CS 11 Data Structures and Algorithms

Lesson 16: Operator Overloading

Skip to Main Content

Section 16.1: Intro to Operator Overloading

C++ allows us to give new meanings to most operators. This is extremely useful when we are defining new types. Let's consider for a moment the example of defining a new type named Fraction, which we would implement as a class. Without operator overloading, we might have a member function in this class named, say, "AddedTo", which might be called like this in the client program:

```
Fraction f1, f2, f3;
f1 = f2.AddedTo(f3);
```

With operator overloading, we could instead overload (i.e. "give a new meaning to") the '+' operator. This allows for much more elegant code in the client program. Once we have overloaded the '+' operator, the code in the example above would be written like this:

```
Fraction f1, f2, f3;
f1 = f2 + f3;
```

I think you'll agree that this is a much more convenient notation for adding two Fraction objects.

Before describing the syntax required to overload an operator like this, let me set up an example that I will use throughout this lesson. Let's develop a class named feetInches. Each feetInches object will store a linear measurement by storing the number of feet and the number of inches. The purpose of having a class like this would be so that if someone were to write a program which required a lot of linear measurements (for example an architectural program), the programmer could use feetInches objects to simplify her code. Here is our initial class definition, along with a client program to illustrate how it might be used, and the output produced by the client program.

For now we will keep all of our code in three files to keep things simple. At the end of this lesson you will see the code divided into three files. If dividing the code into three files is new to you, you should read lesson 15.8 carefully.

```
#include <iostream>
using namespace std;

class feetInches {
   public:
      feetInches(int inFeet = 0, int inInches = 0);
      feetInches addedTo(const feetInches &other) const;
      void print() const;
   private:
      int feet;
      int inches;
};




feetInches::feetInches(int inFeet, int inInches)
{
   feet = inFeet;
   inches = inInches;
}




feetInches feetInches::addedTo(const feetInches &other) const
{
   feetInches temp;
```

```
      temp.feet = feet + other.feet;
      temp.inches = inches + other.inches;
      return temp;
   }




   void feetInches::print() const
   {
      cout << feet << " feet, " << inches << " inches";
   }
```

```
   // CLIENT PROGRAM!

   int main()
   {
      feetInches f1, f2(3), f3(4,7);

      f1 = f2.addedTo(f3);

      f2.print();
      cout << " plus ";
      f3.print();
      cout << " equals ";
      f1.print();
      cout << endl;
   }
```

```
   // OUTPUT

   3 feet 0 inches plus 4 feet 7 inches equals 7 feet 7 inches
```

There are two aspects of the above program that have nothing to do with operator overloading but that you should understand before we proceed. They are (1) default parameters and (2) the reserved word const.

Section 16.2: Default Parameters

First, notice the parameter list in the prototype for the feetInches constructor: int inFeet = 0, int inInches = 0. We are utilizing something here called "default parameters". If the constructor is called and one or both of the arguments is missing, rather than giving a syntax error, the default for that argument is used. In this case, for example, the declaration

```
feetInches f1;
```

in the client code is missing the two arguments. C++ fills in the arguments automatically with the defaults we have provided: in this case, we indicated the value 0 as the default for both arguments. If only one argument is provided, as in the declaration

```
feetInches f2(3);
```

The first parameter, inFeet, gets the value indicated (3), and the second parameter, inInches, gets the default value specified for that parameter (0). Notice that the default values are given by adding the assignment operator (=) and the desired value after the name of the parameter in the prototype only. The default values are not indicated at any time in the function definition.

To say all of this another way, the code we have provided is equivalent to having three independent constructors: one constructor that takes no arguments and sets inches and feet to 0, a second constructor that takes one argument and sets feet to that argument but sets inches to 0, and a third that takes two arguments and sets feet and inches as indicated by the arguments.

Section 16.3: The Reserved Word const

The word const in C++ is used in several different contexts. Its use in the example here is to indicate that a particular value cannot be changed. Let's look at the function header for the addedTo function:

```
feetInches feetInches::addedTo(const feetInches &other) const
```

The parameter here is named other and is of type feetInches. Our program would compile and run correctly if we had left out the word const and the ampersand (&). Why do we include them? First recall that when you use the pass-by-value parameter passing mechanism, a copy of the argument is made. If the argument is a large object, this can be very inefficient, both in terms of the memory used to store the copy and time taken to make the copy. To avoid this, it is generally considered good programming practice to always pass objects by reference. Recall that an object is a variable whose type is a class.

The problem with this is that if we simply use pass-by -reference instead of pass-by-value we have lost the safeguard that pass-by-value provides against the danger of inadvertently modifying the argument. In order to provide this safeguard despite the use of pass-by-reference, we insert the word const in front of the parameter.

To summarize: when you pass an object (as opposed to a normal variable), always insert the word const in front of the parameter and use pass-by-reference.

The word const also appears at the end of the function header. It looks a bit like it is hanging out there in space. In this case it is being used to indicate that the value of the calling object cannot be changed. If you think about it, in a member function, the calling object is a little bit like a pass-by-reference parameter, in that the function has access to it and can change it. In fact, the calling object is sometimes referred to as an "implicit parameter", since it is kind of like a parameter but it is not listed explicitly as most parameters are. If we are writing a member function which is not intended to modify the calling object, then we should put the word const at the end of the function prototype and also at the end of the function header to protect the calling object from being inadvertently modified.

Section 16.4: Overloading '+'

Enough preliminaries. Let's define the '+' operator so that we can use it on feetInches objects. Recall that in our client program, when we wanted to add two feetInches objects, we had to say

```
f1 = f2.addedTo(f3);
```

It would be much more convenient and intuitive to be able to say

```
f1 = f2 + f3;
```

Unfortunately, this won't work because the '+' operator is not defined for feetInches operands. We can however, provide this definition. This is called operator overloading or (more specifically) overloading the '+' operator.

What follows is not necessarily how we would normally overload the '+' operator, but it is a good place to start. For the sake of illustration we will begin by overloading the '+' operator as a global function. (A global function is a function declared outside the scope of a class. In other words, it's the type of function we used to use before we studied classes and learned about member functions.)

Section 16.5: Overloading '+' as a Global Function

The key piece of information you need to know in order to overload '+' as a global function is that when the C++ compiler sees the code

```
f1 = f2 + f3;
```

It translates it into the following code:

```
f1 = operator+(f2,f3);
```

In other words, when C++ encounters the '+' operator, it will call a function (which you the programmer must provide) named operator+. It will treat the left operand (f2, in our example) as the first argument to the function, and the right operand (f3) as the second argument. One result of this is that when we write the operator+ function, it must have two parameters. This makes sense, since '+' is a binary operator (that is, it takes two operands). If we were overloading a unary operator (an operator that takes only one operand), the function would be required to have exactly one parameter.

Adding this function to the code above we get the following (the only changes are the addition of the operator+ function and the rewriting of the statement f1 = f2 + f3; in the client program):

```
// The class declaration and the definitions of
// the member functions remain the same


feetInches operator+(const feetInches &left,
                     const feetInches &right)
{
   feetInches temp;
   temp = left.addedTo(right);
   return temp;
}
```

```
// CLIENT PROGRAM!

int main()
{
   feetInches f1, f2(3), f3(4,7);

   f1 = f2 + f3;

   f2.print();
   cout << " plus ";
   f3.print();
   cout << " equals ";
   f1.print();
   cout << endl;
}
```

```
// OUTPUT

3 feet 0 inches plus 4 feet 7 inches equals 7 feet 7 inches
```

Section 16.6: Overloading '+' as a Member Function

The code above is not how we would normally overload the '+' operator. Normally we would do it as a member function. When '+' is overloaded as a member function, the C++ compiler translates the statement

```
f1 = f2 + f3;
```

into the statement

```
f1 = f2.operator+(f3);
```

This is similar to the situation we had when we were using a global function, except that now instead of having 2 arguments in the function call we only have one. This is because the calling object represents the left operand. In other words, when C++ encounters the '+' operator, it will call a function (which you the programmer must provide) named operator+. It will treat the left operand (f2, in our example) as the calling object, and the right operand (f3) as the argument. One result of this is that when we write the operator+ function as a member function, it must have only one parameter. This makes sense, since one of the two operands is considered to be the calling object and one is considered to be an argument. If we were overloading a unary operator (an operator that takes only one operand) as a member function, the function would be required to have zero parameters.

The code that results from overloading the '+' operator as a member function is below. Notice that we no longer need the addedTo function at all; in fact, one way of summarizing what we have done here is that the addedTo function has been renamed operator+. This is the only change we have made to the class, and this change makes it so we can use the '+' operator with feetInches objects the same way we use it with variables of type int or double.

```
#include <iostream>
using namespace std;
```

```
class feetInches {
   public:
      feetInches(int inFeet = 0, int inInches = 0);
      feetInches operator+(const feetInches &other) const;
      void print() const;
   private:
      int feet;
      int inches;
};




feetInches::feetInches(int inFeet, int inInches)
{
   feet = inFeet;
   inches = inInches;
}




feetInches feetInches::operator+(const feetInches &other) const
{
   feetInches temp;
   temp.feet = feet + other.feet;
   temp.inches = inches + other.inches;
   return temp;
}




void feetInches::print() const
{
   cout << feet << " feet, " << inches << " inches";
}
```

```
// CLIENT PROGRAM!

int main()
{
   feetInches f1, f2(3), f3(4,7);

   f1 = f2 + f3;

   f2.print();
   cout << " plus ";
   f3.print();
   cout << " equals ";
   f1.print();
   cout << endl;
}
```

```
// OUTPUT

3 feet 0 inches plus 4 feet 7 inches equals 7 feet 7 inches
```

Section 16.7: Overloading the Pre-Increment (++) Operator

Since we have successfully overloaded a binary operator, let's move on and overload a unary operator, the Pre-Increment (++) operator. Before we do, we should make sure that we have a correct understanding of how the ++ operator works in C++. When we overload an operator, it is important that the operator retain its usual behaviors so that a client programmer can use the operator as he is accustomed to. For example, it would be poor programming practice to overload the ++ operator to decrement a feetInches object, since this is not what a client programmer would naturally expect the ++ operator to do. In order to correctly overload an operator, then, we need to have a correct and complete understanding of how the operator works.

Many of you have probably never used the ++ operator as an expression, but only as a statement. For example, you probably use ++ like this:

```
count++;
```

but never in the middle of a statement like this:

```
cout << count++;
x = 2 * count++;
```

This is good! Many programmers, myself included, consider it to be poor programming practice to use the ++ operator as an expression as it is in the last two lines. However, when we overload the ++ operator we still need to provide the functionality that allows a client programmer to use the operator in this way.

The pre-increment (++) operator does two things in this order: it (1) increments the operand and (2) determines the value of the expression. It determines the value of the expression by looking at the value of the operand. Since it does this after incrementing the operand, the value of the expression will always be one more than the original value of the operand. Consider the following code:

```
int i;
i = 7;
cout << ++i;
```

What gets printed? Before the cout statement, the value of i is 7. Since i is incremented first, before the value of the expression is determined, the value of the expression will be 8, and 8 will get printed.

The post-increment (++) operator does two things in this order: it (1) determines the value of the expression and (2) increments the operand. It determines the value of the expression by looking at the value of the operand. Since it does this before incrementing the operand, the value of the expression will always be equal to the original value of the operand. Consider the following code:

```
int i;
i = 7;
cout << i++;
```

What gets printed? Before the cout statement, the value of i is 7. Since the value of the expression is determined first, before i is incremented, the value of the expression will be 7, and 7 will get printed.

Now that we understand the post-increment and pre-increment operators, let's move on and overload the pre-increment operator for our feetInches class. If we were going to overload the operator as a global function, it would have one argument, which would represent the operand. Let's instead overload it as a member function (this is the usual method), in which case it will have no arguments, because the operand will be represented by the calling object. In other words, when C++ sees the statement

```
count++;
```

it will translate this into the statement

```
count.operator++();
```

In our function, we should first increment the operand by adding one to the feet data member, and then return this new value of the operand. There is one more problem we need to solve before we can do this. What we want to return in this function is the calling object itself! This is the first time we have needed to access the calling object itself. In the past we have always accessed individual data members of the calling object, but never the entire calling object itself. When we need to refer to the calling object in a member function, we use the identifier *this. We will discuss this in more detail later, for now you just need to know that when you see *this in your program, it represents the calling object. Here is our member function, along with a client program and output to show that the ++ operator is implemented correctly:

```
 feetInches feetInches::operator++()
 {
     feet++;
     return *this;
 }
```

```
// CLIENT PROGRAM!

int main()
{
        feetInches f1, f2(3), f3(4,7);

        f1 = f2 + f3;

        f2.print();
        cout << " plus ";
        f3.print();
        cout << " equals ";
        f1.print();
        cout << endl;

        (++f1).print();
        cout << endl;
        f1.print();
    }
```

```
    // OUTPUT

     3 feet 0 inches plus 4 feet 7 inches equals 7 feet 7 inches
     8 feet 7 inches
     8 feet 7 inches
```

Section 16.8: Overloading the Post-Increment (++) Operator

Now that we have successfully overloaded the pre-increment operator, let's overload the post-increment operator. This will be very similar to overloading the pre-increment operator. First let's think about what the function header for this function will look like:

feetInches feetInches::operator++()
This looks right, but there is a problem. This is exactly the same as the function header for the pre-increment operator! How will C++ know which function to call? In order to distinguish between the pre- and post-increment operators, we insert the word int into the parameter list for the post-increment operator. This is sometimes called a dummy parameter. Really it's not a parameter at all, it's just a syntactic signal to the compiler that this is the post-increment operator, not the pre-increment operator. So our function header will look like this:

```
feetInches feetInches::operator++(int)
```

The body of the function is also a bit trickier than the pre-increment operator was. Recall that with post-increment we want to first determine the value of the expression, and then increment the operand. To implement this we'll need to declare a temporary feetInches object and assign it the value of the operand (which is being represented by the calling object). Then we can increment the operand, and then return the original value of the operand, which we have saved.

```
feetInches feetInches::operator++(int)
{
    feetInches temp(feet, inches);
    feet++;
    return temp;
}
```

To see the prototype of this function and an example of how it is used in a client program, see the version of our class, along with client code and output, which appears at the end of the following section (section 9).

Section 16.9: Simplifying feetInches objects

One problem with our class as it now stands is that we have no mechanism for making sure that the inches data member is always valid -- that is, always between 0 and 12 inclusive. For example, if we were to add 6 feet 7 inches and 2 feet 8 inches, we would get an answer of 8 feet 15 inches. The answer should be 9 feet 3 inches.

A nice way to fix this problem is to provide a private member function named simplify which, when called, ensures that the calling object has valid data. For example, if the calling object was 8 feet 15 inches, the simplify function would modify the calling object so that it contains the value 9 feet 3 inches. Then we simply call this function at the end of any function that might result in invalid data. This would include the constructors, since the client programmer might make a mistake and provide invalid initial data, and the + operator.

Here is a complete feetInches class, along with a client program and the output produced by the client program. The simplify function appears in this class.

```
#include <iostream>
using namespace std;

class feetInches {
   public:
      feetInches(int inFeet = 0, int inInches = 0);
      feetInches operator+(const feetInches &other) const;
      void print() const;
      feetInches operator++();
      feetInches operator++(int);
   private:
      void simplify()
      int feet;
      int inches;
};





feetInches::feetInches(int inFeet, int inInches)
{
   feet = inFeet;
   inches = inInches;
   simplify();
}





feetInches feetInches::operator+(const feetInches &other) const
{
   feetInches temp;
   temp.feet = feet + other.feet;
   temp.inches = inches + other.inches;
   temp.simplify();
   return temp;
}





void feetInches::print() const
{
   cout << feet << " feet, " << inches << " inches";
}





void feetInches::simplify()
{
   if (inches >= 12){
      feet += inches/12;
      inches %= 12;
   } else if (inches < 0){
      feet -= abs(inches)/12 + 1;
      inches = 12 - (abs(inches) % 12);
   }
}





feetInches feetInches::operator++()
{
   feet++;
   return *this;
}
```

```
   feetInches feetInches::operator++(int)
   {
      feetInches temp(feet, inches);
      feet++;
      return temp;
   }
```

```
// CLIENT PROGRAM!

   int main()
   {
      feetInches f1, f2(3), f3(4,7);

      f1 = f2 + f3;

      f2.print();
      cout << " plus ";
      f3.print();
      cout << " equals ";
      f1.print();
      cout << endl;

      (f1++).print();
      cout << endl;
      (++f1).print();
      cout << endl;
      f1.print();
   }
```

```
   // OUTPUT

    3 feet 0 inches plus 4 feet 7 inches equals 7 feet 7 inches
    7 feet 7 inches
    9 feet 7 inches
    9 feet 7 inches
```

Section 16.10: Overloading the Stream Insertion Operator

At this point there really isn't anything new for me to tell you about operator overloading. The rest of this lesson will be primarily spent discussing the precise behavior of a few more operators, so that they can be correctly overloaded.

So far we have had to print our objects using the print member function. It would be very convenient if we were able to use the insertion operator to print instead. Then we could replace the code

```
f2.print();
cout << " plus ";
f3.print();
cout << " equals ";
f1.print();
cout << endl;
```

with the line

```
cout << f2 << " plus " << f3 << " equals " << f1 << endl;
```

I think you'll agree that the latter form is much more convenient. But before overloading the insertion (<<) operator, let's talk about how it works. When C++ encounters the cout statement above, it first applies the leftmost insertion operator, which has a left operand of cout and a right operand of f2. It does two things: (1) it inserts the right operand into the stream represented by the left operand, and (2) it returns the left operand. In other words, the expression cout << f2 evaluates to cout. As a result of applying the leftmost insertion operator, then, f2 gets printed on the screen and we are left with

```
cout << " plus " << f3 << " equals " << f1 << endl;
```

(The expression cout << f2 has been replaced with simply cout.) As a result of applying the next insertion operator, the string " plus " is printed on the screen, and we are left with

```
cout << f3 << " equals " << f1 << endl;
```

and so on.

What this means for us is that when we overload the insertion (<<) operator, we need to (1) insert the right operand into the stream represented by the left operand, and (2) return the left operand.

There is an important complication that arises as we are attempting to overload the insertion operator. If we overload this operator as a member function, C++ considers the left operand to be the calling object. But in order for a feetInches member function to be called, the calling object must be a feeInches object. The left operand for the insertion operator must be an ostream object. So we can't overload the insertion operator as a member function. To summarize:

We canot make the insertion operator a member function because the left operand is not a feetInches object.

To state this more generally:

We cannot overload an operator as a member function if the left operand is not an object of the class.

The next thing to try would be to overload it as a global function, like this:

```
ostream& operator<<(ostream& out, const feetInches &printMe)
{
   out << printMe.feet << " feet, " << printMe.inches << " inches";
   return out;
}
```

The problem with this solution is that the function accesses the private data members of the class, feet and inches. C++ provides a solution to this problem with the friend function. By putting the prototype for a global function in the class declaration and preceding it with the reserved word friend, we give that global function access to the private members of the class. The definition of the operator<< function stays exactly as it appears above -- that is, it is exactly the same as it would be if it were a global function. The only change required is to place the prototype for the function in the class declaration, and to precede the prototype with the reserved word friend. This is illustrated in the final version of the feetInches class that appears at the end of this lesson.

Notice that cout does not appear anywhere in this function. That's because the left operand for the insertion operator can be any output stream. It doesn't have to be cout. The parameter `out` in the function definition above could represent any output stream. Sometimes it will represent cout, but other times it might represent an output file stream.

You might also be wondering about the & that appears next to the word ostream two times in the prototype for the insertion operator. The rule in C++ is that you cannot pass a stream variable by value. It must always be passed by reference. The reason for this is that it doesn't really make sense to pass the stream itself, which might be an entire file or the keyboard or monitor. You can only pass a reference to the stream. In contrast with arrays, which are always automatically pass by reference, with streams we must always explicitly make them pass by reference.

A couple of final points of clarification about friend functions. First, although friend functions are technically global functions that have been given special privileges, in practice they are actually more similar to member functions than they are to global functions. Like member functions, their prototype appears in the class declaration, their definition typically appears along with the definitions of the class member functions, and they have access to the class's private data members. In fact, the only difference between a member function and a friend function is how they are called. A member function must be called with a calling object, but a friend function is called as a global function would normally be called.

Second, there is some debate about when to use friend functions and when to use member functions. Some authors promote the idea of using friend functions anytime you are overloading a binary operator, since the idea of having the left operand be the calling object and the right operand be the argument seems to go against the symmetry of a binary operator. These authors would, for example, implement operator+ as a friend function rather than as a member function as we have done. Personally, I prefer to keep functions as member functions unless the functionality of my operator requires that the left operand sometimes not be an object of the class, as is the case with the insertion operator.

Section 16.11: Revisiting the + Operator

I would now like to add some functionality to my class. I would like my + operator to work not only when both operands are feetInches objects, but also when either the left operand or the right operand is an integer. I will (somewhat arbitrarily) assume that if the client programmer adds a feetInches object and an integer, the integer represents the number of feet (as opposed to the number of inches). Here is an example of an expression like this:

```
int x;
x = 42;
f1 = x + f3;
```

This code should add 42 feet to the feetInches object f1.

As it stands now, our operator+ function would not handle this case. If we were to run this client code, we would get an error message telling us that we have an invalid operand, since + is not defined to work with a left operand of type int and a right operand of type feetInches. One way fix this would be to write 3 separate operator+ functions. One of these operator+ functions would have two feetInches objects, a second one would have a feetInches object for the left argument and an int for the right operand. What about the third operator+ function? It would have an int for the left operand and a feetInches object for the right operand, which would mean we would have to make it a friend function (reread the section on overloading the insertion operator if the need to make this a friend function isn't clear!).

There is, however, a way to make this all work with only one operator+ function. The good news is that when C++ is faced with an expression where the operator is not defined for the operands as given, it tries very hard to find a way to convert one or both operands to some type for which the operator is defined. In fact, if we only provide an operator+ function which requires two feetInches operands, C++ will use a constructor that takes a single int argument to convert the int operand into a feetInches object and then add the two feetInches objects. Fortunately we already have a constructor which takes a single int argument and which treats that single argument as the number of feet.

This means that if we simply leave our operator+ function alone, it will work for the case where the left operand is a feetInches object and the right operand is an int. A problem arises, however, when the left operand is an int. Our operator+ function won't work because it has been defined as a member function. We need to rewrite it as a friend function. This has been done in the code at the end of this lesson. As it is written there, our + function will work if either the left or right operand (or neither) is an int instead of a feetInches object.

Section 16.12: Overloading <

Make sure to study the operator< function provided in the code at the end of this lesson. There aren't any new issues to discuss, but it is another example of needing to make the operator a friend function so that it will work even if one of the operands is an int.

Section 16.13: Overloading +=

There are no new operator overloading techniques to discuss before overloading the += operator; however, we need to make sure that we understand what the operator does. Although we usually use it as a statement and this is good practice, the += operator can also be used as an expression. In that case it does two things. First it adds the right operand to the left operand. Second, it returns (or "evaluates to") the value that it just assigned to the left operand.

This operator is ordinarily written as a member function and not a friend function, since it won't work to have an int as the left operand anyway. Remember that with assignments C++ must be able to convert the right operand into the data type of the left operand. In order to have an int as the left operand, then, you would have to provide C++ with some way of converting a feetInches object into an int. We have not done this, so we cannot use an int as the left operand, and we gain nothing by making += a friend function instead of a member function. As written in the code that appears at the end of this lesson, the operator can take either an int or a feetInches object on the right, but it must have a feetInches object on the left.

Section 16.14: Operator Overloading Trivia

Here are some operator overloading factoids:

It is actually possible to live without friend functions. For example, we could have our operator+ function be a global function and have it call a member function named, say, addedTo which actually does the adding. Friend functions, however, make this process simpler and cleaner.

You can overload any operator in C++ except for the following: the dot operator (.), the scope resolution operator (::), and the .* and ?: operators. If you don't recognize those last two, don't worry about it.

You cannot create new operators, and the operators that you overload retain their precedence, their associativeness, and the number of operands they take.

The assignment operator (=) must be overloaded as a member function (not a friend function.

Section 16.15: Namespaces

This doesn't have anything to do with operator overloading, but we've reached the point in our object oriented programming work where we ought to be doing this correctly.

There's a lot going on conceptually with namespaces. Before you go onto your next Computer Science course, you should make sure you understand this. Here's complete coverage of all the concepts: **Namespaces Information**

Here is what you HAVE to understand for now in order to get your code working. Let's say we are writing a class named "fraction" and we need to place it in a namespace named "cs_fraction". (There's nothing magic about the name of the namespace. It just needs to be something unique. In your assignments you'll probably be given instructions on what to name the namespace.)

1. At the top of your header file -- after any directives, such as #include or #ifndef -- type

   ```
   namespace cs_fraction {
   ```

   Then type a close curly brace at the very bottom of the file. This places your code into the cs_fraction namespace.

2. Now do the SAME EXACT THING for the fraction.cpp file. So, all of your member function definitions (and friend function definitions) will also be placed into the cs_fraction namespace.

3. Add

   ```
   using namespace cs_fraction;
   ```

   to the top of your client file.

That's it! You can see the results in the final version of the feetInches class below.

More Namespace Information

When Identifiers such as ostream and cout are defined by the system, they are defined to "be in" a namespace named std. This simply means that you can't use those identifiers unless you somehow let the compiler know that you are "using" the std namespace.

In the typical case, up until we studied classes, you let the compiler know this by placing "using namespace std;" at the top of your file. So you don't need the std:: prefix.

However, in some cases (such as in a header file) it is poor practice to have the "using" directive at the top of your file. (Some software companies may require that you never use the "using" directive.) In that case, you'll need to specify the std:: namespace every time you use an identifier, such as ostream, that is defined in the std namespace.

Just to complete the explanation, there is a third way to let the compiler know that you are using the std namespace. You can specify a list of identifiers that use the std namespace at the top of your file. For example,

```
    using std::ostream;
    using std::cout;
```

Section 16.16: Final Version of the feetInches Class

In this, our final version of the feetInches class, we are finally dividing the code up into three files as is good practice. If dividing code up into three files like this is new to you, you should read lesson 15.8. There is one matter of good practice that you need to know about that is not discussed in that lesson: **It is poor practice to have a using namespace statement in a header file**. This is because the client program is going to be #including this file, and you shouldn't force the client program to use a namespace it may not want to use (and that might even cause it to not work). Instead, we prefix each identifier that belongs to the namespace "std" with "std::" as demonstrated below.

```
// file: feetinches.h

#ifndef FEETINCHES_H
#define FEETINCHES_H

#include <iostream>

    namespace cs_feetInches {
            class feetInches {
                    public:
                            feetInches(int inFeet=0, int inInches=0);
                            friend feetInches operator+(const feetInches& left,
                                                                                    const feetInches& right);
                            friend bool operator<(const feetInches& left,
                                                                     const feetInches& right);
                            friend std::ostream& operator<<(std::ostream& out,
                                                                               const feetInches& right);
                            feetInches operator++();
                            feetInches operator++(int);
                            feetInches operator+=(const feetInches& right);
                    private:
                            void simplify();
                            int feet;
                            int inches;
            };
    }        // closes the namespace

#endif
```

```
// file: feetinches.cpp

#include <iostream>
#include "feetinches.h"
using namespace std;

namespace cs_feetInches {
            feetInches::feetInches(int inFeet, int inInches)
            {
                    feet = inFeet;
                    inches = inInches;
                    simplify();
            }




            feetInches operator+(const feetInches& left,
                                                 const feetInches& right)
            {
                    feetInches answer;

                    answer.feet = left.feet + right.feet;
                    answer.inches = left.inches + right.inches;

                    answer.simplify();
                    return answer;

                    // or:
                    // return feetInches(left.feet + right.feet,
                    //                     left.inches + right.inches);
            }




            ostream& operator<<(ostream& out, const feetInches& right)
            {
                    out << right.feet << " feet, " << right.inches << " inches";
                    return out;
            }




            void feetInches::simplify()
            {
                    if (inches >= 12){
                            feet += inches/12;
                            inches %= 12;
                    } else if (inches < 0){
                            feet -= abs(inches)/12 + 1;
                            inches = 12 - (abs(inches) % 12);
                    }
            }




            bool operator<(const feetInches& left, const feetInches& right)
            {
                    if (left.feet < right.feet){
                            return true;
                    }

                    if (left.feet > right.feet){
                            return false;
                    }

                    return left.inches < right.inches;
            }




            feetInches feetInches::operator++()
            {
                    feet++;
                    return *this;
            }




            feetInches feetInches::operator++(int)
            {
                    feetInches temp(feet, inches);  //declaration statement
                    feet++;
                    return temp;
            }




            feetInches feetInches::operator+=(const feetInches& right)
            {
                    *this = *this + right;

                    // or:
                    //feet += right.feet;
                    //inches += right.inches;
                    //simplify();

                    return *this;
            }
    }        // this closes the namespace
```

```
// file: feetinchestest.cpp

#include <iostream>
#include "feetinches.h"
using namespace std;
using namespace cs_feetInches;

int main()
{
    feetInches f1, f2(3), f3(4,7);
    int x;
```

```
        f1 = f2 + f3;
        cout << f2 << " plus " << f3 << " equals " << f1 << endl;

        cout << ++f1;
        cout << endl;
        cout << f1++;
        cout << endl;
        cout << "f1 is now " << f1 << endl;

        x = 42;
        f1 = x + f3;
        //f1 = f2.operator+(f3);
        cout << "f1 is now " << f1 << endl;


        if (f2 < f1){
            cout << "f2 is less than f1" << endl;
        } else {
            cout << "f2 is not less than f1" << endl;
        }

        if (x < f1){
            cout << "x is less than f1." << endl;
        } else {
            cout << "x is not less than f1." << endl;
        }

        cout << (f1 += 7) << endl;
    }
```

```
    3 feet 0 inches plus 4 feet 7 inches equals 7 feet 7 inches
    8 feet 7 inches
    8 feet 7 inches
    f1 is now 9 feet 7 inches
    f1 is now 46 feet 7 inches
    f2 is less than f1
    x is less than f1.
    53 feet 7 inches
```