Overloading the << and >> Operators

Reference:

Book: Starting Out with C++ by Tony Gaddis, Chapter 14

Short Tutorial to implement >> and << operators with code example:

```
class FeetInches
{
private:
	int feet;
	int inches;
public:
	// Constructor
	FeetInches(int f = 0, int i = 0);
	// Mutator functions
	void setFeet(int f);
	void setInches(int i);
	// Accessor functions
	int getFeet() const;
	int getInches() const;
	friend ostream &operator << (ostream &strm, const FeetInches &obj)
	{
		strm << obj.feet << " feet, " << obj.inches << " inches";
		return strm;
	}
	friend istream &operator >> (istream &strm, FeetInches &obj)
	{
		// Prompt the user for the feet.
		cout << "Feet: ";
		strm >> obj.feet;
		// Prompt the user for the inches.
		cout << "Inches: ";
		strm >> obj.inches;
		return strm;
	}
	//friend is a keyword which gives access to ostream and istream classes right to
access private data members of FeetInches class
};
```

Detailed Explanation (can read in lab or can take it as home task and compulsory to complete following explanation with crystal clear understanding before Thursday's lecture)

Overloading the << and >> Operators:

Overloading the math and relational operators gives you the ability to write those types of expressions with class objects just as naturally as with integers, floats, and other built-in data types. If an object's primary data members are private, however, you still have to make explicit member function calls to send their values to cout. For example, assume distance is a FeetInches object. The following statements display its internal values:

cout << distance.getFeet() << " feet, ";

```
cout << distance.getInches() << "inches";
```

It is also necessary to explicitly call member functions to set a FeetInches object's data. For instance, the following statements set the distance object to user-specified values:

```
cout << "Enter a value in feet: ";

cin >> f;

distance.setFeet(f);

cout << "Enter a value in inches: ";

cin >> i;

distance.setInches(i);
```

By overloading the stream insertion operator (<<), you could send the distance object to cout, as shown in the following code, and have the screen output automatically formatted in the correct way.

```
cout << distance;
```

Likewise, by overloading the stream extraction operator (>>), the distance object could take values directly from cin, as shown here.

```
cin >> distance;
```

Overloading these operators is done in a slightly different way, however, than overloading other operators. These operators are actually part of the ostream and istream classes defined in the C++ runtime library. (The cout and cin objects are instances of ostream and istream.) You must write operator functions to overload the ostream version of << and the istream version of >>, so they work directly with a class such as FeetInches.

Here is the function that overloads the << operator:

```
ostream &operator << (ostream &strm, const FeetInches &obj)

{

 strm << obj.feet << " feet, " << obj.inches << " inches";

 return strm;

}
```

Notice the function has two parameters: an ostream reference object and a const FeetInches reference object. The ostream parameter will be a reference to the actual ostream object on the left side of the << operator. The second parameter is a reference to a FeetInches object. This parameter will reference the object on the right side of the << operator. This function tells C++ how to handle any expression that has the following form:

```
ostreamObject << FeetInchesObject
```

So, when C++ encounters the following statement, it will call the overloaded operator<<

function:

cout << distance;

Notice that the function's return type is ostream &. This means that the function returns a reference to an ostream object. When the return strm; statement executes, it doesn't return a copy of strm, but a reference to it. This allows you to chain together several expressions using the overloaded << operator, such as:

cout << distance1 << " " << distance2 << endl;

Here is the function that overloads the stream extraction operator to work with the

FeetInches class:

istream &operator >> (istream &strm, FeetInches &obj)

{

// Prompt the user for the feet.

cout << "Feet: ";

strm >> obj.feet;

// Prompt the user for the inches.

cout << "Inches: ";

strm >> obj.inches;

return strm;

}

The same principles hold true for this operator. It tells C++ how to handle any expression in the following form:

istreamObject >> FeetInchesObject

Once again, the function returns a reference to an istream object, so several of these expressions may be chained together.

You have probably realized that neither of these functions is quite ready to work, though. Both functions attempt to directly access the FeetInches object's private members. Because the functions aren't themselves members of the FeetInches class, they don't have this type of access. The next step is to make the operator functions friends of FeetInches.

// Friends

```
friend ostream &operator << (ostream &strm, const FeetInches &obj)
friend istream &operator >> (istream &strm, FeetInches &obj)
```