运算符重载

1、友元

一般来说，类的私有成员只能在类的内部访问，类之外是不能访问它们的。但如果将其他类或函数设置为类的友元（friend），就可以访问了。用这个比喻形容友元可能比较恰当：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。

友元的形式可以分为友元函数和友元类。

```
class 类名
{
    //...
    friend 函数原型;
    friend class 类名;
    //...
}
```

目前为止，我们学过的函数形式有两种：全局函数（自由函数/普通函数）和成员函数。接下来，我们分别进行讨论。现在我们有一个类Point，表示一个二维的点:

```
class Point
{
public:
    Point(int ix = 0, int iy = 0)
    : _ix(ix)
    , _iy(iy)
    {

    }

    void print() const
    {
        cout << "(" << _ix
             << "," << _iy
             << ")";
    }

    friend float distance(const Point &lhs, const Point &rhs);
    friend float Line::distance(const Point &lhs, const Point &rhs);
private:
    int _ix;
    int _iy;
};
```

1.1、友元函数之全局函数

现在我们有一个全局函数distance，通过它计算两个点之间的距离，如果直接访问肯定是不行的，编译会报错。当我们在类Point中将其声明为友元之后，就可以了。

```
float distance(const Point &lhs, const Point &rhs)
{
     return hypot(lhs._ix - rhs._ix, lhs._iy - rhs._iy);
}
```

1.2、友元函数之成员函数

假设类A有一个成员函数，该成员函数想去访问另一个类B类中的私有成员变量。这时候则可以在第二个类B中，声明第一个类A的那个成员函数为类B的友元函数，这样第一个类A的某个成员函数就可以访问第二个类B的私有成员变量了。同样还是求取两个点之间的距离，现在我们再定义一个类Line，由Line中的成员函数distance完成：

```
class Line
{
public:
    float distance(const Point &lhs, const Point &rhs)
    {
          return hypot(lhs._ix - rhs._ix, lhs._iy - rhs._iy);
    }
};
```

1.3、友元之友元类

如上的例子，假设类Line中不止有一个distance成员函数，还有其他成员函数，它们都需要访问Point的私有成员，如果还像上面的方式一个一个设置友元，就比较繁琐了，可以直接将Line类设置为Point的友元。

```
class Point
{
    //...
    friend class Line;
    //...
};
```

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。**

注意：友元的声明是不受public/protected/private关键字限制的。

2、运算符重载

2.1、为什么需要对运算符进行重载?

C++预定义中的运算符的操作对象只局限于基本的内置数据类型，但是对于我们自定义的类型是没有办法操作的。但是大多时候我们需要对我们定义的类型进行类似的运算，这个时候就需要我们对这么运算符进行重新定义，赋予其新的功能，以满足自身的需求。比如:

```
class Complex
{
public:
    Complex(double real = 0, double image = 0)
    : _real(real)
    , _image(image)
    {

    }
private:
    double _real;
    double _image;
};

void test()
{
    Complex c1(1, 2), c2(3, 4);
    Complex c3 = c1 + c2;//编译出错
}
```

为了使对用户自定义数据类型的数据的操作与内置数据类型的数据的操作形式一致，C++提供了运算符的重载,通过把C++中预定义的运算符重载为类的成员函数或者友元函数，使得对用户的自定义数据类型的数据(对象)的操作形式与C++内部定义的类型的数据一致。

运算符重载的实质就是**函数重载**或**函数多态**。运算符重载是一种形式的`C++`多态。目的在于让人能够用同名的函数来完成不同的基本操作。要重载运算符，需要使用被称为运算符函数的特殊函数形式，运算符函数形式:

```
返回类型 operator 运算符(参数表)
{
    //...
}
```

2.2、运算符重载的规则

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);
```

但并不是所有的运算符都可以重载。可以重载的运算符有:

+	-	*	/	%	^
&	|	~	!	=	<
>	+=	-=	*=	/=	%=
^=	&=	|=	>>	<<	>>=
<<=	==	!=	>=	<=	&&
||	++	--	->*	->	,
[]	()	new	delete	new[]	delete[]

不可以重载的运算符: . .* ?: :: sizeof

运算符重载还具有以下规则：

- 为了防止用户对标准类型进行运算符重载，C++规定重载的运算符的操作对象必须至少有一个是自定义类型或枚举类型
- 重载运算符之后，其优先级和结合性还是固定不变的。
- 重载不会改变运算符的用法，原来有几个操作数、操作数在左边还是在右边，这些都不会改变。
- 重载运算符函数不能有默认参数，否则就改变了运算符操作数的个数。
- 重载逻辑运算符（&&,||）后，不再具备短路求值特性。
- 不能臆造一个并不存在的运算符，如@、$等

2.4、运算符重载的形式

运算符重载的形式有三种：

- 采用普通函数的重载形式
- 采用成员函数的重载形式
- 采用友元函数的重载形式

2.4.1、以普通函数形式重载

在上面的例子中，Complex对象无法执行加法操作，接下来我们重载+运算符。由于之前的定义中Complex的成员都设置成了private成员，所以不能访问，我们需要在类中添加2个get函数，获取其值。

```
class Complex
{
public:
    //...
    double getReal() const
    {
        return _real;
    }

    double getImage() const
    {
        return _image;
    }
    //...
};

Complex operator+(const Complex &lhs, const Complex &rhs)
{
```

```
    return Complex(lhs.getReal() + rhs.getReal(), lhs.getImage() +
rhs.getImage());
}

void test()
{
    Complex c1(1, 2), c2(3, 4);
    Complex c3 = c1 + c2;//编译通过
}
```

2.4.2、以成员函数形式重载

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）
{
    //...
}
```

注意：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。

回到Complex的例子，如果以成员函数形式进行重载，则不需要定义get函数：

```
class Complex
{
public:
    //...
    Complex operator+(const Complex & rhs)
    {
        return Complex(_real + rhs._real, _image + rhs._image);
    }
};
```

2.4.3、以友元函数形式重载

如果以友元函数形式进行重载，同样不需要定义get函数：

```
class Complex
{
    //...
    friend Complex operator+(const Complex &lhs, const Complex &rhs);
};

Complex operator+(const Complex &lhs, const Complex &rhs)
{
    return Complex(lhs._real + rhs._real, lhs._image + rhs._image);
}
```

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 &lhs,const Complex &rhs)
{
    return complex(lhs._real - rhs._real, lhs._image - rhs._image);
}
```

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。**

3、特殊运算符的重载

3.1、复合赋值运算符

复合赋值运算符推荐以成员函数的形式进行重载，包括这些(+=,-=,*=, / =,%=,<<=,>>=,&=,^=,|=)，因为对象本身会发生变化。

```
class Complex
{
public:
    //对于复合赋值运算符，对象本身发生了改变，推荐使用成员函数形式
    Complex &operator+=(const Complex &rhs)
    {
        cout << "Complex &operator+=(const Complex &)" << endl;
        _dreal += rhs._dreal;
        _dimag += rhs._dimag;

        return *this;
    }

    Complex &operator-=(const Complex &rhs)
    {
        cout << "Complex &operator-=(const Complex &)" << endl;
        _dreal -= rhs._dreal;
        _dimag -= rhs._dimag;

        return *this;
    }
};
```

3.2、自增自减运算符

自增运算符++和自减运算符--推荐以成员函数形式重载，分别包含两个版本，即运算符前置形式(如 ++x)和运算符后置形式(如 x++)，这两者进行的操作是不一样的。因此，当我们在对这两个运算符进行重载时，就必须区分前置和后置形式。

C++根据参数的个数来区分前置和后置形式。如果按照通常的方法（成员函数不带参数）来重载++/--运算符，那么重载的就是前置版本。要对后置形式进行重载，就必须为重载函数再增加一个int类型的参数，该参数仅仅用来告诉编译器这是一个运算符后置形式，在实际调用时不需要传递实参。

```
class Complex
{
public:
    //...
    //前置形式
    Complex & operator++()
```

```
    {
        ++_real;
        ++_image;
        return *this;
    }

    //后置形式
    Complex operator++(int) //int作为标记，并不传递参数
    {
        Complex tmp(*this);
        ++_real;
        ++_image;
        return tmp;
    }
};

void test()
{
    int a = 3;
    int b = 4;
    (++a);//表达式的值与a的值，需要进行区分，对于重载前置++与后置++是有一定参考价值的
    (a++)
}
```

3.3、赋值运算符

对于赋值运算符=，只能以成员函数形式进行重载，我们已经在类和对象中讲过了，就不再赘述，大家可以翻看前面的内容。

3.4、函数调用运算符

我们知道，普通函数执行时，有一个特点就是无记忆性，一个普通函数执行完毕，它所在的函数栈空间就会被销毁，所以普通函数执行时的状态信息，是无法保存下来的，这就让它无法应用在那些需要对每次的执行状态信息进行维护的场景。大家知道，我们学习了成员函数以后，有了对象的存在，对象执行某些操作之后，只要对象没有销毁，其状态就是可以保留下来的，但在函数作为参数传递时，会有障碍。为了解决这个问题，C++引入了函数调用运算符。函数调用运算符的重载形式只能是**成员函数**形式，其形式为:

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

在定义()运算符的语句中，第一对小括号总是空的，因为它代表着我们定义的运算符名，第二对小括号就是函数参数列表了，它与普通函数的参数列表完全相同。对于其他能够重载的运算符而言，操作数个数都是固定的，但函数调用运算符不同，它的参数是根据需要来确定的， 并不固定。

接下来，我们来看一个例子:

```
class FunctionObject
{
public:
    FunctionObject(int count = 0): _count(count)
    {

    }
```

```
8
9       void operator()(int x)
10      {
11          ++_count;
12          cout << " x = " << x << endl;
13      }
14
15      int operator()(int x, int y)
16      {
17          ++_count;
18         return x + y;
19      }
20
21      int _count;//函数对象的状态
22  };
23
24  void test()
25  {
26      FunctionObject fo;
27      int a = 3, b = 4;
28      fo(a);
29      cout << fo(a, b) << endl;
30  }
```

从例子可以看出，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。对于这种重载了函数调用运算符的类创建的对象，我们称为函数对象（Function Object）。函数也是一种对象，这是泛型思考问题的方式。

3.5、下标访问运算符

下标访问运算符[]通常用于访问数组元素，它是一个二元运算符，如arr[idx]可以理解成arr是左操作数，idx是右操作数。对下标访问运算符进行重载时，只能以成员函数形式进行，如果从函数的观点来看，语句arr[idx];可以解释为arr.operator[](idx);，因此下标访问运算符的重载形式如下：

```
1  返回类型 &类名::operator[](参数类型);
2  返回类型 &类名::operator[](参数类型) const;
```

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访问运算符，编译器将会给出其缺省定义，在其表达对象数组时使用。

```
1   class CharArray
2   {
3   public:
4       CharArray(size_t size = 10)
5       : _size(size)
6       , _array(new char[_size]())
7       {
8
9       }
10
11      char &operator[](size_t idx)
12      {
13         if(idx < _size)
14         {
15              return _array[idx];
16          }
```

```
        else
        {
            static char nullchar = '\0';
            return nullchar;
        }
    }

    const char &operator[](int idx) const//针对的是const对象
    {
        if(idx < _size)
        {
            return _array[idx];
        }
        else
        {
            static char nullchar = '\0';
            return nullchar;
        }
    }

    ~CharArray()
    {
        delete [] _array;
    }
private:
    size_t _size;
    char *_array;
};
```

我们之前使用过的std::string同样也重载了下标访问运算符，这也是为什么它能像数组一样去访问元素的原因。

3.6、成员访问运算符

成员访问运算符包括箭头访问运算符->和解引用运算符*，我们先来看箭头运算符->.

箭头运算符只能以成员函数的形式重载，其返回值必须是一个指针或者重载了箭头运算符的对象。来看下例子：

```
class Data
{
public:
    int getData() const
    {
        return _data;
    }
private:
    int _data;
};

class MiddleLayer
{
public:
    MiddleLayer(Data *pdata)
    : _pdata(pdata)
    {

```

```
    }

    //返回值是一个指针
    Data *operator->()
    {
        return _pdata;
    }

    Data &operator*()
    {
        return *_pdata;
    }

    ~MiddleLayer()
    {
        delete _data;
    }
private:
    Data * _pdata;
};

class ThirdLayer
{
public:
    ThirdLayer(MiddleLayer * ml)
    : _ml(ml)
    {

    }

    //返回一个重载了箭头运算符的对象
    MiddleLayer & operator->()
    {
        return *_ml;
    }

    ~ThirdLayer()
    {
        delete _ml;
    }

private:
    MiddleLayer * _ml;
};

void test()
{
    MiddleLayer ml(new Data());
    cout << ml->getData() << endl;
    cout << (ml.operator->())->getData() << endl;

    cout << (*ml).getData() << endl;

    ThirdLayer tl(new MiddleLayer(new Data()));
    cout << tl->getData() << endl;
    cout << ((tl.operator->()).operator->())->getData() << endl;
}
```

3.7、输入输出流运算符

在之前的例子中，我们如果想打印一个对象时，常用的方法是通过定义一个`print`成员函数来完成，但使用起来不太方便。我们希望打印一个对象，与打印一个整型数据在形式上没有差别(如下例子)，那就必须要重载<<运算符。

```
void test()
{
    int a = 1, b = 2;
    cout << a << b << endl;
    Point pt1(1, 2), pt2(3, 4);
    cout << pt1 << pt2 << endl;
}
```

从上面的形式能看出，cout是左操作数，a或者pt1是右操作数，那输入输出流能重载为成员函数形式吗？我们假设是可以的，由于非静态成员函数的第一个参数是隐含的this指针，代表当前对象本身，这与其要求是冲突的，因此>>和<<不能重载为成员函数，只能是非成员函数，如果涉及到要对类中私有成员进行访问，还得将非成员函数设置为类的友元函数。

```
class Point
{
public:
    //...
    friend ostream &operator<<(ostream &os, const Point &rhs);
    friend istream &operator>>(istream &is, Point &rhs);
private:
    int _ix;
    int _iy;
};

ostream & operator<<(ostream &os, const Point &rhs)
{
    os << "(" << rhs._ix
       << "," << rhs._iy
       << ")";
    return os;
}

istream &operator>>(istream &is, Point &rhs)
{
    is >> rhs._ix;
    is >> rhs._iy;
    return is;
}
```

通常来说，重载输出流运算符用得更多一些。同样的，输入流运算符也可以进行重载，如上。

总结：对于运算符重载时采用的形式的建议：

- 所有的一元运算符，建议以成员函数重载
- 运算符 = () [] -> ->* ，必须以成员函数重载
- 运算符 += -= /= *= %= ^= &= != >>= <<= 建议以成员函数形式重载
- 其它二元运算符，建议以非成员函数重载

4、类型转换

前面介绍过对普通变量的类型转换，比如说`int`型转换为long型，double型转换为int型，接下来我们要讨论下类对象与其他类型的转换。转换的方向有:

- 由其他类型向自定义类型转换
- 由自定义类型向其他类型转换

4.1、由其他类型向自定义类型转换

由其他类型向定义类型转换是由构造函数来实现的，只有当类中定义了合适的构造函数时，转换才能通过。这种转换，一般称为**隐式转换**。下面，我们通过一个例子进行说明:

```
class Point
{
public:
    Point(int ix = 0, int iy = 0)
    : _ix(ix)
    , _iy(iy)
    {}

    //...
    friend ostream &operator<<(ostream &os, const Point &rhs);
private:
    int _ix;
    int _iy;
};

ostream &operator<<(ostream &os, const Point &rhs)
{
    os << "(" << rhs._ix
       << "," << rhs._iy
       << ")";
    return os;
}

void test()
{
    Point pt = 1;//隐式转换
    cout << "pt = " << pt << endl;
}
```

这种隐式转换有时候用起来是挺好的，比如，我们以前学过的std::string，当执行

```
std::string s1 = "hello,world";
```

该语句时，这里其实是有隐式转换的，但该隐式转换的执行很自然，很和谐。而上面把一个int型数据直接赋值给一个Point对象，看起来就是比较诡异的，难以接受，所以这里我们是不希望发生这样的隐式转换的。那怎么禁止隐式转换呢，比较简单，只需要在相应构造函数前面加上explicit关键字就能解决。

4.2、由自定义类型向其他类型转换

由自定义类型向其他类型的转换是由**类型转换函数**完成的，这是一个特殊的成员函数。它的形式如下：

```
operator 目标类型()
{
    //...
}
```

类型转换函数具有以下的特征:

- 必须是成员函数;
- 参数列表中没有参数;
- 没有返回值，但在函数体内必须以return语句返回一个目标类型的变量。

我们来看一个例子:

```
class Fraction
{
public:
    Fraction(double numerator, double denominator)
    : _numerator(numerator)
    , _denominator(denominator)
    {

    }

    operator double()
    {
        return _numerator/_denominator;
    }

    operator Point()
    {
        return Point(_numerator, _denominator);
    }
private:
    double _numerator;
    double _denominator;
};

void test()
{
    Fraction f(2, 4);
    cout << "f = " << f << endl;
    double x = f + 1.11;
    cout << "x = " << x << endl;

    double y = f;
}
```