A Summary of Operator Overloading and Conversion Functions

David Kieras, EECS Dept., Univ. of Michigan
Prepared for EECS 381, Fall 2007

Basic idea

You overload an operator in C++ by defining a function for the operator. Every operator in the language has a corresponding function with a name that is based on the operator. You define a function of this name that has at least one parameter of your own class type, and returns a value of whatever type that you want. Because functions can have the same name if they have different signatures, the compiler will apply the correct operator function depending on the types in the call of it.

Rule #1. *You can't overload an operator that applies only to built-in types; at least one of the operator function parameter must be a "user defined" type*. A "user" here is you, the programmer; a "user defined type" is thus a class or struct type. Note that pointers are a built-in type, no matter what type of thing they point to. This means you can't overload operator+ to redefine what it means to add two integers. Another, and very important example: operator< for two pointers has a built-in meaning, namely to compare the two addresses in the pointers; you can't overload operator< to compare what two pointers point to.

You define operator functions differently depending on whether the operator function is a member of your own type's class, or is a non-member function. A further variation is whether the arguments are of the same or different types. We'll do this first for non-member functions, then for member functions.

In schematic form, when you use a binary operator (`op`), there is a left-hand-side operand (`lhs`) and a right-hand-side operand (`rhs`), and the whole expression has a value.

`lhs op rhs` --- has a value resulting from applying `op` to `lhs` and `rhs`

The operator function's first argument is the `lhs` operand, and the second is the `rhs` operand. The name of the operator function is `operator op`, where `op` is the operator, such as '+', '*', etc. The unary operators are almost all special cases, described later in this handout.

Non-member operator functions

A non-member operator overloading function simply has the operator function name and does whatever you want with the lhs and rhs parameters. For example, suppose we have a simple CoinMoney class that represents money that consists of collections of nickels, dimes, and quarters. A CoinMoney object stores the number of coins of each type, and computes the monetary value on request: A sketch of this class, leaving out members not immediately relevant, is as follows:

```
class CoinMoney {
public:
	CoinMoney(int n = 0, int d=0, int q= 0) : nickels(n), dimes(d), quarters(q) {}

	int get_value() const {return nickels * 5 + dimes * 10 + quarters * 25;}
private:
	int nickels;
	int dimes;
	int quarters;
};
```

Suppose we want to be able to add two CoinMoney objects and get a third CoinMoney object that has the sum of the number of nickles, dimes, and quarters from the two objects. We define the function named `operator+` that takes two arguments of CoinMoney type and returns a CoinMoney object with the correct values. Because the CoinMoney members are private, they will not be available to a non-member version of the function. We need to either add a friend declaration or a bunch of readers functions to the class. Let's assume we have the readers. The operator+ definition would then be:

```
CoinMoney operator+ (const CoinMoney& lhs, const CoinMoney& rhs)
{
	CoinMoney sum (
		(lhs.get_nickels() + rhs.get_nickels()),
		(lhs.get_dimes() + rhs.get_dimes()),
		(lhs.get_quarters() + rhs.get_quarters())
		)
	return sum;
}
```

This function creates a new object initialized with the nickels value from the lhs and rhs objects, and likewise for dimes and quarters, and returns the new object by value.
So now we can write sums of CoinMoney objects:

```
	m3 = m1 + m2;
```

The `m1 + m2` will be compiled into a call to our `operator+` function that takes two CoinMoney objects as arguments. It returns an object containing the sum, whose values then get copied into `m3`. Because `operator+` is just another function, the following two statements do exactly the same thing:

```
	m3 = m1 + m2;

	m3 = operator+ (m1, m2);
```

Explicitly calling operator functions is legal, and is sometimes done in special circumstances, but usually programmers don't bother - that's why the operator was overloaded!

Rule #2. *A non-member operator overload function is just an ordinary function that takes at least one argument of class type and whose name is of the form* operator op.

Left- and right-hand operands can be different types

At least one of the operator function parameters has to be of your type, but the other one can be of any other type, including a built-in type. For example:

```
CoinMoney operator* (CoinMoney lhs, double rhs)
{
	CoinMoney product(
            (lhs.get_nickels() * rhs),
            (lhs.get_dimes() * rhs),
            (lhs.get_quarters() * rhs)
            )
      return product;
}
```

This overload means that we can write:

```
      m1 * 2.0;
```

This gives us a CoinMoney object that has double the number of each type of coin as m1.

Note that if we want to be able to write:

```
      2.0 * m1;
```

We also have to define `CoinMoney operator* (double lhs, CoinMoney rhs)`. The signature is different!

Operator functions that are class member functions
If you write an operator function as a member function, then it automatically has access to all of the member variables and functions of the class. Friend declaration or readers not needed! But the complication is that the left-hand-side operand becomes the hidden "this" parameter of the function - it is "this" object, the current one being worked on. So the member operator function has only one argument, the right-hand-side operand. For example, the overload operator function for + becomes:

```
class CoinMoney
{
...
      // declare the operator overload as a const member function
      CoinMoney operator+ (const CoinMoney& rhs) const;
...
};

CoinMoney CoinMoney::operator+ (const CoinMoney& rhs) const
{
      CoinMoney sum(
            (nickels + rhs.nickels),
            (dimes + rhs.dimes),
            (quarters + rhs.quarters)
            );
      return sum;
}
```

The naked "nickels" is the member variable in the left-hand-side operand, "this" object. The function has only one parameter, the right-hand-side operand. Because this function is a member of the class, it has direct access to the member variables in "this" current object, and dot access to the member variables in the other objects in the same class.

Since a *member* operator function is just an ordinary *member* function, the following statements do the same thing:

```
      m3 = m1 + m2;

      m3 = m1.operator+ (m2);
```

The member version of an operator function is called to work on the left-hand operand, with the right-hand operand being the function argument. Again, calling operator functions explicitly is legal, but rare, and usually pointless.

Left-hand operand for member operator functions must be the class type
As with non-member operator overload functions, you don't have to have both arguments be the same type. However, by definition, the left-hand operand for a member operator function must be an object of the class that the function is a member of.

Rule #3. *An operator overload function written as a member function can only be applied if the lhs is of the class type.*

For example, suppose we wanted to be able to multiply CoinMoney objects by doubles as in the above example. We can define `operator*` as a member function only if a CoinMoney object is the left-hand operand, but not if a double is the left-hand operand:

That is, `CoinMoney operator* (double x)` can be defined as a member function of CoinMoney, and so

```
m1.operator* (2.5);
```

is a legal call. But there is no way to write operator* as a member function of CoinMoney so that you could write either one of
x = my_double_variable * m1;
`x = my_double_variable.operator* (m1);`

or if Thing is some other class type, the same applies:
`x = my_thing * m1;`

You can see why - if operator* is a member of CoinMoney, the lhs has to be a CoinMoney, not a double or a Thing. What do you do in such cases? Simple: define this version of the operator overload using a non-member function.

The member function handles

```
      m2 = m1 * my_double_variable;
```

and the non-member function handles

```
      m2 = my_double_variable * m1;
```

Which operators can and should be overloaded?

Almost all of the operators can be overloaded. But that doesn't mean you *should* overload them! Good OOP practice is to overload operators for a class only when they make obvious sense. For example, what would less-than mean for CoinMoney?

```
	m1 < m2
```

There are several reasonable ways that one collection of coins could be considered to be less than another - total number, total value, number of highest value coin, even total weight! You would define this operator only if there was only one reasonable interpretation in the problem domain you are working in. Finally, there is no clue what some operators might mean, such as:

```
	(m1 % m2++) | m3
```

Note that you don't have to define both "ways" for an overloaded operator. For example, maybe you want half the coins in a CoinMoney object:

```
	m1 / 2.0
```
This might make sense, but why would you want to divide by a CoinMoney object?

```
	2.0 / m1  // What does this mean???
```

You only have to define the versions of the overloads that make sense and that you want to be able to use.

However, common experience is that if it you find yourself writing the code to overload several operators for a class, you should consider overloading all of them that are meaningful - the additional overloads have a funny habit of being needed later!

How do I overload the output operator to output objects of my own type?

Just like any other operator, but you have to get certain things right. Here's the basic pattern to allow you to output CoinMoney objects like any other type, as in

```
cout << m1 << "Hello!" << my_int;  // etc

ostream& operator<< (ostream& os, CoinMoney x)
{
	os << /* whatever you want to output about x */;
	return os;
}
```

An important benefit: Not only will this work to output to `cout`, but also to file output streams! Here are the things you have to get right:

1. The `operator<<` function can't be a member of your own class, because the left-hand operand is an ostream object.

2. The first parameter has to be a reference-type parameter because you want `os` to be the very same stream object that the operator is applying to, and not a copy of it. Thanks to the reference, inside the function, os is an alias, another name, for the original `cout` object.

3. The return type has to be a reference to an ostream object, so that each application of << will produce the same `ostream` object that was originally on the left-hand side, so the next << will take it as its left-hand operand. This is why you can cascade the output operator.

4. You have to be sure to return the `ostream` parameter object so that the cascading in #3 will work.

5. In the body of this function, apply the ordinary output operator to the supplied stream parameter instead of `cout`. For example,

```
os << x.nickels;
```

Pretty Neat!

The `ostream` class in the Standard Library includes an overload of `operator<<` for every built-in type, as in:

```
ostream& operator<< (ostream& os, int x)
{/* output an integer */}
ostream& operator<< (ostream& os, double x)
{/* output a double */}
ostream& operator<< (ostream& os, char * x)
{/* output a C string */}
```

This is why output using the `iostream` library is type-safe - the compiler will make sure the right output function is called for the type of object you are outputting.

How do I overload the input operator to input objects of my own type?

Overloading the input operator is very similar to overloading the output operator, but it is less often done – usually, objects are created and have their values set using a constructor, rather than being first created and then having their member variables set from file or keyboard input.

The key issues in writing an overloaded input operator are to decide whether you are going to insist on a special format for the input, and how you are going to deal with erroneous input. For present purposes, we will ignore these problems and assume for the sake of example that we will read a CoinMoney object as three integers separated by whitespace. The basic pattern for overloading the input operator is illustrated with this example:

```
istream& operator>> (istream& is, CoinMoney& m)
{
      /* whatever you want to input and store in m */
      is >> m.nickels >> m.dimes >> m.quarters;
      return is;
}
```

Such an operator definition would allow you to write code like:

```
CoinMoney m1;
cin >> m1;  // read member variable values
...
```

Here are the things you have to get right:

1. The second parameter, whose type is your own class, also has to be a reference-type parameter, because you will be storing values in the caller's object. A call-by-value parameter would just be a copy of the caller's object; you would store values in the copy, and then it would get thrown away when the function returned. Oops! So this function has to modify the caller's object.

2. The `operator>>` function can't be a member of your own class, because the left-hand operand is an istream object.

3. If the `operator>>` function is going to set private member variables of your own class, it will need to either have friend status, or use public writer functions.

4. The first parameter, `is,` has to be a reference-type parameter because you want `is` to be the very same stream object that the operator is applying to, and not a copy of it. Inside the function, `is` is an alias, another name, for the original `cin` object.

5. The return type has to be a reference to an `istream` object, so that each application of >> will produce the same `istream` object that was originally on the left-hand side, so the next >> will take it as its left-hand operand. This is why you can cascade the input operator.

6. You have to be sure to return the `istream` parameter object so that the cascading in #5 will work.

The Standard Library also includes an overloaded input operator for every built-in type, again resulting in more type-safety. The use of the reference parameter for the destination argument makes it unnecessary to supply an address, as in C's `scanf`. In fact, getting neat and clean operator overloading is a major reason why reference parameters were included in C++.

What about unary operators?

Overloading unary operators works the same way as binary operators, except there is only one parameter to the operator function when it is a non-member, and no parameters when it is a member. For example, suppose we wanted the negation operator (!) to return true if a CoinMoney object was empty (all fields zero), and false otherwise. We just define a nonmember function:

```
bool operator! (const CoinMoney& m)
{
      return (
      m.get_nickels() == 0 &&
      m.get_dimes() == 0 &&
      m.get_quarters() == 0
      );
}
```

or a member function:

```
bool operator! ()
{
      return (
      nickels == 0 &&
      dimes == 0 &&
      quarters == 0
      );
}
```

The istream and ostream classes typically have a similar definition of operator! to implement the test of whether a stream is good by checking the stream object, as in:

```
if(!my_file) {/* oops! something is wrong! */}
```

What about operator++? Which one am I overloading?

The increment operator is commonly overloaded to mean "go to the next thing." But you can write "++" either in front of (prefix) or after (postfix) a variable. Clearly, just writing "operator++" is ambiguous. A *kludge* is used to distinguish the two:

• The prefix `operator++`, like all the other prefix operators, is just a unary operator (see above). If we defined it to do something for CoinMoney, its signature would be:

> `operator++ (CoinMoney&)` if a non-member,
> `operator++ ()` if a member.

• The postfix operator++ is the "odd" one. It has a dummy integer parameter which is never used, but the different signature serves to distinguish its operator function from the prefix operator function. So the postfix increment operator's signature would be

> `operator++ (CoinMoney&, int)` if a non-member,
> `operator++ (int)` if a member.

The same rule also applies to the decrement operator, "--". Note that an unused parameter does not need a variable name - in fact this is the idiom for saying that you have an unused parameter in a function, and usually results in a compiler not complaining about an unused parameter.

Conversion operators - letting the compiler convert types for you

In the CoinMoney example, we might want to treat a CoinMoney object as a single numeric value, say as a double, so we could compute the sales tax for a CoinMoney amount by:

```
tax = .06 * m1;
```

where we want to multiply .06 times the value of m1. If we tell the compiler how to turn a CoinMoney object into a double, then the compiler can just generate code for the multiply easily. We can do this by defining a conversion operator, which is a function whose name is `operator <type>` and whose return type is always the `<type>` and so is not stated. We would write (say as a member function):

```
operator double () const
{
	return get_value();
}
```

Conversion operators are notorious for surprising the programmer. For example, if we had both this conversion operator and the `operator* (double, CoinMoney)` we now have an ambiguity in how the compiler should analyze the above tax statement, producing an error message. So use these **very** sparingly. Usually a specialized get_ function is a better choice because it makes the intent more clear.

The `istream` and `ostream` classes typically contain a conversion operator to convert a `istream` or `ostream` object into a `bool`, so that you can write:

```
if(my_input_file) {/* everything is good! */}
```

You aren't testing for whether the humongous istream object is true or nonzero - it's a big object jammed full of data! But the function

```
operator bool (const istream& is) {whatever}
```

computes a true/false value based on the stream state bits, thereby allowing you to treat the stream object as a single true/false value.

The compiler can use a constructor to convert types - the explicit keyword

Sometimes a constructor function plays the role of a conversion function. For example, consider the following code sketch:

```
class Thing
{
blah blah
};

class Glob
{
	Glob(Thing t);  // construct a Glob from a Thing
blah blah
};

void foo(Glob g);  // function foo takes a Glob parameter

....
Thing my_thing;
...
foo(my_thing);  // call foo with a Thing?
...
```

You can't call foo with a Thing as an argument; foo requires a Glob. But the compiler cleverly notices that it can construct a Glob from a Thing, so it compiles the call as if the programmer had written:

```
foo (Glob(my_thing));
```

This is called an *implicit* conversion. Often this is exactly what you want. Sometimes it is a source of mysterious errors. The keyword `explicit` is used to prevent this use of a constructor, meaning that you want the constructor used only for the purpose of *explicitly* constructing one type from another. So if the Glob(Thing) constructor was defined as follows:

```
class Glob
{
	explicit Glob(Thing t);  // construct a Glob from a Thing
blah blah
};
```

then the `foo(my_thing)` call would result in a compilation error.

Two operators used in "Smart Pointers".

Smart Pointers are template class objects that can be used syntactically as if they were built-in pointers, but instead can do things like automatically manage memory. For them to behave syntactically like pointers, the two key pointer operators must be overloaded. These are operator* (unary) and operator-> (the "arrow" operator). Leaving out all the really interesting parts and just focussing on the overloaded operators, here are the relevant guts of a Smart Pointer class template. It stores a built-in pointer of the relevant type:

```
template typename<T>
class Smart_Pointer {
public:
	// don't worry for now about how the ptr member gets initialized

	// overloaded operators
	T& operator* () {return *ptr;}
	T* operator-> () const {return ptr;}
	// use the following with caution
	operator T*() const {return ptr;}		// conversion to pointer type
private:
	T* ptr;
};
```

If you apply the dereference operator, operator*, to a Smart_Pointer object, the operator function returns the dererenced internal pointer, making the Smart_Pointer object behave just like a built-in pointer in this regard. The arrow pointer, operator-> is more subtle. The definition of this operator is that when overloaded, it needs to return something that another operator-> can be validly applied to, which the compiler goes ahead and does. Thus this function simply returns the internal pointer, and the compiler reapplies operator-> to it, in this case, the built-in one for pointers. Again, the Smart_Pointer object behaves like a built-in pointer. Finally, the conversion operator, operator T*, allows the compiler to convert a Smart_Pointer to a built-in pointer, which can be convenient, but is also a source of errors, so some Smart_Pointer classes do not supply it.