**MSCF Financial Computing II**

**Homework 3**

***Due At 3:30 pm Wednesday, Feb. 7, Before Lecture***

***You will lose 10 points per hour after that time***

1. **(45 points) static Member Functions, inline Functions, Default Parameter Values, and RTTI**
2. Create a new, empty Win32 Console Application project named **FCII\_HW3\_1**. Add the existing items **FCII\_HW3\_1.cpp** (containing the **main()** function), **Employee.h**, **Employee.cpp**, **Manager.h**, **Manager.cpp**, **CEO.h**, and **CEO.cpp**. These files contain the solution code for Homework 1 Part 3, split up into separate code and header files. However, a little more work needs to be done before these will compile and run.

Add *macro guards* in the three header files so that the headers can be **#include**d multiple times without a problem. Also, add the *declaration* of the **static int** data member **next\_id** in the **Employee** class definition in **Employee.h**, and add the *definition and initialization* of **next\_id** in **Employee.cpp**. As you will see in the **Employee** class constructor definition, **next\_id** is used to initialize the **const int id** member of an **Employee** object.

Compile and test, to confirm that the program works as expected.

1. Add these **static** member functions and make other changes as necessary to the **Employee**, **Manager**, and **CEO** classes and their constructors/destructors to support these functions:

**get\_next\_id()** in **Employee** that returns the value of **next\_id**

**get\_num\_emps()** in **Employee** that returns the number of **Employee** objects that currently exist (this count will include the number of **Managers** and number of **CEO**s, since a **Manager** is a kind of **Employee** and a **CEO** is a kind of **Employee**)

**get\_num\_mgrs()** in **Manager** that returns the number of **Manager** objects that currently exist (this count will include the number of **CEO**s, since a **CEO** is a kind of **Manager**)

**get\_num\_ceos()** in **CEO** that returns the number of **CEO** objects that currently exist

“Uncomment” the test code for part (b) in **main()**. Compile and test. Make sure the output makes sense to you and is correct.

1. Revise the Employee, Manager, and CEO classes to use explicit **inline** functions rather than implicit inline functions (defined inside the class definitions) or very short non-inline function definitions (defined in the class code files). You will need to make some judgement calls as to which function definitions should remain in the code files, and which can be moved out of the code files and made explicitly **inline** in the headers.

“Uncomment” the test code for part (c) in **main()**. Compile and test. The explicit inline functions should work identically to the implicit inline functions and non-inline functions that you had before. Make sure the output makes sense to you and is correct.

1. Revise the **Employee**, **Manager**, and **CEO** class constructors to use default parameter values: default rate of **37.5** for **Employee**; default rate of **51.25**, default title of **“Manager”**, and default budget of **350000.0** for **Manager**; and default rate of **92.50**, budget of **1000000.00**, profit\_target of **700000.0**, and bonus\_percentage of **25.0** for **CEO**.

“Uncomment” the test code for part (d) in **main()**. Compile and test. Make sure the output makes sense to you and is correct.

1. Define a function **buy\_soup** that takes a *reference-to* **Employee** as its first argument and a **string** (the name of the soup) as its second argument, and that returns **void**. If the reference argument refers to an actual **Employee**, the function should display the message, “No soup for you!”; if the reference argument actually refers to a **Manager**, the function should display the message, “Chicken noodle soup for you.”; or if the reference argument actually refers to a **CEO**, the function should display the message, “Here is your <*requested\_soup\_type*>. Please come again!”

**buy\_soup** should behave correctly when a **const** **Employee**, **Manager**, or **CEO** is given as the first argument. It should ***not*** be possible to call the function with a string literal as the first argument. Modify your code as needed. (*Hint:* In the implementation of **buy\_soup**, if you take the address of the reference parameter, this will be the same as the address of the argument object.)

“Uncomment” the test code for part (e) in **main()**. Compile and test. Make sure the output makes sense to you and is correct.

1. **(45 points) Algebraic Expressions and shared\_ptr<T>**

In this part of the homework, you will extend the Algebraic Expression classes that we developed in HW2, and make use of the shared\_ptr<T> “smart pointer” class.

1. Create a new, empty Win32 Console Application project named **FCII\_HW3\_2**. In Source Files, add an existing item, **FCII\_HW3\_2.cpp**. This file contains the solution code from **FCII\_HW2\_5.cpp**, with a simplified **main()** function (the five randomly generated arithmetic expressions are displayed, and the algebraic expression involving a single variable is displayed twice with two different values for the variable). We have also modified the pointer types to match the allocated types, rather than using base class (**Term**) pointers, *except* for pointers returned from **rand\_operator()**.

Compile and test, to confirm that the classes and program work as expected.

1. In **main()**, change this statement:

**Times \*t = new Times(c3, c4);**

to this:

**Times \*t = new Times(c3, v1);**

Now, the **Variable** pointed to by **v1** appears ***two*** times in the algebraic expression. Compile and test. Confirm that the expression involving **X** is display correctly, with **X** appearing twice, but then the program crashes. The crash happens because when

**delete m;**

is reached, **~BinaryOp()** deletes **v1** two times.

1. Change the **BinaryOp** class so that it no longer “owns” the **Term**s it points to, that is, so that the destructor no longer does **delete left** and **delete right**. Compile and test. The program should now “work” in the sense that it does not crash. But now the program has a memory leak! Only **p3** in the **for** loop in **main()** and **m** near the end of **main()** are being deleted. **c1**, **c2**, **c3**, **c4**, **p1**, **p2** in the **for** loop, and **c1**, **c2**, **c3**, **c4**, **v1**, **p**, **d**, and **t** later in **main()** are not being deleted. This kind of leak of small objects can be very hard to detect: your expression generator/evaluator might run for months before using up enough of your system’s memory for it to be noticed.
2. Make **main()** the owner of (that is, responsible for) all the pointers: add the missing **delete** statements so that all objects in **main()** allocated with **new** are also deallocated in **main()** with **delete**. Compile and test.
3. Write a top-level function above **main()** with this declaration:

**void add\_x(Term \*pt, double x)**

The **add\_x()** function should add **x** to the value of the **Term** pointed to by **pt**, ***if*** the actual type that **pt** points to is **Variable**. Otherwise, **add\_x()** should do nothing.

Uncomment this code to test your **add\_x()** function:

**add\_x(c1, 1.2); // should do nothing**

**add\_x(v1, 1.1); // should add 1.1 to the Variable**

**add\_x(p, 2.5); // should do nothing**

**cout << "X = " << v1->to\_value() << "\n";**

**cout << m->to\_string() << " = "**

**<< m->to\_value() << "\n";**

Compile and test.

1. Create a new, empty Win32 Console Application project named **FCII\_HW3\_2\_f**. In Source Files, add a new item, **FCII\_HW3\_2\_f.cpp**. Copy-and-paste the contents of **FCII\_HW3\_2.cpp** into this new file. Compile and test, to confirm that the copy-and-paste operation succeeded.

Rather than having **main()** be the owner of all the pointers to the algebraic expression **Term**s, and making **main()** delete all of these objects when done with them, let’s use **shared\_ptr<T>** so that the pointers own themselves: when the last **shared\_ptr<T>** to a given dynamically allocated object goes out of scope, that dynamically allocated object is automagically **delete**d.

First, remove all of the **delete** statements from **main()**. These are no longer necessary when you use **shared\_ptr<T>**.

Next, near the top of **FCII\_HW3\_2\_f.cpp**, add:

**#include <memory>**

Then, throughout the **Term** class hierarchy, change declarations such as

**Term \*p;**

to:

**shared\_ptr<Term> p;**

Change initializations such as:

**Constant \*c1 = new Constant(1.1);**

to:

**shared\_ptr<Constant> c1(new Constant(1.1));**

(you cannot use **=** for initialization of a **shared\_ptr<T>** object).

In the **rand\_operator()** function, you will need to ***explicitly*** construct a **shared\_ptr<T>** object of the appropriate type for each operator. For example:

**shared\_ptr<Term> rand\_operator(shared\_ptr<Term> t1,**

**shared\_ptr<Term> t2)**

**{**

**switch (rand() % 4) {**

**case 0:**

**return shared\_ptr<Plus>(new Plus(t1, t2));**

**...**

**}**

In the **add\_x()** function, you cannot use **dynamic\_cast<T>** on **shared\_ptr<T>** objects. However, **shared\_ptr<T>** provides its own equivalent facility, called **dynamic\_pointer\_cast<T>**:

**void add\_x(shared\_ptr<Term> pt, double x)**

**{**

**if (shared\_ptr<Variable> pv =**

**dynamic\_pointer\_cast<Variable>(pt)) {**

**pv->set\_value(pv->to\_value() + x);**

**}**

**}**

Once you have made all these necessary changes, compile and test. The program should work as before, and have no memory leaks.

1. **Templates (10 points)**
2. Create a new, empty Win32 Console Application project named **FCII\_HW3\_3**. In Source Files, add an existing item, **FCII\_HW3\_3.cpp**. This file contains test code for a couple of template functions, **min\_of<T>** and **abs\_val<T>**. Above **main()**, define template functions **min\_of<T>** which returns the minimum of its two same-typed arguments, and **abs\_val<T>** which returns the absolute value of its argument.

Compile and test, to confirm that the template functions work as expected.

***REMEMBER*** to put all authors’ names into each of your source code and header files.Put your source code and header files into a **.zip** archive and upload to the course web site.