FINM 326: Computing for Finance in C++

Lecture 7

Chanaka Liyanarachchi

February 17, 2023

Inheritance

Black Scholes Pricer

Software Testing

Roadmap

- OOP: Main Concepts (from L3.pdf):
  1. Classes and Objects ✓
  2. Data Abstraction and Encapsulation ✓
  3. Inheritance
  4. Polymorphism
- Applications:
  1. Components in an extensible option pricer:
     - types (European, American, Asian, ...)
     - assumptions (const vol, stochastic vol, ...)
     - models (Black Scholes, stochastic vol models, ...)
     - techniques (Analytical, Monte Carlo, Trees, ...)
  2. Electronic trading (brief look)

Restructuring Code

- In this course:
  - Build applications incrementally.
  - Continuously improve the designs by **restructuring** code as we add new features.
  - e.g. Currency-Converter: changes #1 – #12
- "I think it is cool to keep extending and improving already written code! It is very helpful." – Student

Continuous Improvements: Real Life

- How do we develop software in the industry?
  - We never complete a software projects all at once – software changes.
  - Adding new features and making improvements are common.
  - Changes involves restructuring code to make it easier to make the changes and use.
  - Restructuring is an important part in software development – known as *code refactoring*.
  - Refactoring often is a very good practice in software development.

Inheritance

Commonality Among Classes

- We often come across classes that have common members.
- We would like to exploit the commonality when we design the classes, to:
  1. Reuse code and improve maintainability.
  2. Improve extensibility.

- Why do we see commonality between classes?
- Let's look at a simple example: suppose we want to store info about everyone in this class room.
- We have two main types: *Students* and *Employees*.
- A student has certain attributes:
  - name
  - email
  - major
- An employee has certain attributes:
  - name
  - email
  - job

Example 1

- We can write a simple class to define the Student type:

```
class Student
{
public:
   Student(string name, string email, string major);

   string GetName();
   string GetEmail();
   string GetMajor();

private:
   string name_;
   string email_;
   string major_;
};
```

- And, we can write another simple class to define the Employee type:

```
class Employee
{
public:
   Employee(string name, string email, string job);

   string GetName();
   string GetEmail();
   string GetJob();

private:
   string name_;
   string email_;
   string job_;
};
```

- They are two different roles.
- They have some common members (data and function).
- This should not surprise us: A *Student* is a *Person*; an *Employee* is also a *Person*.
- An employee and a student share attributes and operations common to a person.
- The commonality between a *Student* and an *Employee* is *Person*.

Inheritance

- We use inheritance to *share* common members.
- We write common members in the *base class*.
- Common data members :
  - name_
  - email_
- Common member functions:
  - GetName();
  - GetEmail();

- We would write the *Person* base class:

```
class Person
{
public:
   Person(string name, string email);

   string GetName();
   string GetEmail();

private:
   string name_;
   string email_;
};
```

- All common members are in the base class.

- To represent the specific/specialized types (e.g. Student, Employee), we *derive* classes from the base class. We call them *derived* classes.
- A derived class *inherits* all members from the base class.
- A derived class may define additional members (data and/or function).

- We can define `Student` class using inheritance:

```
class Student : public Person
{
public:
   Student(string name, string email, string major);

   string GetMajor();

private:
   string major_;
};
```

- We can define Employee class using inheritance:

```
class Employee : public Person
{
public:
   Employee(string name, string email, string job);

   string GetJob();

private:
   string job_;
};
```

Base Class: Implemenation

- We can implement the Person class using what we know already.
- For example, we initialize the data members in the base class constructor as usual:

```
Person::Person(string name, string email)
   : name_(name),
     email_(email)
{}
```

- Implementions of the other member functions are pretty straight-forward.

Derived Class: Implemenation

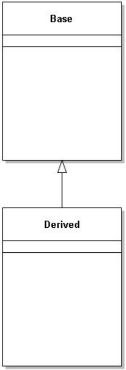
- Derived class constructor is slightly different: it uses the base class constructor to initialize data members of the base class.

```
Student::Student(string name, string email,
                 string major)
    : Person(name, email),
      major_(major)
{}


Employee::Employee(string name, string email,
                   string job)
    : Person(name, email),
      job_(job)
{}
```

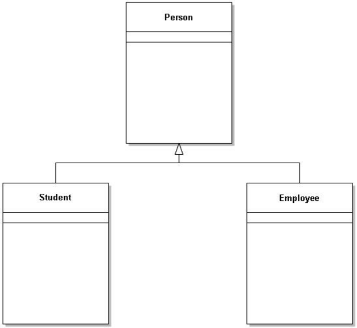
- Other member function implementations in Student and Employee derived classes are pretty straight-forward.

Class Diagrams

- We use class diagrams to illustrate relationships among classes graphically.
- Inheritance relationship between a base and a derived class is shown as:



- Inheritance relationship among classes in Example 1 is shown as:

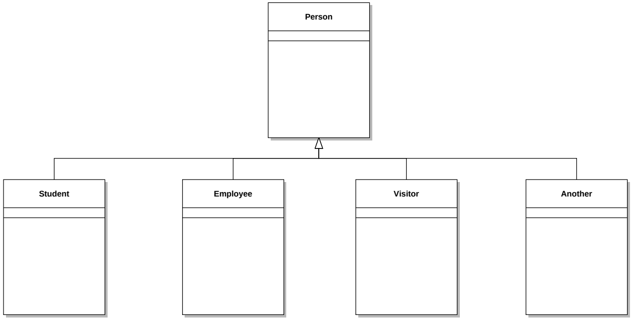


- We will use class diagrams as shown above when we discuss new designs.

Inheritance: Advantages

Inheritance clearly has some advantages:

- We can write common code in a base class – we don't need to repeat code.
- Adding a new type to the class hierarachy (i.e. a derived class) is easy:



  - Don't have to repeat code
  - Don't have to modify existing code
- This design is extensible.

Protection Levels

- A derived class can access members in the base class, subject to protection level restrictions.
- Protection levels `public` and `private` have their regular meanings in an inheritance class hierarchy:
  - A derived class can access `public` members of a base class.
  - A derived class cannot access `private` members of a base class.
- We can have `protected` members in the base class:
  - The members of the (base) class can access them.
  - Members of a derived class can access them.
  - Everyone else cannot access them.

Example 1: Cont..

- In our previous example (Example 1), data members in the base class were `private`.
- Which means, the `Student` class or the `Employee` class cannot access them directly.
- If the `Student` class or the `Employee` class needs to access the data members in the base class, we should mark them as `protected`.

```
class Person
{
public:
   string GetName();
   string GetEmail();

protected:
   string name_;
   string email_;
};
```

The virtual Keyword

- A base class uses the `virtual` keyword to allow a derived class to override (provide a different implementation) a member function.
- Meaning, if a function is virtual (in the base class):
  - The base class has to provide an implementation – it is known as the *default implementation* of that function.
  - A derived class (of that base class) inherits the function interface (definition) and the default implementation.
  - A derived class **can/may** override that function (implementation).

Example 2

- Consider the two classes below[1]:

```
class Base1
{
public:
  virtual void Fun1();

  virtual void Fun2();

  void Fun3();
};
```

- The `Base1` class has two virtual functions and one non-virtual function.

[1] see demo code for details

- The base class has to implement all 3 functions.
- An example implementation:

```
void Base1::Fun1()
{
   cout << "Base1::Fun1" << endl;
}

void Base1::Fun2()
{
   cout << "Base1::Fun2" << endl;
}

void Base1::Fun3()
{
   cout << "Base1::Fun3" << endl;
}
```

A `Derived` class of the Base1 class:

- Can override `Fun1()` and/or `Fun2()` .
- If it doesn't override `Fun1()` and/or `Fun2()`, it will inherit the default implementation from the `Base1` class.
- Should not override `Fun3()`.

We should not override inherited non-virtual functions.

- Shown below is a derived class of Base1:

```
class Derived1 : public Base1
{
public:
   void Fun1() override;
};

void Derived1::Fun1()
{
   cout << "Derived1::Fun1" << endl;
}
```

- In this case, the `Derived1` class:
  - Overrides `Fun1()`.
  - The `override` keyword is used to explicitly indicate this function overrides a virtual function – use of override is optional.
  - Inherits the default implementation of `Fun2()`.
  - Inherits the implementation of `Fun3()`.

Abstract Classes

- Sometimes we may not have a good default implementation for a virtual function in the base class (example next).
- We declare such functions **pure** `virtual` (in the base class).
- A pure virtual function is indicated using the syntax below:

```
class Base2
{
public:
   virtual void Fun1() = 0;
};
```

- The base class does not need to implement a pure virtual function.
- If a class has one or more pure virtual functions, it is called an **abstract class**.
- We cannot instantiate (i.e. create an object of) an abstract class.

Example 3

- `Derived2` class below provides an implementation[2] for `Fun1()`.

  ```
  class Derived2 : public Base2
  {
  public:
     void Fun1() override;
  };
  ```

- We can create an instance of `Derived2`.

[2] implementation not shown here; see demo code for details

Example 4

- `Derived3` class below does not provide an implementation for `Fun1()`, but just inherits the interface.

```
class Base3
{
public:
  virtual void Fun1() = 0;
};

class Derived3 : public Base3
{
public:
    void Fun2();
};
```

- `Derived3` is also an abstract class.

Black Scholes Pricer

Black Scholes Pricer

- Black Scholes model is used to price a European style options.
- Our objectives:
  1. Illustrate inheritance and other OOP concepts.
  2. Introduce some math functions.
- References:
  - http://en.wikipedia.org/wiki/Black%E2%80%93Scholes_model
  - John Hull. Options Futures and Other Derivatives.

Notation

We use the following notation:

S_t	:	Stock price at time t
σ	:	Volatility of the stock (constant)
r	:	Interest rate
T	:	Time to option expiration (in years)
K	:	Strike price
$N(x)$	:	CDF of the standard normal distribution[3]

[3] http://en.wikipedia.org/wiki/Cumulative_distribution_function

Background

- Black Scholes PDE describes the evolution of price process:

$$\frac{\partial V}{\partial t} + rS\frac{\partial V}{\partial S} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} = rV$$

- Call price:

$$V_{call} = S_0 N(d_1) - K \exp^{-rT} N(d_2) \tag{1}$$

- Put price:

$$V_{put} = K \exp^{-rT} N(-d_2) - S_0 N(-d_1) \tag{2}$$

where, N is the CDF of the standard normal distribution, and,

$$d_{1,2} = \frac{\log \frac{S \exp^{rT}}{K}}{\sigma\sqrt{T}} \pm \frac{\sigma\sqrt{T}}{2} \tag{3}$$

- How do we find $N(d_1)$ and $N(d_2)$?

Calculating N(x)

- N(x) here is the CDF of the standard normal distribution

$$N(x) = \int_{-\infty}^{x} \frac{\exp \frac{-z^2}{2}}{\sqrt{2\pi}} \tag{4}$$

- It doesn't have a closed form solution.
- We can rewrite it using the error function (erf)[4] as:

$$N(x) = \frac{1}{2}\Big(1 + erf(\frac{x}{\sqrt{2}})\Big) \tag{5}$$

[4] https://en.wikipedia.org/wiki/Error_function#Cumulative_distribution_function

The Greeks

- The price of an option depends on various parameters (e.g. stock price, interest rate etc.).
- The greeks (risk sensitivities) [5] measure the risk exposures of an option (or a portfolio of options) to factors that affect the option price.
- The greeks are partial derivatives of option price with respect to each parameter.
- Some commonly used greeks:
  - Delta: $\frac{\partial V}{\partial S}$
  - Vega: $\frac{\partial V}{\partial \sigma}$
  - Theta: $\frac{\partial V}{\partial t}$
  - Rho: $\frac{\partial V}{\partial r}$
  - Gamma: $\frac{\partial^2 V}{\partial^2 S}$

[5] http://en.wikipedia.org/wiki/Greeks_(finance)

The Greeks

Greeks for European Vanilla options are given by:

- Call Delta:

$$\frac{\partial C}{\partial S} = N(d_1) \tag{6}$$

- Put Delta:

$$\frac{\partial P}{\partial S} = N(d_1) - 1 \tag{7}$$

- Call Gamma:

$$\frac{\partial^2 C}{\partial^2 S} = \frac{N^{'}(d_1)}{S\sigma\sqrt{T}} \tag{8}$$

- Put Gamma:

$$\frac{\partial^2 P}{\partial^2 S} = \frac{N^{'}(d_1)}{S\sigma\sqrt{T}} \tag{9}$$

where[6],

$$N^{'}(x) = \frac{1}{\sqrt{2\pi}} \exp^{\frac{-x^2}{2}} \tag{10}$$

[6] http://en.wikipedia.org/wiki/Probability_density_function

C++ Implementation

C++ Implementation: Requirements

- Write OO program to:
  - compute option prices
  - compute greeks
- Initially we will consider Put and Call options.
- Design should be extensible.

Math Functions

- We need some math functions:
  - square root function
  - exponential function
  - log function
  - error function
  - π
- C++ supports some math functions:
  - `<cmath>` : `http://www.cplusplus.com/reference/cmath/`
  - `<cstdlib>`: `http://www.cplusplus.com/reference/cstdlib/`
- Until recently (before C++20), C++ did not define the value of π
  - We can define PI using:

    ```
    const double PI = atan(1.0) * 4;
    ```
  - Visual C++ provides this value as a non-standard feature: (`https://msdn.microsoft.com/en-us/library/4hwaceh6.aspx`)

Math Constants in C++20

- C++20 supports several math constants:
- `https://en.cppreference.com/w/cpp/numeric/constants`
- Defined in `<numbers>` in std::numbers namespace.

```
#include <numbers>

cout << std::numbers::pi << endl;
```

- Note: You need to set language standard to C++20. I showed how to do it when we discussed std::optional last week.

Example: CDF and PDF for Normal Distribution

- cdf() and pdf() implementations using some math functions:

```
double cdf(double x)
{
   return 0.5 * (1 + erf(x / sqrt(2)));
}

double pdf(double x)
{
   return exp(-0.5*pow(x, 2)) / sqrt(2 * PI);
}
```

Black Scholes Pricer: Class Design: Questions

- We want to price European Call and Put options.
- How do we represent them?
  - Q1) One class or two classes? and why?
  - Q2) What are the members (data/function)?

Class Members

- Suppose we want to use two classes (Call and Put).
- What are the data members (Attributes)?
  - strike
  - time to maturity
  - symbol [7]
- What are the member functions?
  - constructor
  - price()
  - greeks (e.g. delta(), gamma() etc.)
  - anything else?
- Price/greeks calculations require additional arguments:
  - underlying/stock price
  - rate
  - vol

[7] `https://en.wikipedia.org/wiki/Option_symbol`

- We need `d1` and `d2` for several calculations:
    - price
    - greeks
- Let's add `d1()` and `d2()` as class member functions.

- `EuropeanCall` class:

```
class EuropeanCall
{
   public:
     EuropeanCall(double K, double T);

     double Price(double S0, double r, double v);

     //greeks

   private:
      double d1(double S0, double r, double v);
      double d2(double S0, double r, double v);

      double K_;
      double T_;
};
```

- We use d1() and d2() internally; they are marked `private`.
- Class member functions that we don't want to expose to outside are marked as private.
- Note: symbol (OSI/underlying) is an important attribute of an option, but we won't show it in subsequent slides as it is not used in any computation.

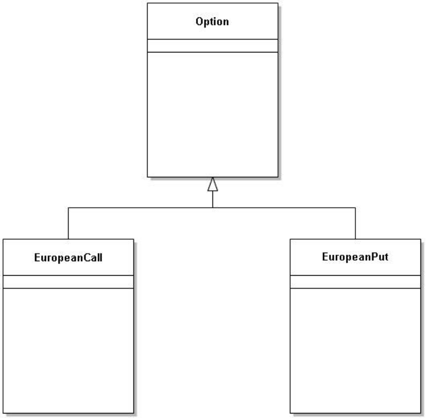
- EuropeanPut class:

```
class EuropeanPut
{
   public:
     EuropeanPut(double K, double T);

     double Price(double S0, double r, double v);

     //greeks

   private:
      double d1(double S0, double r, double v);
      double d2(double S0, double r, double v);

      double K_;
      double T_;
};
```

Using Inheritance: Black Scholes Pricer

- Call and Put options have common code:
  - `EuropeanCall` **is a**n Option.
  - `EuropeanPut` **is a**n Option.
- Let's redesign the option classes using inheritance.

Option Class Hierarchy



Option Base Class

What are the members of the base (Option) class?

1. Data members (strike, maturity):
   - common to all options
   - derived classes need to access them; use protected
2. `Price()` member function:
   - every derived class must support it
   - there's no default implementation that works for all option types
   - it has to be a pure virtual function:
     ```
     virtual double Price(double S0, double r,
                          double v) = 0;
     ```
3. `d1()` and `d2()`:
   - have common implementations
   - they are non virtual functions
   - derived classes need to use them; use protected

- Option base class:

```
class Option
{
   public:
     Option(double K, double T);

     virtual double Price(double S0, double r,
          double v) = 0;

   protected:
      double d1(double S0, double r,
          double v);

      double d2(double S0, double r,
          double v);

      double K_;
      double T_;
};
```

Derived Classes

Derived classes:

- Inherit `Price()` interface from the base class.
- Must to implement `Price()` in each derived class.

Derived Class: European Call

- `EuropeanCall` class[8] is derived from `Option` base class

```
class EuropeanCall : public Option
{
   public:
     EuropeanCall(double K, double T);

     double Price(double S0, double r, double v) override;
};
```

- Overrides `Price()`.

[8] incomplete

Derived Classes: European Put

- Similarly, the `EuropeanPut` class[9] is derived from `Option` base class:

```
class EuropeanPut : public Option
{
   public:
     EuropeanPut(double K, double T);

     double Price(double S0, double r, double v) override;
};
```

- Overrides `Price()`.

[9] incomplete

- Now you can use them to price options:

```
int main()
{
   //parameters
   double S0 = ...;
   double r = ...;
   double v = ...;
   double K = ...;
   double T = ...;

   EuropeanCall c1(K , T);
   cout << "Call price: " << c1.Price(S0, r, v)
      << ", Delta: "<< c1.Delta(S0, r, v)
      << ", Gamma: " << c1.Gamma(S0, r, v) << endl;

   EuropeanPut p1(K, T);
      cout << "Put price: " << p1.Price(S0, r, v)
      << ", Delta: " << p1.Delta(S0, r, v)
      << ", Gamma: " << p1.Gamma(S0, r, v) << endl;
}
```

New Design: Advantages

This new class design has several advantages:

1. Common code in the base class:
   - promotes reusability
   - promote maintainability
2. Adding new types is easy – extensible design:
   - no need to modify existing code to add a new Option type

Assignment 4 (Graded Assignment)

- Use inheritance to write an OO Black Scholes Pricer.
- It should support functions to calculate:
  1. option price
  2. delta
  3. gamma
- Using the pricer you wrote, find the price, delta and gamma of the options given below:
  - Call: $S_0 = 100$, K=100, T = 1, $\sigma = 0.3$, r = 0.05
  - Put: $S_0 = 120$, K=120, T = 2, $\sigma = 0.4$, r = 0.1
- Individual Assignment.
- Due: Friday, Feb 24 by midnight (CST).

Inheritance: More Examples

Examples that use inheritance are very common in real life:

1. Employees at a grocery store/shopping mall/restaurant (manager, security, cashiers)
2. Bank accounts (checking, savings, CD)
3. Geometric shapes (circle, rectangle, triangle)
4. Vehicles (car, truck)

Inheritance: Different Forms

- We can inherit in 3 different ways:
  1. `public`
  2. `private`
  3. `protected`
- We looked at `public` inheritance; it is the most commonly used form.
- We will not discuss the other two forms due to time constraints.
- Their (private/protected) usage is different.

Software Testing

Types of Tests

- Testing is an important part in software development:
  - make sure code works correctly
  - make sure they meet other requirements/specifications
- We use different types of tests in different stages of development:
  1. unit tests (`http://en.wikipedia.org/wiki/Unit_testing`)
  2. integration tests (`https://en.wikipedia.org/wiki/Integration_testing`)
  3. performance tests (discussed in summer HPC course)
- Each test type addresses unique areas/concerns – one type is not a substitute for the other.

Unit Testing

- Used to test individual units of code to make sure they work correctly.
- Remember, we use functions and classes as basic building blocks (units) to create large programs.
- If the basic units do not work correctly, the program as a whole will not work correctly.
- It is much easier to test small units than to test the whole program.

Unit Tests: Example 1

- Suppose we have two add functions:

```
int Add(int i, int j)
{
   return i+j;
}

double Add(double i, double j)
{
   return i+j;
}
```

- How do we test these functions?

- Test important cases and make sure the functions produce correct results.
  1. Add(2, 3) => should get 5
  2. Add(2, -3) => should get -1
  3. Add(0.2+0.3) => should get .5 $\pm\ \epsilon$ (where $\epsilon =$ tolerance)
  4. ...
- Similarly, we can test functions in the Black-Scholes pricer against some known values:
  1. d1, d2
  2. cdf: N(d1), N(d2)
  3. pdf

Unit Tests: Some Guidelines

1. Write one test to test one unit/function.
2. Keep the test short and simple:
   - run from top to bottom
   - avoid complex programming logic
3. Test things that are likely to fail:
   - don't write unnecessary tests that will pass all the time (e.g. get member functions)
   - add tests for things that have failed in the past
   - when a new bug is detected, add a new test

Unit Testing Frameworks for C++

- Software changes.
- Repeating manual-tests is time-consuming.
- Automation using a unit testing framework is the answer.
- Several good unit testing frameworks are available for C++:
    1. Microsoft Unit Testing Framework
    2. Boost.Test
    3. Google Test
    4. Catch2
    5. ...

- Using Microsoft Unit Testing Framework in Visual Studio is easy.
- CLion requires using a framework such as Boost, Google, or Catch.
  - Setting up the projects require some knowledge in cmake (CLion uses cmake to manage the build process).
  - See class demos for an example and additional resources for more info.
- You are not required you to write automated unit tests for assignments or exams.

Microsoft Unit Testing Framework

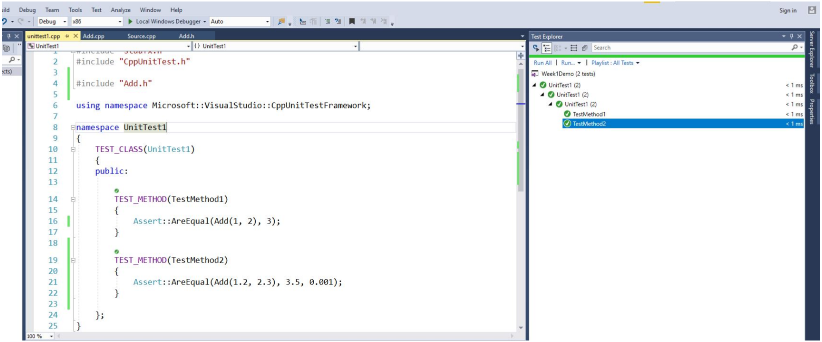
Unit Tests: Example 1

- Microsoft Unit Testing Framework example:

```
TEST_CLASS(UnitTest1)
{
  public:
    TEST_METHOD(TestMethod1)
    {
      Assert::AreEqual(Add(1, 2), 3);
    }

    TEST_METHOD(TestMethod2)
    {
      Assert::AreEqual(Add(1.2, 2.3), 3.5, 0.001);
    }
 };
```

- More info:
  https://docs.microsoft.com/en-us/visualstudio/test/how-to-use-microsoft-test-framework-for-cpp?view=vs-2022

- Using Microsoft Unit Testing Framework:



Catch/Catch2

Unit Tests: Example 2

- Catch2 Example:

```
#define CATCH_CONFIG_MAIN  //Using main() provided by Catch.
                           //Do this in one cpp file.

#include <catch.hpp>
#include <Add.h>

TEST_CASE("adding ints")
{
    REQUIRE(Add(1,2) == 3);
    REQUIRE(Add(11,21) == 32);
}

TEST_CASE("adding doubles")
{
    REQUIRE(Add(1.1,2.2) == Approx(3.3));
}
```

- More info:
  - https://www.jetbrains.com/help/clion/unit-testing-tutorial.html
  - https://github.com/catchorg/Catch2/blob/devel/docs/assertions.md

- Using Catch2 in CLion: