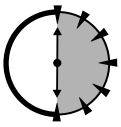
SESSION

Overloading Operators

Session Checklist

- ✔ Overview of overloading operators in C++
- ✔ Discussion of operator format versus function format
- ✔ Implementing operators as a member function versus as a nonmember function
- ✔ The return value from an overloaded operator
- ✔ A special case: the cast operator



30 Min. To Go

Sessions 6 and 7 discussed the mathematical and logical operators that C++ defines for the intrinsic data types.

The *intrinsic* data types are those that are built in the language, such as `int`, `float`, `double`, and so forth, plus the various pointer types.

In addition to these intrinsic operators, C++ enables the programmer to define the operators for classes that the programmer has created. This is called *operator overloading*.

Normally, operator overloading is optional and usually not attempted by beginning C++ programmers. A lot of experienced C++ programmers don't think operator

overloading is such a great idea either. However, there are three operators that you need to learn how to overload: assignment (=), left shift (<<), and right shift (>>). They are important enough that they have been granted their own chapters, which immediately follow this one.

Operator overloading can introduce errors that are very difficult to find. Be sure you know how operator overloading works before attempting to use it.

Why Do I Need to Overload Operators?

C++ considers user-defined types, such as `Student` and `Sofa`, to be just as valid as intrinsic types, such as `int` and `char`. Because operators are defined for the intrinsic types, why not allow them to be defined for user-defined types?

This is a weak argument, but I admit that operator overloading has its uses. Consider the class `USDollar`. Some of the operators make no sense at all when applied to dollars. For example, what would it mean to invert a `USDollar`? On the other hand, some operators definitely are applicable. For example, it makes sense to add a `USDollar` to or subtract a `USDollar` from a `USDollar`, with the result being a `USDollar`. It also makes sense to multiply or divide a `USDollar` by a `double`. However, multiplying a `USDollar` by a `USDollar` does not make much sense.

Operator overloading can improve readability. Consider the following, first without overloaded operators:

```
//expense - calculate the amount of money paid
//          (including both principal and simple interest)
USDollar expense(USDollar principal, double rate)
{
   //calculate the interest expense
USDollar interest = principal.interest(rate);

   //now add this to the principal and return the result
   return principal.add(interest);
}
```

With the proper overloaded operators, the same function looks like the following:

```
//expense - calculate the amount of money paid
//          (including both principal and simple interest)
```

```
USDollar expense(USDollar principal, double rate)
{
   USDollar interest = principal * rate;
   return principal + interest;
}
```

Before we investigate how to overload an operator, we need to understand the relationship between an operator and a function.

What Is the Relationship Between Operators and Functions?

An operator is nothing more than a built-in function with a peculiar syntax. For example, the operator + could just as well have been written `add()`.

C++ gives each operator a function-style name. The functional name of an operator is the operator symbol preceded by the keyword `operator` and followed by the appropriate argument types. For example, the + operator that adds an `int` to an `int` generating an `int` is called `int operator+(int, int)`.

The programmer can overload all operators, except `.`, `::`, `*` (dereference), and `&`, by overloading their functional name with these restrictions:

- The programmer cannot invent new operators. You cannot invent the operation `x $ y`.
- The format of the operators cannot be changed. Thus, you could not define an operation `%i` because `%` is a binary operator.
- The operator precedence cannot change. A program cannot force `operator+` to be evaluated before `operator*`.
- Finally, the operators cannot be redefined when applied to intrinsic types. Existing operators can only be overloaded for newly defined types.

How Does Operator Overloading Work?

Let's see operator overloading in action. Listing 27-1 shows a class `USDollar` with an addition operator and an increment operator defined.

Listing 27-1
USDollar with Overloaded Addition and Increment Operators

```
// USDollarAdd - demonstrate the definition and use
//               of the addition operator for the class
//               USDollar
#include <stdio.h>
#include <iostream.h>

// USDollar - represent the greenback
class USDollar
{
    // make sure that the user-defined operations have
    // access to the protected members of the class
    friend USDollar operator+(USDollar&, USDollar&);
    friend USDollar& operator++(USDollar&);

  public:
    // construct a dollar object with an initial
    // dollar and cent value
    USDollar(int d = 0, int c = 0);

    // rationalize - normalize the number of cents by
    //               adding a dollar for every 100 cents
    void rationalize()
    {
        dollars += (cents / 100);
        cents   %= 100;
    }

    // output- display the value of this object
    //         to the standard output object
    void output()
    {
        cout << "$"
             << dollars
             << "."
             << cents;
    }
```

```
  protected:
    int dollars;
    int cents;
};

USDollar::USDollar(int d, int c)
{
    // store of the initial values locally
    dollars = d;
    cents = c;

    rationalize();
}

//operator+ - add s1 to s2 and return the result
//            in a new object
USDollar operator+(USDollar& s1, USDollar& s2)
{
   int cents   = s1.cents   + s2.cents;
   int dollars = s1.dollars + s2.dollars;
   return USDollar(dollars, cents);
}

//operator++ - increment the specified argument;
//             change the value of the provided object
USDollar& operator++(USDollar& s)
{
    s.cents++;
    s.rationalize();
    return s;
}

int main(int nArgc, char* pszArgs[])
{
   USDollar d1(1, 60);
   USDollar d2(2, 50);
   USDollar d3(0, 0);
```

Continued

Listing 27-1 *Continued*

```
    // first demonstrate a binary operator
    d3 = d1 + d2;
             d1.output();
    cout << " + ";
             d2.output();
    cout << " = ";
             d3.output();
    cout << "\n";

    // now check out a unary operator
    ++d3;
    cout << "After incrementing it equals ";
             d3.output();
    cout << "\n";
    return 0;
}
```

The class `USDollar` is defined as having an integer number of dollars and an integer number of cents less than 100. Starting from the beginning of the class and working our way down, the functions `operator+()` and `operator++()` are declared in the `USDollar` to be friends of the class.

Remember that a class friend is a function that has access to the protected members of the class. Because `operator+()` and `operator++()` are implemented as conventional nonmember functions, they must be declared as friends to be granted access to the protected members.

The constructor for `USDollar` creates an object from an integer number of dollars and an integer number of cents, both of which may be defaulted. Once stored, the constructor calls the rationalize() function which normalizes the amount by adding the number of cents greater than 100 into the dollar amount. The `output()` function displays the `USDollar` to `cout`.

The `operator+()` has been defined with two arguments because addition is a binary operator (that is, it has two arguments). The `operator+()` starts by adding the dollar and cents values of both of its `USDollar` arguments. It then creates a new `USDollar` with these values and returns it to the caller.

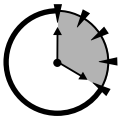
Any operation on the value of a USDollar object should call rationalize() to make sure that number of cents does not exceed 100. The operation+() function calls rationalize() in the USDollar constructor.

The increment operator `operator++()` has only one argument. This function increments the number of cents in object `s`. The function then returns a reference to the object it just incremented.

The `main()` function displays the sum of two dollar amounts. It then increments the result of the addition and displays that. The results of executing `USDollarAdd` are shown below.

```
$1.60 + $2.50 = $4.10
After incrementing it equals $4.11
```

In use, the operators appear very natural. What could be simpler than `d3 = d1 + d2` and `++d3`?



20 Min. To Go

Special considerations

There are a few special considerations to consider when overloading operators. First, C++ does not infer anything about the relationships between operators. Thus, `operator+=()` has nothing to do with either `operator+()` or `operator=()`.

In addition, nothing forces `operator+(USDollar&, USDollar&)` to perform addition. You could have `operator+()` do anything you like; however, doing anything else besides addition is a REALLY BAD IDEA. People are accustomed to their operators performing in certain ways. They don't like them performing other operations.

Originally, there was no way to overload the prefix operator `++x` separately from the postfix version `x++`. Enough programmers complained that the rule was made that `operator++(ClassName)` refers to the prefix operator and `operator++(ClassName, int)` refers to the postfix operator. A zero is always passed as the second argument. The same rule applies to `operator--()`.

If you provide only one `operator++()` or `operator--()`, it is used for both the prefix and postfix versions. The standard for C++ says that a compiler doesn't have to do this, but most compilers do.

A More Detailed Look

Why does `operator+()` return by value, but `operator++()` returns by reference? This is not an accident, but a very important difference.

The addition of two objects changes neither object. That is, `a + b` changes neither `a` nor `b`. Thus, `operator+()` must generate a temporary object in which it can store the result of the addition. This is why `operator+()` constructs an object and returns this object by value to the caller.

Specifically, the following would not work:

```
// this doesn't work
USDollar& operator+(USDollar& s1, USDollar& s2)
{
   s1.cents   += s2.cents;
   s1.dollars += s2.dollars;
   return s1;
}
```

because it modifies `s1`. Thus, after the addition `s1 + s2`, the value of `s1` would be different. In addition, the following does not work:

```
// this doesn't work either
USDollar& operator+(USDollar& s1, USDollar& s2)
{
   int cents = s1.cents + s2.cents;
   int dollars = s1.dollars + s2.dollars;
   USDollar result(dollars, cents);
   return result;
}
```

Although this compiles without complaint, it generates incorrect results. The problem is that the returned reference refers to an object, `result`, whose scope is local to the function. Thus, `result` is out of scope by the time it can be used by the calling function.

Why not allocate a block of memory from the heap, as follows?

```
// this sort of works
USDollar& operator+(USDollar& s1, USDollar& s2)
{
   int cents = s1.cents + s2.cents;
   int dollars = s1.dollars + s2.dollars;
   return *new USDollar(dollars, cents);
}
```

This would be fine except that there is no mechanism to return the allocated block of memory to the heap. Such memory leaks are hard to trace. Memory slowly drains from the heap each time an addition is performed.

Returning by value forces the compiler to generate a temporary object of its own on the caller's stack. The object generated in the function is then copied to the object as part of the return from `operator+()`. But how long does the temporary returned from `operator+()` hang around? A temporary must remain valid until the "extended expression" in which it appears is complete. The extended expression is everything up to the semicolon.

For example, consider the following snippet:

```
SomeClass f();
LotsAClass g();
void fn()
{
    int i;
    i = f() + (2 * g());

    // ...both the temporary object which f() returns
    // and the temporary which g() returns are invalid here...
}
```

The temporary object returned by `f()` remains in existence while `g()` is invoked and while the multiplication is performed. This object becomes invalid at the semicolon.

To return to our `USDollar` example, were the temporary not retained, the following would not work:

```
d1 = d2 + d3 + ++d4;
```

The temporary created by the addition of `d2` to `d3` must remain valid while `d4` is incremented or vice versa.

C++ does not specify the order in which operators are performed. Thus, you do not know whether d2 + d3 or ++d4 is performed first. You must write your functions so that it doesn't matter.

Unlike `operator+()`, the unary `operator++()` does modify its argument. Thus, there is no need to create a temporary or to return by value. The argument provided

can be returned to the caller by reference. In fact, the following function, which returns by value, has a subtle bug:

```
// this isn't 100% reliable either
USDollar operator++(USDollar& s)
{
   s.cents++;
   s.rationalize();
   return s;
}
```

By returning s by value, the function forces the compiler to generate a copy of the object. Most of the time, this is OK. But what happens in an admittedly unusual but legal expression such as ++(++a)? We would expect a to be incremented by 2. With the preceding definition, however, a is incremented by 1 and then a copy of a — not a itself — is incremented a second time.

The general rule is this: If the operator changes the value of its argument, accept the argument by reference so that the original may be modified and return the argument by reference in case the same object is used in a subsequent operation. If the operator does not change the value of either argument, create a new object to hold the results and return that object by value. The input arguments can always be referential to save time in a binary argument, but neither argument should be modified.

There are binary operators that change the value of their arguments such as the special operators +=, *=, and so forth.

Operators as Member Functions

An operator can be a member function in addition to being implemented as a nonmember function. Implemented in this way, our example USDollar class appears as shown in Listing 27-2. (Only the germane portions are shown.)

The complete version of the program shown in Listing 27-2 is contained in the file USDollarMemberAdd found on the accompanying CD-ROM.

Listing 27-2
Implementing an Operator as a Member Function

```
// USDollar - represent the greenback
class USDollar
{
  public:
    // construct a dollar object with an initial
    // dollar and cent value
    USDollar(int d = 0, int c = 0)
    {
        // store of the initial values locally
        dollars = d;
        cents = c;

        rationalize();
    }

    // rationalize - normalize the number of cents by
    //               adding a dollar for every 100 cents
    void rationalize()
    {
        dollars += (cents / 100);
        cents   %= 100;
    }

    // output- display the value of this object
    //         to the standard output object
    void output()
    {
        cout << "$"
             << dollars
             << "."
             << cents;
    }

    //operator+ - add the current object to s2 and
    //            return in a new object
```

Continued

Listing 27-2 *Continued*

```
    USDollar operator+(USDollar& s2)
    {
       int cents   = this->cents   + s2.cents;
       int dollars = this->dollars + s2.dollars;
       return USDollar(dollars, cents);
    }

    //operator++ - increment the current object
    USDollar& operator++()
    {
        cents++;
        rationalize();
        return *this;
    }

  protected:
    int dollars;
    int cents;
};
```

The nonmember function `operator+(USDollar, USDollar)` has been rewritten as the member function `USDollar::operator+(USDollar)`. At first glance, it appears that the member version has one fewer argument than the non-member version. If you think back, however, you'll remember that `this` is the hidden first argument to all member functions.

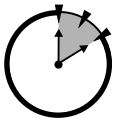
This difference is most obvious in `USDollar::operator+()` itself. Here I show the non-member and member versions in sequence.

```
// operator+ - the nonmember version
USDollar operator+(USDollar& s1, USDollar& s2)
{
    int cents   = s1.cents   + s2.cents;
    int dollars = s1.dollars + s2.dollars;
    USDollar t(dollars, cents);
    return t;
}
//operator+ - the member version
```

```
USDollar USDollar::operator+(USDollar& s2)
{
    int c = this->cents   + s2.cents;
    int d = this->dollars + s2.dollars;
    USDollar t(d, c);
    return t;
}
```

We can see that the functions are nearly identical. However, where the non-member version adds `s1` and `s2`, the member version adds the "current object" — the one pointed at by `this` — to `s2`.

The member version of an operator always has one less argument than the non-member version — the left-hand argument is implicit.



10 Min. To Go

Yet Another Overloading Irritation

Just because you have overloaded `operator*(double, USDollar&)`, that doesn't mean you have overloaded `operator*(USDollar&, double)`. Because these two operators have different arguments, they have to be overloaded separately. This doesn't have to be as big a deal as it would at first appear.

First, there is nothing that keeps one operator from referring to the other. In the case of `operator*()`, we would probably do something similar to the following:

```
USDollar operator*(USDollar& s, double f)
{
   // ...implementation of function here...
}
inline USDollar operator*(double f, USDollar& s)
{
   //use the previous definition
   return s * f;
}
```

The second version merely calls the first version with the order of the operators reversed. Making it inline even avoids any extra overhead.

When Should an Operator be a Member?

There isn't much difference between implementing an operator as a member or as a nonmember with these exceptions:

1. The following operators must be implemented as member functions:
   = Assignment
   () Function call
   [] Subscript
   -> Class membership
2. An operator such as the following could not be implemented as a member function.

```
// operator*(double, USDollar&) - define in terms of
//                        operator*(USDollar&, double)
USDollar operator*(double factor, USDollar& s)
{
    return s * factor;
}
```

   To be a member function, `operator(float, USDollar&)` needs to be a member of class `double`. As noted earlier, we cannot add operators to the intrinsic classes. Thus, an operator in which the class name is the right-hand argument must be a nonmember function.
3. Operators that modify the object on which they operate, such as `operator++()`, should be made a member of the class.

Cast Operator

The cast operator can be overloaded as well. The `USDollarCast` program in Listing 27-3 demonstrates the definition and use of a cast operator that converts a `USDollar` object to and from a `double` value.

Listing 27-3
Overloading the Cast Operator

```
// USDollarCast - demonstrate how to write a cast
//                operator; in this case, the
//                cast operator converts a USDollar
```

```
//                to a double and the constructor
//                converts it back
#include <stdio.h>
#include <iostream.h>

class USDollar
{
  public:
    // constructor to build a dollar from a double
    USDollar(double value = 0.0);

    //the following function acts as a cast operator
    operator double()
    {
        return dollars + cents / 100.0;
    }

    // display - simple debug member function
    void display(char* pszExp, double dV)
    {
        cout << pszExp
             << " = $" << dollars << "." << cents
             << " (" << dV << ")\n";
    }

  protected:
    int dollars;
    int cents;
};

// constructor - divide the double value into its
//               integer and fractional parts
USDollar::USDollar(double value)
{
    dollars = (int)value;
    cents = (int)((value - dollars) * 100 + 0.5);
}
```

Continued

Listing 27-3 *Continued*

```
int main()
{
    USDollar d1(2.0), d2(1.5), d3, d4;

    //invoke cast operator explicitly...
    double dVal1 = (double)d1;
    d1.display("d1", dVal1);

    double dVal2 = (double)d2;
    d2.display("d2", dVal2);

    d3 = USDollar((double)d1 + (double)d2);
    double dVal3 = (double)d3;
    d3.display("d3 (sum of d1 and d2 w/ casts)", dVal3);

    //...or implicitly
    d4 = d1 + d2;
    double dVal4 = (double)d3;
    d4.display("d4 (sum of d1 and d2 w/o casts", dVal4);

    return 0;
}
```

A cast operator is the word `operator` followed by the desired type. The member function `USDollar::operator double()` provides a mechanism for converting an object of class `USDollar` into a `double`. (For reasons that are beyond me, cast operators have no return type.) The `USDollar(double)` constructor provides the conversion path from a `double` back to a `USDollar` object.

As the preceding example shows, conversions using the cast operator can be invoked either explicitly or implicitly. Let's look at the implicit case carefully.

To make sense of the expression `d4 = d1 + d2` in the `USDollarCast` program, C++ goes through these gyrations:

1. First, C++ looks for a member function `USDollar::operator+(USDollar)`.
2. If that can't be found, it looks for the nonmember version of the same thing, `operator+(USDollar, USDollar)`.

3. Lacking that as well, it looks for an `operator+()` that it could use by converting one or the other arguments to a different type. Finally, it finds a match: If it converted both `d1` and `d2` to `double`, it could use the intrinsic `operator+(double, double)`. Of course, it then has to convert the resulting `double` back to `USDollar` using the constructor.

The output from `USDollarCast` appears below. The `USDollar::cast()` function enables the programmer to convert freely from `USDollar` objects to `double` values and back.

```
d1 = $2.0 (2)
d2 = $1.50 (1.5)
d3 (sum of d1 and d2 w/ casts) = $3.50 (3.5)
d4 (sum of d1 and d2 w/o casts = $3.50 (3.5)
```

This demonstrates both the advantage and disadvantage of providing a cast operator. Providing a conversion path from `USDollar` to `double` relieves programmers of the need to provide a complete set of operators. `USDollar` can just piggyback on the operators defined for `double`.

On the other hand, it also removes the ability of programmers to control which operators are defined. By providing a conversion path to `double`, `USDollar` gets all of `double`'s operators whether they make sense or not. For example, I could just as well have written `d4 = d1 * d2`. In addition, the extra conversions may not be the most efficient process in the world. For example, the simple addition just noted involves three type conversions with all of the attendant function calls, multiplications, divisions, and so on.

Be careful not to provide two conversion paths to the same type. For example, the following is asking for trouble:

```
class A
{
  public:
    A(B& b);
};
class B
{
  public:
    operator A();
};
```

Done!

If asked to convert an object of class B to an object of class A, the compiler will not know whether to use B's cast operator `B:operatorA()` or A's constructor `A::A(B&)`, both of which start out with a B and end up making an A out of it.

Perhaps the result of the two conversion paths would be the same, but the compiler doesn't know that. C++ must know which conversion path you intend it to use. The compiler spits out an error if it can't determine this unambiguously,

REVIEW

Overloading a new class with the proper operators can lead to some simple and elegant application code. In most cases, however, operator overloading is not necessary. The following sessions examine two cases in which operator overloading is critical.

- Operator overloading enables the programmer to redefine the existing operators for the programmer's classes. However, the programmer cannot add new operators to nor change the syntax of existing operators.
- There is a significant difference between passing and returning an object by value and by reference. Depending on the operator, this difference may be critical.
- Operators that modify the object should be implemented as member functions. In addition, certain operators must be member functions. Operators in which the left-hand argument is an intrinsic type and not a user-defined class cannot be implemented as member functions. Otherwise, it doesn't make a lot of difference.
- The cast operator enables the programmer to inform C++ how a user-defined class object can be converted into an intrinsic type. For example, casting a `Student` to an `int` might return the student ID number (I didn't say that this conversion was a good idea, only that it is possible.) Having done so, user classes may be mixed with intrinsics in expressions, for example.
- User-defined operators offer the programmer the opportunity to create programs that are easier to read and maintain; however, custom operators can be tricky and should be used carefully.

QUIZ YOURSELF

1. It is important that you be able to overload three operators for any given class. Which operators are they? (See the introduction.)
2. How could the following code ever make sense?

```
USDollar dollar(100, 0);
DM& mark = !dollar;
```

   (See "Why Do I Need to Overload Operators?")

3. Is there another way that I could have written the above without the use of programmer-defined operators, just using "normal" function calls. (See "Why Do I Need to Overload Operators?")