**MSCF**

**Financial Computing I**

**Homework 3, for Lectures 7 and 8**

***Due At 3:30 pm Tuesday, Nov. 21, Before Lecture***

***You will lose 1 point every 10 minutes after that time***

1. **Overloading, References, and ?: (20 points)**
2. Create a new, empty Win32 Console Application project named **hw3.1**. Add the file **hw3.1.cpp** to the project. In this source code file is a statement displaying **“hello, world”** followed by commented-out code for the remaining parts of **hw3.1**. Compile and test.
3. Above **main**, define five functions named **add** that return the sum of their arguments. Your **add** functions should have these interfaces:

**int add(char c1, char c2) { … }**

**int add(int i1, int i2) { … }**

**double add(double x1, double x2) { … }**

**double add(double x1, double x2, double x3) { … }**

**int add(int i1, int i2, int i3, int i4) { … }**

Uncomment these test calls of the overloaded **abc** functions in **main**. ***Predict*** which statements will fail to compile, and the output produced by each statement that will compile. (You don’t need to write down your predictions.) Compile and test; comment out any statements that won’t compile.

**cout << fixed; // display decimal point for double**

**cout << add('a', '\n') << '\n';**

**cout << add(5, 9) << '\n';**

**cout << add('A', 32) << '\n';**

**cout << add(9, 3.3) << '\n';**

**cout << add(3.5, 4.2) << '\n';**

**cout << add(1.1, 2.2, 3.3) << '\n';**

**cout << add(1, 2, 3, 4) << '\n';**

**cout << add(5, 4, 3, 2, 1) << '\n';**

**cout << add(1, 2, 3) << '\n';**

**cout << add(1.1, 2.2, 3.3, 4.4) << '\n';**

**cout << add('A', 2.2, 1) << '\n';**

Be sure you understand why you got the output that you got! (You don’t need to write anything down for this.)

1. Above **main**, define four functions named **swap** that swap two values. Two should swap values via pointer parameters, and two should swap values via reference parameters. That is, the interfaces to these functions should be:

**void swap(int \*pi, int \*pj) { … }**

**void swap(double \*px, double \*py) { … }**

**void swap(int& ri, int& rj) { … }**

**void swap(double& rx, double& ry) { … }**

Uncomment these test calls of the overloaded **swap** functions in **main**. Compile and test.

**int i(7), j(13);**

**double x(3.14), y(2.18);**

**cout << "before swap: i: " << i**

**<< ", j: " << j << '\n';**

**swap(&i, &j); // pass pointers**

**cout << "after swap: i: " << i**

**<< ", j: " << j << '\n';**

**cout << "before swap: i: " << i**

**<< ", j: " << j << '\n';**

**swap(i, j); // pass references**

**cout << "after swap: i: " << i**

**<< ", j: " << j << '\n';**

**cout << "before swap: x: " << x**

**<< ", y: " << y << '\n';**

**swap(&x, &y); // pass pointers**

**cout << "after swap: x: " << x**

**<< ", y: " << y << '\n';**

**cout << "before swap: x: " << x**

**<< ", y: " << y << '\n';**

**swap(x, y); // pass references**

**cout << "after swap: x: " << x**

**<< ", y: " << y << '\n';**

1. Above **main**, define three functions named **max\_of** that return the maximum of two values. One should return the maximum of two **char** values; one should return the maximum of two **int** values; and one should return the maximum of two **double** values. *The body of each function should consist of just a single* **return** *statement,* making use of the **?:** operator. The interfaces to these functions should look like this:

**char max\_of(char c1, char c2) { … }**

**int max\_of(int i1, int i2) { … }**

**double max\_of(double d1, double d2) { … }**

Uncomment these test calls of the overloaded **swap** functions in **main**. Compile and test.

**cout << "max\_of('a', 'b'): " << max\_of('a', 'b') << '\n';**

**cout << "max\_of(-4, 8): " << max\_of(-4, 8) << '\n';**

**cout << "max\_of(2.2, 3.3): " << max\_of(2.2, 3.3) << '\n';**

1. **The BinaryTree Class (80 points)**

In this part of the homework, you will work with the BinaryTree class discussed in the lecture, and make enhancements to it.

1. Create a new, empty Win32 Console Application project named **BinaryTree\_app**.

Three source files have been provided for this part of the homework: (1) **BinaryTree.h**, which contains the **BinaryTree** class definition, and a declaration (***not*** a definition) of the **bt\_node** structure; (2) **BinaryTree.cpp**, which has an **#include "BinaryTree.h"** directive near the top, and implements the **BinaryTree** member functions, the **bt\_node** structure *definition*, and the “helper functions” described in the lecture and previous homework; and, (3) **BinaryTree\_app.cpp**, which is an application (a **main** function) using the **BinaryTree** class. In order to use the **BinaryTree** class, **BinaryTree\_app.cpp** also requires an **#include "BinaryTree.h"** directive near the top.

In Visual Studio, *right*-click on **Source Files** and add the two *existing* items **BinaryTree\_app.cpp** and **BinaryTree.cpp**. Then, *right*-click on **Header Files** and add the *existing* item **BinaryTree.h**. ***Compile and test***, to make sure you have added the correct files to your project.

Examine the file **BinaryTree.h**. You will see that it contains *only* enough information for a user (application developer) to declare and use **BinaryTree** objects. There is *almost no information* about how a **BinaryTree** is implemented, and the single data member in a **BinaryTree** is **private**.

Examine the file **BinaryTree\_app.cpp**. You will see how much cleaner, simpler, and more convenient the code is, compared to our original version where pointer and address notation, clunky function names, lack of automatic initialization, lack of access control to data, and lack of automatic deletion were problems, especially for entry-level application developers.

Examine the file **BinaryTree.cpp**. This implements all of the member functions for the **BinaryTree** class, and contains the *definition* of the **bt\_node** structure. We have chosen to implement the member functions near the top of this code file, but since most of the member functions use previously defined “helper function”, we have put *declarations* of all the needed helper functions above the member function definitions. It would have been fine to move the member function definitions *below* the helper function definitions, so that the helper function declarations would not have been needed. ***Please note*** that the helper functions are ***not*** members of the **BinaryTree** class, so they ***do not*** have the **BinaryTree::** notation indicating class membership.

Compile and test again, after you have examined these source files, to confirm that the member functions used in **main** are behaving as you expect.

1. Add a member function **max** that returns the *maximum* value in a **BinaryTree** object. Add a test of this member function in **main**, below the test of **min**. You may or may not find it convenient to use a helper function to implement **max**. Compile and test.
2. Add a member function **put\_to\_ostream** that behaves like **put**, but that takes a reference to an **ostream** object as an argument (**cout** is an example of an **ostream**), and puts the values in the **BinaryTree** to that specific **ostream**. The *declaration* of this member function in the class definition in **BinaryTree.h** should look like:

**void put\_to\_ostream(ostream&);**

and the *definition* of this member function in **BinaryTree.cpp** should look like:

**void BinaryTree::put\_to\_ostream(ostream& os)**

**{ … }**

***Hint:*** **put\_to\_ostream()** can use a helper function very similar to the one used by **put()**, but with the **ostream&** passed in as an argument instead of being hard-coded as **cout**. In the **main** function in **BinaryTree\_app.cpp**, *after* the *second* occurrence of these statements (the occurrence following the series of **insert**s into **tree1**):

**cout << "tree1 contains: [ ";**

**tree1.put();**

**cout << " ]\n";**

… uncomment these statements to test your **put\_to\_ostream** member function:

**cout << "tree1 contains: [ ";**

**tree1.put\_to\_ostream(cout);**

**cout << " ]\n";**

Compile and test. Confirm that **tree1.put\_to\_ostream(cout)** produces the same output as **tree1.put()**.

1. Define a member function **is\_identical** that tests whether the *current* **BinaryTree** object (**\*this**) is ***identical*** to some *other* **BinaryTree** object, in structure as well as in the values contained, and returns a **bool** value, **true** or **false**. The *declaration* in **BinaryTree.h** should look like this:

**bool is\_identical(const BinaryTree&);**

… and the *definition* in **BinaryTree.cpp** should look like this:

**bool BinaryTree::is\_identical(const BinaryTree& other)**

**{** … **}**

You may find it easier to define a helper function for **is\_identical** to call. Here are hints about how to get this to work:

1. If the current tree and the other tree are both empty, they are identical.
2. If one tree is empty but the other tree is not, they are *not* identical.
3. If the value in the top node of the current tree *is not* equal to the value in the top node of the other tree, the trees are *not* identical.
4. Otherwise, if the value in the top node of the current tree *is* equal to the value in the top node of the other tree, then:
   1. If the two left subtrees are identical ***and*** the two right subtrees are identical, the trees are identical.
   2. Otherwise, the trees are not identical.

Uncomment this code in the **main** function, to test your **is\_identical** function:

**BinaryTree tree4;**

**tree4.insert(12); tree4.insert(5); tree4.insert(30);**

**BinaryTree tree5(tree4);**

**BinaryTree tree6;**

**tree6.insert(30); tree6.insert(12); tree6.insert(5);**

**if (tree4.is\_identical(tree5))**

**cout << "tree4 and tree5 are identical\n";**

**else**

**cout << "tree4 and tree5 are NOT identical\n";**

**if (tree4.is\_identical(tree6))**

**cout << "tree4 and tree6 are identical\n";**

**else**

**cout << "tree4 and tree6 are NOT identical\n";**

Compile and test.

1. Define an overloaded **==** operator, that tests whether the left- and right-hand trees *contain the same values* (but are not necessarily identical in structure), and returns a **bool** value **true** or **false**. The *declaration* in **BinaryTree.h** should look like this:

**bool operator==(const BinaryTree&);**

… and the *definition* in **BinaryTree.cpp** should look like this:

**bool BinaryTree::operator==(const BinaryTree& other)**

**{** … **}**

***Hint:*** You already have a function that returns the size of the **BinaryTree**. If two **BinaryTree** objects have different sizes, they cannot be equal. You already have a function that displays the values in the **BinaryTree** in order to **cout**. You can create a function that stores the values of a **BinaryTree** object into a dynamically allocated array of the correct size. You can loop through the elements in two arrays, comparing for equality.

Write some code that stores the values of **\*this** object in anarray, and the values of the **other** object in an array, and then compares the two arrays. (Don’t forget to deallocate your arrays when done.)

Uncomment this test code in your **main** function, to test your **==** operator:

**if (tree4 == tree5)**

**cout << "tree4 == tree5\n";**

**else**

**cout << "tree4 != tree5\n";**

**if (tree4 == tree6)**

**cout << "tree4 == tree6\n";**

**else**

**cout << "tree4 != tree6\n";**

**if (tree4 == tree1)**

**cout << "tree4 == tree1\n";**

**else**

**cout << "tree4 != tree1\n";**

Compile and test.

1. Define an overloaded **!=** operator, that tests whether the left- and right-hand trees *do not contain the same values*, and returns a **bool** value **true** or **false**. ***Hint:*** *this should require almost no work!* Uncomment the test cases of **!=** below the test cases of **==** from part (e). Compile and test.
2. Overload the **<<** operator for **BinaryTree** objects, so that we can write code like:

**cout << "The values stored in tree4 are\n[ "**

**<< tree4**

**<< " ]\n\n";**

There are a couple of “tricks” to getting this to work.

***First***, notice that **cout** is on the ***left*** side of the **<<** operator, ***not*** on the ***right*** side. In order for our new **<<** operator to be a *member function*, it would have to be a member function of the **ostream** class, so that **cout << tree1** would be treated by the compiler as **cout.operator<<(tree1)**. But the **ostream** class is part of the C++ Standard Library: we cannot (at least, we should not) make modifications to the **ostream** class. (You *could* overload **<<** as a member function of **BinaryTree**, but then you would have to use it in a bizarre way: **tree4 << cout**. Not good!)

So this overloading of **<<** will need to be a *top-level* (or global) function. This will allow the compiler to treat **cout << ltree1** as the function call **operator<<(cout, ltree1)**. For efficiency, we will pass the **ostream** and **BinaryTree** objects *by reference*. So that we can chain **<<** operators in the way we chain **=** operators, the overloading of **<<** must return a *reference* to its **ostream** argument. That is, the definition of this overloading of **<<** will look like:

**ostream& operator<<(ostream& os, BinaryTree& bt)**

**{**

**...**

**return os;**

**}**

***Second***, since this function is ***not*** a member of the **BinaryTree** class, *but it* *might need access to the* **private** *(data) members* of the **BinaryTree** parameter object **bt**, the function must be declared as a **friend** within the **BinaryTree** class definition in **BinaryTree.h**. The **private** (usually data) members of a class can be accessed by the member functions ***and* friend*s*** of the class. Overloaded **<<** is a typical part of the “public” interface to a class, even though it is not a member of the class! (If nothing else, defining **<<** for each class is an aid in debugging.) In the **BinaryTree** class definition in **BinaryTree.h**, you will need this declaration:

**friend ostream& operator<<(ostream&, BinaryTree&);**

A class’s **friend**s are usually declared at the end of the list of **public** members, although technically the position of a **friend** declaration in a class definition does not matter.

Add the **friend** *declaration* of **operator<<** to the **BinaryTree** class definition in **BinaryTree.h**, and add (and finish) the *definition* of **operator<<** in **BinaryTree.cpp**. ***Hint:*** You already have a function you can call that puts a **BinaryTree** object to an arbitrary **ostream** object. Uncomment the test code in **main** for your overloaded **operator<<**. Compile and test.

***REMEMBER*** to put all authors’ names into each of your source code files.Put your **hw3.1.cpp**, **BinaryTree.h, BinaryTree.cpp**, and **BinaryTree\_app.cpp** files into a **.zip** archive and upload to the course web site.