Operator overloading

ThanhNT

Introduction

```
int x = 2;
int y = 3;
```

x+y = ?

```
Mystring string1 = "Hello, ";
Mystring string2 = "World!“
```

string1+string2 = ?

In C++, operators are implemented as functions. By using function overloading on the operator functions, you can define your own versions of the operators that work with different data types (including classes that you’ve written).
Using function overloading to overload operators is called **operator overloading**.

Limitations

1. Almost any existing operator in C++ can be overloaded. The exceptions are: conditional (?:), sizeof, scope (::), member selector (.), and member pointer selector (.*).

2. You can only overload the operators that exist. You can not create new operators or rename existing operators. For example, you could not create an operator ** to do exponents.

3. At least one of the operands in an overloaded operator must be a user-defined type. This means you can not overload the plus operator to work with one integer and one double

4. It is not possible to change the number of operands an operator supports

5. All operators keep their default precedence and associativity

Overloading the arithmetic operators +,-,*,/ (1/3)

```
class MinMax
{
private:
    int m_min; // The min value seen so far
    int m_max; // The max value seen so far

public:
    MinMax(int min, int max)
    {
        m_min = min;
        m_max = max;
    }

    int getMin() { return m_min; }
    int getMax() { return m_max; }

    friend MinMax operator+(const MinMax &m1, const MinMax &m2);
    friend MinMax operator+(const MinMax &m, int value);
    friend MinMax operator+(int value, const MinMax &m);
};
```

Overloading the arithmetic operators +,-,*,/ (2/3)

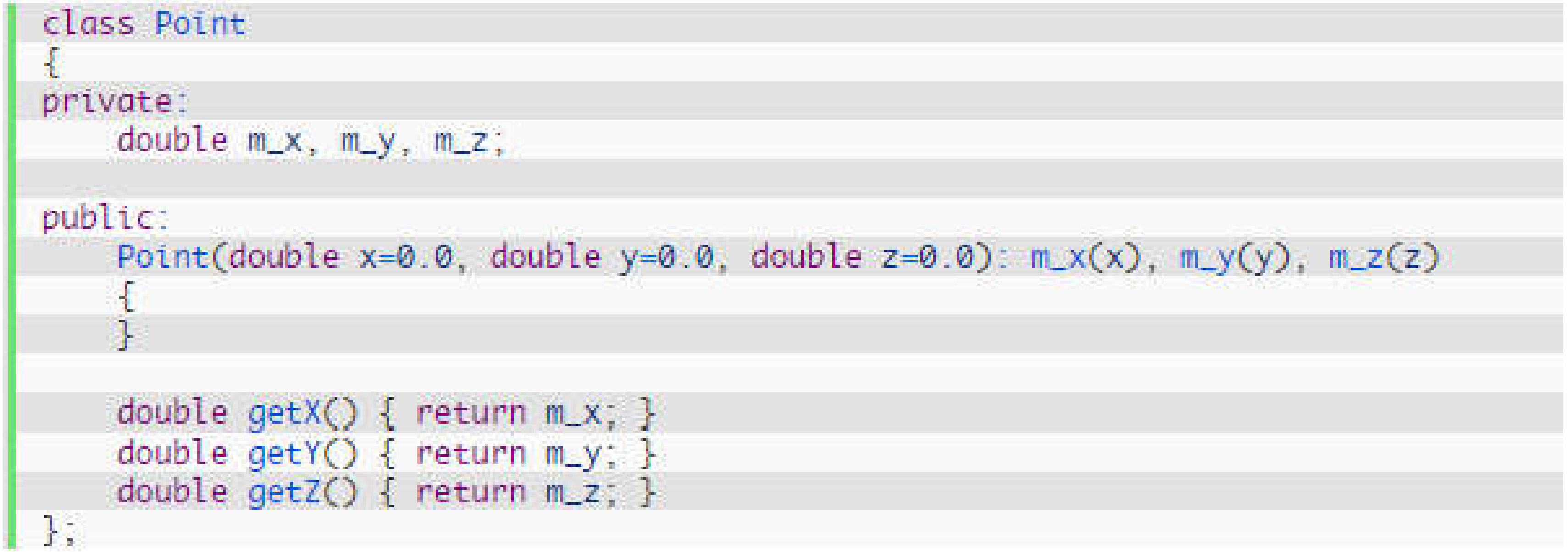
```
MinMax operator+(const MinMax &m1, const MinMax &m2)
{
    // Get the minimum value seen in m1 and m2
    int min = m1.m_min < m2.m_min ? m1.m_min : m2.m_min;
    // Get the maximum value seen in m1 and m2
    int max = m1.m_max > m2.m_max ? m1.m_max : m2.m_max;

    return MinMax(min, max);
}

MinMax operator+(const MinMax &m, int value)
{
    // Get the minimum value seen in m and value
    int min = m.m_min < value ? m.m_min : value;
    // Get the maximum value seen in m and value
    int max = m.m_max > value ? m.m_max : value;

    return MinMax(min, max);
}

MinMax operator+(int value, const MinMax &m)
{
    // call operator+(MinMax, nValue)
    return (m + value);
}
```

Overloading the arithmetic operators +,-,*,/ (3/3)

```
int main()
{
    MinMax m1(10, 15);
    MinMax m2(8, 11);
    MinMax m3(3, 12);

     MinMax mFinal = m1 + m2 + 5 + 8 + m3 + 16;

    std::cout << "Result: (" << mFinal.getMin() << ", " << mFinal.getMax() << ")
\n";

    return 0;
}
```

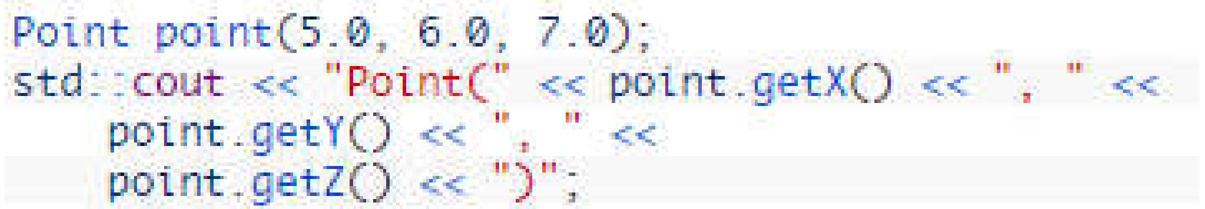
Result

Overloading the I/O operators <<

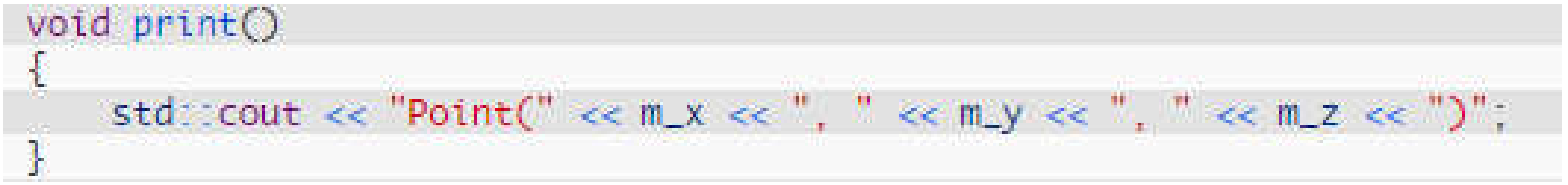


```
class Point
{
private:
    double m_x, m_y, m_z;

public:
    Point(double x=0.0, double y=0.0, double z=0.0): m_x(x), m_y(y), m_z(z)
    {
    }

    double getX() { return m_x; }
    double getY() { return m_y; }
    double getZ() { return m_z; }
};
```

How to print the position of a point???



```
Point point(5.0, 6.0, 7.0);
std::cout << "Point(" << point.getX() << ", " <<
    point.getY() << ", " <<
    point.getZ() << ")";
```



```
void print()
{
    std::cout << "Point(" << m_x << ", " << m_y << ", " << m_z << ")";
}
```

Overloading the I/O operators <<

```cpp
#include <iostream>

class Point
{
private:
    double m_x, m_y, m_z;

public:
    Point(double x=0.0, double y=0.0, double z=0.0): m_x(x), m_y(y), m_z(z)
    {
    }

    friend std::ostream& operator<< (std::ostream &out, const Point &point);
};
```

```cpp
std::ostream& operator<< (std::ostream &out, const Point &point)
{
    // Since operator<< is a friend of the Point class, we can access Point's members directly.
    out << "Point(" << point.m_x << ", " << point.m_y << ", " << point.m_z << ")";

    return out;
}
```

```cpp
int main()
{
    Point point1(2.0, 3.5, 4.0);
    Point point2(6.0, 7.5, 8.0);

    std::cout << point1 << " " << point2 << '\n';

    return 0;
}
```

```
Point(2, 3.0, 4)
```

```
Point(2, 3.5, 4) Point(6, 7.5, 8)
```

Overloading the I/O operators >>

```cpp
class Point
{
private:
    double m_x, m_y, m_z;

public:
    Point(double x=0.0, double y=0.0, double z=0.0): m_x(x), m_y(y), m_z(z)
    {
    }

    friend std::ostream& operator<< (std::ostream &out, const Point &point);
    friend std::istream& operator>> (std::istream &in, Point &point);
};
```

```cpp
std::istream& operator>> (std::istream &in, Point &point)
{
    // Since operator>> is a friend of the Point class, we can access Point's members directly.
    // note that parameter point must be non-const so we can modify the class members with the input values
    in >> point.m_x;
    in >> point.m_y;
    in >> point.m_z;

    return in;
}
```

```cpp
int main()
{
    std::cout << "Enter a point: \n";

    Point point;
    std::cin >> point;

    std::cout << "You entered: " << point << '\n';

    return 0;
}
```

Overloading the comparison operators ==, !=, >, <….

```
class Car
{
private:
    std::string m_make;
    std::string m_model;

public:
    Car(std::string make, std::string model)
        : m_make(make), m_model(model)
    {
    }

    friend bool operator== (const Car &c1, const Car &c2);
    friend bool operator!= (const Car &c1, const Car &c2);
};
```

```
bool operator== (const Car &c1, const Car &c2)
{
    return (c1.m_make== c2.m_make &&
            c1.m_model== c2.m_model);
}
```

```
bool operator!= (Car &c1, Car &c2)
{
    return !(c1== c2);
}
```

```
int main()
{
    Car corolla ("Toyota", "Corolla");
    Car camry ("Toyota", "Camry");

    if (corolla == camry)
        std::cout << "a Corolla and Camry are the same.\n";

    if (corolla != camry )
        std::cout << "a Corolla and Camry are not the same.\n";

    return 0;
}
```

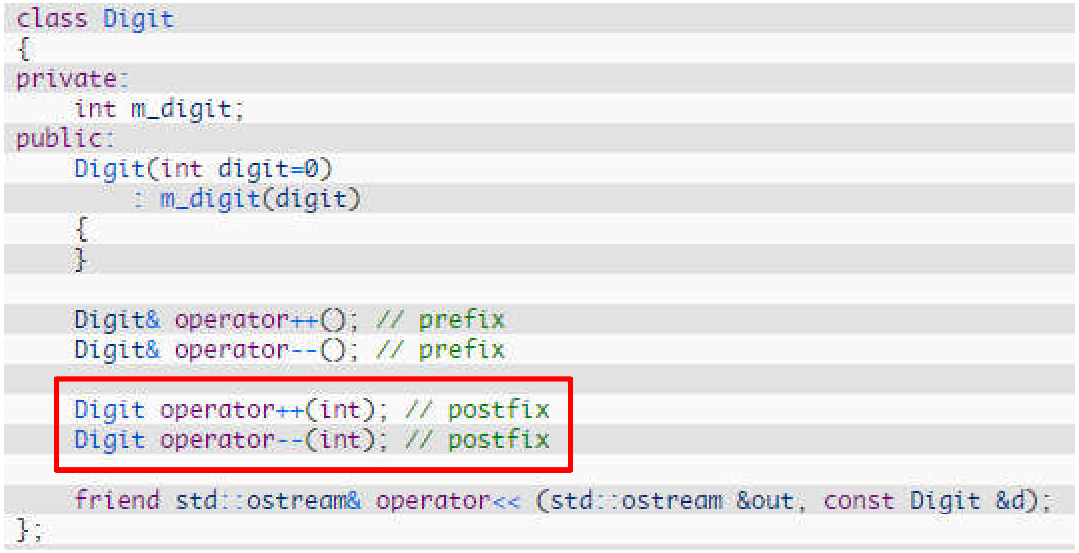
Overloading the increment and decrement operators - prefix

```
class Digit
{
private:
    int m_digit;
public:
    Digit(int digit=0)
        : m_digit(digit)
    {
    }

    Digit& operator++();
    Digit& operator--();

    friend std::ostream& operator<< (std::ostream &out, const Digit &d);
};
```

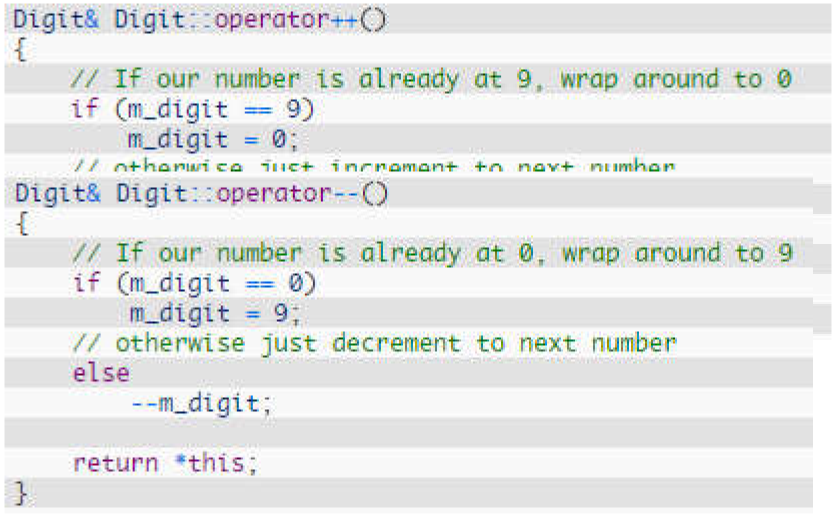
```
Digit& Digit::operator++()
{
    // If our number is already at 9, wrap around to 0
    if (m_digit == 9)
        m_digit = 0;
    // otherwise just increment to next number
    else
        ++m_digit;

    return *this;
}
```

```
Digit& Digit::operator--()
{
    // If our number is already at 0, wrap around to 9
    if (m_digit == 0)
        m_digit = 9;
    // otherwise just decrement to next number
    else
        --m_digit;

    return *this;
}
```

Overloading the increment and decrement operators - postfix

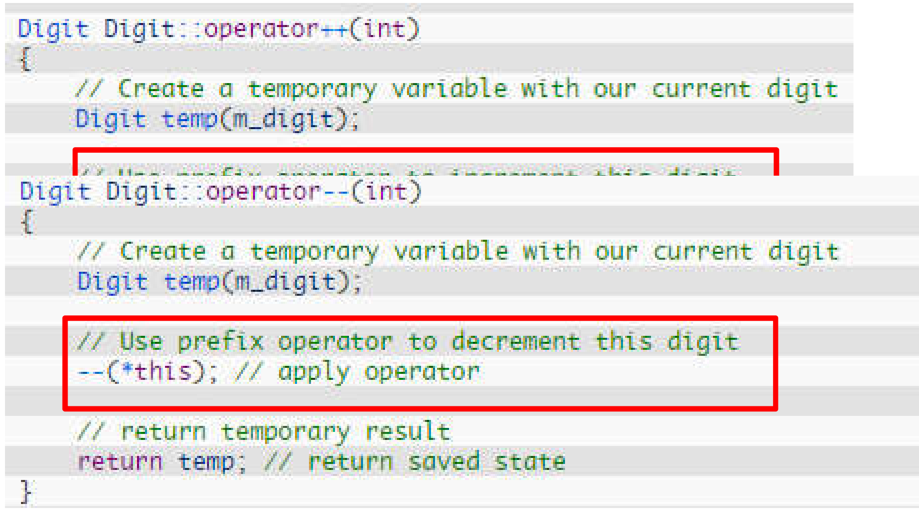


```
class Digit
{
private:
    int m_digit;
public:
    Digit(int digit=0)
        : m_digit(digit)
    {
    }

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

    Digit operator++(int); // postfix
    Digit operator--(int); // postfix

    friend std::ostream& operator<< (std::ostream &out, const Digit &d);
};
```



```
Digit& Digit::operator++()
{
    // If our number is already at 9, wrap around to 0
    if (m_digit == 9)
        m_digit = 0;
    // otherwise just increment to next number
```

```
Digit& Digit::operator--()
{
    // If our number is already at 0, wrap around to 9
    if (m_digit == 0)
        m_digit = 9;
    // otherwise just decrement to next number
    else
        --m_digit;

    return *this;
}
```



```
Digit Digit::operator++(int)
{
    // Create a temporary variable with our current digit
    Digit temp(m_digit);

    // Use prefix operator to increment this digit
```

```
Digit Digit::operator--(int)
{
    // Create a temporary variable with our current digit
    Digit temp(m_digit);

    // Use prefix operator to decrement this digit
    --(*this); // apply operator

    // return temporary result
    return temp; // return saved state
}
```

Overloading the subscript operator

```
class IntList
{
private:
    int m_anList[10];
};
```

How to set/get an item in array?

Solution 1: Create access function

```
public:
    void SetItem(int nIndex, int nData) { m_anList[nIndex] = nData; }
    int GetItem(int nIndex) { return m_anList[nIndex]; }
```

```
int main()
{
    IntList cMyList;
    cMyList.SetItem(2, 3);

    return 0;
}
```

Without seeing the definition of SetItem(), it's simply not clear

Solution 2: overload the subscript operator ([])

```
class IntList
{
private:
    int m_anList[10];

public:
    int& operator[] (const int nIndex);
};

int& IntList::operator[] (const int nIndex)
{
    return m_anList[nIndex];
}
```

```
IntList cMyList;
cMyList[2] = 3; // set a value
cout << cMyList[2]; // get a value
```

Overloading typecasts (1/2)

```
class Cents
{
private:
    int m_nCents;
public:
    Cents(int nCents=0)
    {
        m_nCents = nCents;
    }

    int GetCents() { return m_nCents; }
    void SetCents(int nCents) { m_nCents = nCents; }
};
```

```
int main()
{
    Cents cCents(7);
    PrintInt(cCents.GetCents()); // print 7

    return 0;
}
```

Code becomes messy

```
void PrintInt(int nValue)
{
    cout << nValue;
}
```

How to print the value of m_nCents?

Is it better?
How to implement?

```
int main()
{
    Cents cCents(7);
    PrintInt(cCents); // print 7

    return 0;
}
```

```
// Overloaded int cast
operator int() { return m_nCents; }
```

Note:

- To overload the function that casts our class to an int, we write a new function in our class called operator int(). Note that there is a space between the word operator and the type we are casting to
- Casting operators do not have a return type. C++ assumes you will be returning the correct type

Overloading typecasts (2/2)

1 dollar = 100 cents

```
class Dollars
{
private:
    int m_nDollars;
public:
    Dollars(int nDollars=0)
    {
        m_nDollars = nDollars;
    }
```

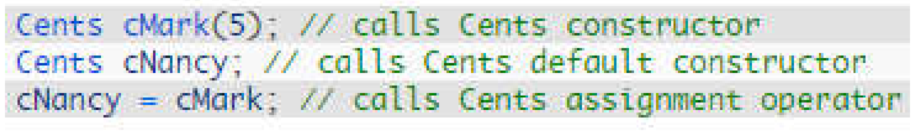
How to convert dollar to cent?

```
// Allow us to convert Dollars into Cents
operator Cents() { return Cents(m_nDollars * 100); }
```

```
void PrintCents(Cents cCents)
{
    cout << cCents.GetCents();
}

int main()
{
    Dollars cDollars(9);
    PrintCents(cDollars); // cDollars will be cast to a Cents

    return 0;
}
```

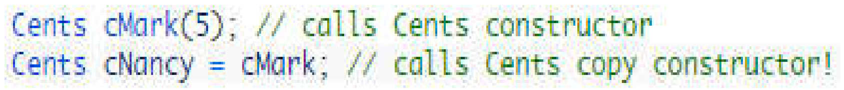
Assignment operator and copy constructor

The **assignment operator** is used to copy the values from one object to another ***already existing object***.



```
Cents cMark(5); // calls Cents constructor
Cents cNancy; // calls Cents default constructor
cNancy = cMark; // calls Cents assignment operator
```

A **copy constructor** is a special constructor that initializes a ***new object*** from an existing object.



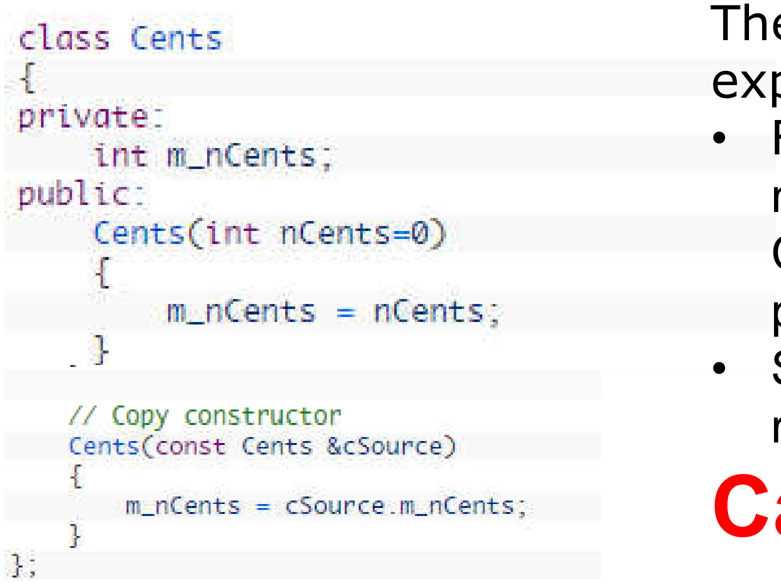
```
Cents cMark(5); // calls Cents constructor
Cents cNancy = cMark; // calls Cents copy constructor!
```

- If a new object has to be created before the copying can occur, the copy constructor is used
- If a new object does not have to be created before the copying can occur, the assignment operator is used

There are three general cases where the copy constructor is called instead of the assignment operator

- When instantiating one object and initializing it with values from another object
- When passing an object by value
- When an object is returned from a function by value

Copy constructor



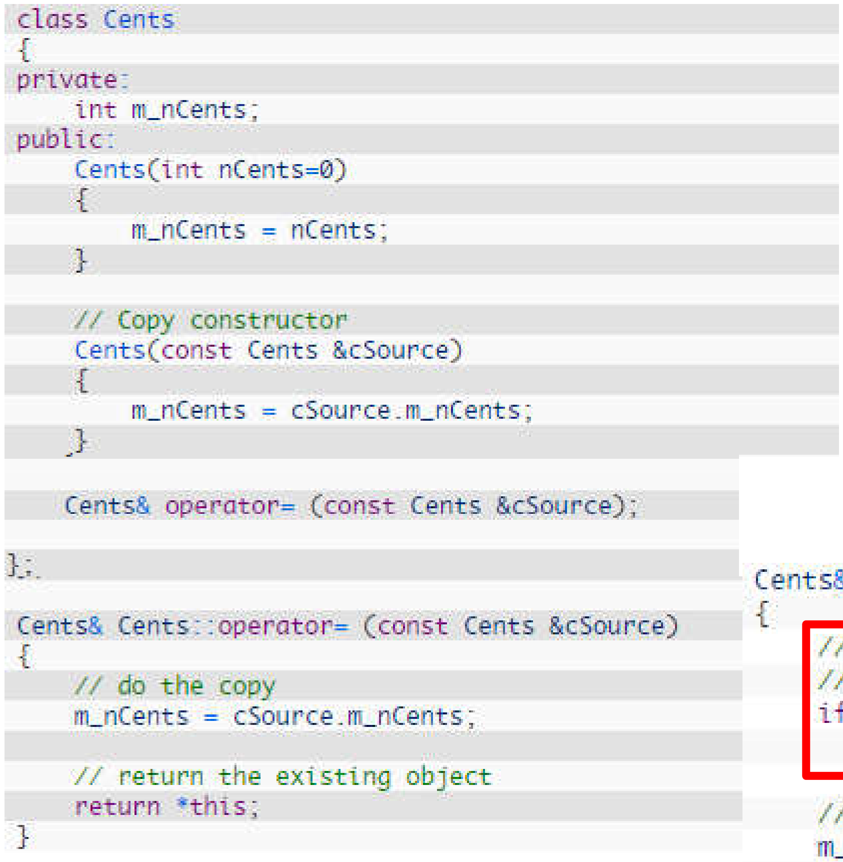
```
class Cents
{
private:
    int m_nCents;
public:
    Cents(int nCents=0)
    {
        m_nCents = nCents;
    }

    // Copy constructor
    Cents(const Cents &cSource)
    {
        m_nCents = cSource.m_nCents;
    }
};
```

There are two things which are worth explicitly mentioning

- First, because our copy constructor is a member of Cents, and our parameter is a Cents, we can directly access the internal private data of our parameter
- Second, the parameter MUST be passed by reference, and not by value

Can you figure out why?

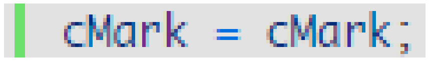
Overload the assignment operator



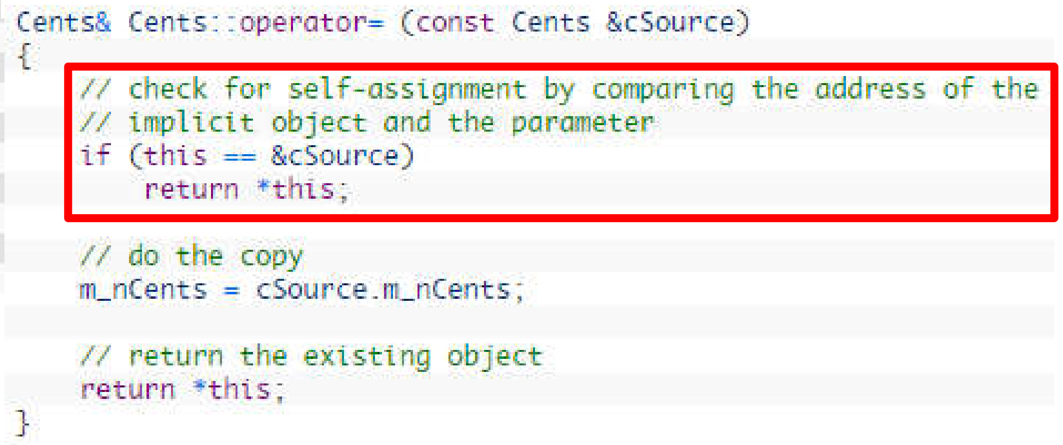
```
class Cents
{
private:
    int m_nCents;
public:
    Cents(int nCents=0)
    {
        m_nCents = nCents;
    }

    // Copy constructor
    Cents(const Cents &cSource)
    {
        m_nCents = cSource.m_nCents;
    }

    Cents& operator= (const Cents &cSource);

};

Cents& Cents::operator= (const Cents &cSource)
{
    // do the copy
    m_nCents = cSource.m_nCents;

    // return the existing object
    return *this;
}
```

2 notes:

- First, the line that does the copying is exactly identical to the one in the copy constructor
- Second, we're returning *this so we can chain multiple assignments together



```
cMark = cMark;
```



```
Cents& Cents::operator= (const Cents &cSource)
{
    // check for self-assignment by comparing the address of the
    // implicit object and the parameter
    if (this == &cSource)
        return *this;

    // do the copy
    m_nCents = cSource.m_nCents;

    // return the existing object
    return *this;
}
```

Q&A

Thanks for your attention!