Operator Overloading

Introduction

One of C++'s most powerful features is *operator overloading*, the ability to define how classes interact with operators applicable to primitive types. Operator overloading is ubiquitous in professional C++ code and, in fact, you've been exposed to operator overloading since the beginning of CS106X . Used correctly, operator overloading can make your programs more concise, more readable, and more template-friendly.

This handout discusses general topics in operator overloading, showcasing how to overload some of the more common operators. It also includes implementation tricks and general pitfalls to avoid when overloading operators. However, it is not a complete treatment of the material, so be sure to consult a reference for some more advanced topics in operator overloading.

A Word on Correctness

I would be remiss to discuss operator overloading without first prefacing it with a warning: operator overloading is a double-edged sword. When used correctly, operator overloading can lead to intuitive, template-friendly code that elegantly performs complex operations behind the scenes. However, incorrectly overloaded operators can lead to incredibly subtle bugs – just think of last week's lecture on the assignment operator. Now that we're delving deeper into operator overloading, you'll encounter more potential for these sorts of mistakes.

Operator overloading can be more dangerous than other parts of the C++ library because incorrectly-overloaded operators can lead to code that is "visually correct" (it just "looks right") but contains serious errors. Since you're redefining basic operators, the mistakes you'll make might show up in seemingly harmless lines of code like `myString += myOtherString` or `myIterator->value = 0`. Thus, when overloading operators, you should make sure to heavily test the overloaded operators in isolation before using them in general code. Additionally, operator overloading can be dangerous because of C++'s general philosophy of "leave it to the programmers to decide what's right." When overloading operators, you have the flexibility to define whatever parameter and returns types you want. For example, you can define the `*=` operator so that expressions like `myVector *= myScalar` is defined. However, this flexibility also means you can define nonsensical operations like `myVector = myMatrix / myString`. Thus, when overloading operators, you should be sure that your operations are *semantically meaningful* – they make sense to a human reader. Furthermore, it's easy to overload an operator but provide the wrong return type. For example, you might provided an overloaded increment operator such that the expression `myClass++` doesn't return a value. However, when applied to basic types, the increment operator *does* return a value, the value the variable had before the increment, and template code expecting the expression `myOtherClass = myClass++` to be meaningful will fail to compile if you forget this.

Ultimately, remember that you are trying to ascribe meaning to operators that by default only apply to primitive types. A good rule of thumb is to define your overloaded operators to have the same properties and returns types they have when applied to `int`s. Thus the expression `one == two` should return a `bool` indicating whether `one` and `two` are equivalent and should return the same result if we had written

`two == one`. In short, operator overloading is a wonderfully powerful tool that can simplify your code in remarkable ways. However, as with all parts of C++, operator overloading carries a good deal of risk and requires you to be more critical of your code.

General Operator Overloading Principles

There are two overarching purposes of operator overloading. First, operator overloading enables your custom classes to act like primitive types. That is, if you have a class like `vector` that mimics a standard C++ array, you'd like to be able to use array notation to access individual elements. Similarly, when designing a class encapsulating a mathematical entity (for example, a complex number), it would make sense for basic numerical operators like `+`, `-`, and `*` should work correctly. Second, operator overloading enables your code to interact correctly with template and library code. For example, you can overload the `<<` operator to make a class compatible with the IOStream library.

To define an overloaded operator, you declare a function whose name is `operator #`, where `#` represents whatever operator you're overloading. Thus the overloaded assignment operator is `operator =`, while the overloaded `<<` operator is `operator <<`. C++ overloads operators by replacing instances of each operator with calls to its `operator` function. So, for example, if you write code like this:

```
string one, two, three;
one = two + three;
```

It's equivalent to

```
string one, two, three;
one.operator =(operator+(two, three));
```

The second version, while syntactically legal, is extremely rare in practice since the point of operator overloading is to make the syntax more intuitive.

When overloading operators, you cannot define brand-new operators like `#` or `@`. After all, C++ wouldn't know the associativity or proper syntax for the function (e.g. is `one # two + three` interpreted as `(one # two) + three` or `one # (two + three)`?) Additionally, you cannot overload any of the following operators:

`::`	`MyClass::value`	Scope resolution
`.`	`one.value`	Member selection
`?:`	`a > b ? -1 : 1`	Ternary conditional
`.*`	(obscure)	Pointer-to-member selection
`sizeof`	`sizeof(MyClass)`	Size of object
`typeid`	`typeid(MyClass)`	Runtime type information operator (beyond the scope of this class)

Beyond these restrictions, however, you're free to overload any operators you see fit!

Overloading Bracket Operators

Let's begin our descent into the realm of operator overloading by discussing the overloaded bracket `[ ]` operator. You've been using the overloaded bracket operator ever since you encountered the `string` and `vector` classes. For example, the following code uses the `vector`'s overloaded brackets:

```
for(int i = 0; i < myVector.size(); i++)
    myVector[i] = 137;
```

In the above example, while it looks like we're treating the `vector` as a primitive array, we are instead calling the function `operator []` with `i` as the parameter. Thus the above code is equivalent to

```
for(int i = 0; i < myVector.size(); i++)
    myVector.operator [](i) = 137;
```

To write a custom bracket operator, you write a member function called `operator []` that accepts as its parameter the element that goes inside the brackets. Note that while this parameter can be of any type (think of the STL `map`), you can only have a single value inside the brackets. When writing `operator []`, as with all overloaded operators, you're free to return objects of whatever type you'd like. However, remember that when overloading operators, it's essential to maintain the same functionality you'd expect from the naturally-occurring uses of the bracket operator. Thus, in almost all cases, you should have the bracket operators return a reference to data that's stored in your class.

For example, here's one possible prototype of the C++ `string`'s bracket operator:

```
char& operator [] (int position);
```

Note that since the brackets return a reference to an element, you should almost always provide a `const`-overloaded version of the bracket operator. Thus the `string` also has a member function that looks something like this:

```
const char& operator [] (int position) const;
```

Another important detail to keep in mind is that when writing the bracket operator, it's totally legal to modify the underlying container in response to a request. For example, with the STL `map`, `operator []` will silently create a new object and return a reference to it if the object isn't already in the `map`. This is part of the beauty of overloaded operators – you're allowed to perform any necessary steps to ensure that the operator makes sense.

Unfortunately, if your class encapsulates a multidimensional object, such as a matrix or hierarchical key-value system, you cannot overload the "`[][]` operator." A class is only allowed to overload one level of the bracket syntax. This is why the CS106 `Grid` ADT doesn't use bracket syntax for access – it's not legal to design objects that doubly-overload `[]`.

Overloading Compound Assignment Operators

The compound assignment operators are operators of the form *op*= (for example, `+=` and `*=`) that update an object's value but do not overwrite it. The basic prototype for a compound assignment operator is

```
MyClass& operator += (ParameterType param)
```

When writing compound assignment operators, you still need to be wary of self-assignment, although in many cases you can ignore it. For example, when writing `Vector3D`, you could define the `+=` operator as

```
Vector3D& operator +=(const Vector3D &other)
{
    for(int i = 0; i < NUM_COORDINATES; i++)
       coordinates[i] += other.coordinates[i];
    return *this;
}
```

Here, since we don't deallocate any memory during the compound assignment, this code won't cause any problems. However, when working with the C++ `string`'s `+=` operator, since the `string` needs to allocate a new buffer capable of holding the current `string` appended to itself, it would need to handle the self-assignment case, either by explicitly checking for self-assignment or through some other means. We'll go over this code later in the handout.

Note that the compound assignment operators should return a non-`const` reference to the current object. Recall that when overloading operators, you should make sure to define your operators such that they work identically to the C++ built-in operators. Thus obscure but legal code should compile using your overloaded operators. Unfortunately, the code below, though the quintessence of abysmal style, is legal:

```
int one, two, three, four;
(one += two) += (three += four);
```

Note that the reference returned by `(one += two)` is then having its own `+=` operator invoked. Had the `+=` operator returned a `const` reference, this code would have been illegal. Thus, unfortunately, you must make sure to have any assignment operator return `*this` as a non-`const` reference.

Unlike the regular assignment operator, with the compound assignment operator it's commonly meaningful to accept objects of different types as parameters. For example, if you were to define a `Vector3D` class that encapsulated a mathematical vector in three-dimensional space, you might want to make expressions like `myVector *= 137` meaningful as a scaling operation. In this case, you'll simply define an `operator *=` that accepts an `int` as its parameter. As with `operator []` (and overloaded operators in general), you're free to specify whatever type you'd like as a parameter, though the operator must accept a single parameter.

Overloading Mathematical Operators

In the previous section, we provided overloaded versions of the `+=` family of operators. Thus, we can now write classes for which expressions of the form `one += two` are valid. However, the seemingly equivalent expression `one = one + two` will still not compile, since we haven't provided an implementation of the lone `+` operator.

When writing mathematical operators like `+` and `*`, it's important to have the return type of the function be a `const`, non-reference version of the class. That is, for a class `MyClass` overloading the `+` operator, the return type should be a `const MyClass`. This may not make much sense, so let's consider what would happen otherwise. Suppose that we declared the return type of the `+` operator non-`const`. Then we'd end up with a function returning a mutable `MyClass` object, so we could do something like this:

```
(one + two) = three;
```

The above expression isn't legal for any built-in type, so we should avoid making it legal for our custom-defined type. This is a great rule of thumb – when overloading operators, make sure that the behavior mimics that of the primitive types, especially `int`. The reason that the overloaded arithmetic operators shouldn't return references is a bit more complicated. Since you're creating a new object for the return value, the object has to either be on the stack or in the heap. If the object is on the stack, when the overloaded function returns, the newly-created object will be cleaned up and the reference will refer to invalid memory. If, however, the memory is on the heap, then you'd need to explicitly deallocate the memory for the temporary object after the arithmetic. Since neither of these cases lead to good code, you should not return references from the arithmetic operators.

Let's consider an implementation of `operator +` for a `CString` type. `CString` has the following definition:

```
class CString
{
    public:
        /* Constructor, destructor, copy constructor, assignment operator */

        CString& operator += (const CString &other);
    private:
        char *theString;
};
```

We can then define `operator +=` as

```
CString& CString::operator +=(const CString &other)
{
    /* Allocate space for the resulting string. */
    char *buffer = new char[strlen(theString) + strlen(other.theString) + 1];

    /* Concatenate the two strings. */
    strcpy(buffer, theString);
    strcat(buffer, other.theString);

    /* Free old memory. */
    delete [] theString;
    theString = buffer;
    return *this;
}
```

Note that while the above code doesn't explicitly check for self-assignment, it will work correctly since we're careful not to deallocate `theString` until after we've created the new string for our `CString`.

We'd like to write an implementation of `operator +`. We already know that we're supposed to return a `const CString`, and based on our knowledge of parameter passing, we know that we should accept as the parameter to `operator +` a `const CString &`. There's one more bit we're forgetting, though, and that's to mark the `operator +` function `const`, since `operator +` creates a new object and doesn't modify either of the values used in the arithmetic statement.

However, we might run into some trouble writing `operator +` since the code for concatenating two C strings is tricky. If you'll notice, though, we already have a working version of string concatenation in the body of `operator +=`. To unify our code, we'll therefore implement `operator +` by making a call to `operator +=`. The full version of this code is shown below:

```
const CString operator +(const CString &other) const
{
    CString result = *this; // Make a deep copy of this CString.
    result += other;    // Use existing concatenation code.
    return result;
}
```

Now, all of the code for `operator +` is unified, which helps cut down on coding errors.

There is an interesting and common case we haven't addressed yet – what if one of the operands isn't of the same type as the class? For example, if you have a `Matrix` class that encapsulates a 3x3 matrix, as shown here:

```
class Matrix
{
    public:
        /* Other member functions. */

        Matrix &operator *= (double scalar); // Scale all entries

    private:
        static const int MATRIX_SIZE = 3;
        double entries[MATRIX_SIZE][MATRIX_SIZE];
};
```

Note that there is a defined `*=` operator that scales all elements in the matrix by a `double` factor. Thus code like `myMatrix *= 2.71828` is well-defined. However, since there's no defined `operator *`, currently we cannot write `myMatrix = myMatrix * 2.71828`.

Initially, you might think that we could define `operator *` just as we did `operator +` in the previous example. While this will work in most cases, it will lead to some problems we'll need to address later. For now, however, let's add the member function `operator *` to `Matrix`, which is defined as

```
const Matrix operator *(double scalar) const
{
    MyMatrix result = *this;
    result *= scalar;
    return result;
}
```

Now, we can write expressions like `myMatrix = myMatrix * 2.71828`. However, what happens if we write code like `myMatrix = 2.71828 * myMatrix`? This is a semantically meaningful expression, but unfortunately it won't compile. When interpreting overloaded operators, C++ will always preserve the order of values in an expression. Thus `2.71828 * myMatrix` is *not* the same as `myMatrix * 2.71828`.* Remember that the reason that `myMatrix * 2.71828` is legal is because it's equivalent to `myMatrix.operator *(2.71828)`. The expression `2.71828 * myMatrix`, on the other hand, is illegal because C++ will try to expand it into `(2.71828).operator *(myMatrix)`, which makes no sense.

---

* One major reason for this is that sometimes the arithmetic operators won't be commutative. For example, given matrices **A** and **B**, **A * B** is not necessarily the same as **B * A**, and if C++ were to arbitrarily flip parameters it could result in some extremely difficult-to-track bugs.

Up to this point, we've only seen overloaded operators as member functions, usually because the operators act relative to some receiving object, but in C++ it's also legal to define overloaded operators as free functions. When defining an overloaded operator as a free function, you simply define a global function named `operator #` (where `#` is the operator in question) that accepts the proper number of parameters. When using this overloaded operator in code, it will expand to a call to the global function. For example, if there is a global `operator +` function, then the line `one + two` would expand into `operator +(one, two)`. This is exactly what we need to solve this problem. Let's make `operator *` a free function that accepts two parameters, a `double` and a `Matrix`, and returns a `const Matrix`. Thus code like `2.71828 * myMatrix` will expand into calls to `operator *(2.71828, myMatrix)`. The new version of `operator *` is defined below:

```
const Matrix operator * (double scalar, const Matrix &matrix)
{
    Matrix result = *matrix;
    matrix *= scalar;
    return result;
}
```

But here we run into the same problem as before if we write `myMatrix * 2.71828`, since we haven't defined a function accepting a `Matrix` as its first parameter and an `double` as its second. To fix this, we'll define a *second* free function `operator *` with the parameters reversed that's implemented as a call to the other version:*

```
const Matrix operator *(const Matrix &matrix, int scalar)
{
    return scalar * matrix;
}
```

You are free to define global operator overloads for most operators, provided that at least one of the parameters is not a primitive type. That is, you can't define `operator +` as a free function that takes two `int`s. Also, there are a few operators you cannot overload as free functions, such as the assignment operator; consult a reference for more information.

One important point to notice about overloading the mathematical operators versus the compound assignment operators is that it's considerably faster to use the compound assignment operators over the relational operators. Not only do the compound assignment operators work in-place (that is, they modify existing objects), but they also return references instead of full objects. From a performance standpoint, this means that given these three `string`s:

```
string one = "This ";
string two = "is a ";
string three = "string!";
```

Consider these two code snippets to concatenate all three strings:

```
/* Using += */
string myString = one;
myString += two;
myString += three;
```

---

* Since this function serves only as a wrapper to the other function, it's an excellent function to mark `inline`.

```
/* Using + */
string myString = one + two + three
```

Oddly, the second version of this code is considerably slower than the first because the + operator generates temporary objects. Remember that when writing `one + two + three`, it's equivalent to `operator +(one, operator +(two, three))`. Each call to `operator +` returns a new `string` formed by concatenating the parameters, so the code `one + two + three` creates a total of two temporary `string` objects (possibly fewer depending on the compiler). The first version, on the other hand, generates no temporary objects since the += operator works in-place. Thus while the first version is less sightly, it is significantly faster than the second version.

Overloading ++ and --

Overloading the increment and decrement operators can be a bit tricky because there are two versions of each function – the prefix and postfix ++ and -- operators. Recall that `x++` and `++x` are different operations – the first will evaluate to the value of `x`, then increment `x`, while the second will increment `x` and then evaluate to the updated value of `x`. You can see this below:

```
int x = 0
cout <<  x++ << endl; // Prints 0
cout << x << endl; // Prints 1

x = 0;
cout <<  ++x << endl; // Prints 1
cout << x << endl; // Prints 1
```

Although this distinction is subtle, it's tremendously important for efficiency reasons. In the postfix version of ++, since we have to return the value of variable was before it was incremented, we'll need to make a full copy of the old version and then return it. With the prefix ++, since we're returning the current value of the variable, we can simply return a reference to it. Thus postfix ++ is noticeably slower than the prefix version.

The next question we need to address is how we can legally use ++ and -- in regular code. Unfortunately, it can get a bit complicated. For example, the following code is totally legal:

```
int x = 0;
++++++++++++++x; // Increments x seven times.
```

This makes sense, since it's equivalent to

```
++(++(++(++(++(++(++x))))));
```

Thus the value we get back from the prefix ++ must not be `const`, or we couldn't do this chained incrementation. However, if we use the postfix version of ++, as seen here:

```
x++++++++++++++; // ERROR!
```

We get a compile-time error, since after the first `x++`, we're left with *the value* of `x` before the incrementation, rather than the variable `x` itself. Thus the value returned by the postfix ++ operator must return a `const` object.

Now, let's actually get into some code. Unfortunately, we can't just sit down and write `operator ++`, since it's unclear *which* `operator ++` we'd be overloading. To differentiate between the two versions, C++ uses a bit of a hack: when overloading the prefix version of `++` or `--`, you simply write `operator ++` as a function that takes no parameters. To overload the postfix version, you'll overload `operator ++`, but the overloaded operator will accept as a parameter the integer value 0. In code, these two declarations look like

```
MyClass& operator ++(); // Prefix
const MyClass operator ++(int dummy); // Postfix
```

We're allowed to implement `++` and `--` in any way we see fit. However, one of the more common tricks is to write the `++` implementation as a wrapped call to `operator +=`. Assuming you've provided this function, we can then write the prefix `operator ++` as

```
MyClass &operator ++()
{
    *this += 1;
    return *this;
}
```

And the postfix `operator ++` as

```
const MyClass operator ++(int dummy)
{
    MyClass oldValue = *this; // Store the current value of the object.
    *this += 1;
    return oldValue;
}
```

Overloading Relational Operators

Perhaps the most commonly overloaded operators (other than `operator =`) are the relational operators like < and ==. Unlike the assignment operator, by default C++ does not provide relational operators for your objects. This means that you must explicitly overload the == and related operators to use them in code. The prototype for the relational operators looks like this (written for <, but can be for any of the relational operators):

```
bool operator < (const MyClass &other) const;
```

You're free to choose any means for defining what it means for one object to be "less than" another. What's important is consistency. That is, if `one < two`, we should also have `one != two` and `!(one >= two)`. In fact, you may want to consider defining just the < operator and then implementing ==, <=, `!=`, >, and >= as wrapper calls.

Functors

Functors (also called *function objects* or *functionoids*) are one of the most powerful features of the C++ language, especially in conjunction with STL algorithms. To understand the motivation behind functors, let's return to STL algorithms and consider a simple task that currently we cannot represent using algorithms.

Let's suppose you have a `vector<string>` containing random strings and you'd like to count the number of elements in the `vector` that have length less than five. Knowing about the `count_if` STL algorithm, you write the following function:

```
bool LengthIsLessThanFive(const string &str)
{
    return str.length() < 5;
}
```

And then call `count_if(myVector.begin(), myVector.end(), LengthIsLessThanFive)` to get the number of short elements. Similarly, if you wanted to count the number of strings with length less than ten, you could write a `LengthIsLessThanTen` function, and so on and so forth. While this approach will work in some cases, it is critically hampered because the maximum length must be determined at compile-time. For example, suppose you want to write a program that prompts the user for a number and then returns the number of strings in the `vector` with fewer than that many characters. Since the user could enter any integer value, you can't use the above approach since you wouldn't know which function to call.

To solve this problem, you might want to try writing a function like this:

```
bool LengthIsLessThan(const string &str, int length)
{
    return str.length() < length;
}
```

While this is indeed far more generic than the above approach, it won't work in conjunction with `count_if` since `count_if` requires a unary function (a function taking only one argument) as a parameter. Somehow you need to build a unary function that stores the value to compare the string length against. While you can do this with global variables, that approach is horrendously unsightly and can lead to all sorts of other problems. Instead, we'll use a *functor*. At a high level, a functor is a class that can be "called" as though it were a function. For example, assuming `MyClass` is a functor, we could write code like this:

```
MyClass myFunctor;
myFunctor(/* insert parameters here. */);
```

While this syntax might look terribly confusing (and admittedly it is a bit bizarre), it's legal C++ code and means "call the function that `myFunctor` provides."

In C++, to create a functor, you create a class that overloads the parentheses operator, `operator ( )`. Unlike other operators we've seen so far, when overloading the parentheses operator, you're free to return an object of any type (or even `void`) and can accept any number of parameters. For example, here's a sample functor that overloads the parentheses operator to print out a string:

```
class MyFunctor
{
    public:
        void operator() (const string &str)
        {
            cout << str << endl;
        }
};
```

Note that there are two sets of parentheses there. The first group is for the function name – `operator ()` – and the second for the parameters to `operator ()`. To use this functor, we can write:

```
MyFunctor functor;
functor("Functor power!");
```

Which prints out "Functor power!" While right now functors might just seem like a curiosity, they are immensely powerful because they can preserve state between function calls. For example, consider the following functor class:

```
class StringAppender
{
    public:
        explicit StringAppender(const string &str) : toAppend(str) {}
        bool operator() (const string &str) const
        {
            cout << str << ' ' << toAppend
        }
    private:
        string toAppend;
};
```

Now, we can write code like this:

```
StringAppender functor("is awesome");
functor("C++");
```

This code will print out "C++ is awesome," since we passed in "C++" as a parameter and the functor appended its stored string "is awesome." Basically, we've written something that looks like a single-parameter function but that has access to extra information. Let's return to the above example with `count_if`. Somehow we need to provide a unary function that can return whether a string is less than an arbitrary length. To solve this problem, instead of writing a unary function, we'll create a unary functor whose constructor stores the maximum length and whose `operator ()` accepts a string and returns whether it's of the correct length. Here's one possible implementation:

```
class ShorterThan
{
    public:
        explicit ShorterThan(int maxLength) : length(maxLength) {}
        bool operator() (const string &str) const
        {
            return str.length() < length;
        }
    private:
        int length;
};
```

Note that while `operator ()` takes in only a single parameter, it has access to the length field that was set up by the constructor. This is exactly what we want – a unary function that somehow knows what value to compare the parameter to. To tie everything together, here's the code we'd use to count the number of strings in the `vector` that are shorter than the specified value:

```
ShorterThan st(length);
count_if(myVector.begin(), myVector.end(), st);
```

Functors are absolutely incredible when combined with STL algorithms for this very reason – they look and act like regular functions but have access to extra information. This is only the tip of the iceberg when it comes to functors, and there are some absolutely incredible things you can do with functors that have significant implications for the way you program using the STL. We'll address some of them next lecture when we talk about the C++ `<functional>` libraries.

Creating Temporary Objects

Commonly, when working with functors, you'll want to create a temporary class instance that exists only in the context of a function call. While right now you might be a bit confused about exactly why you'd ever want to do this, it should become clearer shortly.

In C++, you are allowed to create temporary objects for the duration of a single line by calling the object's constructor with any parameters you see fit. For example, the following code creates a temporary `vector<int>` and prints out its size:

```
cout << vector<int>().size() << endl;
```

Let's analyze exactly what's going on here. The code `vector<int>()` creates a temporary `vector<int>` object by calling the `vector<int>` constructor with no parameters. Therefore, the newly-created `vector` has no elements. We then call the temporary `vector<int>`'s `size` member function, which will return zero since the `vector` is empty. Once this line finishes executing, the `vector`'s destructor will invoke, cleaning up the new object.

While the above example is admittedly quite useless, it is important to know that it's legal to construct objects "on the fly" using this syntax because it frequently arises in professional code, especially when dealing with functors. Look back to the above code with `count_if`. If you'll notice, we're creating a new `ShorterThan` class using the parameter `length`, then feeding the object to `count_if`. After that line, odds are that we'll never use the `ShorterThan` object again. This is an excellent spot to use temporary objects, since we need a new `ShorterThan` for the function call but afterward don't plan on using it. Thus, we can convert this code:

```
ShorterThan st(length)
count_if(myVector.begin(), myVector.end(), st);
```

Into this code:

```
count_if(myVector.begin(), myVector.end(), ShorterThan(length));
```

Here `ShorterThan(length)` constructs a temporary `ShorterThan` functor with parameter `length`, then passes it to the `count_if` algorithm. Don't get tripped up by the syntax – `ShorterThan(length)` is *not* calling the `ShorterThan`'s `operator ()` function. Instead, it's invoking the `ShorterThan` constructor with the parameter `length`.

Storing Objects in STL `sets`

Up to this point we've avoided storing objects in `sets`. Now that we've covered operator overloading, though, you have the necessary knowledge to store objects in the STL `map` and `set` containers.

Internally, the STL `map` and `set` are layered on binary trees that use the relational operators to compare two elements. Due to some clever design decisions, STL containers and algorithms only require the < operator to compare two objects. Thus, to store a custom class inside a `set` or `map`, you simply need to overload the < operator and the STL will handle the rest. For example, here's some code to store a `Point struct` in a `set`:

```
struct Point
{
    int x, y;
};

bool operator < (const Point &one, const Point &two)
{
    if(one.x < two.x)
        return true;
    if(one.x > two.x)
        return false;
    return one.y < two.y;
}

set<Point> mySet; // Now works using the default < operator.
```

You can use a similar trick to store objects as keys in a `map`.

However, what if you want to store elements in a `map` or `set` but not using the default comparison operator? For example, consider a `set<char *>` of C strings. Normally, the < operator will compare two `char *`s by seeing if one references memory with a lower address than the other. This isn't at all the behavior we want. First, it would mean that the `set` would store its elements in a seemingly random order since the comparison is independent of the contents of the C strings. Second, if we tried to call `find` or `count` to determine membership in the `set`, since the `set` compares the *pointers* to the C strings, not the C strings themselves, `find` and `count` would return whether the given pointer, not the string pointed at by the pointer, was contained in the `set`.

We need to tell the `set` that it should not use the < operator to compare C strings, but we can't simply provide an alternative < operator and expect the `set` to use it. Instead, we'll define a functor class whose `operator ()` compares two C strings lexicographically and returns whether one string compares less than the other. Here's one possible implementation:

```
class CStringCompare
{
    public:
        bool operator() (const char *one, const char *two) const
        {
            return strcmp(one, two) < 0; // Use strcmp to do the comparison
        }
};
```

Then, to signal to the `set` that it should store elements using `CStringCompare` instead of the default `<` operator, we'll define the `set` as a `set<char *, CStringCompare>`. Note that we specify the comparison functor class as a template argument to the `set`. This means that `set<char *>` and `set<char *, CStringCompare>` are two different types, so you can only iterate over a `set<char *, CStringCompare>` with a `set<char *, CStringCompare>::iterator`. `typedef` will be your ally here. You can use a similar trick for the `map` by declaring a `map<KeyType, ElemType, CompareType>`, and can also use two-argument functors comparison callback functions for STL algorithms requiring a comparison function.

`friend`

Normally, when you mark a class's data members private, only instances of that class are allowed to access them. However, in some cases you might want to allow specific other classes or functions to modify private data. For example, if you were implementing the STL `map` and wanted to provide an iterator class to traverse it, you'd want that iterator to have access to the `map`'s underlying binary tree. There's a slight problem here, though. Although the iterator is an integral component of the `map`, like all other classes, the iterator cannot access private data.

How are we to resolve this problem? Your initial thought might be to make some public accessor methods that would let the iterator modify the object's internal data representation. Unfortunately, this won't work particularly well, since then *any* class would be allowed to use those functions, something that violates the principle of encapsulation. Instead, to solve this problem, we can use the C++ `friend` keyword to grant the iterator class access to the `map` or `set`'s internals. Inside the `map` declaration, we can write the following:

```
friend class iterator;
class iterator
{
    /* ... iterator implementation here ... */
};
```

Now, since `iterator` is a `friend` of `map`, it can read and modify the `map`'s private data members.

Just as we can grant other classes `friend` access to a class, we can give `friend` access to global functions. For example, if we had a free function `ModifyMyClass` that accepted a `MyClass` object as a reference parameter, we could let `ModifyMyClass` modify the internal data of `MyClass` if inside the `MyClass` declaration we added the line

```
friend void ModifyMyClass(MyClass &param);
```

When using `friend`, there are two key points to be aware of. First, the `friend` declaration must precede the actual implementation of the `friend` class or function. Since C++ compilers only make a single pass over the source file, if they haven't seen a `friend` declaration for a function or class, when the function or class tries to modify your object's internals, the compiler will generate an error. Second, note that while `friend` is quite useful in some circumstances, it can quickly lead to code that entirely defeats the purpose of encapsulation. Before you grant `friend` access to a piece of code, make sure that the code has a legitimate reason to be modifying your object. That is, don't declare a function `friend` simply because it's easier to write that way. Think of `friend` as a way of extending a class definition to include other pieces of code. The class, together with all its `friend` code, should comprise a logical unit that correctly encapsulates data.

When overloading any operator that's a free function, you might want to consider giving that function `friend` access to your class. That way, the functions can efficiently read your object's private data without having to go through getters and setters.

Unfortunately, `friend` does not interact particularly intuitively with template classes. Suppose we want to provide a `friend` function `PQueueFriend` for a template version of the CS106 `PQueue`. If `PQueueFriend` is declared like this:

```
template<typename T>
void PQueueFriend(const PQueue<T> &pq)
{
   // ...
}
```

You'll notice that `PQueueFriend` itself is a template function. This means that when declaring `PQueueFriend` a `friend` of the template `PQueue`, we'll need to make the `friend` declaration templatized, as shown here:

```
template<typename T>
class PQueue
{
    public:
        /* ... */
        template<typename T>
            friend PQueueFriend(const PQueue<T> &pq);
};
```

If you forget the `template` declaration, then your code will compile correctly but will generate a linker error. While this can be a bit of nuisance, it's important to remember since it arises frequently when overloading the stream operators, as you'll see below.

Overloading the Stream Insertion Operator

Have you ever wondered why `cout << "Hello, world!" << endl` is syntactically legal? It's through the overloaded << operator in conjunction with `ostream`s.* In fact, the entire IOStream library can be thought of as a gigantic library of massively overloaded << and >> operators.

The C++ IOStream library is designed to give you maximum flexibility with your input and output routines and even lets you define your own stream insertion and extraction operators. This means that you are allowed to define the << and >> operators so that expressions like `cout << myClass << endl` and `cin >> myClass` are well-defined. However, when writing stream insertion and extraction operators, there are huge number of considerations to keep in mind, many of which are beyond the scope of this class. This next section will discuss basic strategies for overloading the << operator, along with some limitations of the simple approach.

As with all overloaded operators, we need to consider what the parameters and return type should be for our overloaded << operator. Before considering parameters, let's think of the return type. We know that it should be legal to chain stream insertions together – that is, code like `cout << 1 << 2 << endl`

---

* As a reminder, the `ostream` class is the base class for output streams. This has to do with inheritance, which we'll cover next lecture, but for now just realize that it means that `cout`, `stringstream`, and `ofstream` are all specializations of the more generic `ostream` class.

should compile correctly. The << operator associates to the left, so the above code is equal to

```
(((cout << 1) << 2) << endl);
```

Thus, we need the << operator to return an `ostream`. Now, we don't want this stream to be `const`, since then we couldn't write code like this:

```
cout << "This is a string!" << setw(10) << endl;
```

Since if `cout << "This is a string!"` evaluated to a `const` object, we couldn't set its width to 10. Similarly, we do not want to return the stream by value. For example, consider the following code snippet:

```
ofstream output("out-file.txt");
output << "This is a string!" << 137 << endl;
```

If `output << "This is a string!"` returned a copy of the `output` stream, at the end of the line, the temporary stream object's destructor would invoke and close the file handle. Not at all what we had in mind! Putting these two things together, we see that the stream operators should return a non-`const` reference to whatever stream they're referencing.

Now let's consider what parameters we need. We need to know what stream we want to write to or read from, so initially you might think that we'd define overloaded stream operators like this:

```
ostream& operator <<(ostream &input) const;
```

Unfortunately, this isn't correct. Consider the following two code snippets:

```
cout << myClass;
myClass << cout;
```

The first of these two versions makes sense, while the second is backwards. Unfortunately, with the above definition of `operator <<`, we've accidentally made the second version syntactically legal. The reason is that these two lines expand into calls to

```
cout.operator <<(myClass);
myClass.operator <<(cout);
```

The first of these two isn't defined, since `cout` doesn't have a member function capable of writing our object (if it did, we wouldn't need to write a stream operator in the first place!). However, based on our previous definition, the second version, while semantically incorrect, is syntactically legal. Somehow we need to change how we define the stream operator so that we are allowed to write `cout << myClass`. To fix this, we'll make the overloaded stream operator a free function that takes two parameters – an `ostream` to write to and a `myClass` object to write. The code for this is:

```
ostream& operator << (ostream &stream, const MyClass &mc)
{
    /* ... implementation ... */
    return stream;
}
```

While this code will work correctly, since `operator <<` is a free function, it doesn't have access to any of the private data members of `MyClass`. This can be a nuisance, since we'd like to directly write the contents of `MyClass` out to the stream without having to go through the (possibly inefficient) getters and setters. Thus, we'll declare `operator <<` a `friend` inside the `MyClass` declaration, as shown here:

```
class MyClass
{
    public:
        /* More functions. */
        friend ostream& operator <<(ostream &stream, const MyClass &mc);
};
```

Now, we're all set to do reading and writing inside the body of the insertion operator. It's not particularly difficult to write the stream insertion operator – all that you need to do is print out all of the meaningful class information with some formatting information. So, for example, given a `Point` class representing a point in 2-D space, we could write the insertion operator as

```
ostream& operator <<(ostream &stream, const Point &pt)
{
    stream << '(' << pt.x << ", " << pt.y << ')';
    return stream;
}
```

While this code will work in most cases, there are a few spots where it just won't work correctly. For example, suppose we write the following code:

```
cout << "01234567890123456789" << endl; // To see the number of characters.
cout << setw(20) << myPoint << endl;
```

Looking at this code, you'd expect that it would cause `myPoint` to be printed out and padded with space characters until it is at least twenty characters wide. Unfortunately, this isn't what happens. Since `operator <<` writes the object one piece at a time, the output will look something like this:

```
01234567890123456789
                   (0, 4)
```

That's nineteen spaces, followed by the actual `Point` data. The problem is that when we invoke `operator <<`, the function writes a single `(` character to `stream`. It's this operation, not the `Point` as a whole, that will get aligned to 20 characters. There are many ways to circumvent this problem, but perhaps the simplest is to buffer the output into a `stringstream` and then write the contents of the `stringstream` to the destination in a single operation. This can get a bit complicated, especially since you'll need to copy the stream formatting information over.

Writing a correct stream extraction operator (`operator >>`) is incredibly complicated. You'll need to remember all the information you read so that if the operation fails you can revert the state of the stream to the state before the read operation. Similarly, you'll need to manually set the stream's fail state if the operation fails. For more information on writing stream extraction operators, consult a reference.

List of Overloadable Operators

The following table lists the most commonly-used operators you're legally allowed to overload in C++, along with any restrictions about how you should define the operator.

`=`	Assignment	`MyClass &operator =(const MyClass &other);` See the previous handout for more details.
`+= -= *=` `/= %=`	Compound assignment	`MyClass &operator +=(const MyClass &other);` When writing compound assignment operators, make sure that you correctly handle "self-compound-assignment."

`+ - * / %`	Mathematical operators	`const MyClass`
		`  operator + (const MyClass &other) const;`
		`friend`
		`  const MyClass operator +(const OtherClass &one,`
		`  const MyClass &two);`
		
		When overloading arithmetic operators when the parameter isn't of the same type as your class, make sure to define free functions to correctly handle the parameter ordering.

`-`	Unary minus	`const MyClass operator - () const;` The unary minus operator is the minus operator invoked when writing code like `y = -x`. It's easy to forget to overload this one, so be sure to keep an eye out for it.

`< <= ==`
`> >= !=`
	Relational operators	`bool operator < (const MyClass &other) const;`
		`friend bool operator <(const MyClass  &one,`
		`  const MyClass &two);`
		
		If you're planning to use relational operators only for the STL container classes, you just need to overload the < operator. Otherwise, you should overload all six so that users aren't surprised that `one != two` is illegal while `!(one == two)` is defined.

`[]`	Selection	`ElemType &operator [](const KeyType &key);`
		`const ElemType &`
		`  operator [](const KeyType &key) const;`
		
		Most of the time you'll need a `const`-overloaded version of the bracket operator. Forgetting to provide one can lead to a real headache!

`++ --`	Increment/Decrement	Prefix version: `MyClass &operator ++();` Postfix version: `const MyClass operator ++(int dummy);`
`!`	Logical not	`bool operator !() const;` We didn't cover this operator in this handout, but you should be aware that it exists.

`*`	Pointer dereference	`ElemType &operator *();`
		`const ElemType &operator *() const;`
		
		With this function, you're allowing your class to act as though it's a pointer. The return type should be a reference to the object it's "pointing" at. This is how the STL iterators work. Note that this is the unary `*` operator and is not the same as the `*` multiplicative operator.

List of overloadable operators, contd.

`->`	Pointer-to-member	`PointerType operator ->() const;` If the **->** is overloaded for a class, whenever you write `myClass->myMember`, it's equivalent to `myClass.operator ->()->myMember`. Note that the function should be `const` even though the object returned can still modify data. This has to do with how pointers can legally be used in C++. For more information, refer to the handout on `const`.

`<< >>`	Stream operators	`Friend`
		`  ostream& operator << (ostream &out,`
		`  const MyClass &mc);`
		`friend`
		`  istream& operator >> (istream &in,`
		`  MyClass &mc);`

`()`	Function Call	Varies based on the class.

More to Explore

Operator overloading is an enormous topic in C++ and there's simply not enough space to cover it all in this handout. If you're interested in some more advanced topics, consider reading into the following:

1. **Overloaded `new` and `delete`**: You are allowed to overload the `new` and `delete` operators, in case you want to change how memory is allocated for your class. Note that the overloaded `new` and `delete` operators simply change how memory is allocated, not what it means to write `new MyClass`. Overloading `new` and `delete` is a complicated task and requires a solid understanding of how C++ memory management works, so be sure to consult a reference for details.
2. **Conversion functions**: In Handout #17, we covered how to write conversion constructors, functions that convert objects of other types into instances of your new class. However, it's possible to use operator overloading to define an implicit conversion from objects of your class into objects of other types. The syntax is `operator Type()`, where `Type` is the data type to convert your object to. Many professional programmers advise against conversion functions, so make sure that they're really the best option before proceeding.
3. **Custom stream manipulators**: You can also overload the << and >> operators to define your own custom stream manipulators. To write a stream manipulator, simply define a new class that overloads the << or >> operators to modify the properties of a stream.
4. **The Return Value Optimization**: If you'll remember from last week, when returning objects from functions, C++ will create initialize a temporary object to hold the return value. However, using the temporary object syntax, C++ compilers can use the *return value optimization* (RVO) to eliminate an unnecessary object copy. Consult a reference for more information on the RVO, since its especially applicable to overloaded operators.

Practice Problems

Operator overloading is quite difficult because your functions must act as though they're the built-in operators. Here are some practice problems to get you used to overloading operators:

1. If you've overloaded both the prefix and postfix versions of the ++ operator for a certain class, then what does the code `++myClass` expand into? What about `myClass++`?
2. Why is it better to implement + in terms of += instead of += in terms of +?
3. Explain how it's possible to define all of the relational operators in terms of <.
4. Using the `RationalNumber` class from the conversion constructor handout as a starting point, define the relational operators (`< <= == != >= >`) for `RationalNumber`.
5. Recall that member functions defined inside the body of a class are automatically inlined. Knowing this, why is it a good idea to declare a functor's `operator ()` inside the body of the functor?
6. Write a function `AdvancedBiasedSort` that accepts as parameters a `vector<string>` by reference and a `string` "winner" parameter, then sorts the `vector`, except that all strings equal to the winner string are at the front of the `vector`. Do not use any loops.
7. The STL `generate_n` algorithm is defined as `void generate_n(OutputIterator start, size_t count, NullaryFunction fn)` and calls the zero-parameter function `fn count` times, storing the output in the range beginning at `start`. Write a function `FillAscending` that accepts two parameters, an empty `vector<int>` by reference and an `int` called `n` and fills the `vector` with the integers in the range [`0`, `n`). Do not use any loops.
8. Given a `CString` class that stores a C string as `char *theString`, write an overloaded bracket operator for `CString`. Make sure it's `const`-correct!
9. Consider the following definition of a `Span` struct:

```
struct Span
{
    int start;
    int stop;
    Span(int begin, int end) : start(begin), stop(end) {}
};
```

   The `Span` struct allows us to define the range of elements from [`start`, `stop`) as a single variable. Given this definition of `Span` and assuming `start` and `stop` are both non-negative, provide another bracket operator for `CString` that accepts a `Span` by reference-to-`const` and returns another `CString` equal to the substring of the initial `CString` from `start` to `stop`.
10. While legal, is it a good idea to provide either of the operator overloads suggested in the previous problem? Why or why not?