准确的来说就是，其实目前大多数运算符只支持那些最基本的数据类型操作，但是如果我们需要它们也来计算那些类类型的数据，我们就需要重新定义这些运算符操作方式

```
#include <iostream>
using namespace std;
//<return type> operator<the operator>(parameters)
class Vector2D {
public:
    float x, y;

    Vector2D(float x, float y) : x(x), y(y) {}

    // 重载 + 运算符
    Vector2D operator+(const Vector2D& other) const {
        return Vector2D(x + other.x, y + other.y);
    }
    Vector2D operator++()const {
        return Vector2D(x + 1, y + 1);
    }
    Vector2D operator++(int)const {
        return Vector2D(x + 1, y + 1);
    }
};

int main() {
    Vector2D v1(1, 2);
    Vector2D v2(3, 4);
    Vector2D v3(0, 0);
    v3 = v1 + v2;
    //v3 = v1.operator+(v2) 一样的效果
    cout << v3.x << " " << v3.y << endl;
    v1 = ++v1;
    cout << v1.x << " " << v1.y << endl;
    v1 = v1++;//注意如果是后置递增的话就要在重载定义中的参数传入int
    cout << v1.x << " " << v1.y << endl;
}
```

我们这里的例子是将函数重载函数定义在了类的内部，毕竟这样也方便访问到类的成员变量属性嘛。

那么简单说一下这个语法

```
Vector2D operator+(const Vector2D& other) const {
        return Vector2D(x + other.x, y + other.y);
    }
```

类名 operator运算符(const 类名& 对象形参)const{ return 类名(成员变量操作)}

唯一要讲的就是关于传入的参数了，首先如果涉及到其他对象的共同运算的话，那你肯定是要传入它们的，也是一样，用常const操作，带&引用，这些都是好习惯，后面的const修饰函数体的可有可无，自己看着加，只是起到保护数据的作用

你肯定会好奇那我本身类的参数传入要不要写呢，很明显不用，因为你这个重载相当于是类的成员函数了，有对象才能访问这个函数，自己就不需要传入了

当然后面像是<<和>>的重载你就要先引入这两个操作符，来自iostream，具体看后面io操作。

还有一个小点就是++，注意前置和后置递增的区别，后者要传一个int

function overloading and friend function

将函数重载和友元函数结合起来

通常在类的内部先定义一个友元函数，但是不用写这个函数的具体内容，只是把函数名和传入参数描述就行

```
class Vector2D {
public:
    double x, y;

    Vector2D(double x, double y) : x(x), y(y) {}

    // 作为成员函数的重载，正常重载函数
    Vector2D operator*(double a) const {
        return Vector2D(x * a, y * a);
    }

    // 声明一个全局函数为友元，用于实现交换律
    friend Vector2D operator*(double a, const Vector2D& v);//只定义不用写内容，方便传出值
};

// 友元函数的实现
Vector2D operator*(double a, const Vector2D& v) {//在类的外部写函数的具体内容
    return Vector2D(v.x * a, v.y * a);
}
int main(){
    Vector2D v(2.0, 3.0);
    Vector2D result1 = v * 2.0;//正常调用上面的非友元函数
    Vector2D result2 = 2.0 * v;//
    //Vector2D result3 = v.operator*(2.0); 报错，v对象不含有该函数
}
```

注意当这种情况下友元函数非成员函数，是没有this指针的，调用的时候，不能通过类

这段话是关于C++中运算符重载的一些指导原则和规则。让我们逐条解释一下：

1. **通常，一元运算符作为成员函数进行重载**：
   一元运算符（如 `++`, `--`, `!`）通常作为成员函数进行重载。这是因为它们通常只需要操作单个对象（即它们所作用的对象），所以将它们作为成员函数意味着它们可以直接访问对象的私有或受保护的成员。
2. **二元运算符取决于你的设计**：
   二元运算符（如 `+`, `-`, `*`, `/`）可以作为成员函数或非成员函数（通常是友元函数）进行重载。选择取决于是否需要访问类的私有成员，以及操作数的类型。如果运算符的两个操作数类型相同，或者如果对称性不重要，可以选择成员函数重载。否则，可能需要非成员重载。
3. **这四个运算符只能作为成员函数重载：** `=`, `()`, `[]`, `->`:
   - 赋值运算符 `=` 必须是成员函数，因为它涉及到改变对象的内部状态。

- 函数调用运算符 `()`、下标运算符 `[]` 和成员访问箭头 `->` 也必须作为成员函数重载，因为它们直接与对象的表示和状态操作相关联。

4. **如果运算符需要修改类的成员变量，最好作为成员函数进行重载**：
   如果运算符的操作需要修改对象的私有或受保护的成员，那么将其作为成员函数重载更为合适，因为成员函数可以直接访问这些成员。

5. **如果操作数（特别是第一个）希望有隐式类型转换，必须作为友元进行重载**：
   当你希望第一个操作数可以是其他类型，并且希望它能隐式转换为你的类类型时，需要将运算符作为友元函数进行重载。这是因为成员函数的第一个参数总是其所属类型的对象，而友元函数可以有更灵活的参数类型。

6. **交换律**：
   如果你希望你的二元运算符遵循数学上的交换律（例如，`a + b` 和 `b + a` 应该产生相同的结果），那么你可能需要将运算符作为非成员函数进行重载。这样可以更容易地处理不同类型的操作数，特别是当其中一个操作数不是你的类类型时。

总的来说，运算符重载的选择（成员函数还是非成员函数）取决于你的具体需求，包括操作数的类型、是否需要访问类的私有成员以及是否需要支持隐式类型转换等因素。在设计时，应当考虑到这些因素，以实现最直观和有效的行为。