**MSCF Financial Computing II**

**Homework 2**

***Due At 12:30 pm Monday, Nov. 5, Before Lecture***

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

1. **(20 points) Simplifying Option Pricing Using Inheritance**

In this part of the homework, you will make use of inheritance to eliminate all the redundant code you had to copy-and-modify for the European call/put pricing with the Binomial Tree model.

1. Create a new, empty project named **FCII\_HW2\_1**. In Source Files, add an existing item, **PlainVanillaOption.cpp**. This file contains the solution code for the Binomial Tree pricing model for European calls/puts. The file also contains a new class definition, **PlainVanillaOption**, at the top, which is not yet being used.

Compile and test, to confirm that the program works as expected to price a European Call and a European Put.

1. If you examine the implementations of the European Call class and the European Put class, you will notice (or remember) that they contain mostly redundant code. In fact, the only difference between these classes is in how the terminal node option values are computed, and how the interior node option values are computed.

The **PlainVanillaOption** class extracts out what is common among these specific option classes, to make it convenient to eliminate all the redundant code, and to make it convenient to implement more kinds of options derived from the **PlainVanillaOption** class.

The **PlainVanillaOption** class has the initial asset (stock) price **S0**, the risk free rate **r**, asset volatility **sigma**, and expiration time **T** as data members. These data members all have to do with the evolution of the asset price over time, and keeping track of the time value of money, things that don’t vary from one kind of plain vanilla option to another. The **PlainVanillaOption** class ***does not*** have the strike price **K** as a data member, because not all options use a single strike price **K**. This is an option-specific feature.

Define the **put\_BinomialTree()** and **binomialPrice()** member functions for the **PlainVanillaOption** class. Instead of having option-specific formulas for the terminal node and interior node option values, the **PlainVanillaOption::binomialPrice()** function will have to call the **terminal\_node\_price()** and **interior\_node\_price()** functions that have been defined as ***pure virtual*** in the **PlainVanillaOption** class. This means that **PlainVanillaOption** is an ***abstract class***, and that **terminal\_node\_price()** and **interior\_node\_price()** will need concrete definitions in derived classes.

Since **PlainVanillaOption** is abstract, we have made its data members and its pure virtual functions **protected** so they are accessible in derived classes.

Modify the **EuropeanCall** class so that **EuropeanCall** ***is a kind of*** **PlainVanillaOption**, with a strike price **K**, and **EuropeanCall**-specific implementations of **terminal\_node\_price()** and **interior\_node\_price()**. Since EuropeanCall is a concrete class, **K**, **terminal\_node\_price()**, and **interior\_node\_price()** should be **private** within **EuropeanCall**. Modify the **EuropeanCallClass** constructor to pass **S0**, **r**, **sigma** and **T** arguments down to the **PlainVanillaOption** base class constructor.

Compile and test, to confirm that your implementations of **PlainVanillaOption** and **EuropeanCall** work correctly.

1. Modify **EuropeanPut** to be a kind of **PlainVanillaOption**. Compile and test, to confirm that your implementations work correctly.
2. **American Options (20 points)**

In this part of the homework, you will develop code for pricing “plain vanilla” American Call and American Put options. By “plain vanilla” we mean that the underlying asset pays no dividend, that the volatility of the underlying asset’s price movements are constant throughout time, and so forth. Unlike a European Call or Put, an American Call or Put can be exercised *at any time step prior to and including expiration time.*

In this part of the homework, you will add code to your existing **FCII\_HW2\_1** project.

1. The expiration time value of an American Call or Put—the expiration time payoff—is the same as the expiration time value of a European Call or Put, respectively. But since an American Call or Put *can be exercised at any time step*, the backward induction formula at the interior nodes of the Binomial Tree is different. For the European Call (ECi,j) we had:



For the American Call (ACi,j) we have:



That is, if St – K is greater than the value of continuing to hold the option, we will exercise early at time t. The formula for the AmericanPut is similarly changed.

Copy and modify your EuropeanCall and EuropeanPut class code to create AmericanCall and AmericanPut classes. Since most of the code is in the PlainVanillaOption base class, most of the work here will involve changing the payoff formulas.

As an alternative to the **//** and **/\* … \*/** styles of comments, another way to “comment out” code in C++ is to use **#if 0** … **#endif**, which we have done with this testing code for the American options:

AmericanCallOption ac1(50.0, // current stock price, S0

50.0, // option strike price, K

0.10, // risk-free rate

0.40, // stock price volatility

0.4167); // expiration time T (5 months)

cout << "Amer Call price, with " << 1000 << " intervals: "

<< ac1.binomialPrice(1000) << "\n";

AmericanPutOption ap1( 50.0, // current stock price, S0

50.0, // option strike price, K

0.10, // risk-free rate

0.40, // stock price volatility

0.4167); // expiration time T (5 months)

cout << "Amer Put price, with " << 1000 << " intervals: "

<< ap1.binomialPrice(1000) << "\n";

In **#if 0**, change the **0** to **1** so that the American option test code will be compiled into the program. Compile and test. Compare your computed prices for American options with those computed by others, to make sure your values are correct (at least, according to this model!). You can also find American Put/Call price calculators online.

1. **Separate Header and Code Files, and Macro Guards (20 points)**

In this part of the homework, you will split up your option pricing code file into appropriate separate header and code files, and use macro guards, discussed in Lecture 4.

1. In the **FCII\_HW2\_1** project, split up the code into seven separate files: **PlainVanillaOption.h**, containing the PlainVanillaOption class definition, protected by macro guards; **PlainVanillaOption.cpp**, that **#include**s **PlainVanillaOption.h** and defines the class member functions; **EuropeanOptions.h**, containing the EuropeanCall and EuropeanPut class definitions, protected by macro guards; **EuropeanOptions.cpp**, that **#include**s the needed header(s) and defines the class member functions; **AmericanOptions.h**, containing the AmericanCall and AmericanPut class definitions, protected by macro guards; **AmericanOptions.cpp**, that **#include**s the needed header(s) and defines the class member functions; and **main.cpp**, that **#include**s the needed header(s) and defines the **main()** function (from part (2) of the homework, above). Compile and test.

1. **Arithmetic Expressions (20 points)**

In this part of the homework, you will extend the Arithmetic Expression classes that we discussed in Lecture 5.

1. Create a new, empty project named **FCII\_HW2\_4**. In Source Files, add an existing item, **FCII\_HW2\_4.cpp**. This file contains the **Term**, **Constant**, **BinaryOp**, and **Plus** classes and the related **main()** function we looked at in Lecture 5.

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

1. Add classes **Minus**, **Times**, and **Divide**, with the obvious meanings. (You ***do not*** need to separate the various classes into separate **.h** and **.cpp** files, you can just write everything in **FCII\_HW2\_4.cpp**.) At the end of **main()**, below the statement

**delete p2;**

“uncomment” this test code:

**Term \*ca = new Constant(1.5);**

**Term \*cb = new Constant(2.7);**

**Term \*cc = new Constant(0.8);**

**Term \*cd = new Constant(3.3);**

**Term \*ce = new Constant(4.7);**

**Term \*m = new Minus(ce, cc);**

**Term \*d = new Divide(cd, ca);**

**Term \*t = new Times(d, cb);**

**Term \*p = new Plus(m, t);**

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

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

**delete p;**

Compile and test.

1. At the end of **main()**, write code that randomly generates arithmetic expressions with three operators. The form of each expression should be:

((c1 *op1* c2) *op2* (c3 *op3* c4))

where each c*n* is a constant and each *opN* is randomly replaced with one of **+**, **-**, **\***, or **/**. Each constant should be in the range **(0.0,5.0]**, which you can produce (good enough for our purposes) by using

**(rand() % 5000 + 1) / 1000.0**

An example of output for one arithmetic expression and its numeric value would be:

**((3.156 - 0.290) \* (2.831 / 4.106)) = 1.976**

***Your program should display ten such arithmetic expressions, with their numeric values.*** Be careful to **delete** the top (pointer to) **Term** of each expression after displaying it, to avoid memory leak. Compile and test.

1. Modify the **Term** and **Constant** classes to use **static** data members and **static** member functions to keep track of how many **Term** objects and how many **Constant** objects currently exist. You will need to modify the constructors/destructors for **Term** and **Constant**.

“Uncomment” this test code:

**{**

**cout << "# Terms: " << Term::get\_nterms()**

**<< '\n'; // should display 0**

**cout << "# Constants: " << Constant::get\_ncons()**

**<< '\n'; // should display 0**

**Term \*ca = new Constant(0.6);**

**Term \*cb = new Constant(-2.2);**

**Term \*cc = new Constant(0.7);**

**Term \*m = new Minus(ca, cb);**

**Term \*p = new Plus(m, cc);**

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

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

**cout << "# Terms: " << Term::get\_nterms()**

**<< '\n'; // should display 5**

**cout << "# Constants: " << Constant::get\_ncons()**

**<< '\n'; // should display 3**

**delete p;**

**cout << "# Terms: " << Term::get\_nterms()**

**<< '\n'; // should display 0**

**cout << "# Constants: " << Constant::get\_ncons()**

**<< '\n'; // should display 0**

**}**

Notice the use of a *nested block*, so that the variable names **ca**, **cb**, and so forth do not

conflict with the same variable names used earlier.

Compile and test.

1. **Algebraic Expressions (20 points)**
2. Create a new, empty project named **FCII\_HW2\_5**. In Source Files, add a ***new*** item, **FCII\_HW2\_5.cpp**. Copy-and-paste the code from part 4 of this homework, in **FCII\_HW2\_4.cpp**, into this new file, **FCII\_HW1\_2\_5.cpp**.

Compile and test, to confirm that the copy-and-paste operation was successful. Fix bugs as needed.

1. In part 4, we created arithmetic expressions like ((3 + 4) / (2 + 9)). We even were able to numerically evaluate expressions like this. Now we want to create *algebraic expressions* like ((X – 3.1) + (4.1 \* 2.2)), where the expression knows how to evaluate itself if given a value for X.

Of course, the tree representation of the above algebraic expression isn’t really different. It just has a new node type, the **Variable** node:

Turn the Arithmetic example into Algebra. Add a **Variable** class, representing an indeterminate. Like **Constant**, the **Variable** class constructor should take an initial value for the variable as its argument. The **Variable** class should also have a **set\_value(double)** member that allows the value of the variable to be changed. Its symbolic representation in **to\_string()** should be simply “**X**”. Its **to\_value()** member function should return the current value of the variable. Since **set\_value(double)** is only defined for the **Variable** class and not for any other kind of **Term**, you will need to use a *pointer-to* **Variable** to contain the address of a **Variable** object, rather than a *pointer-to* **Term**. That is:

**Variable \*pX = new Variable(5.5);**

**pX->set\_value(9.9); // works fine**

**Term \*pX2 = new Variable(4.56);**

**pX2->set\_value(7.89); // will not compile: Term does**

**// not have a set\_value member**

Add code at the end of **main()** to construct, display (using **to\_string()**), and evaluate (using **to\_value()**) the following expression, where variable X has an initial value of 1.5:

**((X + 3.3) / 0.7) - (4.2 \* 1.1)**

Then, set the value of X to 5.5 and evaluate the expression again. Don’t forget to **delete** the expression after you have evaluated it for the second time. Compile and test.

1. A variable might appear more than once in an expression. An example of a quadratic expression is:

**(X \* X) + ((2.2 \* X) + 3.3)**

Can our **Term** class hierarchy handle something like this? Add code to the end of **main** to test this, and explain what happens. (Add a comment in your code with your explanation.)

*Make sure all authors names are at the top of all source code files you created or modified for HW2. Put all these files in a* **.zip** *file and upload the* **.zip** *file to Canvas.*