C++ Programming

For BCA 4th Semester

Lecture 6

[Operator Overloading in C++ in Details]

Compiled
By
Subhadip Mukherjee
Dept. of Computer Science and BCA
Kharagpur College,
Kharagpur 721305

"Dear students, as we know Operator Overloading a very important topic of C++ programming, therefore, this lecture note is designed in such a way that describes Operator Overloading in C++ in a better way of understanding through Problems, Solutions and Coding"

____ Subhadip Mukherjee.

What's the deal with operator overloading?

It allows you to provide an intuitive interface to users of your class, plus makes it possible for templates to work equally well with classes and built-in/intrinsic types.
Operator overloading allows C/C++ operators to have user-defined meanings on user-defined types (classes). Overloaded operators are syntactic sugar for function calls:

```
class Fred {
public:
  // ...
};

#if 0

  // Without operator overloading:
  Fred add(const Fred& x, const Fred& y);
  Fred mul(const Fred& x, const Fred& y);

  Fred f(const Fred& a, const Fred& b, const Fred& c)
  {
    return add(add(mul(a,b), mul(b,c)), mul(c,a));    // Yuk...
  }

#else

  // With operator overloading:
  Fred operator+ (const Fred& x, const Fred& y);
  Fred operator* (const Fred& x, const Fred& y);

  Fred f(const Fred& a, const Fred& b, const Fred& c)
  {
    return a*b + b*c + c*a;
  }

#endif
```

What are the benefits of operator overloading?

By overloading standard operators on a class, you can exploit the intuition of the users of that class. This lets users program in the language of the problem domain rather than in the language of the machine.

The ultimate goal is to reduce both the learning curve and the defect rate.

What are some examples of operator overloading?

Here are a few of the many examples of operator overloading:

myString + yourString might concatenate two std::string objects
myDate++ might increment a Date object
a * b might multiply two Number objects
a[i] might access an element of an Array object
x = *p might dereference a "smart pointer" that "points" to a disk record — it could seek to the location on disk where p "points" and return the appropriate record into x

What operators can/cannot be overloaded?

Most can be overloaded. The only C operators that can't be are . and ?: (and sizeof, which is technically an operator). C++ adds a few of its own operators, most of which can be overloaded except :: and .*.

Here's an example of the subscript operator (it returns a reference). First with*out* operator overloading:

```
class Array {
public:
  int& elem(unsigned i)       { if (i > 99) error(); return data[i]; }
private:
  int data[100];
};

int main()
{
  Array a;
  a.elem(10) = 42;
  a.elem(12) += a.elem(13);
  // ...
}
```

Now the same logic is presented *with* operator overloading:

```
class Array {
public:
  int& operator[] (unsigned i) { if (i > 99) error(); return data[i]; }
private:
  int data[100];
};

int main()
{
```

```
Array a;
a[10] = 42;
    a[12] += a[13];
     // ...
    }
```

Why can't I Overload . (dot), ::, sizeof, etc.?

Most operators can be overloaded by a programmer. The exceptions are

. (dot) :: ?: sizeof

There is no fundamental reason to disallow overloading of ?:. So far the committee just hasn't seen the need to introduce the special case of overloading a ternary operator. Note that a function overloading expr1?expr2:expr3 would not be able to guarantee that only one of expr2 and expr3 was executed.

sizeof cannot be overloaded because built-in operations, such as incrementing a pointer into an array implicitly depends on it. Consider:

```
   X a[10];
   X* p = &a[3];
   X* q = &a[3];
   p++;    // p points to a[4]
      // thus the integer value of p must be
      // sizeof(X) larger than the integer value of q
```

Thus, sizeof(X) could not be given a new and different meaning by the programmer without violating basic language rules.

What about ::? In N::m neither N nor m are expressions with values; N and m are names known to the compiler and :: performs a (compile time) scope resolution rather than an expression evaluation. One could imagine allowing overloading of x::y where x is an object rather than a namespace or a class, but that would – contrary to first appearances – involve introducing new syntax (to allow expr::expr). It is not obvious what benefits such a complication would bring.

operator. (dot) could in principle be overloaded using the same technique as used for ->. However, doing so can lead to questions about whether an operation is meant for the object overloading . or an object referred to by .. For example:

```
class Y {
public:
   void f();
   // ...
};

class X {   // assume that you can overload .
   Y* p;
   Y& operator.() { return *p; }
   void f();
   // ...
};

void g(X& x)
{
   x.f();  // X::f or Y::f or error?
```

```
}
```

This problem can be solved in several ways. So far in standardization, it has not been obvious which way would be best.

Can I define my own operators?

Sorry, no. The possibility has been considered several times, but each time it was decided that the likely problems outweighed the likely benefits.

It's not a language-technical problem. Even when Stroustrup first considered it in 1983, he knew how it could be implemented. However, the experience has been that when we go beyond the most trivial examples people seem to have subtly different opinions of "the obvious" meaning of uses of an operator. A classical example is a**b**c. Assume that ** has been made to mean exponentiation. Now should a**b**c mean (a**b)**c or a**(b**c)? Experts have thought the answer was obvious and their friends agreed – and then found that they didn't agree on which resolution was the obvious one. Such problems seem prone to lead to subtle bugs.

Can I overload operator== so it lets me compare two char[] using a string comparison?

No: at least one operand of any overloaded operator must be of some user-defined type (most of the time that means a class).
But even if C++ allowed you to do this, which it doesn't, you wouldn't want to do it anyway since you really should be using a std::string-like class rather than an array of char in the first place since arrays are evil.

Can I create a operator** for "to-the-power-of" operations?

Nope.
The names of, precedence of, associativity of, and arity of operators is fixed by the language. There is no operator** in C++, so you cannot create one for a class type.

If you're in doubt, consider that x ** y is the same as x * (*y) (in other words, the compiler assumes y is a pointer). Besides, operator overloading is just syntactic sugar for function calls. Although this particular syntactic sugar can be very sweet, it doesn't add anything fundamental. I suggest you overload pow(base,exponent) (a double precision version is in <cmath>).

By the way, operator^ can work for to-the-power-of, except it has the wrong precedence and associativity.

How do I create a subscript operator for a Matrix class?

Use operator() rather than operator[].

When you have multiple subscripts, the cleanest way to do it is with operator() rather than with operator[]. The reason is that operator[] always takes exactly one parameter, but operator() can take any number of parameters (in the case of a rectangular matrix, two parameters are needed).

For example:

```
class Matrix {
public:
  Matrix(unsigned rows, unsigned cols);
  double& operator() (unsigned row, unsigned col);       // Subscript operators often
come in pairs
  double  operator() (unsigned row, unsigned col) const;  // Subscript operators often
come in pairs
  // ...
 ~Matrix();                              // Destructor
  Matrix(const Matrix& m);               // Copy constructor
  Matrix& operator= (const Matrix& m);   // Assignment operator
  // ...
private:
  unsigned rows_, cols_;
  double* data_;
};

inline
Matrix::Matrix(unsigned rows, unsigned cols)
  : rows_ (rows)
  , cols_ (cols)
//, data_ ← initialized below after the if...throw statement
{
  if (rows == 0 || cols == 0)
    throw BadIndex("Matrix constructor has 0 size");
  data_ = new double[rows * cols];
}

inline
Matrix::~Matrix()
{
  delete[] data_;
}

inline
double& Matrix::operator() (unsigned row, unsigned col)
{
  if (row >= rows_ || col >= cols_)
    throw BadIndex("Matrix subscript out of bounds");
  return data_[cols_*row + col];
}

inline
double Matrix::operator() (unsigned row, unsigned col) const
{
  if (row >= rows_ || col >= cols_)
    throw BadIndex("const Matrix subscript out of bounds");
```

```
  return data_[cols_*row + col];
}
```

Then you can access an element of Matrix m using m(i,j) rather than m[i][j]:

```
int main()
{
  Matrix m(10,10);
  m(5,8) = 106.15;
  std::cout << m(5,8);
  // ...
}
```

How can I overload the prefix and postfix forms of operators ++ and --?

Via a dummy parameter.

Since the prefix and postfix ++ operators can have two definitions, the C++ language gives us two different signatures. Both are called operator++(), but the prefix version takes no parameters and the postfix version takes a dummy int. (Although this discussion revolves around the ++ operator, the -- operator is completely symmetric, and all the rules and guidelines that apply to one also apply to the other.)

```
class Number {
public:
  Number& operator++ ();    // prefix ++
  Number  operator++ (int); // postfix ++
};
```

Note the different return types: the prefix version returns by reference, the postfix version by value. If that's not immediately obvious to you, it should be after you see the definitions (and after you remember that y = x++ and y = ++x set y to different things).

```
Number& Number::operator++ ()
{
  // ...
  return *this;
}

Number Number::operator++ (int)
{
  Number ans = *this;
  ++(*this);  // or just call operator++()
  return ans;
}
```

The other option for the postfix version is to return nothing:

```
class Number {
public:
  Number& operator++ ();
```

```
  void    operator++ (int);
};

Number& Number::operator++ ()
{
  // ...
  return *this;
}

void Number::operator++ (int)
{
  ++(*this);  // or just call operator++()
}
```

However you must *not* make the postfix version return the this object by reference; you have been warned.
Here's how you use these operators:

```
Number x = /* ... */;
++x;  // calls Number::operator++(), i.e., calls x.operator++()
x++;  // calls Number::operator++(int), i.e., calls x.operator++(0)
```

Assuming the return types are not 'void', you can use them in larger expressions:

```
Number x = /* ... */;
Number y = ++x;  // y will be the new value of x
Number z = x++;  // z will be the old value of x
```


Example 1: Overloading Unary Operator

```
#include <iostream>
using namespace std;
class Height {
public:
// Member Objects
int feet, inch;
// Constructor to initialize the object's value
Height(int f, int i)
{
feet = f;
inch = i;
}
// Overloading(-) operator to perform decrement
// operation of Height object
void operator-()
{
feet--;
inch--;
cout << "Feet & Inches after decrement: " << feet << " ' " << inch <<endl;
```

```
}
};
int main()
{
//Declare and Initialize the constructor of class Height
Height h1(6, 2);
//Use (-) unary operator by single operand
-h1;
return 0;
}
```

Example 2: Overloading Binary Operator

```
#include <iostream>
using namespace std;
class Height
{
public:
int feet, inch;
Height()
{
feet = 0;
inch = 0;
}
Height(int f, int i)
{
feet = f;
inch = i;
}
// Overloading (+) operator to perform addition of
// two distance object using binary operator
Height operator+(Height& d2) // Call by reference
{
// Create an object to return
Height h3;
// Perform addition of feet and inches
h3.feet = feet + d2.feet;
h3.inch = inch + d2.inch;
// Return the resulting object
return h3;
}
};
int main()
{
```

```
Height h1(3, 7);
Height h2(6, 1);
Height h3;
//Use overloaded operator
h3 = h1 + h2;
cout << "Sum of  Feet & Inches: " << h3.feet << "'" << h3.inch << endl;
return 0;
}
```

Rules for Operator Overloading

- Only the existing operators can be overloaded and new operators cannot be overloaded
- The overloaded operator must contain at least one operand of the user-defined data type.
- We do not use a friend function to overload certain operators. However, the member functions can be used to overload those operators.
- When unary operators are overloaded through a member function they take no explicit arguments, but, if they are overloaded by a friend function they take one argument.
- When binary operators are overloaded through a member function they take one explicit argument, and if they are overloaded through a friend function they take two explicit arguments.