CLine 内两个 CPoint 对象成员

public:
    CLine(int x1, int y1, int x2, int y2):p1(x1, y1), p2(x2, y2)//构造函数，初始化成员列表
    {
    }

    void printline() //输出提示信息
    {
        p1.printpoint(); //调用对象成员的公共接口
        cout << "    ------->    ";
        p2.printpoint(); //调用对象成员的公共接口
        cout << endl;
    }
};

void CLine::CPoint::printpoint() //CPoint 类中函数 printpoint()的实现，注意双重作用域限定符
{
    cout << "( " << m_iX << " , " << m_iY << " )";
```

```
}

int main()
{
    CLine line (1, 2, 3, 4); //调用 CLine 类的构造函数，声明一个 CLine 类的对象 line
    line.printline();        //输出提示信息
    CLine::CPoint pt(1, 3);//以 CLine::CPoint 访问 CPoint 类定义，声明一个 CPoint 类的对象 pt
    pt.printpoint();         //输出提示信息
    cout << endl;            //为整齐美观，换行
    return 0;
}
```

程序运行结果如下：

```
( 1 , 2 )  -------> ( 3 , 4 )
( 1 , 3 )
Press any key to continue
```

注意：在声明一个嵌套类的对象时，必须使用 B::A 的形式。

3. 块作用域

类的定义在代码块中，这是所谓局部类，该类完全被块包含，其作用域仅仅局限于其定义所在的块，不能在块外使用类名声明该类的对象，如例 3.3 所示。

例 3.3 块作用域。

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

int main()
{
    void Point(int, int);
    Point(6, 4);
    return 0;
}

void Point(int a, int b)
{
    class CPoint                    //类定义在函数内，在函数外无法使用 CPoint 创建对象
    {
    private:
        int m_iX, m_iY;
    public:
        CPoint(int x = 0, int y= 0)
        {
```

```
            m_iX= x;
            m_iY= y;
        }
        void print()
        {
            cout << m_iX << "," << m_iX << endl;
        }
    };

    CPoint pt(a, b);  //在函数内创建 CPoint 类的对象 pt
    pt.print();
}
```

程序运行结果如下：

```
6,4
Press any key to continue
```

和普通变量的覆盖原则一样，类名也存在“屏蔽”和“覆盖”，不过，依旧可使用作用域运算符“::”指定具体使用的类名，如“::类名”访问的是全局类，使用“外部类::嵌套类”来访问嵌套类。

3.2 友元

在前面章节中讲过，当把一个成员的访问属性设置成为非公有属性后，则不能从类的外部访问。但有时候，需要在某个函数或者类中访问这些非公有成员。

在 C++中有这样的机制使得这一愿望得以实现，这就是友元机制有以下三种友元形式：

1. 非成员函数作为友元函数

在类的定义中用 friend 声明了一个外部函数或其他类的成员函数（public 和 private 均可）后，这个外部函数称为类的友元函数。

友元函数声明的基本格式为：

```
friend  函数原型;
```

一个类的友元函数可访问该类的 private 成员。

例 3.4 将一个外部函数声明为类的友元函数。

```
#include <cmath>       //使用计算平方根的函数 sqrt 要用到的头文件
#include <iostream>
using namespace std;

class CPoint
{
private:
```

```
        int m_iX,m_iY;
        //友元函数的声明，声明位置没有关系，可以是 public，也可是 private
        friend float Cal(CPoint &p1, CPoint & p2);
    public:
        CPoint(int x= 0, int y = 0)
        {
            m_iX = x;
            m_iY =y;
        }
        void print()
        {
            cout << "(" << m_iX << "," << m_iY << ")";
        }
    };

    float Cal(CPoint & p1, CPoint & p2)     //友元函数的实现，它是一个全局函数
    {
        //友元函数中可访问类的 private 成员
        float d = sqrt((p1.m_iX - p2.m_iX)*(p1.m_iX - p2.m_iX)+(p1.m_iY - p2.m_iY)*(p1.m_iY - p2.m_iY));
        return d;
    }
    int main()
    {
        CPoint p1(2, 3), p2(5, 7);
        p1.print();
        cout << "与";
        p2.print();
        cout << "两点距离=" << Cal(p1, p2) << endl;//利用友元函数计算两点间的距离
        return 0;
    }
```

程序运行结果如下：

```
(2,3)与(5,7)两点距离=5
Press any key to continue
```

2. 一个类的成员函数作为另外一个类的友元函数

A 类的成员函数作为 B 类的友元函数时，必须先定义 A 类，而不仅仅是声明它。

注意：将其他类的成员函数声明为本类的友元函数后，该友元函数并不能变成本类的成员函数。也就是说朋友并不能变成家人。

成员函数形式的友元函数如例 3.5 所示。

在例 3.5 中，CLine 类的成员函数 Cal()的实现必须在类外进行，且必须在 CPoint 类的定

义之后。因为其参数中包含了 CPoint 这种类型。

CLine 类的 Cal()函数本来不能访问 CPoint 类中的 private 成员，但在 CPoint 类中将 Cal()声明为友元函数后就能访问了。请注意 Cal()函数依然不是 CPoint 类的成员函数。也就是说，Cal()只是 CPoint 类的朋友，可以访问 CPoint 类的私有成员变量 m_iX 和 m_iY。

例 3.5 将 CLine 类的成员函数 Cal()作为 CPoint 类的友元函数。

```
#include <cmath>                    //使用计算平方根的函数 sqrt 要用到的头文件
#include<iostream>
using namespace std;
class CPoint;
class CLine
{
public:
    float Cal(CPoint& p1, CPoint& p2); //友元函数的原型，作为 CLine 类的成员函数
};
class CPoint
{
private:
    int m_iX,m_iY;
    friend float CLine::Cal(CPoint &p1, CPoint &p2);  //友元函数的声明
public:
    CPoint(int x = 0, int y = 0)
    {
        m_iX = x;
        m_iY = y;
    }
    void print()
    {
        cout << "(" << m_iX << "," << m_iY << ")";
    }
};

//CLine 类内成员函数 Cal 的实现，作为 CPoint 类的友元函数
float CLine::Cal(CPoint &p1, CPoint &p2)
{
    float d = sqrt((p1.m_iX - p2.m_iX) * (p1.m_iX - p2.m_iX) + (p1.m_iY - p2.m_iY) * (p1.m_iY -
p2.m_iY));
    //友元函数可访问 CPoint 类对象的 private 成员
    return d;
}
int main()
```

```
{
    CLine line;
    CPoint p1(2,3), p2(7,5);
    p1.print();
    cout << " 与 ";
    p2.print();
    cout << " 两点的距离 = " << line.Cal(p1, p2) << endl;
    return 0;
}
```

程序运行结果如下：

```
(2,3) 与 (7,5) 两点的距离 = 5.38516
Press any key to continue
```

3. 友元类

类 A 作为类 B 的友元时，类 A 称为友元类。A 中的所有成员函数都是 B 的友元函数，都可以访问 B 中的所有成员。

类 A 可以在类 B 的 public 部分或 private 部分进行声明，方法如下：

```
friend  <类名>；//友元类类名
```

例 3.6　友元类。

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

class CLine;

class CPoint //定义 CPoint 类
{
private:
    int   m_iX, m_iY;
    friend CLine; //友元类的声明，位置同样不受限制

public:
    CPoint(int x= 0, int y = 0)
    {
        m_iX = x;
        m_iY = y;
    }

    void print()
    {
```

```
        cout << "(" << m_iX << "," << m_iY << ")";
    }
};

class CLine                 //类 CLine 的定义，其中所有的函数都是 CPoint 类的友元函数
{
public:
    float Cal(CPoint& p1,CPoint& p2)            //可访问 p1 和 p2 的 private 成员
    {
        float d;
        d=sqrt((p1.m_iX-p2.m_iX)*(p1.m_iX-p2.m_iX)+(p1.m_iY-p2.m_iY)*(p1.m_iY-p2.m_iY));
        return d;
    }
    void SetPoint(CPoint* p1,int x,int y)       //可访问 p1 和 p2 的 private 成员
    {
        p1->m_iX = x;
        p1->m_iY = y;
    }
};
int main()
{
    CLine line;
    CPoint p1(2,3),p2(7,5);
    p1.print();
    cout<<"与";
    p2.print();
    cout<<"两点的距离="<<line.Cal(p1,p2)<<endl;
    line.SetPoint(&p1,4,5);     //调用 line 的成员函数 SetPoint 改写 p1 中的 private 成员 x 和 y 的值
    p1.print();                 //修改后点 p1 的信息输出
    cout<<endl;
    return 0;
}
```

程序运行结果如下：

```
(2,3)与(7,5)两点的距离=5.38516
(4,5)
Press any key to continue
```

注意：不可否认，友元在一定程度上将类的私有成员暴露出来，破坏了信息隐藏机制，似乎是种“副作用很大的药”。但俗话说“良药苦口”，好工具总是要付出点代价的，拿把锋利的刀砍瓜切菜，总是要注意不要割到手指。

友元的存在使得类的接口扩展更为灵活，使用友元机制进行运算符重载（将在 3.3 节中具体介绍）从概念上也更容易理解一些，而且，C++规则已经极力地将友元的使用限制在了一定范围内，它是单向的、不具备传递性、不能被继承，所以应尽力合理使用友元。

3.3 运算符重载

3.3.1 运算符重载的基本概念

C++内部定义的数据类型（int、float、……)的数据操作可以用运算符号直接来表示，其使用形式是表达式。用户自定义类型的数据操作则用函数表示，其使用形式是函数调用。为了使用户自定义数据类型的数据操作与系统内部定义的数据类型的数据操作形式一致，C++提供了运算符的重载，通过把 C++中预定义的基本运算符重载为类的成员函数或者友员函数，使得对用户自定义数据类型的数据对象的操作形式与 C++内部定义保持一致。运算符重载就是赋予已有的运算符多重含义。C++中通过重新定义运算符，使它能够用于特定类的对象执行特定的操作，这便增强了 C++语言的处理能力。

3.3.2 运算符重载的基本规则

（1）允许重载的运算符。标准 C++中提供的运算符很多，但允许重载的只能是表 3-1 中所列的运算符。

表 3-1 允许重载的运算符列表

双目运算符	+ - * / %
关系运算符	== != < > <= >=
逻辑运算符	|| && +
单目运算符	+ - * &
自增自减运算符	++ --
位运算符	| & ~ ^ << >>
赋值运算符	= += -= *= /= %= &= |= ^= <<= >>=
空间申请和释放	New delete new[] delete[]
其他运算符	() -> ->* , []

（2）不允许重载的运算符。不允许重载的运算符只有 5 个：

- typeof（静态获取参数类型）
- typedef（定义类型别名）
- .（成员访问符）
- .*（成员指针访问运算符）
- ::（作用域运算符）
- sizeof（长度运算符）
- ?:（条件运算符）

（3）不允许自己定义新的运算符，只能对已有的运算符进行重载。

（4）重载不能改变运算符运算对象的个数，如>和<是双目运算符，重载后仍为双目运算符，需要两个参数。

（5）重载不能改变运算符的结合性，如=是从右至左，重载后仍然为从右至左。

（6）重载不能改变运算符的优先级别，例如*、／优先于＋、－，那么重载后也是同样的优先级。

（7）重载运算符的函数不能有默认的参数，否则就改变了运算符参数的个数，与（4）矛盾。

（8）重载的运算符必须和用户的自定义数据类型一起使用，其参数至少有一个是类对象（或者类对象的引用），或者说参数不能全部是 C++的标准类型。

（9）运算符重载函数可以是类的成员函数，也可以是类的友元函数，也可以是普通函数。

3.3.3 运算符重载的两种方式

对同一个运算符，往往可以通过普通函数、友元函数和类成员函数这三种方式实现重载，以完成同样的功能。通过普通函数实现运算符重载的特点是自定义类不得不将其成员变量公开，以便让普通函数可以访问类的成员变量，这破坏了类的封装性，所以这种重载方式要少用或不用，因为不提倡使用，本文也不做详细说明。C++语言之所以提供此种方式，是为了保持与 C 语言的兼容。通过友元函数重载运算符的特点是不破坏类的封装性，类的成员变量可以是私有的，但这种方式需要在类中定义友元函数，以允许友元函数可以操作类的私有成员变量，这在实际使用中也很不方便，所以非必要时不要使用，这里的必要指的是 C++中有一些运算符不能用类成员函数的方式实现重载，比如流提取符 cout 和 cin 就必须使用友元函数实现重载。

通过类成员函数重载运算符是我们推荐使用的，运算符重载函数是类的成员函数，正好满足了类的封装性要求。这种运算符重载的特点是类本身就是一个运算符参数，参见实例 3.7。以下分别对一些主要的运算符重载的具体情况展开讨论。

1. 以类成员函数形式重载运算符

例 3.7 类成员函数形式的运算符重载。

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

```
class complex //定义复数类  complex
{
private:
    double real, imag; //private  成员，分别代表实部和虚部
public:
    complex(double r = 0.0, double i = 0.0) //带缺省参数值的构造函数
    {
        real = r;
        imag = i;
    }
    complex operator += (const complex &); //成员函数形式重载+=运算符
    complex operator + (const complex &); //成员函数形式重载+运算符
    complex operator - (const complex &); //成员函数形式重载-运算符
    complex operator - (); //成员函数形式重载一元运算符-（取反）
    complex operator * (const complex &); //成员函数形式重载*运算符
    complex operator / (const complex &); //成员函数形式重载/运算符
    complex & operator++ (); //成员函数形式重载前置++运算符
    complex operator++ (int); //成员函数形式重载后置++运算符
    void disp() //成员函数，输出复数
    {
        cout << real << " + " << "i*" << imag << endl;
    }
};
complex complex::operator += (const complex & CC) //+=的实现
{
    real += CC.real;
    imag += CC.imag;
    return (*this);
}
complex complex::operator + (const complex & CC) //+的实现
{
    return complex(real + CC.real, imag + CC.imag);
}
complex complex::operator - (const complex & CC) //-的实现
{
    return complex(real - CC.real, imag - CC.imag);
}
complex complex::operator * (const complex & CC) //*的实现
{
    return complex(real * CC.real - imag * CC.imag, real * CC.imag + imag * CC.real);
```

```
}

complex complex::operator / (const complex & CC) //运算符/的实现
{
    return complex((real * CC.real + imag * CC.imag) / (CC.real * CC.real + CC.imag * CC.imag), (imag *
CC.real - real * CC.imag) / (CC.real * CC.real + CC.imag * CC.imag));
}
complex complex::operator - () //单目运算符-即取反的实现
{
    return complex(-real, -imag);
}
complex & complex::operator++ () //前置++的实现
{
    cout << "前置++" << endl;
    ++real;
    ++imag;
    return (*this);
}
complex complex::operator++(int) //后置++的实现，体会和前置++的区别
{
    cout << "后置++" << endl;
    complex cTemp = (*this); //最终的返回值是原来的值，因此需要先保存原来的值
    real++;
    imag++;
    return cTemp;
}
int main()
{
    complex cx1(1.0, 2.0), cx2(3.0, 4.0), cxRes;
    cxRes += cx2; //相当于 cxRes.operator+=(cx2)
    cxRes.disp();
    cxRes = cx1 + cx2; //相当于 cx1.operator+(cx2)
    cxRes.disp();
    cxRes = cx1 - cx2; //相当于 cx1.operator-(cx2)
    cxRes.disp();
    cxRes = cx1 * cx2; //相当于 cx1.operator*(cx2)
    cxRes.disp();
    cxRes = cx1 / cx2; //相当于 cx1.operator/(cx2)
    cxRes.disp();
    cxRes = -cx1; //相当于 cx1.operator-()
```

```
    cxRes.disp();
    cout << endl;
    complex cx3(1.0, 1.0), cx4(5.0, 5.0);
    cxRes = ++cx3; //相当于 cx3.operator++()
    cxRes.disp();
    cx3.disp();
    cout << endl;
    cxRes = cx4++; //相当于 cx4.operator++(0)
    cxRes.disp();
    cx4.disp();
    cout << endl;
    //注意下述语句在友元函数形式和成员函数形式中的对比
    cxRes = cx1 + 5; //相当于 cx1.operator+(5)  或  cx1.operator+(complex(5))
    cxRes.disp();
    //cxRes = 5 + cx1; //错误，相当于 5.operator+(cx1);
    //cxRes.disp();
    return 0;
}
```

程序运行结果如下：

```
3 + i*4
4 + i*6
-2 + i*-2
-5 + i*10
0.36 + i*0.08
-1 + i*-2

前置++
2 + i*2
2 + i*2

后置++
5 + i*5
6 + i*6


6 + i*2
Press any key to continue
```

注意：成员函数形式的运算符函数声明和实现与成员函数类似，首先应当在类定义中声明该运算符函数，声明的具体形式为：

```
返回类型  operator 运算符（参数列表）;
```

也可以在类定义之外定义运算符函数，但要使用作用域运算符“::”，类外定义的基本格

式为:

```
返回类型  类名::operator 运算符（参数列表）
{
    ...
}
```

2. 以友元函数形式重载运算符

例 3.8 友元函数形式的运算符重载。

```
#include <iostream>
using namespace std;
class complex                                   //定义复数类 complex
{
private:
    double real,imag;                           //private 成员，分别代表实部和虚部
public:
    complex(double r=0.0,double i=0.0)          //带缺省参数值的构造函数
    {
        real=r;
        imag=i;
    }
    friend complex operator + (const complex &,const complex &);//友元函数形式重载+
    friend complex operator - (const complex &,const complex &); //友元函数形式重载-
    friend complex operator - (const complex &); //友元函数形式重载一元运算符-（取反）
    friend complex operator * (const complex &,const complex &);//友元函数形式重载*
    friend complex operator / (const complex &,const complex &); //友元函数形式重载/
    friend complex & operator ++(complex &); //友元函数形式重载前置++
    friend complex operator ++(complex &,int); //友元函数形式重载后置++
    void disp() //成员函数，输出复数
    {
        cout<<real<<" + "<<"i*"<<imag<<endl;
    }
};
complex operator +(const complex & C1,const complex & C2)      //+的实现
{
    return complex(C1.real + C2.real, C1.imag + C2.imag);
}
complex operator -(const complex & C1,const complex & C2)      //-的实现
{
    return complex(C1.real - C2.real, C1.imag - C2.imag);
}
complex operator -(const complex & C1)      //单目运算符-即取反的实现
```

```
{
    return complex(-C1.real, -C1.imag);
}
complex operator *(const complex & C1,const complex & C2)  //*的实现
{
    return complex(C1.real * C2.real - C1.imag * C2.imag, C1.real * C2.imag + C1.imag * C2.real);
}
complex operator /(const complex & C1,const complex & C2)  //运算符/的实现
{
    return complex((C1.real * C2.real + C1.imag * C2.imag) / (C2.real * C2.real + C2.imag * C2.imag),
    (C1.imag * C2.real - C1.real * C2.imag) / (C2.real * C2.real + C2.imag * C2.imag));
}
Complex & operator ++(complex & C1)        //前置++的实现
{
    cout << "前置++" << endl;
    C1.real += 1;
    C1.imag += 1;
    return C1;
}
complex operator ++(complex & C1,int)      //后置++的实现，体会和前置++的区别
{
    cout << "后置++" << endl;
    complex ctemp = C1;
    ++C1;
    return ctemp;
}
int main()
{
    complex cx1(1.0, 2.0), cx2(3.0, 4.0), cxRes;
    cxRes = cx1 - cx2;      //相当于 operator-(cx1, cx2)
    cxRes.disp();
    cxRes = -cx1;           //相当于 operator-(cx1)
    cxRes.disp();
    cxRes = cx1 + cx2;      //相当于 operator+(cx1, cx2)
    cxRes.disp();
    cxRes = cx1 * cx2;      //相当于 operator*(cx1, cx2)
    cxRes.disp();
    cxRes = cx1 / cx2;      //相当于 operator/(cx1, cx2)
    cxRes.disp();
    complex cx3(1.0, 1.0), cx4(5.0, 5.0);
```

```
    cxRes = ++cx3;          //相当于 operator++(cx3)
    cxRes.disp();
    cx3.disp();
    cxRes = cx4++;          //相当于 operator++(cx4, 0)
    cxRes.disp();
    cx4.disp();
    //注意下述语句在友元函数形式和成员函数形式的差别
    cxRes = cx1 + 5; //相当于 operator+(cx1, 5);
    cxRes.disp();
    cxRes = 5 + cx1; //相当于 operator+(5, cx1);
    cxRes.disp();
    return 0;
}
```

程序运行结果如下：

```
-2 + i*-2
-1 + i*-2
4 + i*6
-5 + i*10
0.36 + i*0.08
前置++
2 + i*2
2 + i*2
后置++
前置++
5 + i*5
6 + i*6
6 + i*2
6 + i*2
Press any key to continue
```

注意：重载为友元函数的运算符重载函数的声明格式为：

```
friend 返回类型 operator 运算符 (参数表);
```

我们知道单目运算符是指该运算符只操作一个数据，由此产生另一个数据，如正数运算符和负数运算符。双目运算符则是要操作两个数据，由此产生另一个数据，如一般的算术运算符和比较运算符都是双目运算符。

①在类中重载一元运算符@为类运算符的形式为：

```
返回值类型 类名::operator@()
{
    //…
}
```

②在类中重载一元运算符@为友元运算符的形式为：

```
返回值类型 operator@(const 类名 & obj)
{
    //…
}
```

③在类中重载二元运算符@为类运算符的形式为：

```
返回值类型 类名::operator@(参数 1)
{
    //…
}
```

双目运算符需要两个操作数，这里的函数原型为何仅列出了一个参数？这是因为类本身也是一个参数，存取该参数可以通过 this 指针来实现。

④在类中重载二元运算符@为友元运算符的形式为：

```
返回值类型 operator@(参数 1，参数 2)
{
    //…
}
```

3.3.4 几种特殊运算符的重载

1. 赋值（=）运算符的重载

缺省的赋值运算符是实现对象间的按位拷贝，如果类成员中含有指针类型的成员变量，一般应该重载该类的赋值运算符，因为这时调用缺省的赋值运算符没有意义。

例 3.9 赋值（=）运算符的重载。

```
#include <iostream>
using namespace std;
class CMyString
{
private:
    char *m_pszData;
public:
    CMyString(char *pszData);                       //普通构造函数
    CMyString(CMyString & objStr);                  //拷贝构造函数
    CMyString & operator=(CMyString & objStr);      //重载=操作符
    CMyString & operator=(char *pszData);           //重载=操作符
    printS()
    {
        cout<<m_pszData<<endl;
```

```
    };
    ~CMyString()
    {
        delete []m_pszData;
    }
};
CMyString::CMyString(char *pszData)
{
    m_pszData=new char[strlen(pszData)+1];
    strcpy(m_pszData,pszData);
}
CMyString::CMyString(CMyString & objStr)
{
    m_pszData=new char[strlen(objStr.m_pszData)+1];
    strcpy(m_pszData,objStr.m_pszData);
}
CMyString & CMyString::operator=(CMyString & objStr)
{
    if(this==&objStr)
        return *this;
    delete []m_pszData;
    m_pszData=new char[strlen(objStr.m_pszData)+1];
    strcpy(m_pszData,objStr.m_pszData);
    return *this;
}
CMyString & CMyString::operator=(char *pszData)
{
    delete []m_pszData;
    m_pszData=new char[strlen(pszData)+1];
    strcpy(m_pszData,pszData);
    return *this;
}

int main()
{
    CMyString s1="abc";//调用拷贝构造函数 CMyString(char *pszData)
    s1.printS();
    CMyString s2="xyz";//调用拷贝构造函数 CMyString(char *pszData)
    s2.printS();
    s1="123";//调用赋值函数 CMyString &operator=(char *pszData)
```

```
    s1.printS();
    s1=s2; //调用赋值函数 CMyString & operator=(CMyString &objStr)
    s1.printS();
    return 0;
}
```

程序运行结果如下：

```
abc
xyz
123
xyz
Press any key to continue
```

注意：赋值函数和拷贝构造函数的区别和联系。

相同点：都是将一个对象拷贝到另一个中去。

不同点：拷贝构造函数涉及到要新建立一个对象。如下三种情况需要调用拷贝构造函数：

①形如 A a = b；由于对象 a 是新建的对象，所以这种情况并不调用赋值运算符，而是调用拷贝构造函数。

②对象做函数参数时，由于要在堆栈中建立新对象，所以调用拷贝构造函数。

③对象是函数的返回值，而赋值运算符不涉及到对象的建立。

在类中如果不定义赋值运算符和拷贝构造函数，编译器将为之生成默认的赋值运算符函数和拷贝构造函数，它只进行按位拷贝，来实现对象的拷贝或赋值。一般来讲当一个类定义有指针，默认的赋值运算符函数和拷贝构造函数将不能完成对象的赋值和拷贝，因为按位拷贝的结果是将一对象的指针值拷贝到另一个对象中，这会导致相同的内存被多个对象引用的错误。

要注意一个问题，引用或指针作为函数参数或返回值，函数调用时会不会调用类的拷贝构造函数呢？答案是不会的，因为函数在调用时不会涉及到新对象的建立，仅仅是将原对象的地址进行了拷贝。

2. []下标运算符重载

标准情况下，[]运算符用于访问数组的元素。可以通过重载下标运算符为类运算符。使得可以像访问数组元素一样来访问对象中的成员变量。C++只允许把下标运算符重载为非静态的成员函数。

下标运算符函数的定义形式为：

```
T1 T::operator[] (index);
```

其中 T1 为希望返回的数据类型，T 为类名，index 为下标，一般情况下是整型的数据类型。如需访问例 3.10 中的 CMyString 的某个字符，在类中可声明重载[]运算符函数。

```
char operator[](int iIndex);
```

在外部完成该运算符重载函数的实现。

```
char CMyString::operator[](int iIndex)
{
```

```
        if(iIndex<strlen(m_pszData))
            return m_pszData[iIndex];
        return 0;
    }
```

3. 输入>>、输出<<运算符的重载

>>和<<运算符只能重载为友元函数形式。

例 3.10 对操作符>>、<<的重载。

```
    #include <iostream>
    using namespace std;
    /*解决 VC 6.0 中友元方式重载运算符时无法访问类私有成员的方法，在类定义之前将类和友元操作符函数的原型提前声明一下。*/
    class Complex;
    ostream & operator<<(ostream &os, Complex &C1);          //对操作符<<的重载
    istream & operator>>(istream &is, Complex &C1);          //对操作符>>的重载
    class Complex
    {
    private:
        double imag;                                         //虚部
        double real;                                         //实部
    public:
        Complex(double r=0.0,double i=0.0)                   //构造函数
        {
            real=r;
            imag=i;
        }
        friend ostream & operator<<(ostream & ,Complex & );  //友元函数声明
        friend istream & operator>>(istream & ,Complex & );
    };
    ostream & operator<<(ostream & os,Complex & C1)          //对操作符<<的重载
    {
        os<<C1.real<<"+i*"<<C1.imag<<endl;
        return os;
    }
    istream & operator>>(istream & is,Complex& C1)           //对操作符>>的重载
    {
        is>>C1.real;
        while (is.get()!='*')
        {
        }
        cin>>C1.imag;
```

```
    return is;
}

int main()
{
    Complex c1(2.5,3.1);
    cin>>c1;
    cout<<c1;
    return 0;
}
```

运行结果：

```
2 //键盘输入 2 回车
* //键盘输入*回车
6 //键盘输入 6 回车
2+i*6
Press any key to continue
```

3.4 本章小结

在类中定义的成员变量和成员函数的作用域是整个类，这些名称只在类中可见，在类外是不可见的。如果在成员函数中的局部变量和该类的成员变量同名时，引用类成员变量可以用 this 指针或“类名::”的形式。如果某个函数或者类需要访问某类中的非公有成员，就需要使用友元。友元有三种形式，即非成员函数作为友元；一个类的成员函数作为另外一个类的友元；一个类作为另一个类的友元。友元的声明位置可以是公有的，也可以是私有的。为了使用户自定义数据类型的数据操作与系统内部定义的数据类型的数据操作形式一致，C++提供了运算符的重载机制。C++中有的运算符可以重载，有的运算符不能重载，详见本章 3.3.2 节中的说明，而且不能自己定义新的运算符。

习题

1．实现友元有哪几种方式？
2．运算符重载的意义？运算符重载的三种方式？不允许重载的 5 个运算符是哪些？
3．赋值运算符函数和拷贝构造函数的区别与联系？
4．流运算符为什么不能通过类的成员函数重载？一般怎么解决？
5．分别对友元的三种情况，写一个 CStudent 举例说明。
6．已知类 CString 的原型为：

```
class CString
```

```
{
private:
    char *m_pData; //用于保存字符串
public:
    CString(const char *str = NULL); //普通构造函数
    CString(const String &other); //拷贝构造函数
    CString& operator=(const char* cstr);
    CString& operator=(const CString & str);
    CString& operator+=( const char* pszSrc);//字符串的连接
    CString& operator+=(const CString & pszSrc);
    friend bool operator!=( const CString & str1, const CString & str2) ;//是否不同
    friend bool operator==( const CString & str1, const CString & str2) ;//是否相同
    friend bool operator<( const CString & str1, const CString & str2) ;//是否小于
    friend bool operator>(const CString & str1, const CString & str2) ;//是否大于
    ~ CString( ); //析构函数
};
```

请完成类中声明的成员函数和友元函数的实现。

7．请上机运行例 3.1～例 3.10。