第五讲 操作符重载（一）

一、操作符重载定义

二、算术运算符重载

三、输出运算符重载

四、下标运算符重载

一、操作符重载定义

操作符通常用来操作操作数，但有时我们所操作的操作数的类型并不是默认类型。那么，这时我们需要对操作符进行重载（重新定义），以便它可以操作我们想操作的操作数类型，这种重载就称为操作符重载。

注意:

⑴ 只能重载已经存在的操作符;

⑵ 操作数的数目应该和原始定义一致;

⑶ 至少有一个操作数是用户自定义类型;

例如: Person + Person 正确

Person + int 正确

int + int 错误）;

⑷ 重载后优先级不变;

二、算术运算符重载

【示例2-1】算术运算符重载:

```
/** Point.h **/
class Point {
    friend const Point operator+(const Point &p1,const Point &p2);
    friend const Point operator-(const Point &p1,const Point &p2);

public:
    Point(float _x = 0,float _y = 0);
    void print() const;

private:
    float x;
    float y;
};
```

```
/** Point.cpp **/
Point::Point(float _x,float _y):x(_x),y(_y){}

void Point::print() const {
    cout<<"x = "<<x<<"\ty = "<<y<<endl;
}

const Point operator+(const Point &p1,const Point &p2) {
    return Point(p1.x+p2.x,p1.y+p2.y);
}

const Point operator-(const Point &p1,const Point &p2) {
    return Point(p1.x-p2.x,p1.y-p2.y);
}
```

```
/** main.c **/
int main(int argc, const char * argv[]) {

    Point p1(4,7);
    Point p2(2,3);

    (p1+p2).print();
    (p1-p2).print();

    return 0;
}
```

运行结果如下：

```
x = 6  y = 10
x = 2  y = 4
```

示例分析：

(1) 函数重载运算符+、-、*定义为内联函数减少函数调用的开销，而运算符/没有定义为内联函数，原因是一些编译器不允许内联函数中有if语句；

(2) 重载运算符函数的参数为常引用，避免拷贝；

(3) 算术运算符重载既可以是友元函数，也可以是成员函数。

【示例2-2】运算符重载函数为成员函数的代码示例

```
const Point Point::operator+(const Point &p) {
    return Point(x+p.x,y+p.y);
}
```

在这个定义中,只有一个明确的操作数,即第二个操作数。第一个操作数是通过隐式指针将自身传给重载函数。我们宁愿使用友元函数,因为这个成员函数定义的操作数看起来并不对称。

三、输出操作符重载

在上面的例子中我们会发现，打印函数看起来有写复杂，这是我们也可以对输出操作符进行重载。

输出操作符重载函数不能被定义成一个成员函数，因为输出操作符第一个操作数是ostream对象。

因为输出操作符通常是需要访问类的私有成员,它经常被声明为类的友元函数。

注意：输出操作符<<的重载，返回的ostream对象一定不能是const类型。而且，输出操作符<<函数里的ostream参数，也不能是const类型。因为，如果将ostream对象声明成const类型的对象，将不能再向ostream对象中写入任何数据，就不能输出数据了。

【示例3-1】输出运算符重载示例：

```
friend ostream &operator<<(ostream &out, const Point &p);//声明
stream &operator<<(ostream &out, const Point &p) {//实现
    out<<"x = "<<p.x<<"\ty = "<<p.y<<endl;
    return out;
}
cout<<p1<<endl;//调用
```

四、下标运算符重载

如果一个类的成员是一个数组,我们可以使用下标操作符来访问数组的成员。例如,一个类定义如下:

【示例4-1】下标运算符重载

```
#define size 100
class A {
    //private members
public:
    int a_member[size];
    //other public members
};

int main(int argc, const char * argv[]) {
    A an_obj;
    //...
    an_obj.a_member[3] = 5;
    //an_obj[3] = 5;   这是我们期望的
    return 0;
}
```

这时会存在两个问题：第一,这只适用于这个数组成员公开。第二,这使得代码更复杂。我们可能重载这个数组的下标运算符来直接访问组件成员,这样数组成员不必公开。通过这种方式,该对象看起来很像一个“数组”。

下面例子来说明下标运算符重载：

【示例4-2】下标运算符重载

```
/** Vector.h **/
#define SIZE 20
class Vector {
public:
    const int &operator[](int index) const;//const版本
    int &operator[](int index);//非const版本

private:
    int rep[SIZE];
};
```

```
/** Vector.cpp **/
const int &Vector::operator[](int index) const {
    return rep[index];
}
int &Vector::operator[](int index) {
    return rep[index];
}
ostream &operator<<(ostream &out, const Vector &v) {
    for (int i = 0; i < SIZE; i++) {
        out<<v[i]<<"\t";
    }
    return out;
}
/** main.c **/
int main(int argc, const char * argv[]) {
    Vector v;
    cout<<"Enter "<<SIZE<<" integers"<<endl;
    for (int i = 0; i < SIZE; i++) {
        cin>>v[i];
    }
    cout<<v<<endl;
    return 0;
}
```

示例分析

(1) 在主函数中，v用起来就像一个数组一样，直接通过下标修改和访问rep中的元素；

(2) 非const版本下标操作符返回引用，使他可以当左值使用；而const版本的下标操作符返回值不可以当左值；

(3) 通常情况下，下标符[]重载函数有两个版本:const版本和非const版本。const版本用于访问数组的成员，非const版本用于修改数组的成员。如果只定义const版本，那么数组成员不允许被改变；如果只定义非const版本，那么const对象无法调用；

(4) 下标操作符重载函数必须作为成员函数实现，而不能用友元。因为操作的是一个对象；

The End

渥瑞达·教学部