**[C++运算符重载三种形式（成员函数，友元函数，普通函数）详解](https://www.cnblogs.com/Mayfly-nymph/p/9034936.html)**

**三种重载方式**

首先，介绍三种重载方式：

//作为成员函数重载（常见）

class Person{

private:

string name;

int age;

public:

Person(const char\* name, int age):name(name),age(age){}

bool operator<(const Person& b);

};

bool Person::operator<(const Person& b)

{

//作为成员函数时，\*this即为左操作数a

...

}

//作为友元函数重载

class Person{

private:

string name;

int age;

public:

Person(const char\* name, int age):name(name),age(age){}

friend bool operator<(const Person& a,const Person& b);

};

bool operator<(const Person& a,const Person& b)

{

...

}

//作为普通函数重载（不推荐）

class Person{

public://注意，重载运算符为普通函数时，使用到的类成员必须为public

string name;

int age;

public:

Person(const char\* name, int age):name(name),age(age){}

};

bool operator<(const Person& a,const Person& b)

{

...

}

**作为成员函数重载**

先介绍第一种：

bool Person::operator<(const Person& b)，bool是函数返回类型，Person::只是指定了成员函数所属类名。

 在作为函数成员重载中，先看下这句话：单目运算符作为类成员函数重载时没有型参（除了后置自增（自减）有一个整型参数：[详细点击](http://www.jb51.net/article/72424.htm" \t "_blank)），双目运算符作为类成员函数重载时只有一个型参，作为运算符的右操作数，其左操作数就是本对象自己，也就是this。

 单目运算符一般重载为成员函数。

 因此在作为成员函数进行重载时，是以

#include <iostream>

#include <cstdlib>

using namespace std;

class A {

private:

int a;

int b;

public:

A(int x = 0, int y = 0):a(x), b(y){}

A operator+ (A &C);

};

A A::operator+ (A &C) {

A G;

G.a = this->a + C.b;

return G;

}

int main()

{

A G(5, 6);

A J(7, 8);

A K;

K = G + J;

return 0;

}

而计算机对于K = G + j;进行重载后形式是

K = G.operator+(J);

G为对象，J为参数。

**作为友元函数重载**

而对于第二种形式的重载： 友元函数（友元函数则是指某些虽然不是类成员却能够访问类的所有成员的函数）进行重载，那么它就不存在this指针了，所以需要定义两个参数来运算（对于双目运算符），而友元函数的实现可以在外面定义，但必须在类内部声明。

#include <iostream>

#include <cstdlib>

using namespace std;

class A {

private:

int a;

int b;

public:

A(int x = 0, int y = 0):a(x), b(y){}

friend A operator+ (A &C, A &D);

};

A operator+ (A &C, A &D) {

A G;

G.a = D.a + C.b;

return G;

}

int main()

{

A G(5, 6);

A J(7, 8);

A K;

K = G + J;

return 0;

}

推荐类内声明，外部定义，这样不会显得类臃肿。

对于K = G + J;计算机将重载为：

K = operator+(G, J);

 声明为友元函数的好处：

1.和普通函数重载相比，它能够访问非公有成员。

2.将双目运算符重载为友元函数，这样就可以使用交换律。

弊端：

 友元可以像类成员一样访问类的成员和函数，但是使用不慎会造成破坏类的封装性。

 第一条：没解释的必要，跳过，哈哈哈哈嗝。        =。=

第二条：交换律也可以理解成对操作数对称处理。

#include <iostream>

using namespace std;

//复数类

class Complex{

public:

Complex(): m\_real(0.0), m\_imag(0.0){ }

Complex(double real, double imag): m\_real(real), m\_imag(imag){ }

Complex(double real): m\_real(real), m\_imag(0.0){ } //转换构造函数

public:

friend Complex operator+(const Complex &c1, const Complex &c2);

public:

double real() const{ return m\_real; }

double imag() const{ return m\_imag; }

private:

double m\_real; //实部

double m\_imag; //虚部

};

//重载+运算符

Complex operator+(const Complex &c1, const Complex &c2){

Complex c;

c.m\_real = c1.m\_real + c2.m\_real;

c.m\_imag = c1.m\_imag + c2.m\_imag;

return c;

}

int main(){

Complex c1(25, 35);

Complex c2 = c1 + 15.6;

Complex c3 = 28.23 + c1;

cout<<c2.real()<<" + "<<c2.imag()<<"i"<<endl;

cout<<c3.real()<<" + "<<c3.imag()<<"i"<<endl;

return 0;

}

如果将 operator+ 定义为成员函数，根据“+” 运算符具有左结合性”这条原则，Complex c2 = c1 + 15.6;会被转换为下面的形式：

Complex c2 = c1.operator+(Complex(15.6));

这就是通过对象调用成员函数，是正确的。而对于Complex c3 = 28.23 + c1;，编译器会尝试转换为不同的形式：

Complex c3 = (28.23).operator+(c1);

很显然这是错误的，因为 double 类型并没有以成员函数的形式重载 +。  
也就是说，以成员函数的形式重载 +，只能计算 c1 + 15.6，不能计算 28.23 + c1，这是不对称的

将22-25行代码可替换为

return (c1.real + c2.real, c1.image + c2.image);//调用构造函数

将会创建一个匿名对象返回。

使用&的好处：

将重载的返回类型定义为引用类型，能够实现连续输入（输出）。

#include <iostream>

#include <cstdlib>

using namespace std;

class A {

private:

int a;

int b;

public:

A(int x = 0, int y = 0):a(x), b(y){}

friend A operator+ (A &C, A &D);

friend ostream& operator<<(ostream & out, A &W);

};

A operator+ (A &C, A &D) {

A G;

G.a = D.a + C.b;

return G;

}

ostream& operator<<(ostream & out, A &W) {

out << W.a << " " << W.b;

return out;

}

int main()

{

A G(5, 6);

A J(7, 8);

A K;

K = G + J;

cout << K << " " << J;

return 0;

}

将流提取运算符 >> 或流插入运算符 << 声明为友元函数，能够访问非公有成员。

**作为普通函数重载**

对于第三种普通函数重载：

因为不属于类了，自然也就没比较加Person::

 要注意的是这种形式无法访问非公有成员。

 其他知识点：

为什么我们要使用两个参数（非成员函数形式）来重载流提取运算符 >> 和流插入运算符 << 呢？

 如果我们要用成员函数,则会有cout.operator<<(const A& W),但重载双目操作符（即为类的成员函数），就只要设置一个参数作为右侧运算量，而左侧运算量就是对象本身，而cin和cout并不是对象本身（你如果声明为对象，下面岂不是this->a + W.b,注意this是类对象的）

 如果一定要声明为成员函数，只能成为如下的形式：

#include <iostream>

#include <string>

using namespace std;

class A {

public:

A(const string &s = "hello"):str(s){}

ostream& operator<<(ostream &ou) {

ou << this->str;

return ou;

}

private:

string str;

};

int main()

{

A a;

a << cout;//输出"hello"

system("PAUSE");

return 0;

}

所以在运用这个<<运算符时就变为这种形式了：

t<<cout;

而不是

cout<<t

而且也无法链式使用了

cout<<t<<t<<t<<endl;

对于t << cout 我们可以反向理解cout << t，cout >> t会被重载为cout.operator(t)，错误。

 这段借用网上的：

不能重载的根本原因在于， 大部份的标准库实现中，对ostream,istream类体系采用了构造函数保护继承的方式。。。致使即使以继承的方式来扩展流类，也会在对象实例化时遭遇阻碍。。。 另一方面，标准库中的流类，其插入符函数没有声明为虚函数，因此子类不能对其实现进行覆盖，所以也使成员函数重载遭遇到实质的困难。。。 总的来说，C++标准I/O库非常繁杂且难，其实现思想很多都与常规的OOP有所出入。。。在使用的时候要谨慎，并最好遵从惯例。。。

 至于单目运算符的声明定义就不多做介绍了，要看的[点击此处](https://blog.csdn.net/maoliran/article/details/51583165)，前面脚本之家的链接也有单目运算符重载的介绍。

 附：今天早上上机时发现VC++6.0不支持流提取运算符和流插入运算符的友元函数形式重载。