**MSCF Financial Computing II**

**Homework 1**

***Due At 12:30 pm Monday, Oct. 29, Before the Quizzes***

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

1. **European Options (40 points)**

In this part of the homework, you will develop code for pricing “plain vanilla” European Call and European Put options. By “plain vanilla” we mean that the underlying asset pays no dividend, that the volatility of the underlying asset’s price movements is constant throughout time, that the payoff of the call/put is made at expiration time T and cannot be collected at any prior time (as it can in an American Call or Put), and so forth.

1. Create a new, empty project named **EuropeanOptions**. Add the existing item **EuropeanOptions.cpp**, provided with this homework. **EuropeanOptions.cpp** contains the **EuropeanCallOption** class code that we saw in Lecture 1, with the addition of a “pretty printing” function, **put\_BinomialTree**, that lets you display small **binomialTree** objects in a clean way.

Compile and test. You will see that the program displays the **binomialTree** at each step in the **binomialPrice** function’s algorithm. The backward induction code is yours to write—**Your code here**—so the reported option price at this point is 0.0.

1. Add the backward induction code, the formula for which is shown on slide 17 of the Lecture 1 notes. Compile and test. Here is the output I get for a 5-step **binomialPrice**:

Call price, with 5 intervals: 6.35983

1. The call value with 5 time steps is pretty far off from the BSM price shown on slide 6 of the notes: **$6.12**. As you add time steps to the Binomial Tree, the **binomialPrice** should converge to the BSM price. In **main**, add more calls of **binomialPrice**, with time steps of 5, 10, 20, 50, 100, 200, 500, and 1000. Compile and test. Do the computed prices appear to converge?
2. Given identical volatility, interest rate, and expiration time, the price of an option on a $5 stock with strike price $5 should be exactly 1/10 of the price of an option on a $50 stock with strike price $50. Likewise, the price of an option on a $500 stock with a strike price of $500 should be 10 times the price of $50 stock, $50 strike option. That is, there should be no arbitrage opportunity on these expected future cash flows, given identical risk, interest rate, and time profiles. Add tests in **main** to show that this is the case for the Binomial Tree pricing model. Just use 1000 time intervals for each test. Compile and test.
3. The option price will decrease as volatility decreases, and increase as volatility increases. Add tests in **main** to show the price of the European Call on the $50 stock with $50 strike, for volatilities of .05, .10, .20, .40, .80, and 1.60. Compile and test. Is the price linear or non-linear with increasing volatility? Try plotting this using Excel or some other graphing application (you don’t need to turn in your plot).
4. Implement a **EuropeanPutOption** class, which is essentially the same as the **EuropeanCallOption** class but with a different payoff function. You can define the **EuropeanPutOption** class and member function(s) in **EuropeanOptions.cpp**, below or above the **EuropeanCallOption** class code. In **main**, add the same series of tests for the **EuropeanPutOption** that we performed on the **EuropeanCallOption**. Compile and test.
5. For the **EuropeanCallOption** class, implement a **BSMPrice** function that computes the option price according to the Black-Scholes-Merton formula, shown on slide 5 of the Lecture 1 notes.

The C++ Standard Library (as of C++17, anyway) does not include a Standard Normal CDF function, but does include the Gauss Error Function, **erf**. It turns out that that

**N(x) == (1.0 + erf(x / sqrt(2))) / 2.0**

so you can easily compute N(d1) and N(d2) using **erf** (**#include <cmath>**). Confirm that the **BSMPrice** is very close to the **binomialPrice** (with a large number of time steps) for the various call options you have defined in this part of the homework.

1. For the **EuropeanPutOption** class, implement a **BSMPrice** function that computes the option price according to the Black-Scholes-Merton formula for a European Put. (You will need to look up the formula.)

Confirm that the **BSMPrice** is very close to the **binomialPrice** (with a large number of time steps) for the put options you have defined in this part of the homework.

1. ERI, the Economic Research Institute, provides a convenient online BSM calculator, with the same units that we have used in our European option classes.

**https://www.erieri.com/blackscholes**

Confirm that your binomial (with large N) and BSM prices match (within a penny) the

prices produced by this calculator. (Make a comment about this in your source code.)

1. **Function Pointers and typedef (20 points)**

In this part of the homework, you will experiment with function pointers, **typedef**, and **const**.

1. Create a new, empty Win32 Console Application project named **FCII\_HW1\_2**. In Source Files, add an existing item, **FCII\_HW1\_2.cpp**. This file declares and initializes a **set<int>** object **si1**, and then uses a range **for** loop to display the elements.

Compile and test. Confirm that the **set<int>** object stores values in ascending order by default, with no duplicates.

1. You can use a *pointer-to* function to specify a non-default ordering rule when you create a **set**. The pointed-to function must take two arguments of the same type as the **set** element type, and return **true** if the first argument value should come *before* the second argument value in the **set**, or **false** otherwise. (This is called a *strict weak ordering*.)

For example, here is an ordering function that defines *descending* order for **int** values:

**bool gtr\_than(int i, int j) { return i > j; }**

The name **gtr\_than** denotes the address of this function. The data type of the name **gtr\_than** is *pointer-to* *function-taking* **int***-arg* *and* **int***-arg* *returning* **bool**.

To create a **set***-of-***int** named **si1** with the default ordering rule, you can write:

**set<int> si1;**

To create a **set***-of-***int** named **si2** using function *order* as the ordering rule, you can write:

**set<int,** *type-of-order***> si2(***order***);**

So using **gtr\_than**, you can write:

**set<int, bool (\*)(int,int)> si2(gtr\_than);**

Add the **gtr\_than** function definition and this declaration of **si2** to your program at the end of **main**, then use a range **for** loop on **si1** with an **insert** into **si2** so that after the loop **si2** contains the same values as **si1**. Use a range **for** loop on **si2** to display **si2**’s elements. Compile and test. Confirm that **si2**’s values are in descending order rather than ascending order.

1. Define a **typedef** named **pFbii** for *pointer-to* *function-taking* **int***-arg* *and* **int***-arg* *returning* **bool.**

Use your **pFbii** **typedef** to create a **set***-of-***int** in descending order named **si3**:

**set<int, pFbii> si3(gtr\_than);**

Write test code in **main** to confirm that **si3** stores **int** values in descending order. Compile and test. Would you say the **typedef** makes the code more clear, or less clear, or doesn’t matter?

1. Define an ordering function **even\_up\_odd\_down** that orders all even **int**s ahead of all odd **int**s, with even numbers ascending and odd numbers descending. Declare **si4** as a **set***-of-***int** in **even\_up\_odd\_down** order. Write test code in **main** to confirm that **si4** stores **int** values in the specified order. Compile and test.
2. **Employees, Managers, and the CEO (40 points)**

In this part of the homework, you will experiment with a simple inheritance hierarchy.

1. Create a new, empty Win32 Console Application project named **FCII\_HW1\_3**. In Source Files, add an existing item, **FCII\_HW1\_3.cpp**. This file contains the **Employee** and **Manager** classes described in Lecture 3, and a **main** function for testing. In this version of the code, the **Manager** *derived class* has an ***override*** of the **print()** member function, ***but*** **print()** is ***not*** declared **virtual** in the **Employee** base class.

Compile and test. Confirm that a **Manager** “is a kind of” **Employee**, and that it is okay to have a pointer-to-**Employee** point to a **Manager**. However, because **print()** in the base class is ***not*** declared **virtual**, the base class **print()** is called when a pointer-to-**Employee** contains the address of a **Manager** object.

1. Declare **print()** to be **virtual** in the **Employee** class. Compile and test. Confirm that because **print()** is virtual, **Manager::print()** is called when a pointer-to-**Employee** contains the address of a **Manager**. But also confirm that the **Employee::print()** can be called for a **Manager** via the **Employee::** prefix in front of **print()** (add appropriate test code in **main**, below the test code for part (a), above).
2. Since **id** in **Employee** is **const**, it is okay to make **id** public, since no code can change the value of **id** after **id** has been initialized. However, this is unconventional, and usually frowned upon. Move the **id** data member from the public section to the private section of **Employee**, and add a public **get\_id()** member function that returns an **Employee** object’s id number.

Uncomment this testing code in your **main** function; compile and test. (Debug if necessary.)

**Employee gene(10, "Gene", 68.25);**

**const Employee hank(12, "Hank", 67.0);**

**Manager ian(13, "Ian", 71.75, "Boss", 250000.0);**

**cout << "Gene's id: " << gene.get\_id() << '\n';**

**cout << "Hank's id: " << hank.get\_id() << '\n';**

**cout << "Ian's id: " << ian.get\_id() << '\n';**

1. The **print()** members of **Employee** and **Manager** are declared **const**, that is, these member functions promise not to change the **Employee** or **Manager** objects on which they are invoked. In **main**, there is a pointer-to-**Employee** variable, **p**. Uncomment this testing code in **main**. Should this work? Compile and test. Change the declaration of **p**, if necessary.

**p = &gene;**

**p->print();**

**p = &hank;**

**p->print();**

**p = &ian;**

**p->print();**

1. In the **Manager** class definition, declare **print()** to be **virtual**. Compile and test, to confirm that there is no change in behavior, since **Employee** (the base class) already has **print()** declared **virtual**.
2. Define a new class **CEO** as a kind of **Manager**, with all of the attributes and behaviors of a **Manager**, but also a **profit\_target** and a **bonus\_percentage**. Override **print()** for **CEO**, and add constructor, **set** and **get** member functions as needed. Uncomment this testing code in **main**. Compile and test. Make corrections and/or enhancements to the various classes as necessary.

**CEO joe(15, "Joe", 120.0,**

**// joe's title will be "CEO"**

**100000000.0, // budget**

**4000000.0, // profit target**

**250.0); // bonus percentage**

**joe.print();**

**cout << "Joe's id: " << joe.get\_id() << '\n';**

**cout << "Joe's rate: " << joe.get\_rate() << '\n';**

**cout << "Joe's title: " << joe.get\_title() << '\n';**

**cout << "Joe's budget: " << joe.get\_budget() << '\n';**

**cout << "Joe's profit target: "**

**<< joe.get\_profit\_target() << '\n';**

**cout << "Joe's bonus percentage: "**

**<< joe.get\_bonus\_percentage() << '\n';**

**// 10% raise**

**joe.set\_rate(joe.get\_rate() \* 1.1);**

**// 20% budget increase**

**joe.set\_budget(joe.get\_budget() \* 1.2);**

**// 10% profit target reduction**

**joe.set\_profit\_target(**

**joe.get\_profit\_target() \* 0.90);**

**// 30% increase in bonus percentage**

**joe.set\_bonus\_percentage(**

**joe.get\_bonus\_percentage() \* 1.3);**

**p = &joe;**

**p->print();**

***REMEMBER*** to put all authors’ names into each of your source code files. Put your source code files into a .zip archive, and upload to Canvas.