

第三章 类域、友元、运算符重载

前边介绍了 C++ 中面向对象编程的基本概念，讨论了类的定义、实现以及对象的创建和组织，特殊数据成员和函数成员的用法等，本章将在前文介绍的基础上，探讨关于对象的一些深层次内容。

3.1 类作用域(就近原则)

作用域可分为类作用域、类名的作用域 以及对象的作用域几部分内容。

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，也要通过对象名来调用，对于 static 成员，要指定类名来调用。

如果发生“屏蔽”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，这有些类似于使用“::”访问全局变量。

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

int x = 100;           //定义性声明，全局int型变量x
int z = 200;           //定义性声明，全局int型变量z

class Example          //Example类定义
{
    int x;
    int y;
public:
    Example(int xp = 0, int yp = 0) //构造函数
    {
        x = xp;
        y = yp;
    }
    void print(int x) //成员函数print，形参为x
    {
        cout << "传递来的参数: " << x << endl;
        cout << "成员x: " << (this->x) << ", 成员y: " << y << endl; //形参x覆盖掉了成员x和全局变量x
        cout << "除了this指针外，还可以使用类名::的形式: " << endl; //此处的y指的是成员y，如果要访问成员x，可使用this指
        cout << "成员x: " << Example::x << ", 成员y: " << y << endl; //或使用类名加作用域限定符的形式指明要访问成员x
        cout << "全局x: " << (::x) << endl; //访问全局变量x
        cout << "全局z: " << z << endl; //没有形参和数据成员会对全局变量z构成屏蔽，直接访问
    }
};

int main()
{
    Example ex1; //声明一个Example类的对象ex1
    ex1.print(5); //调用成员函数print()
    return 0;
}
```

执行结果：

```
传递来的参数: 5
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: 100
全局z: 200
Press any key to continue
```

3.2 类定义的作用域与可见域

和函数一样，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 3 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++ 类是定义在该作用域中，我们在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)

一个类可以定义在另一类的定义中，这是所谓**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 的类名。当然，如果 A 的访问权限是 private，则只能在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。见下边代码 1

（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，其作用域仅仅限于定义所在块，不能在块外使用类名声明该类的对象。见代码 2

（4）类名也存在覆盖

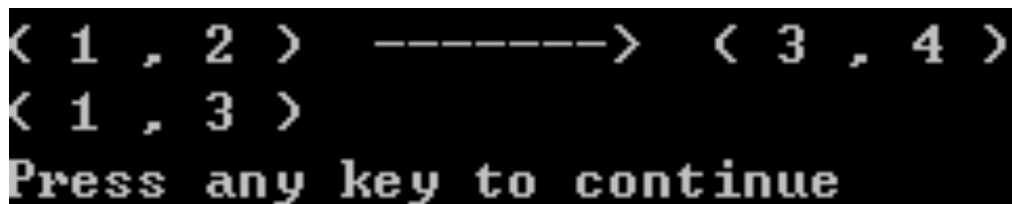
和普通变量的覆盖原则一样，类名也存在“屏蔽”和“覆盖”，不过，依旧可使用作用域声明符“::”指定具体使用的类名，如“::类名”访问的是全局类，使用“外部类::嵌套类”访问嵌套类。

例程：嵌套类(类中类)

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

class line //line类定义
{
public:
    class point //point类定义在line类内，且为public属性，外部可访问
    {
    private: //point类内私有成员列表
        int x;
        int y;
    public:
        point(int xp = 0, int yp = 0) //point类构造函数，带缺省参数值
        {
            x = xp;
            y = yp;
        }
        void printpoint(); //point类成员函数原型，外部实现
    };
private:
    point p1, p2; //line内两个point对象成员
public:
    line(int x1, int y1, int x2, int y2):p1(x1, y1), p2(x2, y2) //构造函数，初始化表
    {
    }
    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;
    }
};
```

执行结果：



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

例程：块作用域

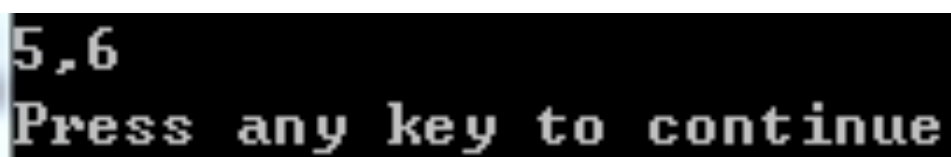
```
#include <iostream>
using namespace std;

int main()
{
    void Work(int, int);    //work函数原型声明
    Work(5, 6);            //work函数调用
    return 0;
}

void Work(int a, int b)
{
    class point            //类定义在函数内，在函数外无法使用point创建对象
    {
    private:
        int x, y;
    public:
        point(int xp = 0, int yp = 0)
        {
            x = xp;
            y = yp;
        }
        void print()
        {
            cout << x << "," << y << endl;
        }
    };

    point pt(a, b);        //函数内，创建point类的对象pt
    pt.print();            //输出提示信息
}
```

执行结果：

A screenshot of a terminal window with a black background and white text. The first line displays '5,6'. The second line displays 'Press any key to continue'.

3.3 对象的生存期、作用域和可见域

类名无生存期，只有作用域和可见域。

对象有生存期，对象的生存期也是对象中所有非静态成员变量的生存期，这些成员变量都随着对象的创建而创建，随着对象的撤销而撤销。

对象的生存期、作用域和可见域取决于对象的创建位置，同样有全局、局部、类内之分，和前面关于普通变量的介绍并无区别，这里便不在赘述。

关于对象创建有几点问题需要强调。

3.3.1 先定义，后实例化

类的定义一定要在类对象声明之前，因为编译器要知道需要为类分配多大的内存空间，仅仅对类进行声明是不够的，如：

```
class B;           //声明
B objectB;         //创建B类的对象。 错误
class B           //B类定义
{
    .....
};
```

但是，如果不创建 B 类的对象，而仅仅是声明一个指向类型 B 对象的指针（或引用），是可行的。如：

```
class B;           //声明
B* pB=NULL;        //创建B类的对象。正确
B* pC=new B;        //创建B类的对象。错误
class B           //B类定义
{
    .....
};
```

例程：

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

class point;        //类声明
point ptA;          //错误，类对象的创建必须在类定义之后，因为该语句需要调用构造函数
point *pB = NULL;   //正确
point *pC = new point(); //错误，因为new语句会导致调用构造函数

class point
{
private:
    int x;
    int y;
public:
    point(int ix = 0, int iy = 0):x(ix), y(iy) {}
};
int main()
{
    return 0;
}
```

3.3.2 对象内存释放与堆内存

一种普遍的误解是“如果对象被撤销，其占据的内存空间被释放，那么对象创建时和函数执行中通过 new 和 malloc 申请的动态内存也会被自动释放”，实际上，除非显式地用 delete 或 free 释放，申请的动态内存不会随着对象的撤销而撤销，相反，撤销了对象，却没有释放动态内存反而会引起内存泄露。

当然，在程序结束时，操作系统会回收程序所开辟的所有内存。尽管如此，还是要养成 new/delete、malloc/free 配对的编程习惯，及时释放已经无用的内存。

3.4 友元

一般来说，类的私有成员只能在类的内部访问，类外的函数是不能访问它们的。

但是，可以将一个函数定义为类的友元函数，这时该函数就可以访问该类的私有成员了。

同时，也可以把另一个类 B 定义为本类 A 的友元类，这样 B 类就可以访问 A 类的任何成员了。

下面来具体讨论友元函数和友元类的概念。

3.4.1 友元之非成员函数

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;

友元函数中可访问类的 private 成员。

用下面的比喻形容友元函数可能比较恰当，将类比作一个家庭，类的 **private 成员相当于家庭的秘密**，一般的外人是不允许探听这些秘密的，只有 **friend (朋友)** 才有能力探听这些秘密。

例程:

```
#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 Point //Point类定义
{
private:
    int x, y;
    friend float dis(Point &p1, Point &p2); //友元函数的声明, 声明位置没有关系, 可以是public, 也可能是private
public:
    Point(int i = 0, int j = 0) //构造函数, 带缺省参数值
    {
        x = i;
        y = j;
    }
    void disp() //成员函数
    {
        cout << "(" << x << "," << y << ")";
    }
};
float dis(Point &p1, Point &p2) //友元函数中可访问类的private成员
{
    float d = sqrt((p1.x - p2.x)*(p1.x - p2.x) + (p1.y - p2.y)*(p1.y - p2.y));
    return d;
}
int main()
{
    Point p1(1, 2), p2(4, 5); //声明两个Point类的对象p1和p2
    p1.disp(); //显示点p1的信息
    cout << "与";
    p2.disp(); //显示点p2的信息
    cout << "距离=" << dis(p1, p2) << endl; //利用友元函数计算两点举例

    return 0;
}
```

执行结果:



```
<1,2>与<4,5>距离=4.24264
Press any key to continue
```

3.4.2 友元之成员函数

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。

成员形式的友元函数见下边代码:

```

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

class Point; //声明Point类

class Line //定义Line类
{
public:
    float dis(Point& p1, Point& p2); //友元函数的原型，作为Line类的成员函数
};
class Point //定义Point类
{
private:
    int x,y; //private型数据成员x和y
    friend float Line::dis(Point &p1, Point &p2); //友元的声明
public:
    Point(int i = 0, int j = 0) //Point类的构造函数，带缺省参数
    {
        x = i;
        y = j;
    }
    void disp() //成员函数disp(), 用来输出点的信息
    {
        cout << "(" << x << "," << y << ")";
    }
};
float Line::dis(Point &p1, Point &p2) //Line类内成员函数dis的实现，作为Point类的友元函数
{
    float d = sqrt((p1.x - p2.x) * (p1.x - p2.x) + (p1.y - p2.y) * (p1.y - p2.y));
    return d; //可访问Point类对象的private成员
}

int main()
{
    Line line1; //声明一个Line类的对象line1
    Point p1(1,2), p2(4,5); //声明两个Point类的对象p1和p2

    p1.disp(); //输出点p1的信息
    cout << " 与 ";
    p2.disp(); //输出点p2的信息

    cout << " 的距离 = " << line1.dis(p1, p2) << endl;
    //通过调用line1的成员函数dis计算两个点间的距离

    return 0;
}

```

执行结果：

```

<1,2> 与 <4,5> 的距离 = 4.24264
Press any key to continue

```

上述代码中，Line 类的成员函数 dis(...)的实现必须在类外进行，且必须在 Point 类的定义之后。因为其参数中包含了 Point 这种类型。

Line 类的 `dis()` 函数本来是不能访问 `Point.x` 和 `Pint.y` 这种 `Point` 类的 `private` 成员的,但在 `Point` 类中将 `dis()` 申明为友元函数后就能访问了。但 `dis()` 函数依然不是 `Point` 类的成员函数。**也就是说, `dis()` 只是 `Point` 类的朋友了,** 可以访问 `Point` 类的私有成员变量 `x` 和 `y` 了。

3.4.3 友元函数的重载

要想使得一组重载函数全部成为类的友元,必须一一声明,否则只有匹配的那个函数会成为类的友元,编译器仍将其他的当成普通函数来处理。

```
class Exp
{
public:
    friend void test(int);
};
void test();
void test(int);
void test(double);
```

上述代码中,只有“`void test(int)`”函数是 `Exp` 类的友元函数,“`void test()`”和“`void test(double)`”函数都只是普通函数。

3.4.4 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。

A 可以在 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
```

例程:

```

#include<iostream>
#include <cmath>
using namespace std;
class CLine;           //声明类CLine
class CPoint           //定义CPoint类
{
private:
    int x, y;
    friend CLine;       //友元类的声明，位置同样不受限制
public:
    CPoint(int i = 0, int j = 0) //构造函数，带缺省参数值
    {
        x = i;
        y = j;
    }
    void disp()              //成员函数，输出点的信息
    {
        cout << "(" << x << "," << y << ")";
    }
};
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 dis(CPoint& p1,CPoint& p2) //可访问p1和p2的private成员
    {
        float d;
        d=sqrt((p1.x-p2.x)*(p1.x-p2.x)+(p1.y-p2.y)*(p1.y-p2.y));
        return d;
    }
    void Set(CPoint* p1,int a,int b) //可访问p1和p2的private成员
    {
        p1->x=a;
        p1->y=b;
    }
};

int main()
{
    CLine cz1;           //声明一个CLine类的对象cz1
    CPoint p1(1,2),p2(4,5); //声明两个CPoint类对象p1和p2

    p1.disp();           //输出点p1的信息
    cout<<"与";
    p2.disp();           //输出点p2的信息
    cout<<"距离="<<cz1.dis(p1,p2)<<endl; //调用cz1的成员函数dis计算两点间距
    cz1.Set(&p1,3,4);    //调用cz1的成员函数Set改写p1中的private成员x和y
    p1.disp();           //修改后的点p1信息输出
    cout<<endl;         //换行
    return 0;
}

```

执行结果:

```

<1,2>与<4,5>距离=4.24264
<3,4>
Press any key to continue

```

3.4.5 友元是否破坏了封装性

不可否认，友元在一定程序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的。

友元的存在，使得类的接口扩展更为灵活，使用友元进行运算符重载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。

3.5 运算符重载

首先来看一个问题：下边代码定义了一个复数类 complex，然后用 complex 定义了 2 个复数，如何实现这 2 个复数的加法呢？

这个问题的解决就需要用到运算符重载的知识，下面详细讲解。

```
#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) //构造函数，带缺省参数值
    {
        real = r;
        imag = i;
    }
    void disp()                             //成员函数，输出复数
    {
        cout << real << " + " << "i*" << imag << endl;
    }
};

int main()
{
    complex cx1(1.0, 2.0);
    complex cx2(3.0, 4.0);
    //complex cxRes = cx1 + cx2;             //错误

    return 0;
}
```

3.5.1 运算符重载规则

运算符是一种通俗、直观的函数，比如：`int x=2+3;`语句中的“+”操作符，**系统本身就提供了很多个重载版本：**

```
int operator + (int, int);
```

```
double operator + (double, double);
```

可以重载的运算符有：

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() -> ->* , []

不能重载的运算符有：

成员访问符.

成员指针访问运算符.*

域运算符::

长度运算符 sizeof

条件运算符?:

不能臆造并重载一个不存在的运算符：如@, #, \$等。

注意：既然是操作符重载，就必然会访问类的私有成员变量，根据类的封装性要求，除了友元函数和成员函数外，其他任何外部操作都是违规的，所以**不能用普通函数来重载操作符**。

3.5.2 以成员函数形式重载运算符

成员函数形式的运算符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，声明的具体形式为：

返回类型 operator 运算符（参数列表）；

既可以在类定义的同时定义运算符函数使其成为 inline 型，也可以在类定义之外定义运算符函数，但要使用作用域限定符“::”，类外定义的基本格式为：

返回类型 类名::operator 运算符（参数列表）例程：

```
#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) //构造函数，带缺省参数值
    {
        real = r;
        imag = i;
    }
    complex operator+=(const complex &);    //成员函数形式重载加+=
    complex operator+(const complex &);    //成员函数形式重载加+
    complex operator-(const complex &);    //成员函数形式重载减-
    complex operator-();                   //成员函数形式重载一元-（取反）
    complex operator*(const complex &);    //成员函数形式重载乘*
    complex operator/(const complex &);    //成员函数形式重载除/
    complex& operator++();                 //成员函数形式重载前置++
    complex operator++(int);               //成员函数形式重载后置++
    void disp()                            //成员函数，输出复数
    {
        cout << real << " + " << "i*" << imag << endl;
    }
};

complex complex::operator+=(const complex& CC) //加+=的实现
{
    real += CC.real;
    imag += CC.imag;
    return (*this);
}
complex complex::operator-(const complex& CC) //减-的实现
{
    return complex(real - CC.real, imag - CC.imag);
}
complex complex::operator*(const complex& CC) //乘*的实现
{
    return complex(real * CC.real - imag * CC.imag, real * CC.imag + imag * CC.
real);
}
complex complex::operator/(const complex& CC) //除/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-，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);
    //最终的返回值的是原来的值，因此需要先保存原来的值
    ++(*this);                                //返回后原来的值需要加1
    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;
}

```


3.5.3 以友元函数形式重载运算符

用成员函数重载双目运算符时，左操作数无须用参数输入，而是通过隐含的 `this` 指针传入，这种做法的效率比较高

此外，操作符还可重载为友元函数形式，**这将没有隐含的参数 `this` 指针**。
对双目运算符，友元函数有 2 个参数，对单目运算符，友元函数有一个参数。

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);`

注意，在 VC 6.0 如果用

```
#include <iostream>
```

```
using namespace std;
```

编译器会出错，将以上两行替换为

```
#include <iostream.h> 即可通过
```

因为 VC 6.0 编译器的命名空间不包含以友元方式的运算符重载（了解即可），但是在其它编译器中一般不会报错的

运算符重载的主要目的是为了让类对象能像普通数据类型一样能够进行加减乘除，自加自减等操作，非常直观方便。现在来回顾 C++ 的自加减(分前置与后置)以及不等号非运算符，赋值运算符的重载。

注意和友元函数实现的区别：

++重载

(1)前置++运算符的重载方式：

成员函数的重载：函数类型& operator++()

友元函数的重载：friend 函数类型& operator++(类的类型&)

(2)后置++运算符的重载方式：

成员函数的重载：函数类型& operator++(int)

友元函数的重载：friend 函数类型& operator++(类的类型&, int)

注意，为了区分前置++与后置++的区别，需要在参数后增加一个"int"以示区分。含有"int"的重载方式为后置++，否则为前置++。前置--与后置--类似用法。前面说过，**成员函数与友元函数的重载如果同时存在时，会先调用成员函数的重载，但是在++或--时，成员函数与友元函数的重载是不能同时存在的。**

3.5.4 友元函数形式和成员函数形式的比较

对于绝大多数可重载操作符来说，两种重载形式都是允许的。但对下标运算符[]、赋值运算符=、函数调用运算符()、指针运算符->，只能使用成员函数形式重载，因为他们都和构造函数的重载有关，不是普通的成员函数。（知道即可）

对于如下代码：

```
complex c1(1.0, 2.0), cRes;
```

```
cRes = c1 + 5; // #1 合法
```

```
cRes = 5 + c1; // #2 非法
```

友元函数形式重载的都是合法的，可转换成：

```
cRes = operator+(c1, 5); // #1 合法
```

```
cRes = operator+(5, c1); // #2 合法
```

3.5.5 对运算符重载的补充说明

运算符重载可以改变运算符内置的语义，如以友元函数形式定义的加操作符：

```
complex operator +(const complex& C1, const complex& C2)
{
    return complex(C1.real-C2.real, C1.imag-C2.imag);
}
```

明明是加操作符，但函数内却进行的是减法运算，这是合乎语法规则的，不过却有悖于人们的直觉思维，会引起不必要的混乱，因此，除非有特别的理由，尽量使重载的运算符与其内置的、广为接受的语义保持一致。

此外，还要注意各运算符之间的关联，比如下列几个指针相关的操作符：

[]、*（取内容）、&（取地址）、->

编译器对这些操作符的解释有一种“等价”关系，因此，如果对其中一个进行了重载，其他对应的操作符也应被重载，使等价操作符完成等价的功能。

3.6 运算符重载范例

本节重点演示几种特殊运算符的重载示例，运算符的重载是很灵活的工具，使用得当，会带来意想不到的便利。

. 3. 6. 1 赋值运算符=

赋值运算是一种很常见的运算，如果不重载赋值运算符，编译器会自动为每个类生成一个缺省的赋值运算符重载函数，先看下面的语句：

对象 1=对象 2;

实际上是完成了由对象 2 各个成员到对象 1 相应成员的复制，其中包括指针成员，这和第二章中复制构造函数和缺省复制构造函数有些类似，如果对象 1 中含指针成员，并且牵扯到类内指针成员动态申请内存时，问题就会出现。

注意下述两个代码的不同：

类名 对象 1=对象 2; //复制构造函数

和

类名 对象 1; //默认构造函数

对象 1=对象 2; //赋值运算符函数

注意：区分复制构造函数和复制运算符函数的区别

例程：

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

class computer
{
private:
    char *brand;           //字符指针brand
    float price;
public:
    computer()             //无参构造函数
    {
        brand = NULL;     //brand初始化为NULL
        price = 0;
        cout << "无参构造函数被调用" << endl;
    }
    computer(const char* sz,float p)
    {
        brand = new char[strlen(sz)+1]; //构造函数中为brand分配一块动态内存
        strcpy(brand, sz);              //字符串复制
        price = p;
        cout << "带参构造函数被调用" << endl;
    }
    computer(const computer& cp)        //复制构造函数
    {
        brand = new char[strlen(cp.brand) + 1]; //为brand分配动态内存
        strcpy(brand, cp.brand);              //字符串复制
        price = cp.price;
        cout << "复制构造函数被调用" << endl;
    }
}
```

```

/*如果我们没有重载=，则系统会隐式的重载成如下形式：
computer &operator=(const computer &cp)
{
    price = cp.price;
    brand = cp.brand;
    cout<<"系统默认赋值函数被调用"<<endl;
    return (*this);
}*/
//应该使用下述函数取代上述系统隐式的定义
computer &operator=(const computer &cp)    //成员函数形式重载赋值运算符
{
    if (this==&cp)                        //首先判断是否是自赋值，是的话返回当前对象
        return (*this);

    price=cp.price;                        //如果不是自赋值，先对price赋值
    delete[] brand;                       //防止内存泄露，先释放brand指向的内容
    brand=new char[strlen(cp.brand)+1];    //为brand重新开辟一块内存空间
    if (brand!=NULL)                      //如果开辟成功
    {
        strcpy(brand,cp.brand);           //复制字符串
    }
    cout<<"赋值运算符重载函数被调用"<<endl;

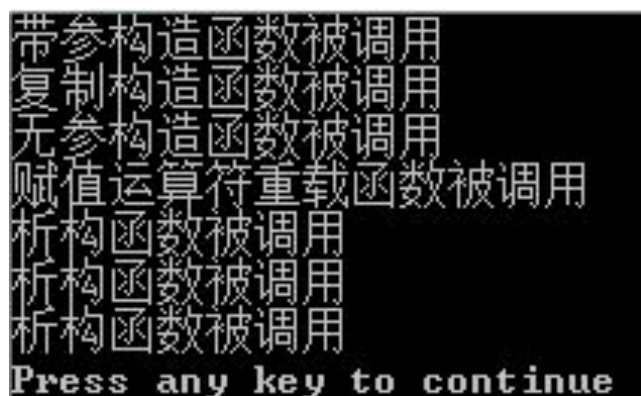
    return (*this);                       //返回当前对象的引用，为的是实现链式赋值
}

~computer()                              //析构函数，释放动态内存，delete[] NULL不会出错
{
    delete[] brand;
    cout << "析构函数被调用" << endl;
}

void print()                             //成员函数，输出信息
{
    cout << "品牌: " << brand << endl;
    cout << "价格: " << price << endl;
}
};
int main()
{
    computer com1("Dell", 2000);           //调用含参构造函数声明对象com1
    computer com2 = com1;                  //调用复制构造函数
    if (true)
    {
        computer com3;                    //调用无参构造函数
        com3 = com1;                       //调用赋值运算符重载函数
    }
    return 0;
}

```

执行结果：



```

带参构造函数被调用
复制构造函数被调用
无参构造函数被调用
赋值运算符重载函数被调用
析构函数被调用
析构函数被调用
析构函数被调用
Press any key to continue

```

3.6.2 函数调用运算符()

函数调用运算符()同样只能重载为成员函数形式。其形式为:

返回类型 operator() (arg1, arg2, ……)

参数个数可以有多个, 没有限制。

针对如下定义:

```
void computer::operator() () {};
```

```
int computer::operator() (int x) {};
```

```
char computer::operator() (char x, char y) {};
```

可以这样调用:

```
computer com1;
```

```
int z = com1(3200);
```

```
char c = com1( 'a' , 'b' );
```

一个类如果重载了函数调用 operator(), 就可以将该类对象作为一个函数使用, 这样的类对象也称为函数对象。函数也是一种对象, 这是泛型思考问题的方式。

例程:

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

class Demo
{
public:
    double operator()(double x, double y)           //重载函数调用符(),两个double型参数
    {
        return x > y ? x : y;                       //返回两个参数中较大的一个
    }
    double operator()(double x, double y, double z) //重载函数调用符(),3个double型参数
    {
        return (x + y) * z;                          //将前两个相加,与第3个参数相乘,返回最后的结果
    }
};

void main()
{
    Demo de;                                           //声明一个类对象
    cout << de(2.5, 0.2) << endl;                    //可以将对象像函数一样使用
    cout << de(1.2, 1.5, 7.0) << endl;
}
```

执行结果:

```
2.5
18.9
Press any key to continue
```

3.6.3 下标运算符[]

下标运算符是个二元运算符，C++编译器将表达式

sz[x];解释为 sz.operator[](x);

一般情况下，下标运算符的重载函数原型如下：

返回类型& operator[](参数类型);

下标运算符的重载函数**只能有一个参数**，不过该参数并没有类型限制，任何类型都可以，如果类中未重载下标运算符，**编译器将会给出下标运算符的缺省定义，此时，参数必须是 int 型**，并且要声明数组名才能使用下标变量，如

computer com[3];

则 com[1]等价于 com.operator[](1)，如果[]中的参数类型非 int 型，或者非对象数组要使用下标运算符时，需要重载下标运算符[]。

例程：

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

class CharSZ //类CharSZ的定义
{
private:
    int iLen;
    char *pBuf;
public:
    CharSZ(int size) //构造函数
    {
        iLen= size;
        pBuf = new char[iLen]; //开辟一块动态内存，字符数组
    }
    ~CharSZ() //析构函数
    {
        delete pBuf; //释放申请的动态内存
    }
    int GetLen() //读取private成员iLen的值
    {
        return iLen;
    }
    char& operator [](int i); //以成员函数形式重载下标运算符
};
```



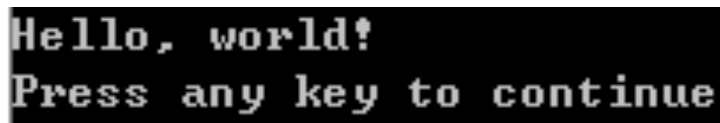
```

char & CharSZ::operator[](int i)//下标运算符重载的实现
{
    static char szNull = '\0'; //用于返回空字符时，由于返回类型为char &，不能直接return '\0';
    if (i < iLen && i >= 0) //如果参数i在有效范围内
    {
        return pBuf[i]; //返回字符数组的第i+1个元素
    }
    else
    {
        cout << "下标越界" << endl; //参数i不合法
        return szNull; //输出空字符，不能直接return '\0';
    }
}
int main()
{
    char *sz = "Hello, world!";
    CharSZ de(strlen(sz) + 1); //对象de中申请的动态内存大小为n，可存放n-1个有效字符（除开'\0'）

    //将sz的内容复制给de
    int i;
    for (i = 0; i < (strlen(sz) + 1); i++)
    {
        de[i] = sz[i];
    }
    //逐个输出de的值
    for (i = 0; i < de.GetLen(); i++)
    {
        cout << de[i];
    }
    cout << endl;
    return 0;
}

```

执行结果：



```

Hello, world!
Press any key to continue

```

3.6.4 new 和 delete 重载

通过重载 new 和 delete, 我们可以自己实现内存的管理策略。new 和 delete 只能重载为类的静态运算符。而且重载时，无论是否显示指定 static 关键字，编译器都认为是静态的运算符重载函数。

为什么必须是静态的函数：（补充）

第一：这是 C++ 的语法规定的

第二：new 是对全局的堆内存操作的，不是对对象本身的内存进行操作的，因此必须为 static，不是属于对象的成员函数，而是类的成员函数

重载 new 时，必须返回一个 void * 类型的指针，它可以带多个参数，但第 1 个参数必须是 size_t 类型，该参数的值由系统确定。

注意：我们在重载 new 函数中调用 new 会进入系统默认的 new，而不会造成

递归调用。

```
static void * operator new(size_t nSize)
{
    cout << "new 操作符被调用, size = " << nSize << endl;
    void * pRet = new char[nSize];
    return pRet;
}
```

重载 delete 时**必须**返回 void 类型, 它可以带有多个参数, 第 1 个参数必须是要释放的内存的地址 void*。如果有第二个参数, 那么第二个参数必须为 size_t 类型。

```
static void operator delete(void * pVoid)
{
    cout << "delete 操作符被调用." << endl;
    delete [] pVoid;
}
```

例程:

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

class CStudent
{
public:
    int iId;
    char szName[10];
public:
    static void * operator new(size_t nSize)
    {
        cout << "new 操作符被调用, size = " << nSize << endl;
        void * pRet = new char[nSize];
        return pRet;
    }
    static void operator delete(void * pVoid)
    {
        cout << "delete 操作符被调用." << endl;
        delete [] pVoid;
    }
};


int main()
{
    CStudent *pStu = new CStudent();

    pStu->iId = 101;
    strcpy(pStu->szName, "Tony");

    delete pStu;

    return 0;
}
```

执行结果：



```
new 操作符被调用, size = 16
delete 操作符被调用.
Press any key to continue
```

注意：

`new point` -> `operator new(int n)` `n` 就是系统传递过来的

`new point[5]` -> `operator new[](int n)` 可需要开辟的个数（字节）

`delete` -> `operator delete(void*)` `void*`就是要释放的地址

`delete[]` -> `operator delete[](void*)`

此处的 `new[]` 中的 “`[]`” 不会调用下标运算符 `operator[](int i)`。

3.6.5 输入>>输出<<的重载

>>和<<运算符只能重载为友元函数形式。

为什么只能重载为友元函数：（了解）

因为<<和>>属于输入和输出类的函数，在 `std` 这个作用域空间下的 `iostream` 头文件里有对他的定义，属于 `iostream` 之下某一个类的成员函数。如果我们把它定义为我们的成员函数，那么<<和>>的 `this` 指针就指向的我们自己的类。但是它应该指向的是 `iostream` 里面的类，否则会在输入输出时导致错误；如果我们想重载，又想调用我们自己的私有数据，只能以友元函数重载（友元之成员函数）。

对操作符<<（输出流）的重载 （>>输入流）

```
friend ostream& operator<< (ostream& os, Complex& C1
{
    os<<C1.real<<" + i*"<<C1.imag<<endl;
    return os;
}
```

对操作符>>的重载

```

friend ostream& operator<<(ostream& os,Complex& C1)
{
    os<<C1.real;
    while (os.get()!='*');
    cin>>C1.imag;
    return os;
}

```

例程：

```

#include <iostream>
using namespace std;
//解决VC6.0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
class Complex;
ostream & operator<<(ostream &os, Complex &C1); //对操作符<<的重载
istream & operator>>(istream &is, Complex &C1);
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;
}

```

3.6.6 指针运算符->的重载

前边有讲：只能以成员函数方式重载（里面关系到构造函数的调用）

直接看例程：

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

class CData
{
public:
    int GetLen() {return 5;}
};

class CDataPtr
{
private:
    CData *m_pData;
public:
    CDataPtr(){m_pData = new CData;}
    ~CDataPtr(){delete m_pData;}
    CData * operator->()           //->操作符重载
    {
        cout << "->操作符重载函数被调用." << endl;
        return m_pData;
    }
};

int main()
{
    CDataPtr p;
    cout << p->GetLen() << endl;  //等价于下面1句：
    cout << (p.operator->())->GetLen() << endl;
    return 0;
}
```

执行结果：

->操作符重载函数被调用.

5

->操作符重载函数被调用.

5

Press any key to continue

(以下内用可不掌握)

3.7 类型转换

前面已经对普通变量的类型转换进行了介绍，本节来讨论类对象和其他类型对象的转换

转换场合有：赋值转换、表达式中的转换、显式转换、函数调用、传递参数时的转换。

转换方向有：由定义类向其他类型的转换、由其他类型向定义类的转换。

下面分别展开讨论

3.7.1 由其他类型向定义类的转换

由其他类型（如 `int`、`double`）等向自定义类的转换是由构造函数来实现的，只有当类的定义和实现中提供了合适的构造函数时，转换才能通过。什么样的构造函数才是合适的构造函数呢？主要有以下几种情况，为便于说明，假设由 `int` 类型向自定义 `point` 类转换：

(1) `point` 类的定义和实现中给出了仅包括只有一个 `int` 类型参数的构造函数

(2) `point` 类的定义和实现中给出了包含一个 `int` 类型参数，且其他参数都有缺省值的构造函数

(3) `point` 类的定义和实现中虽然不包含 `int` 类型参数，但包含一个非 `int` 类型参数如 `float` 类型，此外没有其他参数或者其他参数都有缺省值，且 `int` 类型参数可隐式转换为 `float` 类型参数。

(4) 在构造函数前加上关键字 `explicit` 可以关闭隐式类型转换

例程：

```

#include <iostream>
using namespace std;

class point;

class anotherPoint
{
private:
    double x;
    double y;
public:
    anotherPoint(double xx = 1, double yy = 1) //构造函数，带缺省参数值
    {
        x = xx;
        y = yy;
    }
    void print() //输出函数，点的信息
    {
        cout << "( " << x << " , " << y << " )";
    }
    friend class point; //使point类成为本类的友元类，这样point
                        //类就可以访问anotherPoint 类的private变量了
};

class point
{
private:
    int xPos;
    int yPos;
public:
    point(int x = 0, int y = 0) //构造函数，带缺省参数，两个int型变量
    {
        xPos = x;
        yPos = y;
    }

    void print() //输出函数，点的信息
    {
        cout << "( " << x << " , " << y << " )";
    }
    friend class point; //使point类成为本类的友元类，这样point
                        //类就可以访问anotherPoint 类的private变量了
};

class point
{
private:
    int xPos;
    int yPos;
public:
    point(int x = 0, int y = 0) //构造函数，带缺省参数，两个int型变量
    {
        xPos = x;
        yPos = y;
    }
    point(anotherPoint aP) //构造函数，参数为anotherPoint类对象
    {
        xPos = aP.x; //由于point类是anotherPoint类的友元类，
        yPos = aP.y; //因此这里可以访问anotherPoint的私有变量x和y
    }
    void print() //输出函数，点的信息
    {
        cout << "( " << xPos << " , " << yPos << " )" << endl;
    }
};

```

```

int main()
{
    //1. 将int类型数字5转换成point类型
    point p1; //创建point类对象p1, 采用带缺省参数的构造函数, 即x=0、y=0
    cout << 5 << " 转换成 ";
    p1 = 5; //等价于p1=point(5,0);
    p1.print(); //输出点p1的信息

    //2. 将double类型变量dX转换成point类型
    double dX = 1.2; //声明一个double变量dX
    cout << dX << " 转换成 ";
    p1 = dX; //等价于p1=point(int(dX),0)
    p1.print(); //输出点p1的信息

    //3. 将anotherPoint类型转换成point类型
    anotherPoint p2(12.34, 56.78); //创建anotherPoint类的对象p2
    p2.print();
    cout << " 转换成 ";
    p1 = p2; //等价于p1=point(p2);
    p1.print(); //输出点p1的信息

    return 0;
}

```

执行结果:



```

5 转换成 < 5 , 0 >
1.2 转换成 < 1 , 0 >
< 12.34 , 56.78 > 转换成 < 12 , 56 >
Press any key to continue

```

3.7.2 由定义类型向其它类型转换（转换函数）

可以通过 `operator int()` 这种类似操作符重载函数的类型转换函数来实现由自定义类型向其他类型的转换。如将 `point` 类转换成 `int` 类型等。

在类中定义类型转换函数的形式一般为:

`operator 目标类型名();`

有以下几个使用要点:

- 转换函数必须是成员函数，不能是友元形式（因为我们只想在这个类中这样转换，其它的还是照旧）。
- 转换函数不能指定返回类型，但在函数体内必须用 `return` 语句以传值方式返回一个目标类型的变量。
- 转换函数不能有参数。

例程:

//类型转换函数(由类转换成其他类型)

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

```
class anotherPoint          //anotherPoint类定义
{
private:                   //private成员列表
    double x;
    double y;
public:
    anotherPoint(double xx = 1.11, double yy = 1.11) //构造函数, 带缺省参数值
    {
        x = xx;
        y = yy;
    }
    void print()            //成员函数, 输出点的信息
    {
        cout << "( " << x << " , " << y << " )" << endl;
    }
};
```

```
class point                //Point类定义
{
private:                   //private成员列表
    int xPos;
    int yPos;
public:
    point(int x = 0, int y = 0) //构造函数, 带缺省参数值
    {
        xPos = x;
        yPos = y;
    }
    void print()            //成员函数, 输出点的信息
    {
        cout << "( " << xPos << " , " << yPos << " )" << endl;
    }
    operator int()          //定义Point向int型的转换函数int()
    {
        return xPos;
    }
    operator double()       //定义Point向double型的转换函数double()
    {
        return xPos * yPos;
    }
    operator anotherPoint() //定义Point向anotherPoint型的转换函数another
    {
        return anotherPoint(xPos, yPos);
    }
};
```

```

int main()
{
    point p1(4, 5);           //声明一个point类变量p1
    p1.print();

    //1. point转换成int
    int x1 = p1;              //p1赋值给一个int型变量，point中的转换函数int()被隐式调用
    cout << x1 << endl;

    //2. point转换成double
    double dX = p1;           //p1赋值给一个double型变量，point中的转换函数double()被隐式调用
    cout << dX << endl;

    //3. point转换成anotherPoint
    anotherPoint p2;           //声明anotherPoint类对象p2，构造函数采用缺省值
    p2 = p1;                   //p1赋值给p2，point中的转换函数anotherPoint()被隐式调用
                                //等价于p2=another(p1.xPos,p1.yPos)
    p2.print();                //看p2是否修改成功

    return 0;
}

```

执行结果：

```

< 4 , 5 >
4
20
< 4 , 5 >
Press any key to continue

```

3.8 小结

- 本章继续讨论了面向对象编程的一些概念，首先，对类作用域、类定义的作用域和可见域以及对象的生存期、作用域和可见性进行了介绍
- C++引入了友元机制来对类的接口进行扩展，大大提高了外部访问的灵活性，但这在一定程度上也破坏了类的封装性、违反了信息隐藏的原则，因此，对友元的使用要合理，好钢用在刀刃上。
- 运算符重载是很重要的内容，合理重载运算符使得程序编写简便灵活而且高效，除了极个别的运算符外，绝大多数的运算符都可被重载。运算符重载有成员函数形式和友元函数形式两种，两种方式各有优缺点，对一些运算符来说，只能采用成员函数的形式。

复习:

- (1) 类的作用域(可以在不同类中使用相同的成员名,如果发生“屏蔽”现象,可借助 this 指针或“类名::”形式指明所访问的是类成员,这有些类似于使用“::”访问全局变量。)(类种类,块作用域)(注意优先级顺序)
- (2) 先定义,后实例化(仅仅声明是不够的,但是声明后可以用指针)
- (3) 友元(友元之成员函数、友元之非成员函数、友元类)(特点,可访问私有成员,但不属于类,不可继承)
- (4) 友元函数支持重载
- (5) 运算符重载(operator)(以成员函数的形式重载)(以友元函数形式重载)(注意前置++和后置++)
- (6) []、=、()、->, 只能使用成员函数形式重载
- (7) new 和 delete 重载(无论是否显示指定 static 关键字,编译器都认为是静态的运算符重载函数。)(重载 new 时,必须返回一个 void*类型的指针,它可以带多个参数,但第 1 个参数必须是 size_t 类型,该参数的值由系统确定)(重载 delete 时必须返回 void 类型,它可以带有多个参数,第 1 个参数必须是要释放的内存的地址 void*。如果有第二个参数,那么第二个参数必须为 size_t 类型。)
- (8) 输入输出的重载(只能重载为友元函数形式)
- (9) 由其他类型向定义类的转换(由构造函数实现)
- (10) 由定义类型向其它类型转换(operator int())(必须是成员函数、不能有返回值,但是要 return 一个目标类型、不能有参数)