Object-Oriented Programming

Lesson 8

Operator overloading

Operator overloading

Operator overloading is just "syntactic sugar," which means it is simply another way for you to make a function call.

Operator definition is just like an ordinary function definition except that the name of the function consists of the keyword operator followed by the operator.

There are certain operators in the available set that cannot be overloaded. The general reason for the restriction is safety.

There are no user-defined operators. That is, you can't make up new operators that aren't currently in the set. Part of the problem is how to determine precedence, and part of the problem is an insufficient need to account for the necessary trouble.

Operator overloading

```
class CDate
{
   unsigned m_Year;
   short int m_Month;
   short int m_Day;

public:
   CDate(); //initialize with the today’s date
   CDate(short int d, short int m, unsigned y);
   ~CDate();
};
```

```
int main()
{
     CDate today;
     CDate new_year = CDate(1, 1, 2015);

     return 0;
}
```

How to calculate the difference (in days) between two dates?

Operator overloading

```
class CDate
{
   // ...
public:
   //...

   int Difference (const CDate& d) const;
   friend int Difference(const CDate& d1, const CDate& d2);
};
```

```
int main()
{
     CDate today;
     CDate new_year = CDate(1, 1, 2015);

     int how_many = new_year.Difference(today);
     how_many = Difference(new_year, today);
     return 0;
}
```

Operator overloading

```
class CDate
{
   // ...
public:
   //...
   int operator- (const CDate& d) const;
};
```

```
int Difference (const CDate& d) const;
```

```
int main()
{
     CDate today;
     CDate new_year = CDate(1, 1, 2015);

     int how_many = new_year - today;

     return 0;
}
```

```
int how_many = new_year.Difference(today);
```

Operator overloading

To overload the operator + so as to be able to add two objects of type X:

```
class X                              //class definition
{      int a;
public:
       X(int aa) {a=aa;}
       // operator + overloading
       const X operator+(const X&)const;
};

const X X::operator+(const X& xr) const
{
       X tmp(this->a);
       tmp.a += xr.a;                   return X (a + xr.a);
       return tmp;
}
```

Operator overloading

The overloaded operator can be called in two different manners:

```
X x1(1), x2(2), x3(3);
x1 = x2 + x3;              //implicit call
x1 = x2.operator+(x3);     //explicit call
```

```
x1 = x1 + x2 + x3;
x1 = x1.operator+(x2).operator+(x3);
```

Unary and assignment operators are executed starting from the right-hand side of an expression.

The remaining operators are executed starting from the left-hand side.

Operator overloading

Not all the operators can be overloaded.

The operators that can be overloaded in C++ are the following:

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

The operators that cannot be overloaded in C++ are:

```
.*    ::    .     ?:    sizeof    typeid
```

Operator overloading

You cannot change the evaluation precedence of operators (only with parentheses).

A = B + C*D;

3 2 1

A = (B + C)*D;

3 1 2

You cannot change the number of arguments required by an operator. Unary operators require one operand and binary operators require two operands. The operators & , * , + , -, ~, ! exist in both versions: unary and binary, each of which can be overloaded separately.

A = B ++C;

You cannot create new operators.

A= B C;**

Operator overloading

Example:

Suppose that new operators would be allowed and we created the operator **<-** .

Then, in the following expression:

`a<--b`

the compiler does not know how to interpret the expression:

`a<-(-b)` or **`a<(--b)`**

Operator overloading

Only an expression containing a user-defined type can have an overloaded operator.

At least one of the operands must be either an object of a class or a reference to an objects of a class. It is not possible to overload an operator which works exclusively with pointers.

Example:

A programmer cannot overload the operator + to add two integers but it is possible to overload the operator + to add two objects of a user-defined class X.

Operator overloading

Overloading a certain operator does not imply authomatic overloading of the related operators, i.e. the operators can only be overloaded explicitly, never implicitly.

Example:

If the class X has two operators overloaded: + and = then it is possible to apply the following expression to the objects x1 and x2 of type X:

```
x1 = x1 + x2;          // Ok
```

In meantime, the operator += may not be applied:

```
x1 += x2;          // Error
```

To make the previous code compile, an explicit overloading of the operator += is required.

Operator overloading

A binary operator can be overloaded as either a member-function with one argument or a global function with two arguments.

```
class X
{   int a;
public:
    X(int aa) {a=aa;}
    const X operator+(const X&) const;
    friend const X operator+(const X& xl, const X& xr);
};
```

OR

```
const X X::operator+(const X& xr) const
{     return X(a + xr.a);      }
```

```
const X operator+ (const X& xl, const X& xr)
{    return X(xl.a + xr.a);    }
```

Operator overloading

A unary operator can be overloaded as either a member-function with no arguments or a global function with one argument.

```
class X
{   int a;
public:
    X(int aa) {a=aa;}
    const X operator-() const;          OR
    friend const X operator-(const X& xr);
};
```

```
const X X::operator- () const
{    return X(-a);        }
```

```
const X operator-(const X& xr)
{     return X(-xr.a);  }
```

Operator overloading

```
class X   {
//...
public:
    const X operator+(const X&)
     const;
//...

X x1(1) ,x2(2), x3(3);
// or x1 = x2.operator+(x3);
x1 = x2 + x3;
```

1st argument — 2nd argument

The 1st argument is passed implicitly through this pointer.

```
class X   {
//...
public:
    friend const X operator+
         (const X&, const X&);
//...

X x1(1) ,x2(2), x3(3);
// or x1 = operator+(x2, x3);
x1 = x2 + x3;
```

1st argument — 2nd argument

The 1st argument is passed explicitly.

Overloading assignment

```
class CVector
{  unsigned m_nElements;
   int* m_arElements;
public:
   CVector& operator=(const CVector& rv);
   CVector(unsigned el) ;
   virtual ~CVector();
};
```

```
int main ()
{
   CVector v1 (5);
   v1 = v1;

   return 0;
}
```

```
CVector& CVector::operator=(const CVector& rv)
{
   {   delete [] m_arElements;
       m_nElements = rv.m_nElements;
       m_arElements = new int[m_nElements];
       memcpy(m_arElements, rv.m_arElements,
              sizeof (int) * m_nElements);
   }
   return *this;
}
```

All of the assignment operators must include code to check for self-assignment !!!

Operator = vs. copy-constructor

```
CVector::CVector(const CVector& rv)
{
    m_nElements = rv.m_nElements;
    m_arElements = new int[m_nElements];
    memcpy(m_arElements, rv.m_arElements,
           sizeof (int) * m_nElements);
}
```

Copy-constructor

```
CVector v1(3);
CVector v2 = v1;
CVector v3;
v3 = v1;
```

```
CVector& CVector::operator=(const CVector& rv)
{  if (this != &rv)
   {    delete [] m_arElements;
        m_nElements = rv.m_nElements;
        m_arElements = new int[m_nElements];
        memcpy(m_arElements, rv.m_arElements,
               sizeof (int) * m_nElements);
   }
   return *this;
}
```

operator =

Operator = vs. copy-constructor

Example: identify lines where the copy-constructor and operator = are called

```
class CSetInt
{       //…
public:
         CSetInt& operator+= (const CSetInt& rv);
         const CSetInt operator+ (const CSetInt& rv) const
         { CSetInt aux(/*..*/); /*…*/ return aux; }
         CSetInt& operator= (const CSetInt& right);
        //…
};

CSetInt function (CSetInt& my_set)
{
        CSetInt my_set1 = my_set;      //1
        CSetInt my_set2 = my_set1;     //2
        my_set1 = my_set;              //3
        my_set2 += my_set;             //4
        my_set2 = my_set + my_set1;    //5
        return my_set2;                //6
}
```

Operator =

The operator = , when overloaded, must be declared as a class member!

It is not possible to overload the global operator = !

Do not forget to check for self-assignment.

Before starting to reserve memory for the object member, first release all the memory which was reserved before!

Because assigning an object to another object *of the same type* is an activity most people expect to be possible, the compiler will automatically create a type::operator=(type) if you don't make one.

The synthesized operator = will perform bitwise copy of data members (and will call recursively the operator = for all the subobjects).

⇒ The synthesized operator = is not suitable for complex objects (which reserve memory dynamically).

⇒ The operator = is not inherited by the derived classes.

For complex classes always define yourself the operator =, as well as the copy-constructor and the destructor.

Operator []

The overloaded operador **[ ]** must be a class member!

```
class CVector
{       unsigned m_nElements;
        int* m_arElements;
public:
        int& operator[](unsigned pos);

        // …
};
```

```
int main ()
{
   CVector v (5);
   v [3]= 3;
   int k = v[2];

   return 0;
}
```

```
int& CVector::operator[](unsigned pos)
{
        //check the pos
        return m_arElements[pos];
}
```

Arguments in operator overloading

The number of arguments in an overaloaded operator is:

0 – unary/member;
1 – unary/global or binary/member;
2 – binary/global.

As with any function argument, if you only need to read from the argument and not change it, default to passing it as a const reference:

```
... operator + (const type& r);
```

Only with the operator-assignments (like **+=**) and the **operator=**, which change the left-hand argument, is the left argument *not* a constant, but it's still passed in as an address because it will be changed.

Types of return value

The type of return value you should select depends on the expected meaning of the operator!

To allow the result of the assignment to be used in chained expressions, like **a=b=c**, it's expected that you will return a reference to that same lvalue that was just modified. The assignment operators usually return non-const references:

```
type& operator = (const type& r);
```

Logical operators usually return bool values:

```
bool operator >= (const type& r);
```

If the effect of the operator is to produce a new value, you will need to generate a new object as the return value. This object is returned by value as a const, so the result cannot be modified as an lvalue:

```
const type operator - (const type& r);
```

Member or non-member?

When the left-hand operand is a class object, the operator can be defined as a class member function.

When the left-hand operand is not a class object, the operator has to be defined as a global function.

```
friend ostream& operator << (ostream& os, const type& r);
```

```
X p(5);

cout << p << endl;
```

Input/output operators

```
friend std::ostream& operator << (std::ostream& os, const X& x);
friend std::istream& operator >> (std::istream& is, X& x);
```

```
ostream& operator << (ostream& os, const X& x)
{
        return os << x.a;
}


istream& operator >> (istream& is, X& x)
{
        return is >> x.a;
}
```

Member or non-member?

Operator	Recommended use
All unary operators	member
= () [] -> ->*	**must** be member
Assignment operators (excepting =): += –= /= *= ^= &= |= %= >>= <<=	member
All other binary operators	non-member

Bibliography

Bruce Eckel, Thinking in C++, 2nd edition, MindView, Inc., 2003

=> Chapter 12