Objective Viewpoint

The ABCs of Writing C++ Classes: Operators

G. Bowden Wise

In my last column [1], I explored some of the subtle intricacies of C++ that are often forgotten, even by expert C++ programmers. One of the best ways to remind ourselves of these subtleties is to develop our own set of guidelines, short, to the point reminders that you can glance at and quickly remember how to go about writing your code. Last time, I provided guidelines for the special member functions: constructors, destructors, and assignment operators. Special functions are important because every C++ class you write will have them. However, there are other guidelines which are also useful. In this issue we will focus on guidelines for operators.

Operators

The public interface of a class gives users the ability to create objects of that type and manipulate them through the public member functions. Operators are usually added after you have implemented the public interface and wish to provide a short-hand form for performing common operations. For example, an iterator class for a linked list might provide a member function to check whether the iterator is at the end of the list:

```
while (!myIter.AtEnd())
{
   // do something
}
```

Rather than have users remember the `AtEnd()` function, the iterator also uses the logical not operator to provide the same functionality, which might be implemented like this:

```
int operator! () { return !AtEnd(); }
```

Now users can write more compact code like this:

```
while (!myIter)
```

```
    {
       // do something
    }
```

C++ provides a number of operators, including: additive (`'+', '-'`); multiplicative (`'*', '/', '%'`); shift (`'<<', '>>'`); relational (`'>', '<', '<=', '>='`); equality (`'==', '!='`); bitwise (`'&', '|', '!', '^', '~'`); logical (`'&&', '||'`); assignment (`'=', '+=', '-=', '*=', '/=', '%=', '&=', '|=', '<<=', '>>=', '^=', '~='`); address-of (`'&'`); increment and decrement (`'++', '--'`); free store (`new, delete`); class member access (`'.'` and `'->'`); pointer-to-member (`'.*'` and `'->*'`); a conditional (`'?:'`); a scope (`'::'`); a function call (`'()'`); and a subscript (`'[]'`).

With the exception of a few operators (`'.', '.*', '::', '?:', sizeof`), only the pre-defined C++ operators may be defined in your classes. The compiler only provides an assignment (`'='`) and an address-of (`'&'`) operator for your class. If you want any of the others for your class, you must define them.

An operator is defined like an ordinary C++ member function except that its name consists of the keyword `operator` followed by one of the above fixed C++ operators that can be overloaded. This means you cannot introduce any new operators for the built-in types. For example, you cannot define `**` to be an operator for exponentiation. Overloaded operators can only be defined for class types. C++ prevents you from defining new operators by requiring all overloaded operators to have at least one class type argument.

We begin with some general guidelines to follow when overloading operators:

Guideline 1. *The pre-defined meaning of an operator for the built-in types may not be changed.*

This language restriction is to make C++ extensible but at the same time prevent unwanted abuse of operator overloading. For example, you cannot make the addition operator for integers mean subtraction.

When implementing your own classes, you can, of course, make operators behave as you like. But, as the next guideline suggests, you should try to be consistent with the intended meanings of the operators.

Guideline 2. *Try to preserve the semantics or intended meaning of the operator.*

In other words, the addition operator should always be used to represent addition or an operation that has similar semantics for the objects of your class. For example, the addition operator (`'+'`) often denotes denote concatenation for a `String` class.

Guideline 3. *The pre-defined ``arity'' of the operator must be preserved.*

The unary increment operator (`'++'`), for example, cannot be defined as the binary operator to append two linked list objects into one linked list. However, remember that the following four operators have both unary and binary forms: `'+', '-', '*', and '&'`. Either or both of these arities may be defined for these operators. See Guideline 10 for more information.

Guideline 4. *The pre-defined precedence of an operator may not be changed.*

Precedence rules govern which operators are invoked first by the compiler when there are multiple operators in a single expression. For example, multiplications are done before additions. These rules are fixed by the compiler and cannot be overridden. You may, of course, use parentheses to override precedence in expressions.

Guideline 5. *Make non-member operators `friend`s only when necessary.*

Below we will discuss when operators should be made member or non-member functions. Making a non-member function a `friend` of a class gives the function access to the private and protected members of the class. You do not have to make non-member operators `friend` automatically. For example, you may not need to make an operator a `friend` if you can access the data members the operator needs through the public interface of the class and they are inline so that the performance penalty for calling them is not too great.

Aside from these general guidelines, the only real determining factor when implementing implementing an operator is whether to implement it as a member or non-member function.

Member, Non-Member, or Friend

Like any other function for a class, operators may be implemented as a member or a non-member. In addition, for non-members the operator may be made a `friend` of the class. Some operators **must** be implemented as members. The other remaining operators may be implemented as we deem appropriate.

Guideline 6. *The assignment operator (`'='`), function call operator (`'()'`), subscript operator (`'[]'`), and the member selection operator (`'->'`)* ***must*** *be made member functions.*

There is no question about whether any of these should be member or friend, since all of them are required to be members.

Usually we determine whether to implement an operator as a member or a non-member depending on whether implicit type conversions should be allowed. Let's take a closer look at implicit type conversion.

Implicit Type Conversion

An *implicit type conversion* from type `S` to type `T` is declared by either:

- A `T` constructor that can take a single `S` argument, such as, `T::T(S s)` or `T::T(S s, int i=0)`.
- An `operator T` conversion function that is a member of `S`.

Consider a class for `Complex` numbers, which has an addition operator defined as a member:

```
class Complex
{
public:
 Complex(double re, double im = 0);
 Complex operator+(const Complex& rhs);
private:
 double re, im;
};
```

We would like our users to be able to write code like this:

```
Complex c1(1,1);
Complex c2 = c1 + 4.5;
Complex c3 = 10 + c1;
```

The first statement simply constructs the complex number `c1 = 1 + 1i`. The complex number `c2` is obtained by adding `4.5` to `c1`. Writing the assignmentin functional form yields:

```
c2.operator= ( c1.operator+ ( 4.5 ) );
```

The addition operator takes a `Complex` argument. So how does this work when a real number, `4.5`, is passed as the argument to `operator+`? It turns out that the compiler uses the constructor to implicitly convert `4.5` into a `Complex` by creating a temporary variable in order to perform the addition:

```
Complex c1 (1,1);
{
   Complex t1 (4.5);
   Complex c2 = c1 + t1;
}
```

The braces are used to show the scope of the temporary `t1` which only exists for the duration of the addition. In functional form this is can be written:

```
c2.operator= ( c1.operator+ ( t1 ) );
```

Next consider the assignment of `c3`. In functional form, the statement is written:

```
c3.operator= ((10.0).operator+ ( c1 ));
```

In order for this to work, `10.0` must be converted to a `Complex`. However, the compiler will not implicitly convert the left operand to call a member function on it.

If the addition operator is implemented as a non-member:

```
Complex operator+ (const Complex& a,
                   const Complex& b);
```

the compiler will implicitly convert both arguments as needed so that the assignment of `c3` will compile. The operator may or may not be a friend depending on how it is implemented.

Implicit conversions are also used for conversion of initializers, function arguments, function return values, expression operands, expressions controlling iteration and selection statements, and explicit type conversions.

Guideline 7. *Make operators non-members if implicit type conversions of left-operands must be allowed.*

Implicit conversions (for left operands) will be performed implicitly by the compiler when the operator is a non-member. If type conversions are undesirable, the operator should be implemented as a member function. Most binary operators, such as addition, should be implemented as non-members since implicit type conversions are desired.

Binary operators

A binary operator @, for `x @ y` is functionally evaluated by the compiler as:

- `x.operator@ ( y );`
  when @ is implemented as a member;
- `operator@ ( x, y );`
  when @ is implemented as a non-member.

Guideline 8. *Design binary operators non-member functions to allow implicit conversions.*

Recall that when an operator is implemented as a member function, implicit conversions are only performed on the right operand. Binary operators should be made non-members so that implicit conversions are performed on both left and right operands.

Unary operators

A unary operator `@`, for `@x` is functionally evaluated by the compiler as:

- `x.operator@ ();`
  when `@` is implemented as a member;
- `operator@ ( x );`
  when `@` is implemented as a non-member.

Guideline 9. *Make unary operators member functions to suppress implicit conversions if you can.*

Often, having implicit conversions can make your code difficult to understand because the conversions may occur without you being aware of them. When writing unary operators, implement them as members first. Only make them non-members if absolutely necessary.

Guideline 10. *The operators `'+', '-', '*',` and `'&'` have both unary and binary forms. Make sure that you are providing both forms if your class needs them.*

When you implement one of the forms of these operators, consider whether you need to provide the other form for your class. Whether you need both forms will certainly depend on the functionality of your class, however, do not forget to include both forms when needed.

Other assignment operators

Guideline 11. *The other forms of assignment (e.g., `'+=', '-=', '/=', '*=', '&=', '|=', '%=', '<<=', '>>=', '~=', and '^='`) should be made member functions just like the assignment operator.*

The assignment operator must be a member function. Although, the other forms of assignment are not required to be members, they should be made members so that they behave similarly to the normal assignment operator.

Increment and Decrement Operators

The built-in increment and decrement operators provide a convenient way to add and subtract `1` from a variable and are most often used in `for` loops. These operators are commonly used in user-defined classes where it makes sense to provide an increment and decrement capability, such as for, iterators.

Guideline 12. *When implementing increment or decrement operators for your class, remember to provide implementations for both the prefix and postfix forms of the operators.*

The increment and decrement operators, `'++'` and `'--'` respectively, can have both prefix and postfix forms. When we write either `i++` (postfix) or `--i` (prefix) the value of `i` is returned by the expression. However, in the postfix form, `i` is incremented after the value is returned, whereas in the prefix form, `i` is incremented before the value is returned.

C++ allows us to provide implementations for both of these forms so that the compiler can make appropriate calls to the functions. How do we distinguish between them when we implement our classes? We give the postfix version of the function a dummy `int` argument:

```
class Z
{
public:
   Z operator++ ();    // Prefix
   Z operator++ (int); // Postfix
};
```

When the compiler encounters the postfix version of an operator, it passes a dummy argument of `0` automatically. For example,

```
++z; // z.operator++()
z++; // z.operator++(0)
```

The Subscript Operator

The subscript operator (`` `[]' ``) deserves special mention due to its capability of being used on both sides of an assignment statement. We already know that it must be implemented as a member (see Guideline 6). Suppose we have implemented an array class for integers, called `IArray`. We can write:

```
IArray a;
a[1] = 4;
a[2] = a[1] * 3;
```

Note that the operator can appear on either side of the assignment operator. In order to appear on the left-hand side, the operator must return an lvalue (e.g., a reference).

Guideline 13. *When implementing the operator `[]` be sure to return a reference so that it may appear on the left-hand side of an assignment.*

References allow you to change whatever the reference refers to. For example,

```
int  i = 0;
int& ref = i;
ref = 2;
```

The assignment to the reference `ref` effectively changes the value of `i` from `0` to `2`. Similarly, whenever a function returns a reference, you may use that return value as an lvalue. To enable `operator[]` to be used on the left-hand side of an assignment, we would define it as follows:

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

You are not required to return a reference, however, if you return an `int` instead of an `int&` users of your class will not be able to use the operator on the left-hand side of an assignment.

I hope these guidelines will help you implement most of the operators for your classes. The free store operators (`new` and `delete`) and the user conversion operators are quite detailed and an entire column can easily be devoted to either one of them (perhaps, in a future issue, I will).

As you write your own programs, you will soon notice the beauties of programming in C++ (as well as some of the not-so beautiful constructs of the language!). When you come across a good rule of thumb, jot it down in a notebook. You will soon have your own set of guidelines.

References

1

WISE, G. B. The ABCs of Writing C++ Classes: Constructors, Destructors, and Assignment Operators. *Crossroads: The ACM Student Magazine 1*, 4 (May 1995).