Overloading Operators in C++

Stewart Weiss

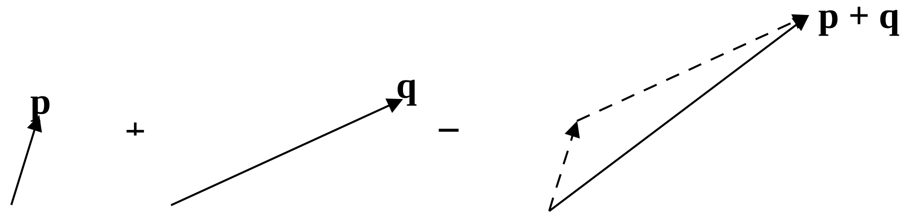
Defining the Overloaded Operators

C++ provides a standard set of operators for its built-in types, like +, /, and %. When you begin to learn programming, you learn that the "/" in x = 8 / 3 is the integer division operator. C++ overloads some of these operators itself. Thus, in "x = 8.0 / 4" the "/" is the floating point division operator, not the integer division operator. It behaves differently because the result is a floating point number, not an integer.

Sometimes, when you create a class to represent an application object, you often find yourself thinking that it would be nice if you also could extend the meanings of some of those standard operators so that they represented similar operators for that type of object. For example, it is useful to add points, fractions, vectors, or matrices using the same notation as we use for adding numbers. Wouldn't it be easy to understand the following bit of code if the operators had the natural meaning?

```
Point p,q;
cin >> p >> q;
cout << "The sum of " << p
     << " and " << q << " is " << p+q
     << endl;
```

If you know a bit about matrix algebra then you know that the sum of two points is the point that you get by thinking of each point as a vector with its source at the origin and its end at the point, and putting the source of one vector at the head of the other, as in the following picture.



The points p and q are represented by vectors from the origin (0,0). Their sum is the vector starting at the origin shown above.

Overloads do not provide any extra computational power for a class. They are nothing more than a notational convenience that makes it more natural for the client programmer to write code.

I will demonstrate how to use overloaded operators with a very simple class named `CPoint` that represents two-dimensional points. The `CPoint` class interface that follows demonstrates how the overloads can be achieved.

```
class CPoint {
private:
    int h;
    int v;
public:
    CPoint(int x = 0, int y = 0);
    CPoint(const CPoint& P );   //copy constructor
    CPoint& operator+= (const CPoint& R);
    CPoint  operator=  (const CPoint& C);
    double  operator() (const double D);

    friend ostream& operator<<(ostream& out,CPoint& P);
    friend CPoint operator+ (const CPoint& lhs, const CPoint& rhs);

};
```

In this interface, the following operators are overloaded:

```
     +=  =  ()  +  <<
```

Not all C++ operators can be overloaded. The ones that cannot be redefined by the user are:

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

Even the `new` and `delete` operators can be redefined by the user!

Implementing the Overloads

First notice that there are two ways in which operators are overloaded in the above example. One way is to make the overloaded operator a member of the class itself. The other way is to make it a non-member, but a *friend*, of the class.

If you make the operator a member of the class, it has an implicit argument that is the class object itself. To demonstrate,

```
class C {
    C operator+ (C);// + is a binary operator with *this as left operand
    C operator- (C,C); // error -- this makes operator- a ternary operation.
};
C operator*(C,C);  // * is a binary non-member operator
```

So in the above example, there are three member operators, +=, =, and (), and two non-members, + and <<, implemented as friends of the class.

Some of the operators require more explanation than others. I will explain how to implement them by example. To start, here is the first implementation.

```
CPoint CPoint::operator= (const CPoint& P)
//overload assignment operator
{
      this->h = P.h;
      this->v = P.v;
      return *this;
}
```

The assignment operator is required to return a value. In this case it returns its result by value rather than by reference. In essence, it is copying the members of the argument object P into the object on which it is called. If you write

```
      q = p;
```

it really means `q.operator=(p),` meaning, p is copied into q. The `this` pointer is a pointer to the object on which the call was made. There is no other way to return a copy of this object.

The next example shows that this could be done by reference as well.

```
CPoint&  CPoint:: operator+=(const CPoint& P)
{
      this->h += P.h;
      this->v += P.v;
      return *this;
}
```

It works just as well because the this pointer does not point to an object that will disappear within the lifetime of the call on which it is made. In other words, returning a reference to the object itself is safe in this case.

The function call operator, properly known as the application operator, (), must be a member of the class. It can have any number of arguments. In this example, it has just one:

```
double CPoint::operator() (const double D)
{
      double result;
      result = D*h + D+v;
      return result;
}
```

The unusual thing about this operator is that it can turn the object itself into something that looks like a function. Thus, in the client program, we can write

```
      const CPoint r(3,4);
      cout << "Value of r(4) is " << r(4) << endl;
```

Here, the expression, `r(4)` looks like a call to the function r with argument `4.` In fact it is a call to the `()` operator of the object `r`.

Using friend operators

An alternative is to use a non-member function to implement an operator. The non-member will have to be a friend if it needs access to protected or private data. In the above example the stream insertion operator and the addition operator are non-members. Their implementations follow:

```
ostream& operator<<(ostream& out, CPoint& P)
{
     out << "(" << P.h << "," << P.v << ")";
     return out;
}
```

Notice that the return value of the insertion operator is a reference to a stream. This is because when something is written to a stream, the stream itself is modified. The pointer to the next point of writing is advanced, among other things, so the stream must be passed as a reference result. It must also appear as an argument so that it can be written onto by the function. The function is a non-member, so it is passed any object of the given class as a parameter.

The addition operator is a simpler example of a friend. It is given two arguments and returns in this case an object of the same class. It declares a local variable that must be returned by value, not reference.

```
CPoint operator+ (const CPoint& lhs, const CPoint& rhs)
{
     CPoint result = rhs;
     result.h = lhs.h + rhs.h;
     result.v = lhs.v + rhs.v;
     return result;
}
```

Some Other Rules To Remember

There are a few things you must remember about overloading operators. Some are what you might expect, others perhaps not:

- You cannot change the number of operands of an operator that you are overloading. For example, if the standard operator has two operands, so must your overload.
- You can only overload operators that act on classes. In other words, at least one of the operands must be an instance of a class.
- You cannot change the precedence of an operator that you are overloading.
- You cannot create new operators by overloading other symbols.