数据结构与算法

DATA STRUCTURE

第四讲 函数重载和运算符重载

信息管理与工程学院

2017 - 2018 第一学期

重载（Overload）

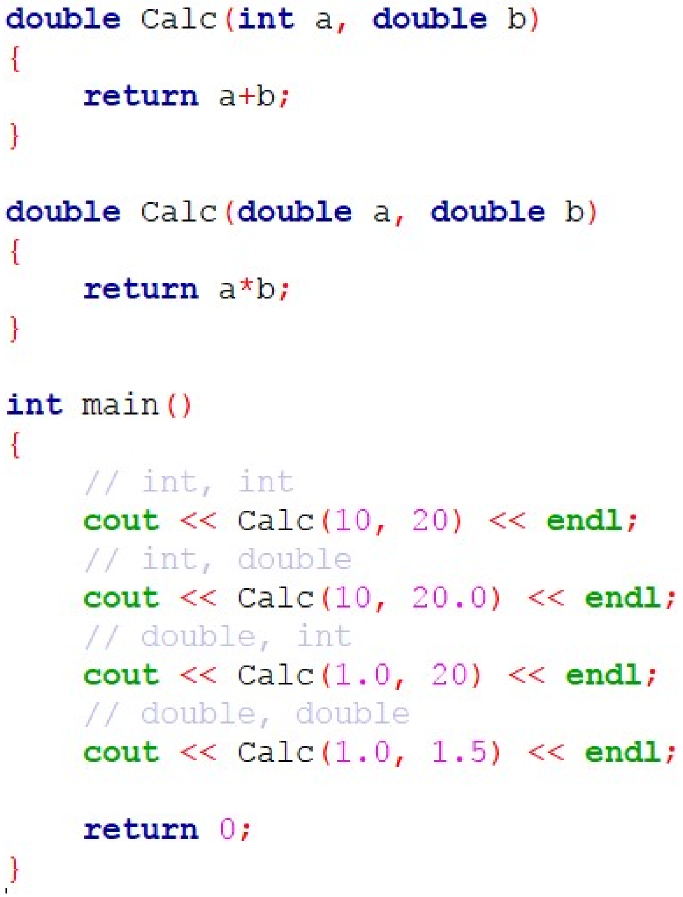
- 函数重载 (function overload)
- 运算符重载 (operator overload)

函数重载

函数重载

- 同一个函数名，参数不同
- 编译器根据参数来识别调用

- 注意返回类型不能区别
- 注意同一个函数实现应该一致，不要一会儿a+b，一会儿a*b

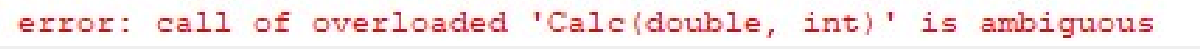


```
double Calc(int a, double b)
{
    return a+b;
}

double Calc(double a, double b)
{
    return a*b;
}

int main()
{
    // int, int
    cout << Calc(10, 20) << endl;
    // int, double
    cout << Calc(10, 20.0) << endl;
    // double, int
    cout << Calc(1.0, 20) << endl;
    // double, double
    cout << Calc(1.0, 1.5) << endl;

    return 0;
}
```

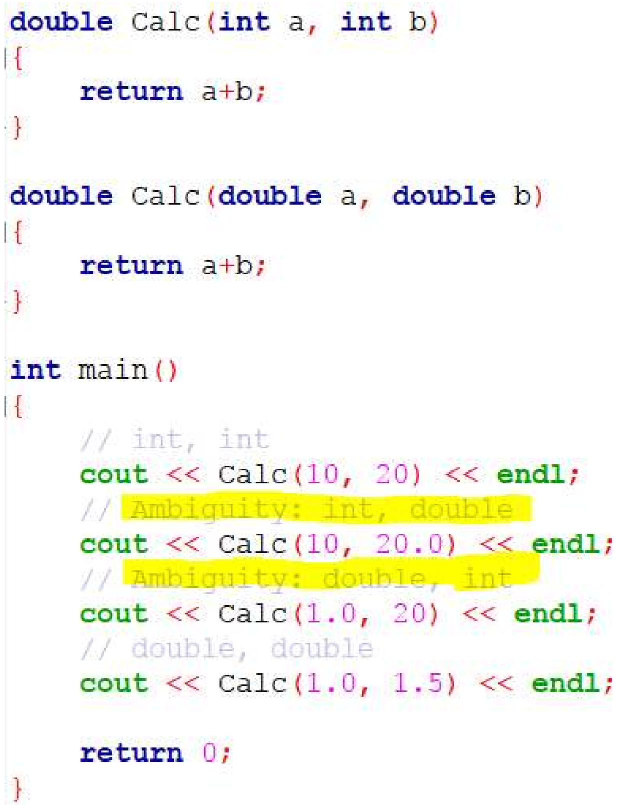
函数重载调用歧义

1. C++首先找完全匹配
2. 兼容数据类型匹配
   - Char， short => int
   - Float => double
   - Enum => int
3. 内置数据类型转换
4. 自定义数据类型转换
5. 最后编译出错

这里(3)类型转换时发现有两个选择



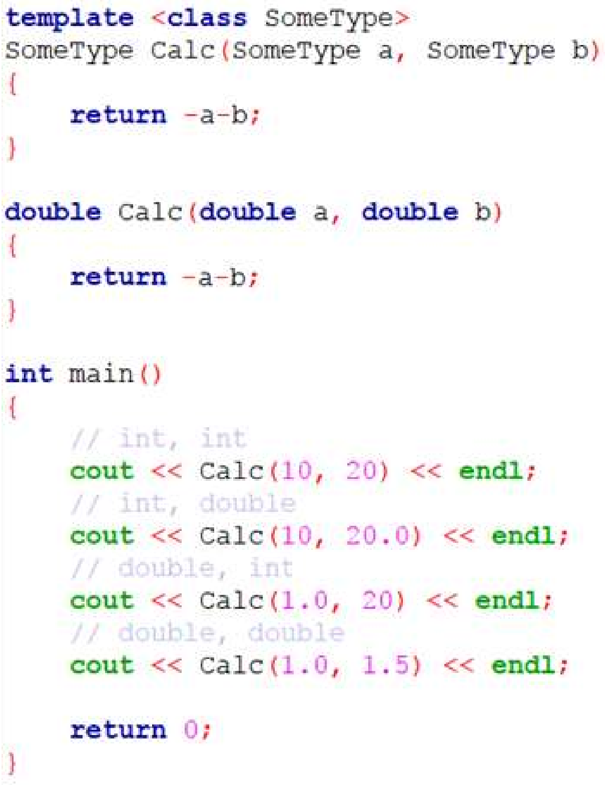
```
error: call of overloaded 'Calc(double, int)' is ambiguous
```



```
double Calc(int a, int b)
{
    return a+b;
}

double Calc(double a, double b)
{
    return a+b;
}

int main()
{
    // int, int
    cout << Calc(10, 20) << endl;
    // Ambiguity: int, double
    cout << Calc(10, 20.0) << endl;
    // Ambiguity: double, int
    cout << Calc(1.0, 20) << endl;
    // double, double
    cout << Calc(1.0, 1.5) << endl;

    return 0;
}
```

函数重载和模板

1. 函数重载目的是让程序简单
2. 模板同样可以达成这个目的
3. 如果重载的类型很多的话，模板更方便

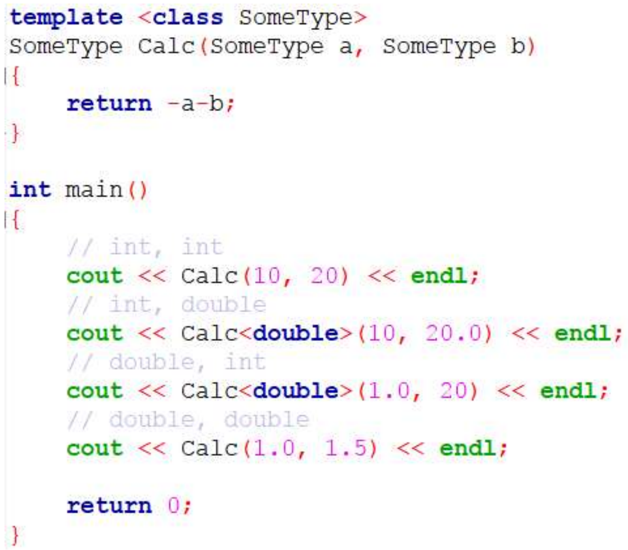


```
template <class SomeType>
SomeType Calc(SomeType a, SomeType b)
{
    return -a-b;
}

double Calc(double a, double b)
{
    return -a-b;
}

int main()
{
    // int, int
    cout << Calc(10, 20) << endl;
    // int, double
    cout << Calc(10, 20.0) << endl;
    // double, int
    cout << Calc(1.0, 20) << endl;
    // double, double
    cout << Calc(1.0, 1.5) << endl;

    return 0;
}
```

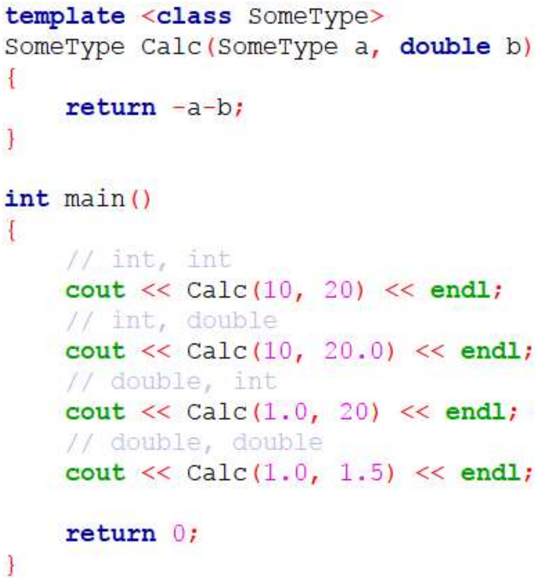
注意这里没有歧义，因为内置类型转换匹配 优先于 自定义类型匹配

函数重载和模板

- 完全使用模板



```cpp
template <class SomeType>
SomeType Calc(SomeType a, SomeType b)
{
    return -a-b;
}

int main()
{
    // int, int
    cout << Calc(10, 20) << endl;
    // int, double
    cout << Calc<double>(10, 20.0) << endl;
    // double, int
    cout << Calc<double>(1.0, 20) << endl;
    // double, double
    cout << Calc(1.0, 1.5) << endl;

    return 0;
}
```



```cpp
template <class SomeType>
SomeType Calc(SomeType a, double b)
{
    return -a-b;
}

int main()
{
    // int, int
    cout << Calc(10, 20) << endl;
    // int, double
    cout << Calc(10, 20.0) << endl;
    // double, int
    cout << Calc(1.0, 20) << endl;
    // double, double
    cout << Calc(1.0, 1.5) << endl;

    return 0;
}
```

运算符重载

运算符重载

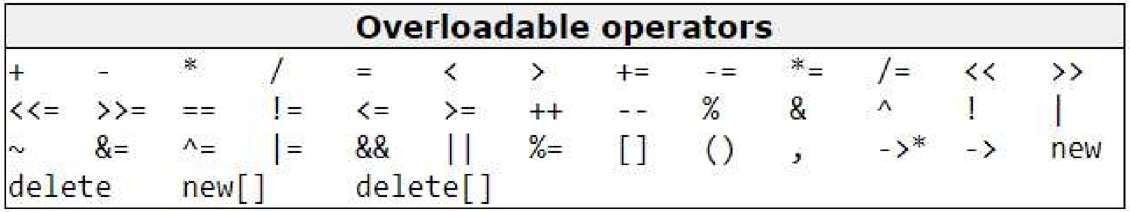
- 算术符其实也是函数
  - datatype operator sign (parameters) { /*... body ...*/ }

- 在自定义数据类型里重载算术符

- C++看到算术符时，
  - 所有操作数都是内置类型，调用缺省操作，或者编译错误
  - 至少一个是自定义数据类型，看有没有重载定义，不行就试类型转换，再不行就编译错误

运算符重载限制

- 不能Overload的运算符：
  - conditional (?:)
  - Sizeof
  - scope (::)
  - member selector (.)
  - member pointer selector (.*)
- 只能overload已经存在的算术符
- 算术符优先级，操作数数量不变
- 至少有一个参数是自定义数据类型



Overloadable operators												
+	-	*	/	=	<	>	+=	-=	*=	/=	<<	>>
<<=	>>=	==	!=	<=	>=	++	--	%	&	^	!	|
~	&=	^=	|=	&&	||	%=	[]	()	,	->*	->	new
delete		new[]		delete[]								

- 重载符号意义最好和之前类似，比如不要把XOR(^)定义为指数操作。
- 对于意义不清楚的运算符最好使用更容易理解的函数，比如string - string

运算符重载例子

- String + String
- Matrix A * double
- cout << Point << endl;
- Faction A == Fraction B

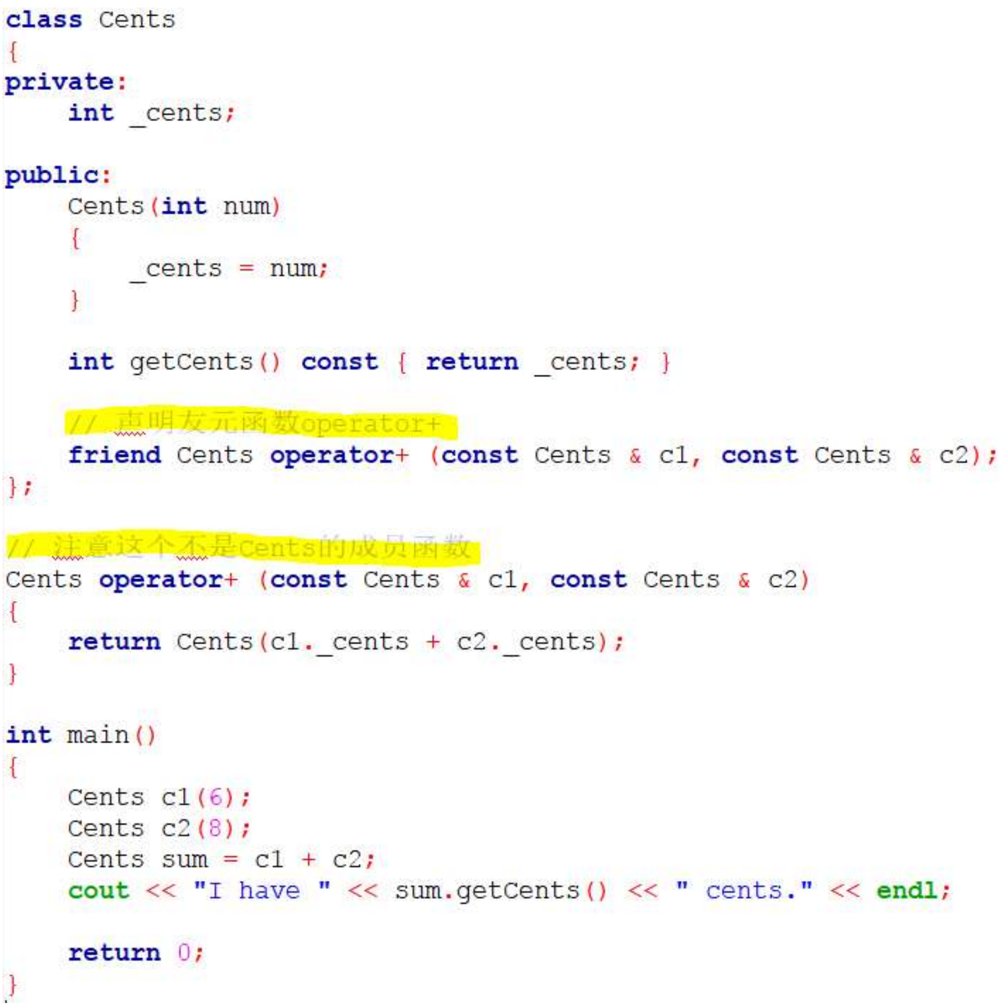
使得程序更直观，简单。

运算符重载

1. 用友元函数
2. 用正常函数
3. 用类成员函数

友元函数重载

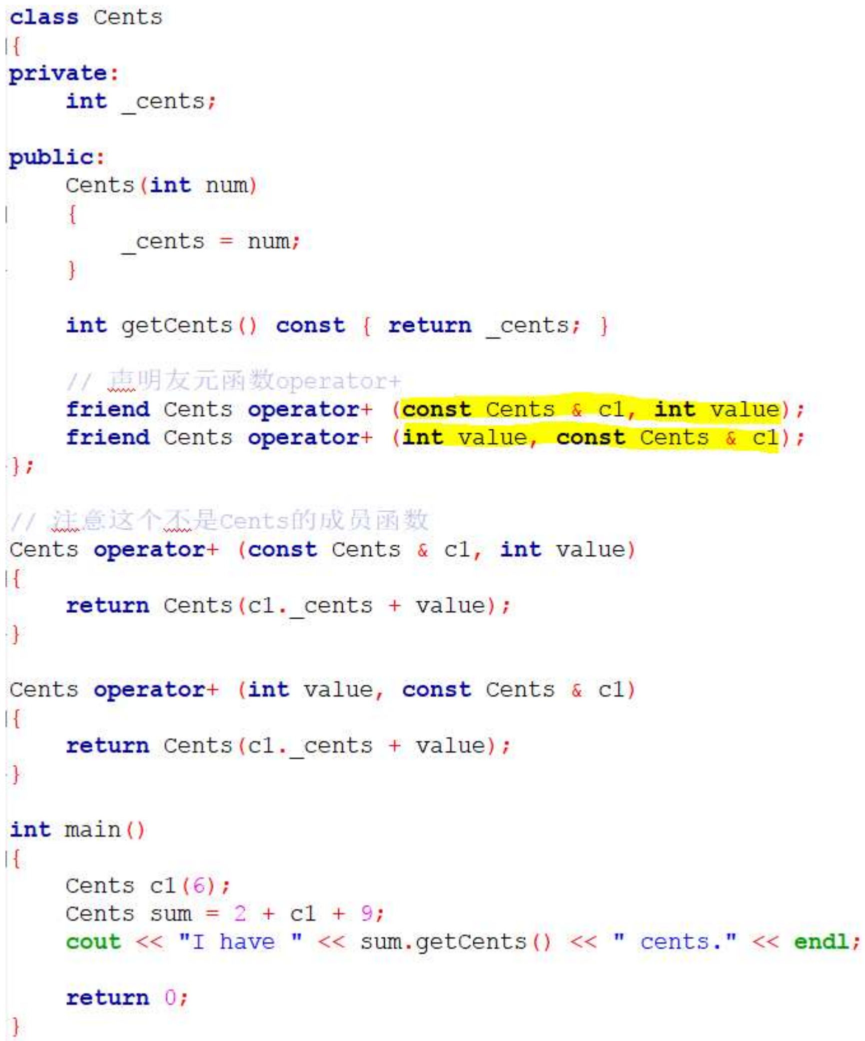
- 对算术运算符+重载
- 在类里声明了友元函数，从而是的operator+可以直接读私有数据
- 注意友元函数不是成员函数



```
class Cents
{
private:
    int _cents;

public:
    Cents(int num)
    {
        _cents = num;
    }

    int getCents() const { return _cents; }

    // 声明友元函数operator+
    friend Cents operator+ (const Cents & c1, const Cents & c2);
};

// 注意这个不是Cents的成员函数
Cents operator+ (const Cents & c1, const Cents & c2)
{
    return Cents(c1._cents + c2._cents);
}

int main()
{
    Cents c1(6);
    Cents c2(8);
    Cents sum = c1 + c2;
    cout << "I have " << sum.getCents() << " cents." << endl;

    return 0;
}
```

友元函数重载

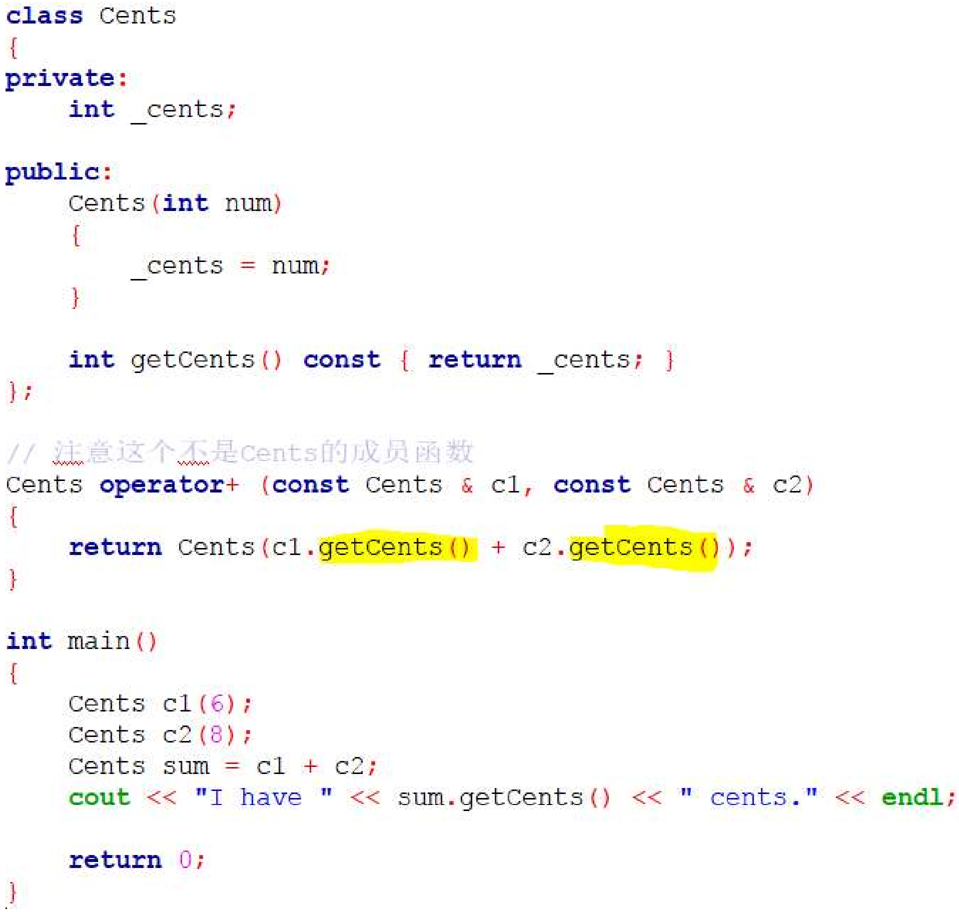
- 对算术运算符+重载，操作数类型不一样
- 一般来说要根据次序定义多个



```
class Cents
{
private:
    int _cents;

public:
    Cents(int num)
    {
        _cents = num;
    }

    int getCents() const { return _cents; }

    // 声明友元函数operator+
    friend Cents operator+ (const Cents & c1, int value);
    friend Cents operator+ (int value, const Cents & c1);
};

// 注意这个不是Cents的成员函数
Cents operator+ (const Cents & c1, int value)
{
    return Cents(c1._cents + value);
}

Cents operator+ (int value, const Cents & c1)
{
    return Cents(c1._cents + value);
}

int main()
{
    Cents c1(6);
    Cents sum = 2 + c1 + 9;
    cout << "I have " << sum.getCents() << " cents." << endl;

    return 0;
}
```

正常函数重载

- 几乎和之前一样
- 声明不在类里面，这样类更安全
- 重载定义里不能使用私有数据



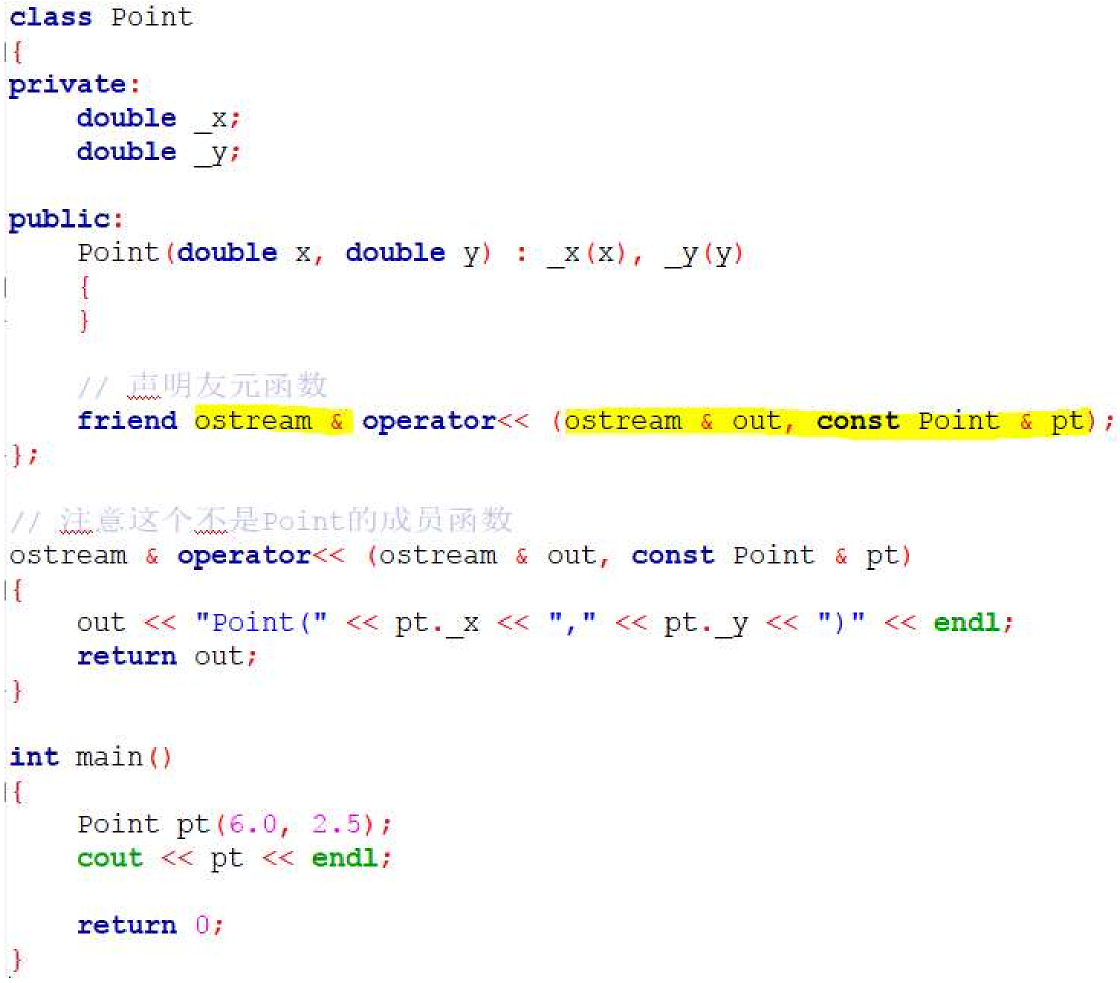
```
class Cents
{
private:
    int _cents;

public:
    Cents(int num)
    {
        _cents = num;
    }

    int getCents() const { return _cents; }
};

// 注意这个不是Cents的成员函数
Cents operator+ (const Cents & c1, const Cents & c2)
{
    return Cents(c1.getCents() + c2.getCents());
}

int main()
{
    Cents c1(6);
    Cents c2(8);
    Cents sum = c1 + c2;
    cout << "I have " << sum.getCents() << " cents." << endl;

    return 0;
}
```

输出<<重载

- std::cout << pt，操作数是ostream和□Point
- 返回值是ostream引用，这样才可以多重调用

  std::cout << pt << endl

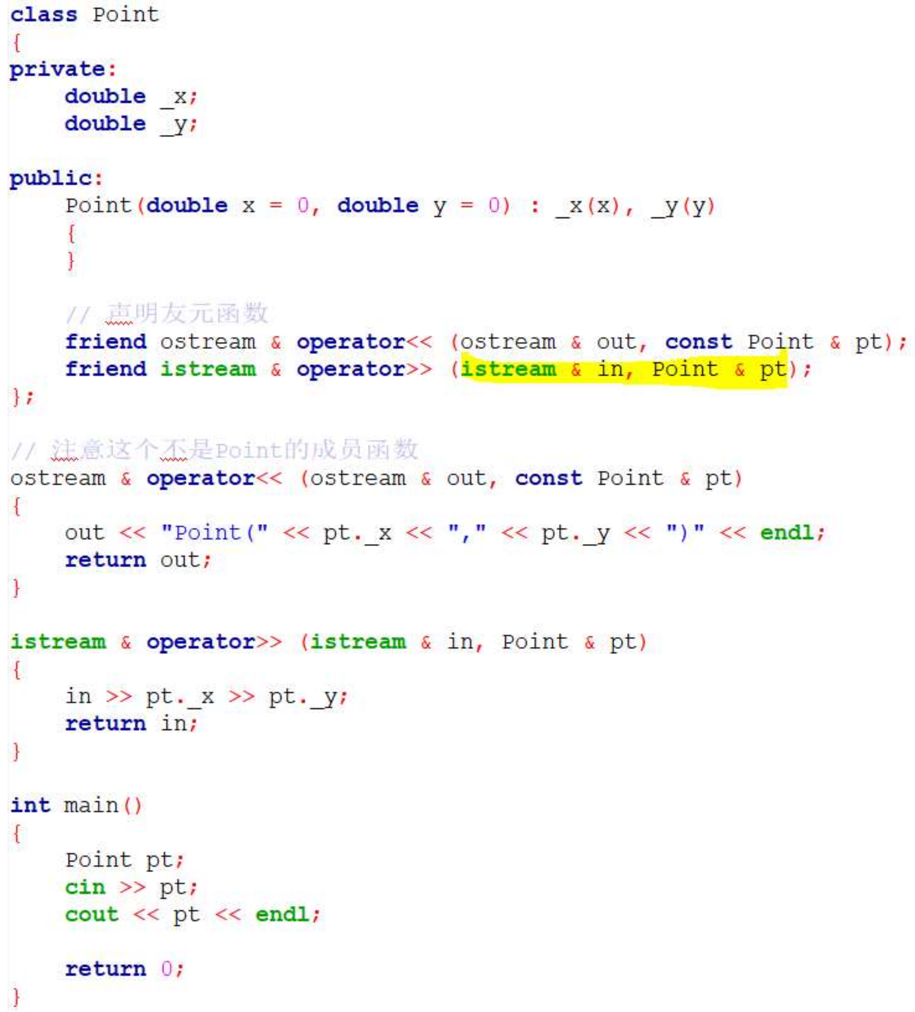
注意这里返回的ostream不能是临时变量的引用



```
class Point
{
private:
    double _x;
    double _y;

public:
    Point(double x, double y) : _x(x), _y(y)
    {
    }

    // 声明友元函数
    friend ostream & operator<< (ostream & out, const Point & pt);
};

// 注意这个不是Point的成员函数
ostream & operator<< (ostream & out, const Point & pt)
{
    out << "Point(" << pt._x << "," << pt._y << ")" << endl;
    return out;
}

int main()
{
    Point pt(6.0, 2.5);
    cout << pt << endl;

    return 0;
}
```

输入>>重载

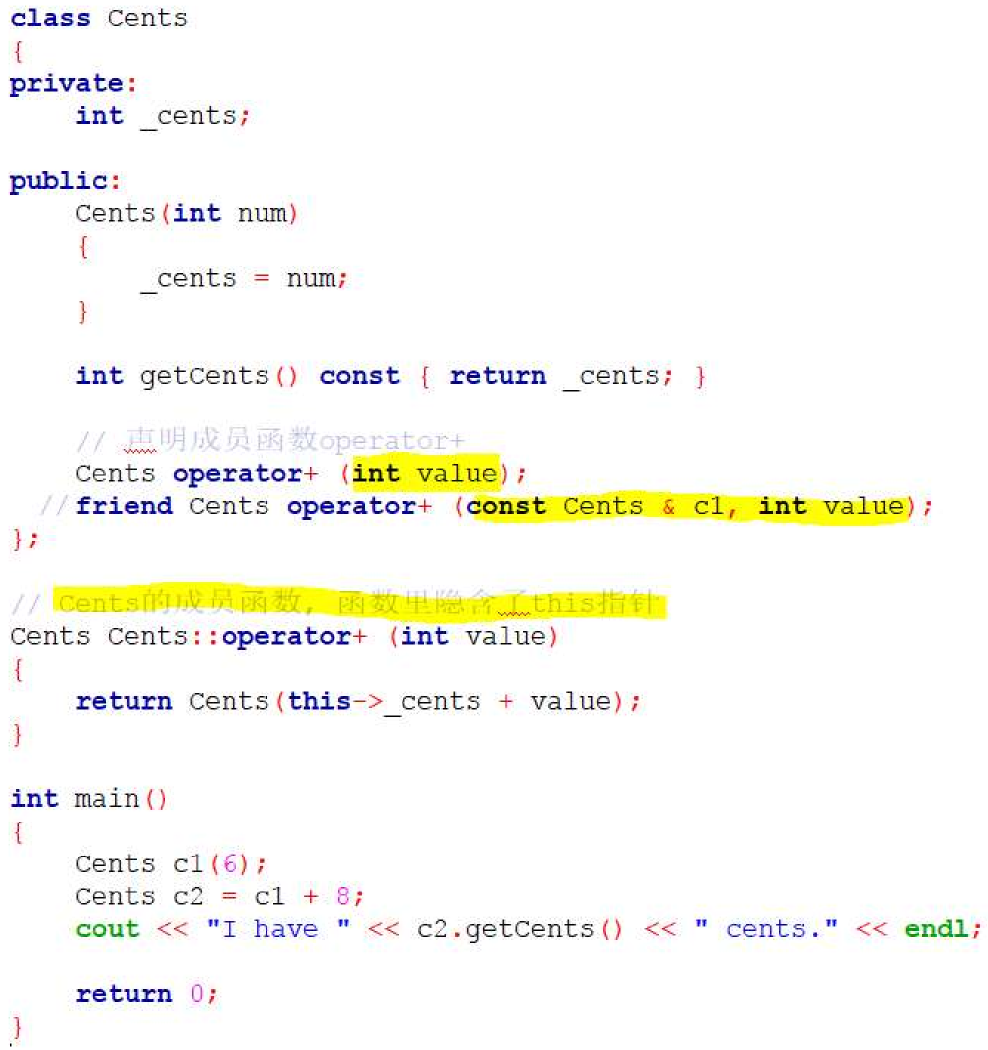
- std::cin >> pt， 操作数是istream和Point
- 和输出<<几乎一样，但是第二个参数不能加const



```
class Point
{
private:
    double _x;
    double _y;

public:
    Point(double x = 0, double y = 0) : _x(x), _y(y)
    {
    }

    // 声明友元函数
    friend ostream & operator<< (ostream & out, const Point & pt);
    friend istream & operator>> (istream & in, Point & pt);
};

// 注意这个不是Point的成员函数
ostream & operator<< (ostream & out, const Point & pt)
{
    out << "Point(" << pt._x << "," << pt._y << ")" << endl;
    return out;
}

istream & operator>> (istream & in, Point & pt)
{
    in >> pt._x >> pt._y;
    return in;
}

int main()
{
    Point pt;
    cin >> pt;
    cout << pt << endl;

    return 0;
}
```

成员函数重载

- 左边的参数移去了，因为隐含的this指针
- c1+8其实就是operator+(c1, 8)，或者说c1.operator+(2)



```
class Cents
{
private:
    int _cents;

public:
    Cents(int num)
    {
        _cents = num;
    }

    int getCents() const { return _cents; }

    // 声明成员函数operator+
    Cents operator+ (int value);
  //friend Cents operator+ (const Cents & c1, int value);
};

// Cents的成员函数，函数里隐含了this指针
Cents Cents::operator+ (int value)
{
    return Cents(this->_cents + value);
}

int main()
{
    Cents c1(6);
    Cents c2 = c1 + 8;
    cout << "I have " << c2.getCents() << " cents." << endl;

    return 0;
}
```

小结

- 有些不能用友元函数重载

  =，[]，->，()，必须用成员函数重载

- 有些不能用成员函数重载，除非最左边的参数是自定义类

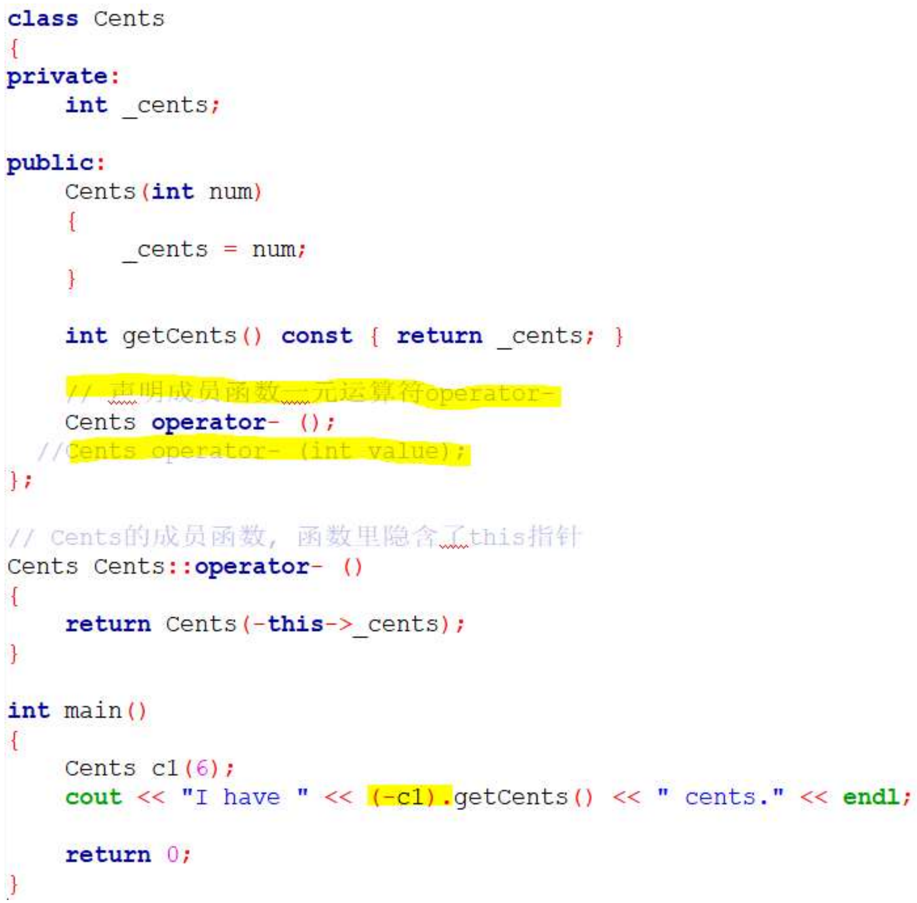
  <<，>>，必须用友元/正常函数重载

  Operator+(int, Cents)，必须用友元/正常函数重载

- 如果运算符修改了左边参数，比如+=，用成员函数重载
- 如果运算符不修改了左边参数，比如+，用友元/正常函数重载
- 一元运算符一般用成员函数重载

一元运算符重载

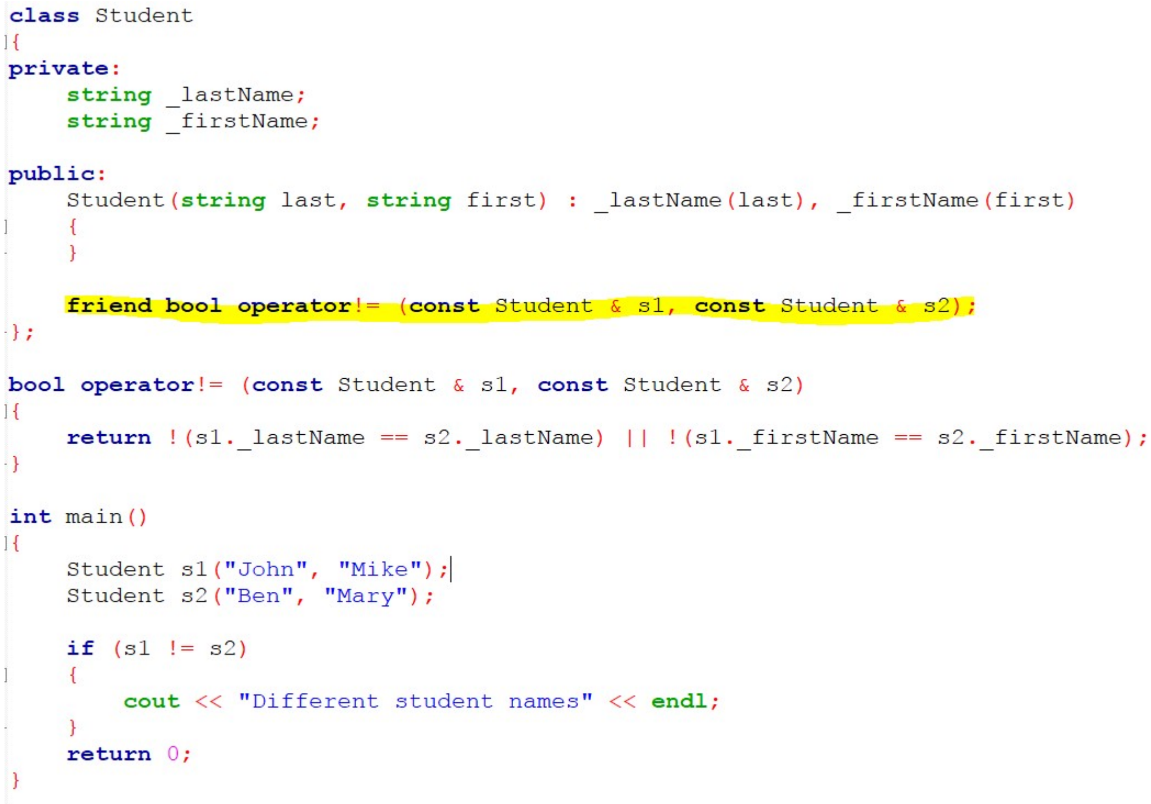
- 一元运算符+，-，！
- 这里和二元运算符+没有歧义，用参数数量识别



```
class Cents
{
private:
    int _cents;

public:
    Cents(int num)
    {
        _cents = num;
    }

    int getCents() const { return _cents; }

    // 声明成员函数一元运算符operator-
    Cents operator- ();
  //Cents operator- (int value);
};

// Cents的成员函数, 函数里隐含了this指针
Cents Cents::operator- ()
{
    return Cents(-this->_cents);
}

int main()
{
    Cents c1(6);
    cout << "I have " << (-c1).getCents() << " cents." << endl;

    return 0;
}
```

比较运算符重载

- <, >, ==, !=
- 因为不改变左边参数，所以用友元函数重载
- 这里定义>不是很有意义



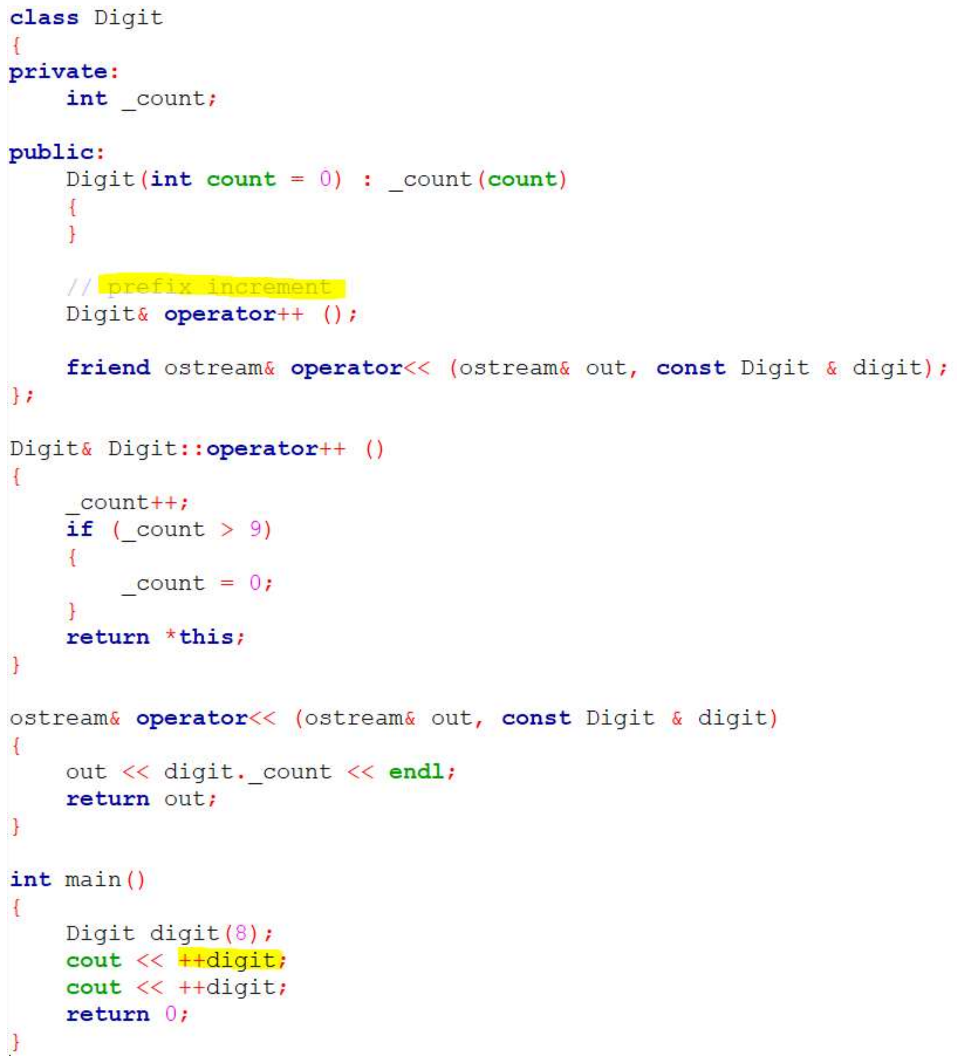
```
class Student
{
private:
    string _lastName;
    string _firstName;

public:
    Student(string last, string first) : _lastName(last), _firstName(first)
    {
    }

    friend bool operator!= (const Student & s1, const Student & s2);
};

bool operator!= (const Student & s1, const Student & s2)
{
    return !(s1._lastName == s2._lastName) || !(s1._firstName == s2._firstName);
}

int main()
{
    Student s1("John", "Mike");
    Student s2("Ben", "Mary");

    if (s1 != s2)
    {
        cout << "Different student names" << endl;
    }
    return 0;
}
```

自增减运算符重载

前缀++， --

首先自加/减一，

然后再进行表达式计算



```cpp
class Digit
{
private:
    int _count;

public:
    Digit(int count = 0) : _count(count)
    {
    }

    // prefix increment
    Digit& operator++ ();

    friend ostream& operator<< (ostream& out, const Digit & digit);
};

Digit& Digit::operator++ ()
{
    _count++;
    if (_count > 9)
    {
        _count = 0;
    }
    return *this;
}

ostream& operator<< (ostream& out, const Digit & digit)
{
    out << digit._count << endl;
    return out;
}

int main()
{
    Digit digit(8);
    cout << ++digit;
    cout << ++digit;
    return 0;
}
```

自增减运算符重载

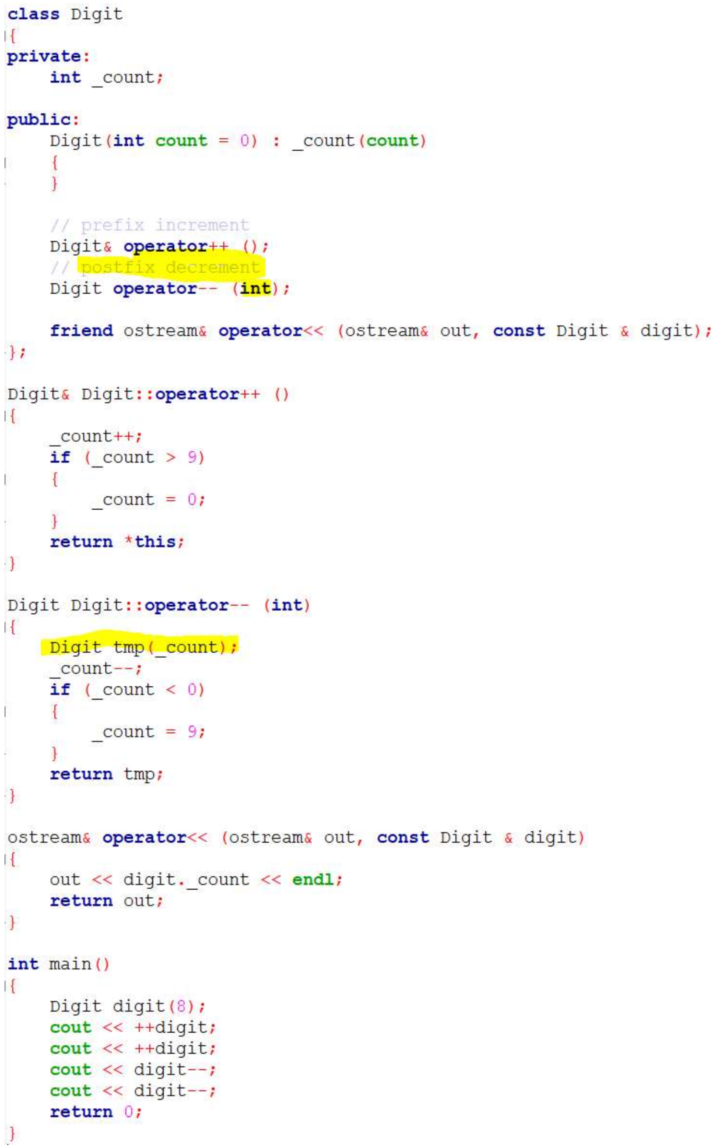
后缀++， --

首先进行表达式计算，

然后自加/减一，

需要加dummy int来区别前缀

返回的是临时变量，开销比前缀大

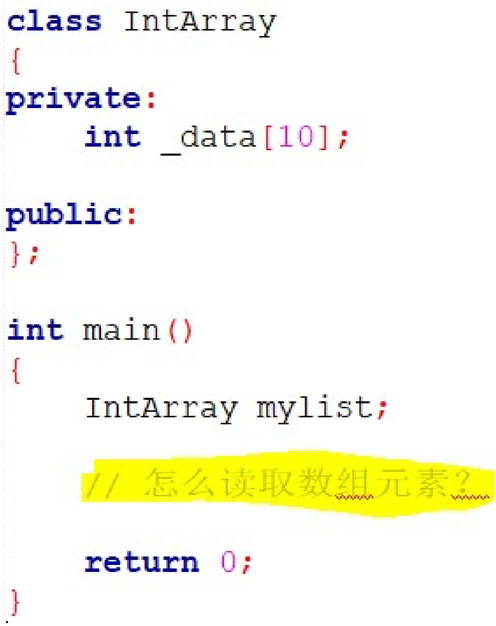


```
class Digit
{
private:
    int _count;

public:
    Digit(int count = 0) : _count(count)
    {
    }

    // prefix increment
    Digit& operator++ ();
    // postfix decrement
    Digit operator-- (int);

    friend ostream& operator<< (ostream& out, const Digit & digit);
};

Digit& Digit::operator++ ()
{
    _count++;
    if (_count > 9)
    {
        _count = 0;
    }
    return *this;
}

Digit Digit::operator-- (int)
{
    Digit tmp(_count);
    _count--;
    if (_count < 0)
    {
        _count = 9;
    }
    return tmp;
}

ostream& operator<< (ostream& out, const Digit & digit)
{
    out << digit._count << endl;
    return out;
}

int main()
{
    Digit digit(8);
    cout << ++digit;
    cout << ++digit;
    cout << digit--;
    cout << digit--;
    return 0;
}
```

下标运算符[]重载

假如我们封装了一个静态数组

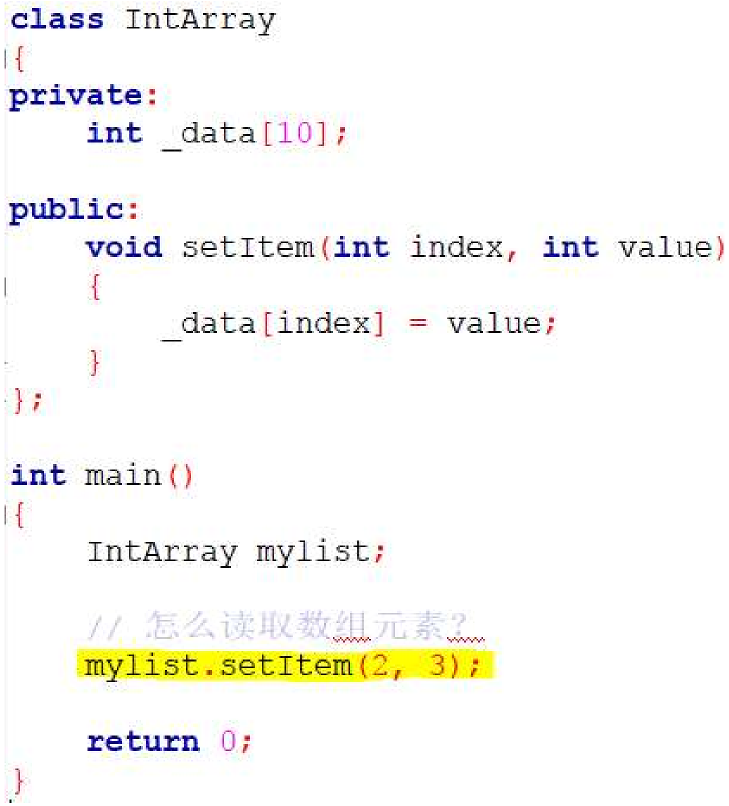
怎么读取数组元素？



```
class IntArray
{
private:
    int _data[10];

public:
};

int main()
{
    IntArray mylist;

    // 怎么读取数组元素？

    return 0;
}
```

下标运算符[]重载

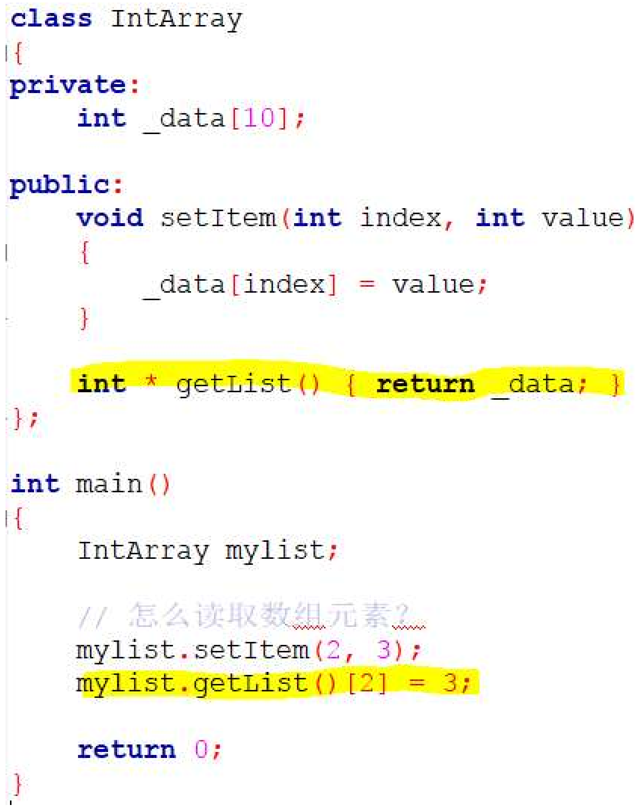
1. 定义一个函数setItem



```
class IntArray
{
private:
    int _data[10];

public:
    void setItem(int index, int value)
    {
        _data[index] = value;
    }
};

int main()
{
    IntArray mylist;

    // 怎么读取数组元素？
    mylist.setItem(2, 3);

    return 0;
}
```

下标运算符[]重载

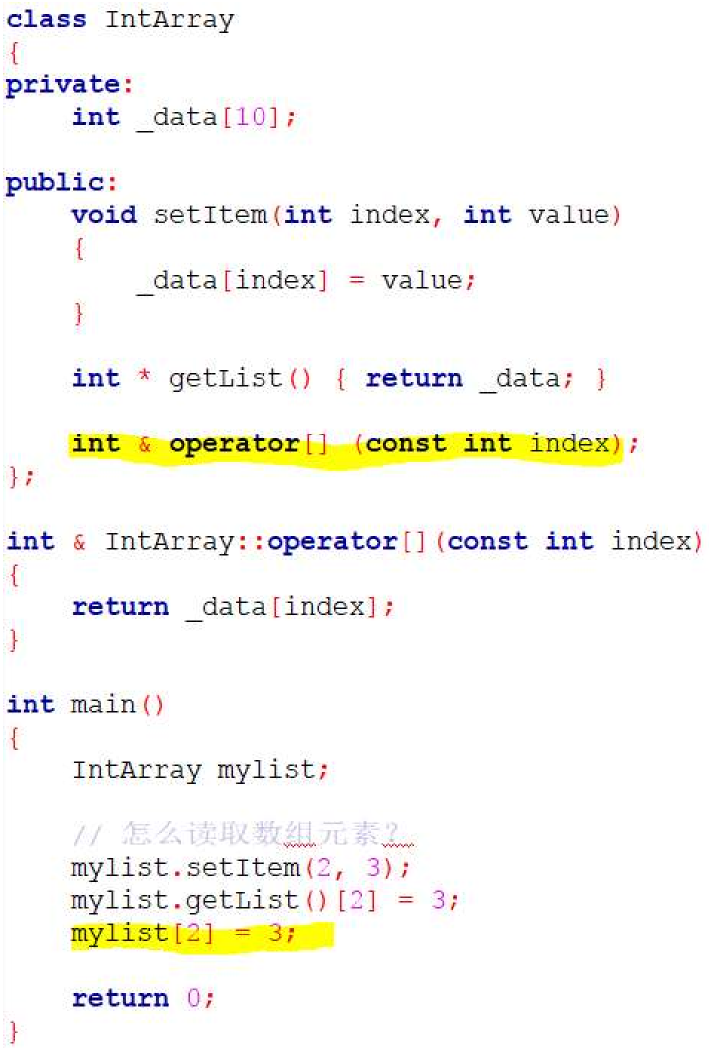
1. 定义一个函数setItem
2. 返回内部数组地址



```
class IntArray
{
private:
    int _data[10];

public:
    void setItem(int index, int value)
    {
        _data[index] = value;
    }

    int * getList() { return _data; }
};

int main()
{
    IntArray mylist;

    // 怎么读取数组元素？
    mylist.setItem(2, 3);
    mylist.getList()[2] = 3;

    return 0;
}
```

下标运算符[]重载

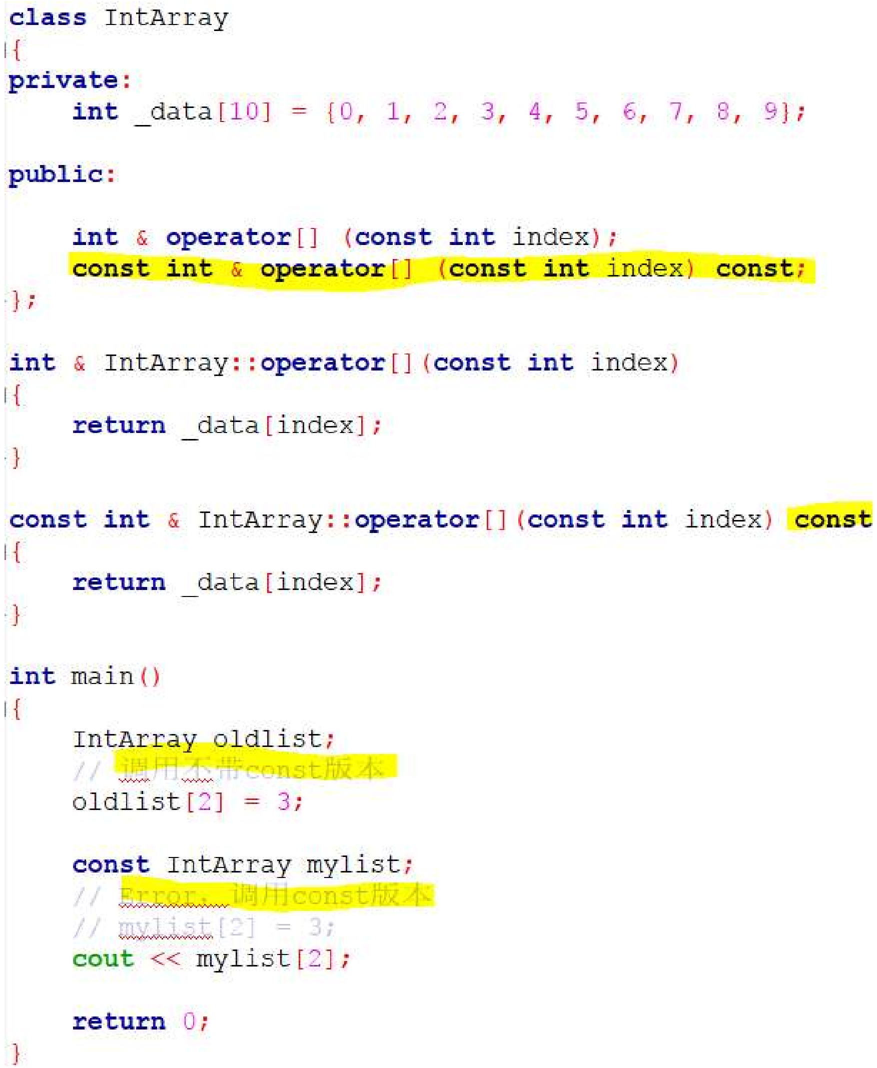
1. 定义一个函数setItem
2. 返回内部数组地址
3. 重载[]



```cpp
class IntArray
{
private:
    int _data[10];

public:
    void setItem(int index, int value)
    {
        _data[index] = value;
    }

    int * getList() { return _data; }

    int & operator[] (const int index);
};

int & IntArray::operator[](const int index)
{
    return _data[index];
}

int main()
{
    IntArray mylist;

    // 怎么读取数组元素？
    mylist.setItem(2, 3);
    mylist.getList()[2] = 3;
    mylist[2] = 3;

    return 0;
}
```

下标运算符[]重载

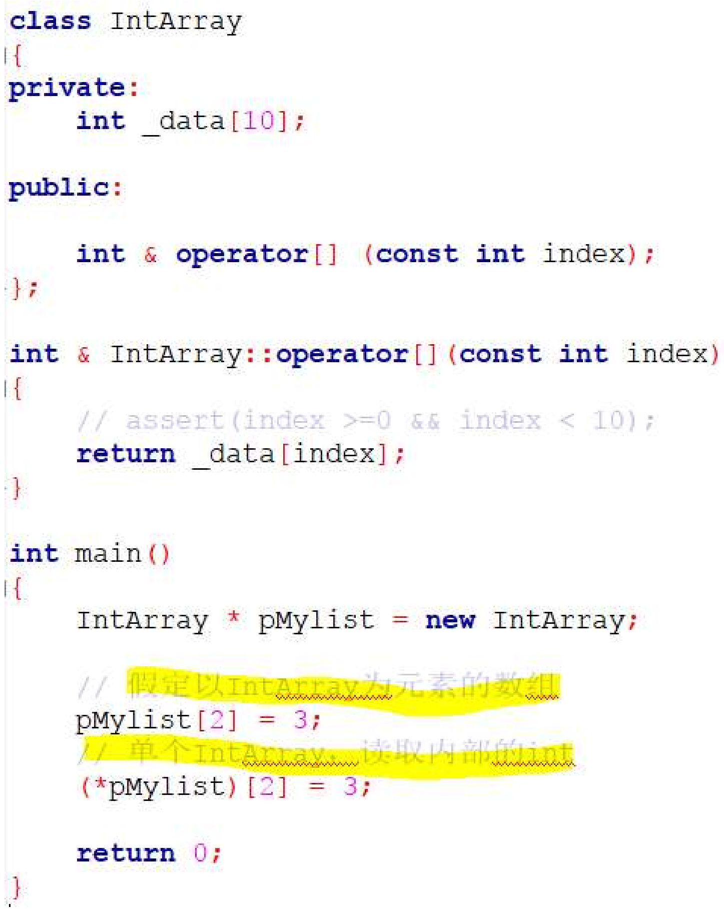
1. Operator[]必须返回引用
2. 或者说必须是左值（有实际地址）
3. Const object
4. 注意类函数可以用const区分
5. 重载[]后可以检查数组越界



```
class IntArray
{
private:
    int _data[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

public:

    int & operator[] (const int index);
    const int & operator[] (const int index) const;
};

int & IntArray::operator[](const int index)
{
    return _data[index];
}

const int & IntArray::operator[](const int index) const
{
    return _data[index];
}

int main()
{
    IntArray oldlist;
    // 调用不带const版本
    oldlist[2] = 3;

    const IntArray mylist;
    // Error, 调用const版本
    // mylist[2] = 3;
    cout << mylist[2];

    return 0;
}
```

下标运算符[]重载

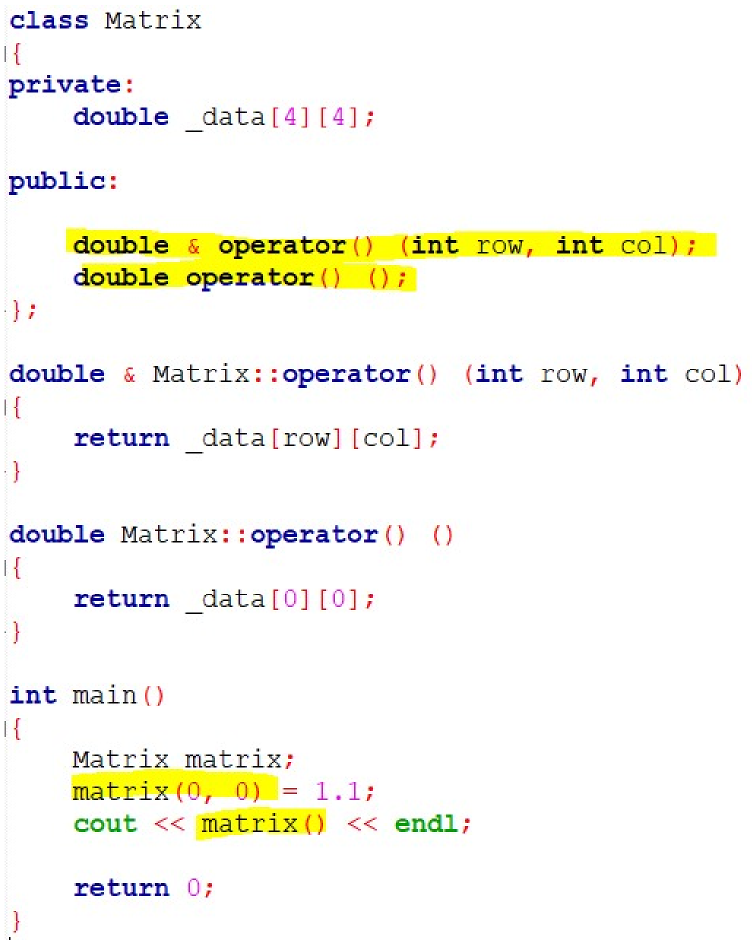
1. 注意指针和[]混淆
2. 重载[]的参数不限于int，可以是别的，比如string



```
class IntArray
{
private:
    int _data[10];

public:

    int & operator[] (const int index);
};

int & IntArray::operator[](const int index)
{
    // assert(index >=0 && index < 10);
    return _data[index];
}

int main()
{
    IntArray * pMylist = new IntArray;

    // 假定以IntArray为元素的数组
    pMylist[2] = 3;
    // 单个IntArray，读取内部的int
    (*pMylist)[2] = 3;

    return 0;
}
```

函数运算符()重载

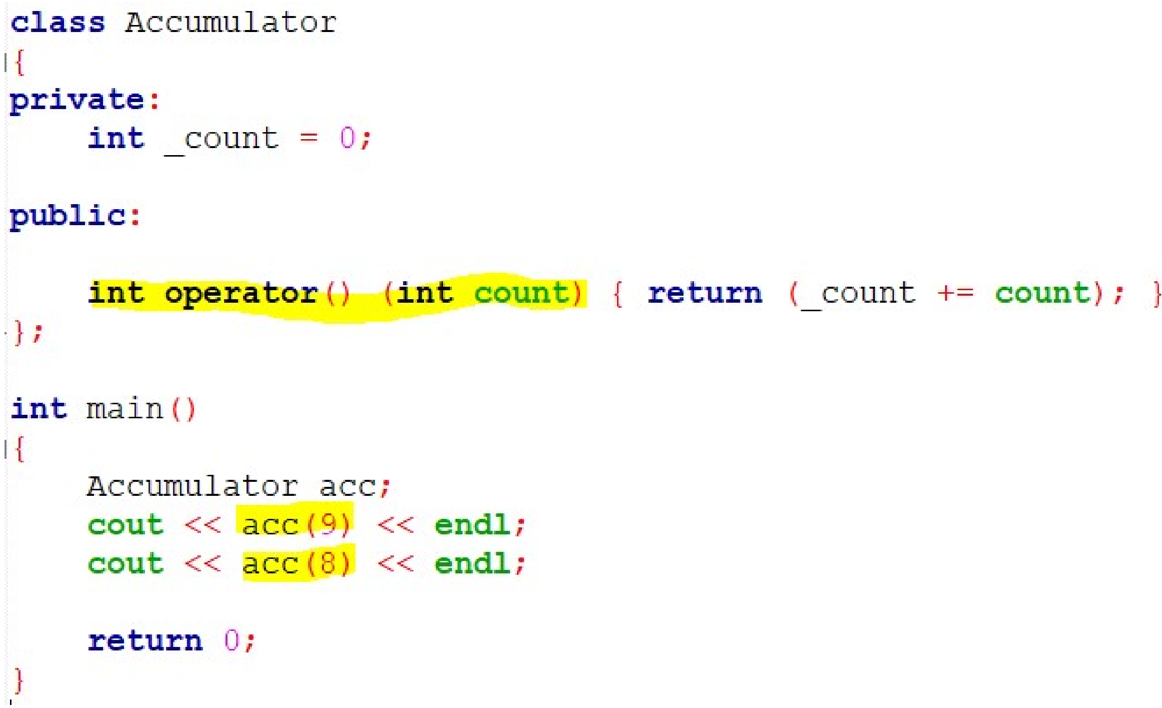
- 重载()的参数的类型和数目都可以变动
- 必须是成员函数
- 谨慎使用
- 可以用来读取二位数组
- 可以用来读取一维数组一批元素



```
class Matrix
{
private:
    double _data[4][4];

public:

    double & operator() (int row, int col);
    double operator() ();
};

double & Matrix::operator() (int row, int col)
{
    return _data[row][col];
}

double Matrix::operator() ()
{
    return _data[0][0];
}

int main()
{
    Matrix matrix;
    matrix(0, 0) = 1.1;
    cout << matrix() << endl;

    return 0;
}
```

Function object(functor)

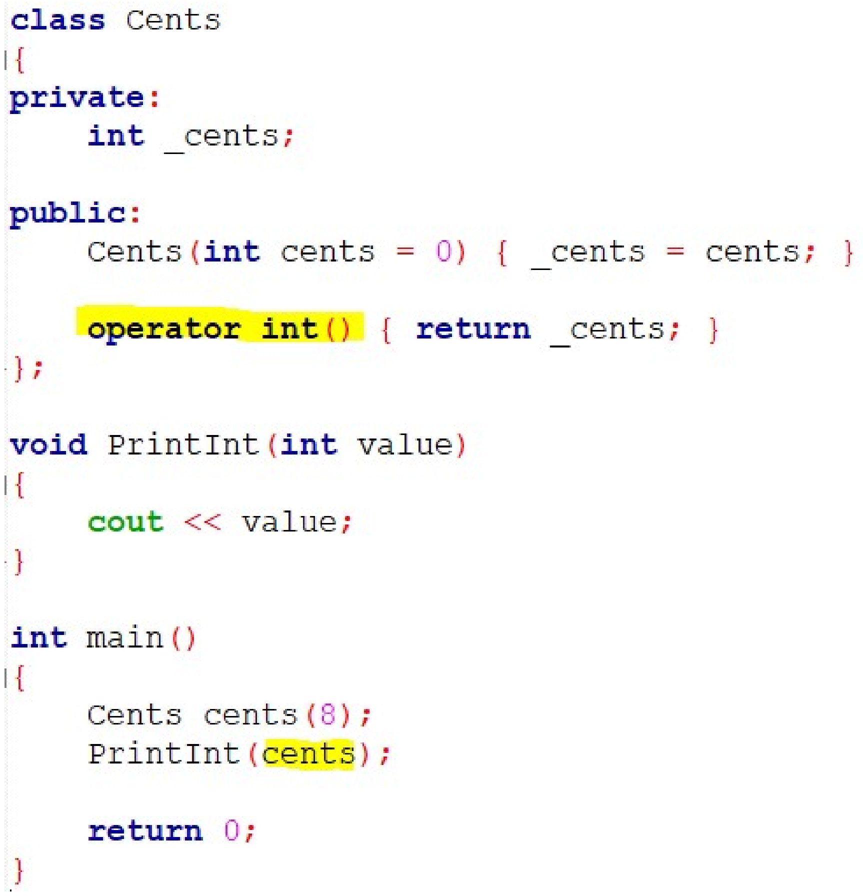
- 使用时像函数，其实是class，保存了数据
- 类似于函数里定义了static局部变量



```
class Accumulator
{
private:
    int _count = 0;

public:

    int operator() (int count) { return (_count += count); }
};

int main()
{
    Accumulator acc;
    cout << acc(9) << endl;
    cout << acc(8) << endl;

    return 0;
}
```

重载typecast

- 比如operator int()
- 没有返回类型
- 可以重载自定义类型cast



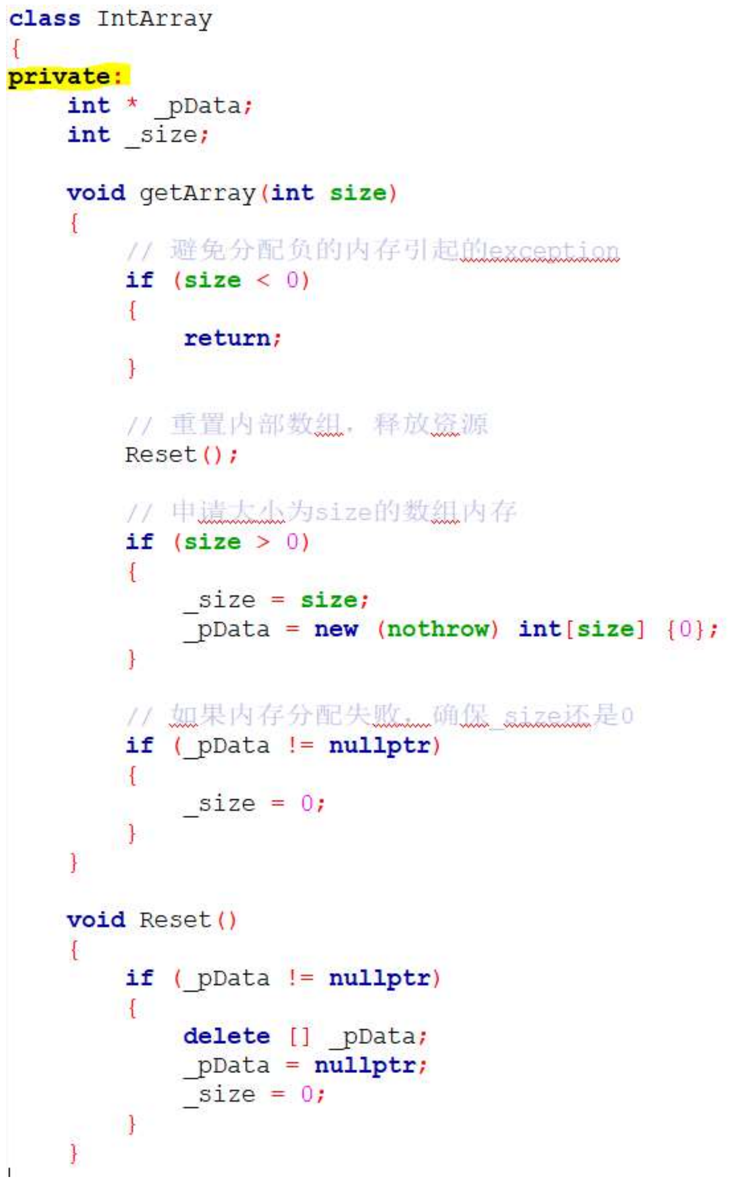
```
class Cents
{
private:
    int _cents;

public:
    Cents(int cents = 0) { _cents = cents; }

    operator int() { return _cents; }
};

void PrintInt(int value)
{
    cout << value;
}

int main()
{
    Cents cents(8);
    PrintInt(cents);

    return 0;
}
```

上机实验

- 创建类IntArray
  - 要求内部动态分配数组内存
  - 注意释放资源（rule of three）
  - 支持常用操作（Length(), [], <<, etc)
  - 注意边界检查
  - 注意常量const对象使用

IntArray

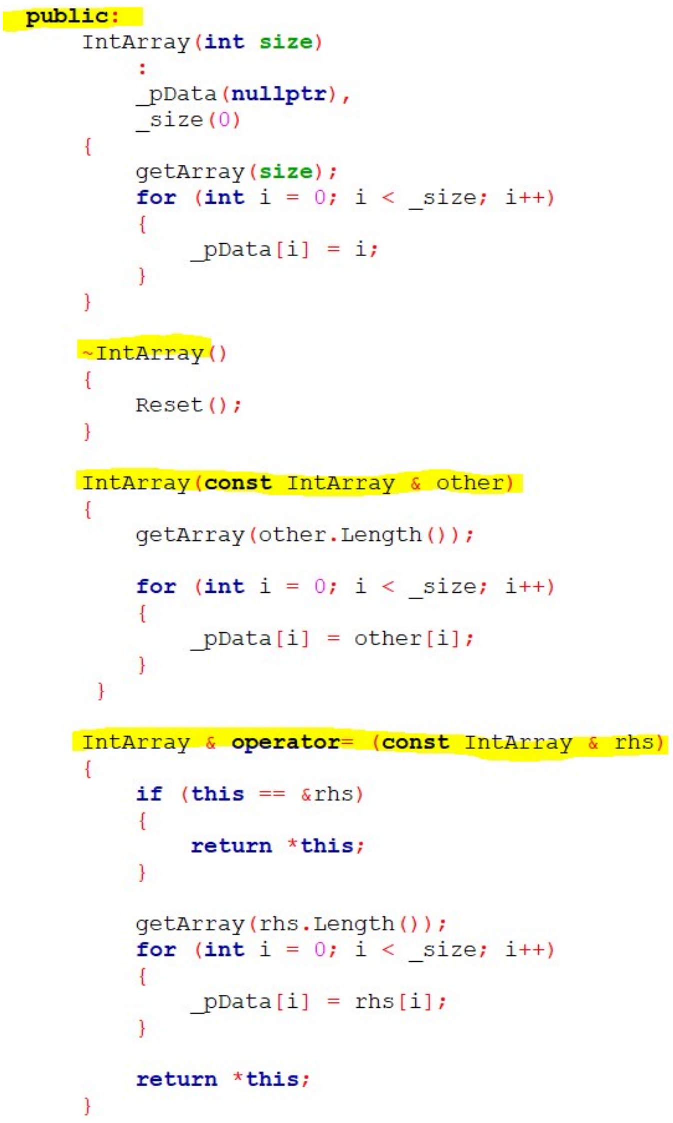
- 私有数据和方法



```
class IntArray
{
private:
    int * _pData;
    int _size;

    void getArray(int size)
    {
        // 避免分配负的内存引起的exception
        if (size < 0)
        {
            return;
        }

        // 重置内部数组，释放资源
        Reset();

        // 申请大小为size的数组内存
        if (size > 0)
        {
            _size = size;
            _pData = new (nothrow) int[size] {0};
        }

        // 如果内存分配失败，确保_size还是0
        if (_pData != nullptr)
        {
            _size = 0;
        }
    }

    void Reset()
    {
        if (_pData != nullptr)
        {
            delete [] _pData;
            _pData = nullptr;
            _size = 0;
        }
    }
```

IntArray

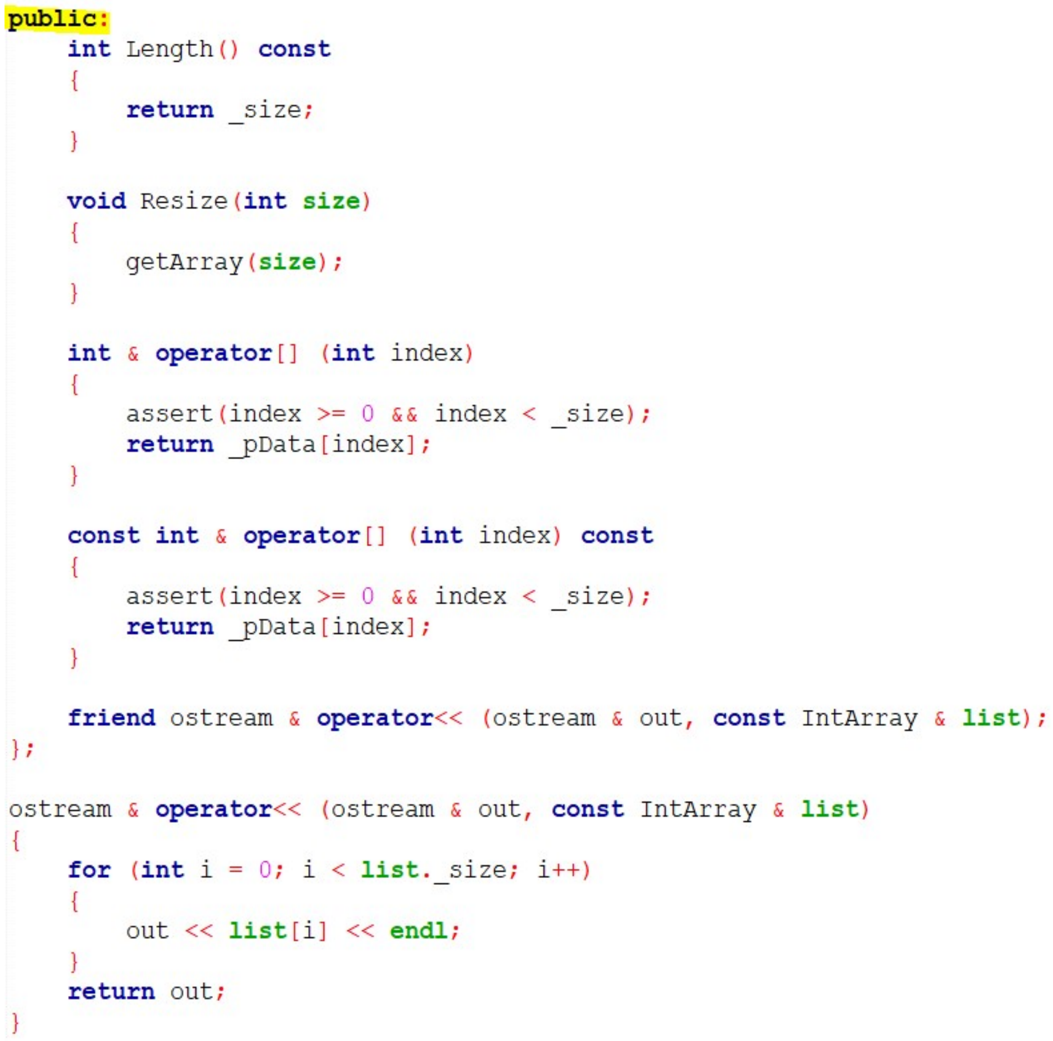
- 公开方法
- 尤其是析构函数，赋值构造函数，赋值重载操作符



```
public:
    IntArray(int size)
        :
        _pData(nullptr),
        _size(0)
    {
        getArray(size);
        for (int i = 0; i < _size; i++)
        {
            _pData[i] = i;
        }
    }

    ~IntArray()
    {
        Reset();
    }

    IntArray(const IntArray & other)
    {
        getArray(other.Length());

        for (int i = 0; i < _size; i++)
        {
            _pData[i] = other[i];
        }
     }

    IntArray & operator= (const IntArray & rhs)
    {
        if (this == &rhs)
        {
            return *this;
        }

        getArray(rhs.Length());
        for (int i = 0; i < _size; i++)
        {
            _pData[i] = rhs[i];
        }

        return *this;
    }
```

IntArray

- 其余公开方法



```
public:
    int Length() const
    {
        return _size;
    }

    void Resize(int size)
    {
        getArray(size);
    }

    int & operator[] (int index)
    {
        assert(index >= 0 && index < _size);
        return _pData[index];
    }

    const int & operator[] (int index) const
    {
        assert(index >= 0 && index < _size);
        return _pData[index];
    }

    friend ostream & operator<< (ostream & out, const IntArray & list);
};

ostream & operator<< (ostream & out, const IntArray & list)
{
    for (int i = 0; i < list._size; i++)
    {
        out << list[i] << endl;
    }
    return out;
}
```

Q&A

Thanks!