第7章-重载操作符

author：岳石磊 copyright：科林明伦 内部资料禁止外泄

1. 概念

C++ 提供的运算符，通常只支持对于基本数据类型和标准库中提供的类进行操作，对于自定义类型如果想通过操作符实现对应的操作，需要自定义重载的操作符并实现具体的功能。
重载操作符：也称为重载运算符，本质上是一个函数，而且是一种特殊的重载操作符函数，并告诉C++编译器，当遇到该运算符时就调用此函数来行使运算符功能，本质是通过函数扩展了操作符的功能。重载操作符函数需要使用关键字operator作为函数名的一部分，重载操作符一般要有返回值，方便继续和其他操作符去操作。
可分两类：类内重载、类外重载。

2. 类内重载操作符

在类内重载，作为类成员函数，需要用对象调用，使用场景需要根据函数的参数一致（包括类型和顺序），注意在类内重载的操作符函数有隐藏的this指针作为第一个参数。
在使用是要注意重载操作符的参数类型和顺序，可以直接使用操作符，也可以显示通过对象调用重载的操作符。

```
class CTest{
private:
    int m_a;
public:
    CTest(){
        m_a  = 10;
    }
    int operator+(int a){
        return m_a+a;
    }
};

int main(){
    CTest tst;
    int a = tst+20;
    int b = tst.operator+(20);
    //int c = 20+tst;   //非法
    return 0;
}
```

对于单目运算符++，有左++ 和 右++ 两种，为了区分右++，我们需要额外指定一个int类型的参数，这个参数只是用来区分，并无实际意义。

```
class CTest{
    //类内
    int operator++();   //左++
    int operator++(int a); //右++
};
```

3. 类外重载操作符

在类外重载操作符时，至少需要包含一个**自定义类型**，在类内的重载操作符函数有默认的this且为自定义类型，所以在定义函数时忽略了第一个参数，但类外重载没有隐藏的参数了，一般比类内要多一个参数。但要注意是否与类内重载的函数有冲突。

```
class CTest{
public:
    int m_a;
    CTest(){
        m_a = 10;
    }
    int operator+(int a){
        return this->m_a+a;
    }
};

int operator+(int a,CTest &tst){
    return tst.m_a+a;
}

int main(){
    CTest tst;
    tst+10;    //匹配类内重载函数
    10+tst;    //匹配类外重载函数
    return 0;
}
```

类内、类外产生歧义，调用不明确。

```
class CTest{
public:
    int m_a;
```

```
    CTest(){
        m_a = 10;
    }
    int operator+(int a){
        return this->m_a+a;
    }
};
int operator+(CTest &tst,int a){
    return tst.m_a+a;
}

int main(){
    CTest tst;
    tst+10;    //error C2593: “operator +”不明确

    //消除歧义:
    tst.operator+(10);   //类内
    ::operator+(tst,10); //类外

    return 0;
}
```

自定义重载输入、输出操作符，一般在类外重载。

```
class CTest{
public:
    int m_a;
};

//输入，注意参数为类、结构体类型，最好用引用而非值传递
istream& operator>>(istream& is, CTest& tst) {
    is >> tst.m_a;
    return is;
}
//输出,
ostream& operator<<(ostream& os, CTest& tst) {
    os << tst.m_a;
    return os;
}
```

4. 使用注意

对于同一个操作符来说，写在不同的位置代表不同的含义，*p 和 a*b，那么重载这个操作符需要注意参数的数量、顺序不同代表不同的含义。

```
//类内重载
int opertor*();     //间接引用
int opertor*(int); //乘法
```

注意:

1. 不能重载的运算符：长度运算符sizeof、条件运算符: ?、成员选择符.、作用域运算符::等
2. 还有一些操作符只能在类内重载，**赋值=** ，**下标[]**，**调用()** ，**和成员指向->** 操作符必须被定义为类成员操作符。
3. 重载操作符不能改变操作符的用法，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。
4. 运算符重载函数不能有默认的参数，否则就改变了运算符操作数的个数，这显然是错误的。
5. 重载操作符不能改变运算符的优先级和结合性。
6. 不能创建新的运算符。

5. 对象类型转换

上面重载等号操作符 operator=，能让其他的类型赋值到当前类对象中，但是如果反过来写则会报错，类型不匹配，因为operator=只能在类内重载。
此时可以重载某个类型，这样定义该类对象就可以像这个类型一样去使用。
函数格式为：

```
operator type(){
    return type_value;  // 类型要和type 一致。
}
```

函数在写法上无参数，无返回值，但函数体中应该有return，且return 的变量类型要和重载的类型一致。

```
operator int(){
    int a=10;
    return a;
}
```

下面两种都是错误的写法：

```
int operator int()      //error C2549: 用户定义的转换不能指定返回类型
operator int(int a)    //error C2835: 用户定义的转换“CTest::operator int”不接受形参
```

如果同时存在重载操作符 和重载类型，那么优先匹配重载的操作符

```
int a = tst+10;                 //operator+
a = 10+tst;                     //operator int
a = tst;                        //operator int
```

当然也可以显示的调用 类型转换函数

```
int a = tst.operator int()+10;  //operator int
```

6. 封装Iterator

```
class CMyList {

    ...

    void ShowList() {
        //Node* pTemp = m_pHead;  //定义指针并初始化 ，对应构造
        Node* pTemp = nullptr;
        pTemp = m_pHead;          //赋值    ，operator=

        while (pTemp!=nullptr) {   //判断是否等于、不等于某个节点  operator== operator!=
            cout << pTemp->val << "    ";   // 间接引用取值     operator*
            pTemp = pTemp->pNext;           //向后移动     operator++
        }
    }

    ...

};
```

封装迭代器，包含临时指针 和 对应操作的 重载操作符。

```
class CMyIterator {
public:
    Node* m_pTemp;
    CMyIterator() {
        m_pTemp = nullptr;
```

```
    }
    CMyIterator(Node* pNode) {
        m_pTemp = pNode;
    }
    ~CMyIterator(){}
public:
    Node* operator=(Node* pNode) {
        m_pTemp = pNode;
        return pNode;
    }
    bool operator==(Node* pNode) {
        return m_pTemp == pNode;
    }
    bool operator!=(Node* pNode) {
        return m_pTemp != pNode;
    }

    int operator*() {
        if (m_pTemp) {
            return m_pTemp->val;
        }
        return -1;
    }

    Node*  operator++() { //左++
        if (m_pTemp) {
            m_pTemp = m_pTemp->pNext;
        }
        return m_pTemp;
    }

    Node* operator++(int) { //右++
        Node* t = m_pTemp;  //标记加之前
        if (m_pTemp) {
            m_pTemp = m_pTemp->pNext;
        }
        return t;
    }
};
```

使用迭代器遍历链表

```
class CMyList {

    ...

    void ShowList() {
        CMyIterator ite = m_pHead;
        while (ite != nullptr) {
            cout << *ite << "    ";
            ite++;
        }
    }

    ...

};
```