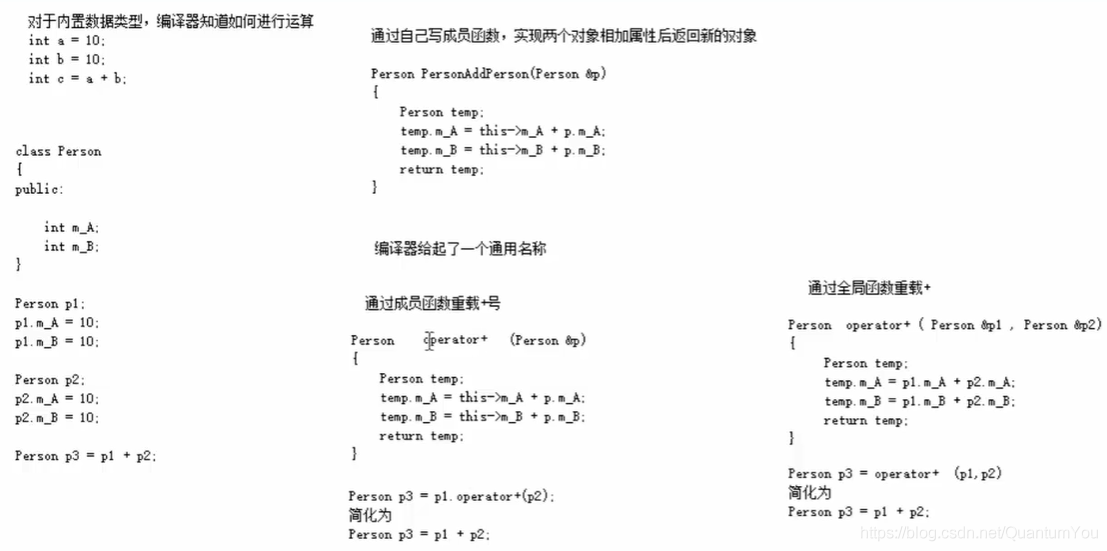
* 我希望你没有刻意为追求一个数字而生活，我希望你找到了真正的价值所在
* 你一定要**坚持**做自己，**静下心**来做自己喜欢的事，然后把自己交给**命运**

  
@[toc]

# 运算符重载

利用成员函数实现：  


* **运算符重载概念**：对已有的运算符重新进行定义，赋予其另一种功能，以适应不同的数据类型

## 加号运算符重载

作用：实现两个自定义数据类型相加的运算

class Person {  
public:  
 Person() {};  
 Person(int a, int b)  
 {  
 this->m\_A = a;  
 this->m\_B = b;  
 }  
 //成员函数实现 + 号运算符重载  
 Person operator+(const Person& p) {  
 Person temp;  
 temp.m\_A = this->m\_A + p.m\_A;  
 temp.m\_B = this->m\_B + p.m\_B;  
 return temp;  
 }  
  
  
public:  
 int m\_A;  
 int m\_B;  
};  
  
//全局函数实现 + 号运算符重载  
//Person operator+(const Person& p1, const Person& p2) {  
// Person temp(0, 0);  
// temp.m\_A = p1.m\_A + p2.m\_A;  
// temp.m\_B = p1.m\_B + p2.m\_B;  
// return temp;  
//}  
  
//运算符重载 可以发生函数重载   
Person operator+(const Person& p2, int val)   
{  
 Person temp;  
 temp.m\_A = p2.m\_A + val;  
 temp.m\_B = p2.m\_B + val;  
 return temp;  
}  
  
void test() {  
  
 Person p1(10, 10);  
 Person p2(20, 20);  
  
 //成员函数方式  
 Person p3 = p2 + p1; //相当于 p2.operaor+(p1)  
 cout << "mA:" << p3.m\_A << " mB:" << p3.m\_B << endl;  
  
  
 Person p4 = p3 + 10; //相当于 operator+(p3,10)  
 cout << "mA:" << p4.m\_A << " mB:" << p4.m\_B << endl;  
  
}  
  
int main() {  
  
 test();  
  
 system("pause");  
  
 return 0;  
}

总结1：对于内置的数据类型的表达式的的运算符是不可能改变的

总结2：不要滥用运算符重载

**注意**：  
成员函数重载本质调用 Person p3 = p1.operator+(p2) ;  
全局函数重载本质调用 Person p3 = operator+(p1,p2)

## 左移运算符重载

* **作用**：可以输出自定义数据类型

class Person {  
 friend ostream& operator<<(ostream& out, Person& p);  
  
public:  
  
 Person(int a, int b)  
 {  
 this->m\_A = a;  
 this->m\_B = b;  
 }  
  
 //成员函数 实现不了 p << cout 不是我们想要的效果  
 //void operator<<(Person& p){  
 //}  
  
private:  
 int m\_A;  
 int m\_B;  
};  
  
//全局函数实现左移重载  
//ostream对象只能有一个  
ostream& operator<<(ostream& out, Person& p) {  
 out << "a:" << p.m\_A << " b:" << p.m\_B;  
 return out;  
}  
  
void test() {  
  
 Person p1(10, 20);  
  
 cout << p1 << "hello world" << endl; //链式编程  
}  
  
int main() {  
  
 test();  
  
 system("pause");  
  
 return 0;  
}

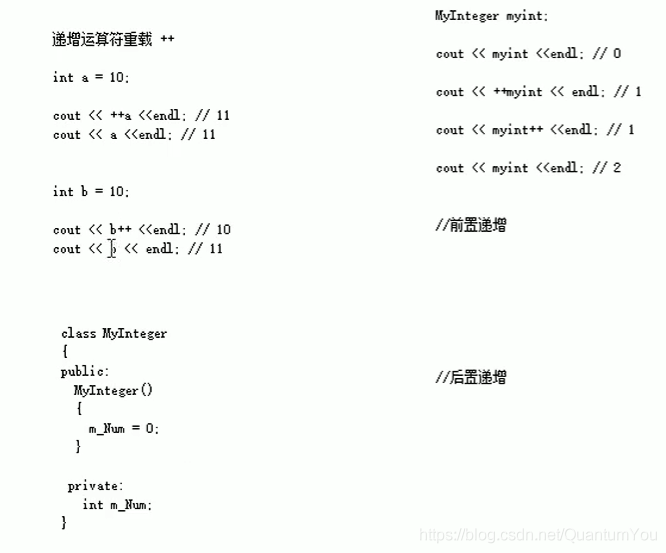
总结：重载左移运算符配合友元可以实现输出自定义数据类型

**核心**：

* 利用全局函数实现左移运算符的重载，返回引用的形式便于实现链式编程

ostream& operator<<(ostream& out, Person& p) {  
 out << "a:" << p.m\_A << " b:" << p.m\_B;  
 return out;  
}

## 递增运算符重载

主要分为前置递增和后置递增 运算符  


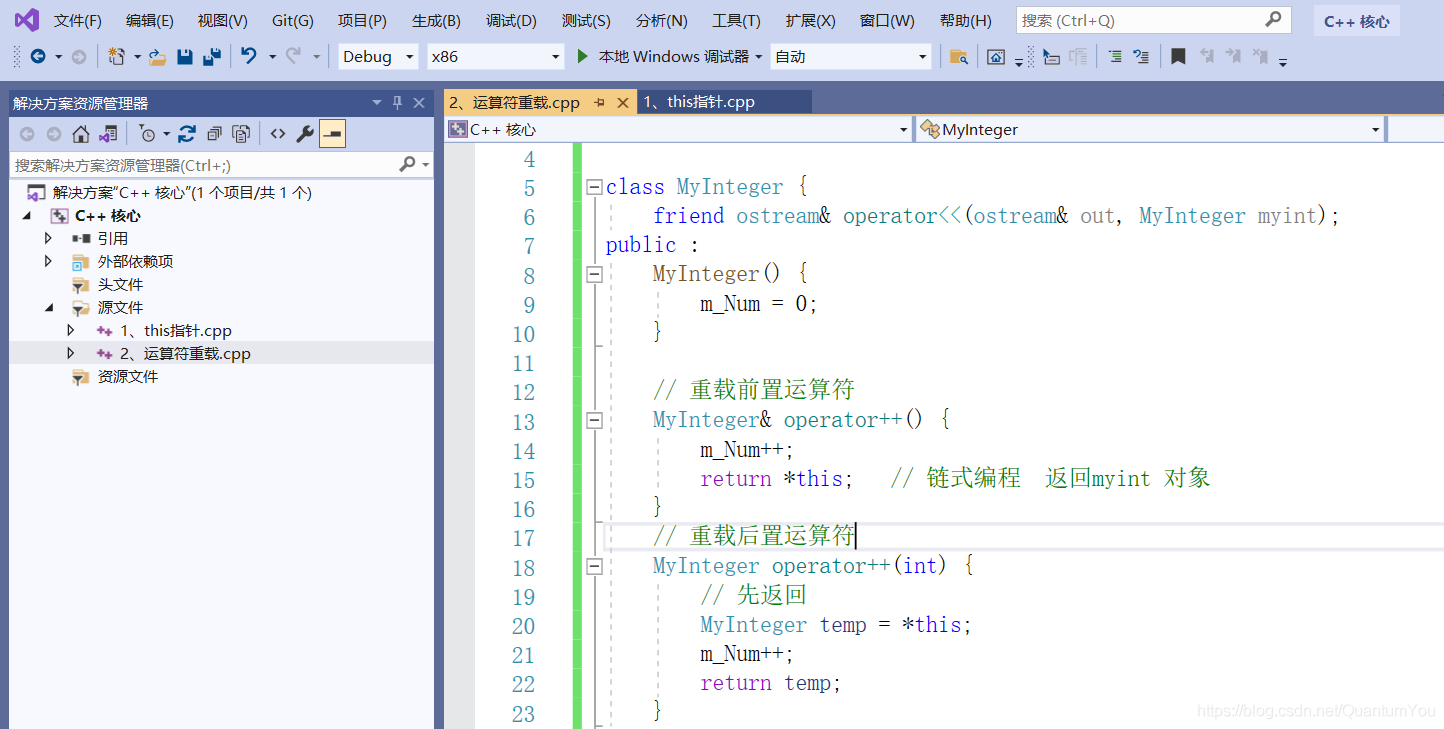
* **作用**： 通过重载递增运算符，实现自己的整型数据

class MyInteger {  
  
 friend ostream& operator<<(ostream& out, MyInteger myint);  
  
public:  
 MyInteger() {  
 m\_Num = 0;  
 }  
 //前置++  
 MyInteger& operator++() {  
 //先++  
 m\_Num++;  
 //再返回  
 return \*this;  
 }  
  
 //后置++  
 MyInteger operator++(int) {  
 //先返回  
 MyInteger temp = \*this; //记录当前本身的值，然后让本身的值加1，但是返回的是以前的值，达到先返回后++；  
 m\_Num++;  
 return temp;  
 }  
  
private:  
 int m\_Num;  
};  
  
  
ostream& operator<<(ostream& out, MyInteger myint) {  
 out << myint.m\_Num;  
 return out;  
}  
  
  
//前置++ 先++ 再返回  
void test01() {  
 MyInteger myInt;  
 cout << ++myInt << endl;  
 cout << myInt << endl;  
}  
  
//后置++ 先返回 再++  
void test02() {  
  
 MyInteger myInt;  
 cout << myInt++ << endl;  
 cout << myInt << endl;  
}  
  
int main() {  
  
 test01();  
 //test02();  
  
 system("pause");  
  
 return 0;  
}

总结： 前置递增返回引用，后置递增返回值

**注意**：

* 关于前置运算符重载返回的是引用（因为核心为一直在对一个数进行操作）
* void operator++(int) int 代表占位参数，可以用于区分前置和后置，因为函数的重载，返回值不作为区分（所以会和前置运算符发生函数名相同的冲突）
* 后置运算符不返回引用的原因MyInteger operator++(int) {}：返回局部对象的引用，局部对象之后会释放，如果再引用会造成非法操作。



## 赋值运算符重载

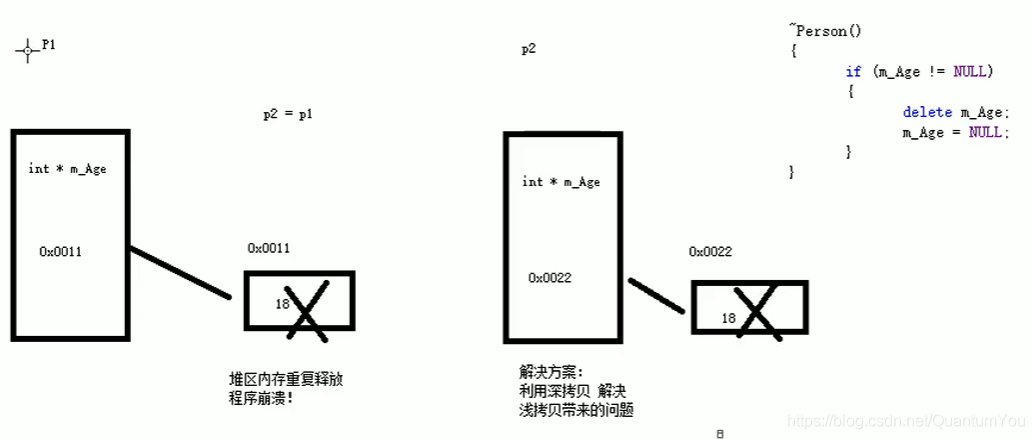
c++编译器至少给一个类添加4个函数

1. 默认构造函数(无参，函数体为空)
2. 默认析构函数(无参，函数体为空)
3. 默认拷贝构造函数，对属性进行值拷贝
4. 赋值运算符 operator=, 对属性进行值拷贝

如果类中有属性指向堆区，做赋值操作时也会出现深浅拷贝问题

**示例：**

class Person  
{  
public:  
  
 Person(int age)  
 {  
 //将年龄数据开辟到堆区  
 m\_Age = new int(age);  
 }  
  
 //重载赋值运算符   
 Person& operator=(Person &p)  
 {  
 if (m\_Age != NULL)  
 {  
 delete m\_Age;  
 m\_Age = NULL;  
 }  
 //编译器提供的代码是浅拷贝  
 //m\_Age = p.m\_Age;  
  
 //提供深拷贝 解决浅拷贝的问题  
 m\_Age = new int(\*p.m\_Age);  
  
 //返回自身  
 return \*this;  
 }  
  
  
 ~Person()  
 {  
 if (m\_Age != NULL)  
 {  
 delete m\_Age;  
 m\_Age = NULL;  
 }  
 }  
  
 //年龄的指针  
 int \*m\_Age;  
  
};  
  
  
void test01()  
{  
 Person p1(18);  
  
 Person p2(20);  
  
 Person p3(30);  
  
 p3 = p2 = p1; //赋值操作  
  
 cout << "p1的年龄为：" << \*p1.m\_Age << endl;  
  
 cout << "p2的年龄为：" << \*p2.m\_Age << endl;  
  
 cout << "p3的年龄为：" << \*p3.m\_Age << endl;  
}  
  
int main() {  
  
 test01();  
  
 //int a = 10;  
 //int b = 20;  
 //int c = 30;  
  
 //c = b = a;  
 //cout << "a = " << a << endl;  
 //cout << "b = " << b << endl;  
 //cout << "c = " << c << endl;  
  
 system("pause");  
  
 return 0;  
}



**注意：**关于指针的使用（指针维护堆区的数据）

Person (int age){  
 m\_Age = new int(age) ;  
}  
  
int \* m\_Age ;

[参考视频](https://www.bilibili.com/video/BV1et411b73Z?p=124&spm_id_from=pageDriver)

## 关系运算符重载

* **作用** :重载关系运算符，可以让两个自定义类型对象进行对比操作

**示例：**

class Person  
{  
public:  
 Person(string name, int age)  
 {  
 this->m\_Name = name;  
 this->m\_Age = age;  
 };  
  
 bool operator==(Person & p)  
 {  
 if (this->m\_Name == p.m\_Name && this->m\_Age == p.m\_Age)  
 {  
 return true;  
 }  
 else  
 {  
 return false;  
 }  
 }  
  
 bool operator!=(Person & p)  
 {  
 if (this->m\_Name == p.m\_Name && this->m\_Age == p.m\_Age)  
 {  
 return false;  
 }  
 else  
 {  
 return true;  
 }  
 }  
  
 string m\_Name;  
 int m\_Age;  
};  
  
void test01()  
{  
 //int a = 0;  
 //int b = 0;  
  
 Person a("孙悟空", 18);  
 Person b("孙悟空", 18);  
  
 if (a == b)  
 {  
 cout << "a和b相等" << endl;  
 }  
 else  
 {  
 cout << "a和b不相等" << endl;  
 }  
  
 if (a != b)  
 {  
 cout << "a和b不相等" << endl;  
 }  
 else  
 {  
 cout << "a和b相等" << endl;  
 }  
}  
  
  
int main() {  
  
 test01();  
  
 system("pause");  
  
 return 0;  
}

## 函数调用运算符重载

* 函数调用运算符 () 也可以重载
* 由于重载后使用的方式非常像函数的调用，因此称为仿函数
* 仿函数没有固定写法，非常灵活

**示例：**

class MyPrint  
{  
public:  
 void operator()(string text)  
 {  
 cout << text << endl;  
 }  
  
};  
void test01()  
{  
 //重载的（）操作符 也称为仿函数  
 MyPrint myFunc;  
 myFunc("hello world");  
}  
  
  
class MyAdd  
{  
public:  
 int operator()(int v1, int v2)  
 {  
 return v1 + v2;  
 }  
};  
  
void test02()  
{  
 MyAdd add;  
 int ret = add(10, 10);  
 cout << "ret = " << ret << endl;  
  
 //匿名对象调用   
 cout << "MyAdd()(100,100) = " << MyAdd()(100, 100) << endl;  
}  
  
int main() {  
  
 test01();  
 test02();  
  
 system("pause");  
  
 return 0;  
}

注意： 匿名函数的使用