一、

短整形运算符重载

```
//短整形运算符重载
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <string.h>
using namespace std;
class Int
{
public:
	Int(long i = 0) : m_i(i)
	{}
	Int(const Int& x)
	{
		this->m_i = x.m_i;
	}
public:
	Int operator+(const Int& x)
	{
		return Int(this->m_i + x.m_i);
	}
	Int operator-(const Int& x)
	{
		return Int(this->m_i - x.m_i);
	}
	Int operator*(const Int& x)
	{
		return Int(this->m_i * x.m_i);
	}
	Int operator/(const Int& x)
	{
		return Int(this->m_i / x.m_i);
	}
	Int operator%(const Int& x)
	{
		return Int(this->m_i % x.m_i);
	}
public:
	Int& operator+=(const Int& x) // a += b;
	{
		this->m_i += x.m_i;
		return *this;
```

```
	}
	Int& operator-=(const Int& x)
	{
		this->m_i -= x.m_i;
		return *this;
	}
	Int& operator*=(const Int& x)
	{
		this->m_i *= x.m_i;
		return *this;
	}
	Int& operator/=(const Int& x)
	{
		this->m_i /= x.m_i;
		return *this;
	}
	Int& operator%=(const Int& x)
	{
		this->m_i %= x.m_i;
		return *this;
	}
public:
	Int operator>>(const Int& x) //a >> b
	{
		return Int(this->m_i >> x.m_i);
	}
	Int operator<<(const Int& x)
	{
		return Int(this->m_i << x.m_i);
	}
	Int& operator>>=(const Int& x)
	{
		this->m_i >>= x.m_i;
		return *this;
	}
	Int& operator<<=(const Int& x)
	{
		this->m_i <<= x.m_i;
		return *this;
	}
public:
	bool operator==(const Int& x)
	{
		if (this->m_i == x.m_i)
			return true;
		return false;
	}
```

```
	bool operator!=(const Int& x)
	{
		if (this->m_i != x.m_i)
			return true;
		return false;
	}
public:
	Int& operator++() //前置++   比后置++效率   可以引用返回
	{
		this->m_i++;
		return *this;
	}
	Int operator++(int)  //后置++
	{
		Int temp(*this);   //调用拷贝构造函数或:
		//Int tmp(this->m_i);  //用构造函数
		this->m_i++;      //*this++
		return temp;
	}
	Int& operator--() //前置--   比后置++效率   可以引用返回
	{
		this->m_i--;
		return *this;
	}
	Int operator--(int)  //后置--
	{
		Int Tmp(m_i);
		this->m_i--;
		return Tmp;
	}
public:
	int GetData()const
	{
		return this->m_i;
	}
private:
	long m_i;
};

int main()
{
	Int a(1);  //int a = 1;  a++  ++a  a-- --a
	Int b(2);

	cout << "a = " << a.GetData() << endl;
	cout << "b = " << b.GetData() << endl;
```

```
    Int result;
    result = a + b; //result = 1 + 2;
    cout << "result = " << result.GetData() << endl;

    result = ++a;
    result = a++;

    return 0;
}
```

以上同类型的符号重载只分析一种，因为+ - += -= * *= / /= % %= 都是一样的逻辑，知一晓百

+号的重载

有整数 a, b = 1, c = 2, d = 3

a = b + c + d;

上述加法要能正常执行就必须等号右边每两个数相加并返回结果，计算顺序是无关的

即要先算出(b+c) = 3, 然后再算出来3 + d = 6 或者先算出来(c + d) = 5, 然后再算出来b + 5 = 6

最终将结果6返回给a

有Test类的对象 a, b (1), c(2), d(3)

对象之间是无法直接相加的，因此需要运算符重载

对+号进行重载成类的成员函数如下：

a = b + c + d;

相当于：a = b + c.operator+(d); a = b.operator+(c.operator+(d));

或： a = b.operator+(c) + d; a = (b.operator+(c)).operator+(d);

```
Int operator+(const Int& x)
    {
        return Int(this->m_i + x.m_i);
    }
```

类的成员函数都是用对象来驱动的，被调用的成员方法的this指针便指向驱动它的对象

用this指针访问前对象的私有数据成员this->m_i，用常引用的形参const Int& x访问后对象的私有数据成员x.m_i，将数据相加然后构造一个临时对象，返回值按值返回，便实现了将两个对象相加，并将结果返回，于是连续的对象相加也能实现

+=重载

对象加等于a += b; 就是 a = a + b ;

单看a += b 则重载的+=是用对象a驱动的，a.operator+=(b); 最终只有a的值发生了改变，所以b可以用常引用。对象a的生命不受重载的+=函数影响，所以a的值发生改变后可以引用返回，以值的方式返回也不会有任何问题。我们都知道函数参数按值返回会借助临时变量，由于是对象按值返回，这样的话就会多调用构造函数，即临时变量占用了内存空间又调用了构造函数，导致效率明显降低了。

所以函数中不受函数控制生存周期的变量，强烈建议用引用返回

```
Int& operator+=(const Int& x) // a += b;
    {
        this->m_i += x.m_i;
        return *this;
    }
```

以下右移位操作符>>和右移位等的操作符>>=重载和上面分析的+和+=是一样的道理

```
Int operator>>(const Int& x) //a >> b
    {
        return Int(this->m_i >> x.m_i);
    }

Int& operator>>=(const Int& x)
    {
        this->m_i >>= x.m_i;
        return *this;
    }
```

==操作符重载如下

```
bool operator==(const Int& x)
    {
        if (this->m_i == x.m_i)
            return true;
        return false;
    }
```

觉得上述啰嗦吧，也可以如下的写法：

但是上面的有更好的逻辑性，且返回值类型匹配都是布尔类型

```
bool operator==(const Int& x)
{
        return (this->m_i == x.m_i)
}
```

前置++和后置++的重载

```
Int& operator++() //前置++    比后置++效率    可以引用返回
{
        this->m_i++;
        return *this;
}

Int operator++(int)  //后置++
{
        Int temp(*this);   //调用拷贝构造函数或:
        //Int tmp(this->m_i);  //用构造函数
        this->m_i++;      //*this++
        return temp;
}
```

前置++和后置++的重载，其实相当简单，只需要知道后置++要在函数列表设一个int就OK

前置++，是给对象先+1，然后以引用返回

后置++，是先返回对象，再给对象+1，错错错，大错特错。再函数中一但返回，函数就结束了，return后面的语句就不再执行了

正确的思路是先借助临时变量记录一下++的左值，然后给左值+1，然后按值返回临时变量（不能用引用返回，语义上临时变量不能用引用返回，临时变量一出作用 域就死亡了，死亡的东西就是未知的也许，那块临时空间还保留着出函数的值，恰巧结果可以正确，但未知和不确定的实物可能就是整个程序死亡的诱因，要严谨要严谨）。

二

二

模拟String类运算符重载

对于3中对模拟String类实现的剖析的例子，我们继续对其进行运算符重载

实现友元函数：length()

重载以下运算符：

operator[]

operator+

operator+=

operator > < >= <= == !=

```
#define _CRT_SECURE_NO_WARNINGS
#include<iostream>
#include<assert.h>
#include<string.h>
using namespace std;

//////////////////////////////////////
//length()
//operator[]
//operator+
//operator+=
//operator > < >= <= == !=
//////////////////////////////////////
class String
{
public:
	String(const char* str = "")  //常类型到常类型
	{
		m_data = new char[strlen(str) + 1];
		strcpy(m_data, str);
	}
	String(const String& s)
	{
		m_data = new char[strlen(s.m_data) + 1];
		strcpy(m_data, s.m_data);
	}
	~String()
	{
		delete[]m_data;
		m_data = NULL;
	}
public:
	size_t length()const
	{
		return strlen(m_data);
	}

	char operator[](int i)
	{
		assert(i >= 0 && i < length());
		return m_data[i];
	}
```

```
    String& operator=(const String& s)
    {
        if (this != &s)
        {
            delete[](this->m_data);
            //new char[s.length() + 1];
            this->m_data = new char[s.length() + 1];
            strcpy(this->m_data, s.m_data);
        }
        return *this;
    }

    String operator+(const String& s)
    {
        char* tmp = new char[length() + s.length() + 1];    //??????
        strcpy(tmp, this->m_data);
        strcat(tmp, s.m_data);

        String temp(tmp);
        delete []tmp;
        return temp;
    }

    String& operator+=(const String& s)
    {
        char* tmp = new char[strlen(this->m_data) + strlen(s.m_data) + 1];
        strcpy(tmp, this->m_data);
        strcat(tmp, s.m_data);

        delete[]m_data;
        this->m_data = tmp;
        return *this;
    }

public:
    bool operator==(const String& s)
    {
        if (strcmp(this->m_data, s.m_data) == 0)
            return true;
        return false;
    }
    bool operator!=(const String& s)
    {
        if (strcmp(this->m_data, s.m_data) != 0)
            return true;
        return false;
    }
```

```
	bool operator>(const String& s)
	{
		if (strcmp(this->m_data, s.m_data) > 0)
			return true;
		return false;
	}
	bool operator<(const String& s)
	{
		if (strcmp(this->m_data, s.m_data) < 0)
			return true;
		return false;
	}
	bool operator>=(const String& s)
	{
		if (strcmp(this->m_data, s.m_data) < 0)
			return false;
		return true;
	}
	bool operator<=(const String& s)
	{
		if (strcmp(this->m_data, s.m_data) > 0)
			return false;
		return true;
	}
private:
	char* m_data;
};


int main()
{
	String s1("Hello"); //s1[0] ==> H
	String s2("Bit.");

	s1 = s2;
	for (int i = 0; i < s1.length(); ++i)
		cout << s1[i];
	cout << endl;


	String s = s1 + s2; //s = HelloBit
	for (int i = 0; i < s.length(); ++i)
		cout << s[i];
	cout << endl;


	s1 += s2;    //
```

```
	for (int i = 0; i < s1.length(); ++i)
		cout << s1[i];
	cout << endl;

	return 0;
}
```

三

运算符重载之重载为成员函数：

下面的程序定义了一个简单的SmallInt类，用来表示从-128到127之间的整数。

类的唯一的数据成员val存放一个-128到127（包含-128和127这两个数）之间的整数，为了方便，

类SmallInt还重载了一些运算符。

阅读SmallInt的定义，回答题目后面的问题。

```
#define _CRT_SECURE_NO_WARNINGS
#include<iostream>
using namespace std;
class SmallInt;
ostream& operator<<(ostream& os, const SmallInt& si);
istream& operator>>(istream& is, SmallInt& si);

//SmallInt si(1000);

class SmallInt
{
public:
	SmallInt(int i = 0);
	//重载插入和抽取运算符
	friend ostream& operator<<(ostream& os, const SmallInt& si);
```

```
        friend istream& operator>>(istream& is, SmallInt& si);

        //重载算术运算符
        SmallInt operator+(const SmallInt& si) { return SmallInt(val + si.val); }
        SmallInt operator-(const SmallInt& si) { return SmallInt(val - si.val); }
        SmallInt operator*(const SmallInt& si) { return SmallInt(val * si.val); }
        SmallInt operator/(const SmallInt& si) { return SmallInt(val / si.val); }
        //重载比较运算符
        bool operator==(const SmallInt& si) { return (val == si.val); }
private:
        char val;
};
SmallInt::SmallInt(int i)
{
        while (i > 127)
                i -= 256;
        while (i < -128)
                i += 256;
        val = i;
}

ostream& operator<<(ostream& os, const SmallInt& si)
{
        os << (int)si.val;
        return os;
}

istream& operator>>(istream& is, SmallInt& si)
{
        int tmp;
        is >> tmp;
        si = SmallInt(tmp);
        return is;
}

int  main()
{
        SmallInt si(1000);
        cout << si << endl;
        SmallInt si1;
        cin >> si1;
        cout << si1 << endl;
        return 0;
}
```

问题：（本小题4分）上面的类定义中，
重载的插入运算符和抽取运算符被定义为类的友元函数，
能不能将这两个运算符定义为类的成员函数？//不能
如果能，写出函数原型，如果不能，说明理由。

语法上可以，语义上不可以代码如下：

```
//////////////////////////////////////
#define _CRT_SECURE_NO_WARNINGS
#include<iostream>
using namespace std;

//SmallInt si(1000);

class SmallInt
{
public:
	SmallInt(int i = 0);
	//重载插入和抽取运算符
	ostream& operator<<(ostream& os)
	{
		os << (int)this->val;
		return os;
	}
	istream& operator>>(istream& is)
	{
		int tmp;
		is >> tmp;
		this->val = tmp;
		return is;
	}

	//重载算术运算符
	SmallInt operator+(const SmallInt& si) { return SmallInt(val + si.val); }
	SmallInt operator-(const SmallInt& si) { return SmallInt(val - si.val); }
	SmallInt operator*(const SmallInt& si) { return SmallInt(val * si.val); }
	SmallInt operator/(const SmallInt& si) { return SmallInt(val / si.val); }
	//重载比较运算符
	bool operator==(const SmallInt& si) { return (val == si.val); }
private:
	char val;
};
SmallInt::SmallInt(int i)
```

```
{
	while (i > 127)
		i -= 256;
	while (i < -128)
		i += 256;
	val = i;
}
int  main()
{
	SmallInt si(1000);
	si << cout << endl;
	SmallInt si1;
	si1 >> cin;
	si1 << cout << endl;
	return 0;
}
```

按题目要求重载，只能像知识总结中一中的那种不符合使用习惯（使用就必须如下）的重载办法

```
   si << cout << endl;
	si1 >> cin;
	si1 << cout << endl;
```

四

explicit关键字

```
#define _CRT_SECURE_NO_WARNINGS
#include<iostream>
using namespace std;
class Test
{
public:
	Test(int d = 0) : m_data(d)      //构造函数的类型转换（隐式）Test t1;   t1 = 998;
	{
		m_count++;
	}
	~Test()
	{
		m_count--;
```

```
    }
public:
    int GetData()const
    {return this->m_data;}
public:
    operator int() //强制转换
    {
        return this->m_data;
    }
public:
    void list()
    {
        fun();
    }
    static void fun()
    {
        m_count = 10;
        cout<<"Test::fun() static"<<endl;
    }
private:
    int m_data;
    static int m_count;
};

int Test::m_count = 0; //

int main()
{
    Test t1;
    t1 = 998;
    return 0;
}
```

```
t1 = 998;       //理论上我们知道普通变量是不能给对象赋值的
```

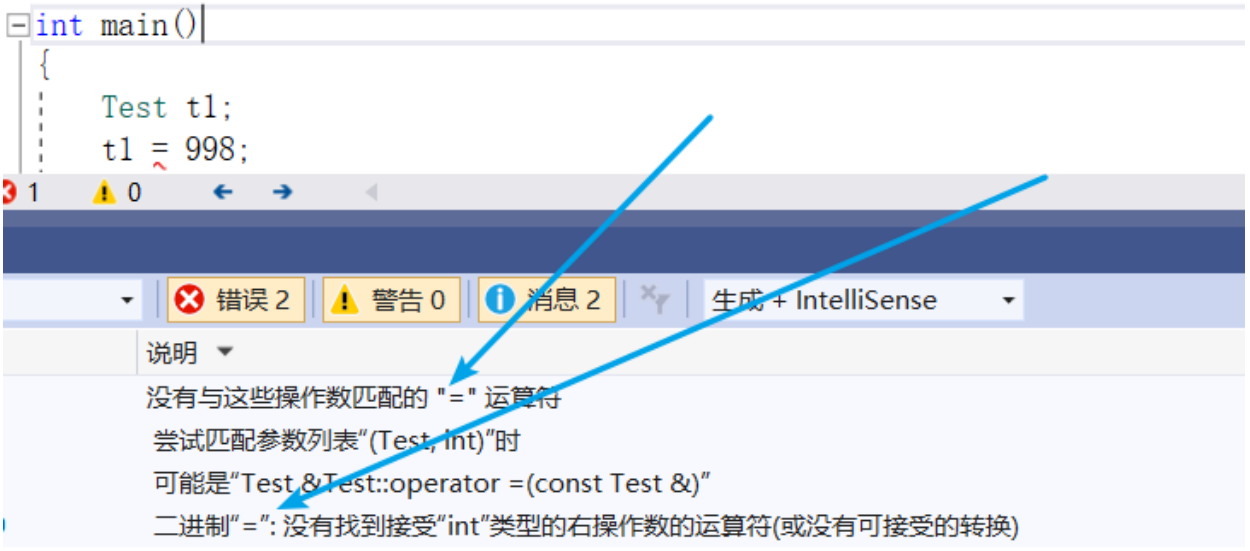
但是实际上构造函数不仅可以构造对象还能够隐式的做类型转换，998给对象赋值，会首先调用构造函数将整形998构造成一个匿名对象，然后用匿名对象去给对象t1赋值。

如果我们给上述代码的构造函数前加上explicit 关键字，构造函数便不能做隐式的类型转化了

```
explicit Test(int d = 0) : m_data(d)
{
        m_count++;
```

}

这样写编译就无法通过如下图：



五

哑成员

```
#define _CRT_SECURE_NO_WARNINGS
#include<iostream>
using namespace std;
class Test
{
public:
	Test(int d = 0) :m_data(d)
	{}
public:
	class Tmp //内部类
	{
	public:
		Tmp(int a , int b) : x(a), y(b)
		{}
	public:
		int GetX()
		{return x;}
	public:
```

```
        int x;  //
        int y;  //哑成员
    };
public:
    int m_data;
};
int main()
{
    Test T;
    cout <<"sizeof(T) = "<< sizeof(T) << endl;
    Test::Tmp tp(1,2);
    cout<<"sizeof(tp) = "<<sizeof(tp) << endl;
    return 0;
}
```

创建类中类的对象时必须加外部类的作用域访问符：

Test::Tmp tp(1,2);　　//创建内部类Tmp的对象时就得加外部类Test的作用域访问符Test::

创建的外部类的对象大小不带内部类

```cpp
class Test
{
public:
    Test(int d = 0) :m_data(d)
    {}
public:
    class Tmp //内部类
    {
    public:
        Tmp(int a , int b) : x(a), y(b)
        {}
    public:
        int GetX()
        {return x;}
    public:
        int x;  //
        int y;  //哑成员
    };
public:
    int m_data;
};
int main()
{
    Test T;
    cout <<"sizeof(T) = "<< sizeof(T) << endl;
    Test::Tmp tp(1,2);
    cout<<"sizeof(tp) = "<<sizeof(tp) << endl;
    return 0;
}
```

外部类对象的大小不包括内部类