C++ fundamentals with use cases from finance

Lecture 2: Object Oriented Programming

Ivan Zhdankin

String

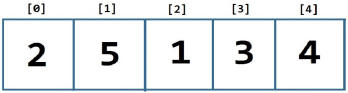
- One of the key type in C++ is *strings* - the collections of characters
- String can be words, sentences, user passwords and IDs
- C++ has a *class* which implements string
- To use stings we should include it:

```
#include <string>
```

- We can compare stings, combine them and perform other manipulation
- Can search for substring, swap and replace the substrings
- The various manipulation include:
  - To combine strings: $+, +=$
  - The member functions: *length*, *substr*, *find*
- Let us take a look at demo "String"

Vector

- Often we need to work with many similar items (of the same type)
  - Transactions in an account
  - Students in a class
  - Trades in a book
- In other words we have a *collection* of items
- We can perform different actions on the items in a collection:
  - To sum the items
  - To average the items
  - To sort the items
- The Standard Library provides different collections within it
- On of the simplest is a *vector*:
  - Holds a number of items of the same type
  - Size does not need to be known in advance
  - Easy to access a specific items or all of them



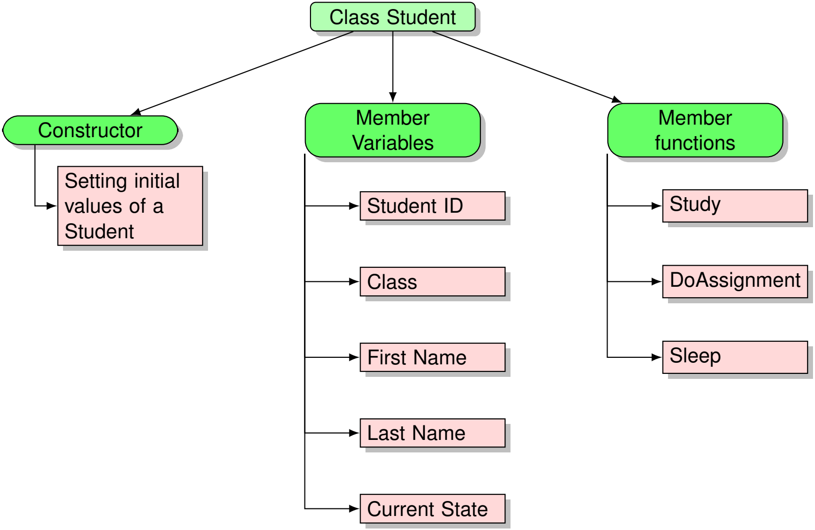
- Some operations with a vector:
  - *push_back*() to add the item
  - Type of the item added must match the type of items in a vector
  - To access the items we can use [] or iterators *begin*() and *end*()
  - Some free functions work with a vector: *count*() or *sort*()
- To use a vector we need to include it

```
#include <vector>
```

- Let us take a look at demo "Vector"

OOP - Object Oriented Programming

- C++ is object oriented programming language
- C++ is made up not just of functions but also of *Classes* and *Objects*
- A class defines the template for objects:
  - What the object contains?
  - What the object can do with anything it contains



Class and design

- An object is an *instance* of a class
- Function that are inside a class are called *member functions*
- The functions which are not part of a class (discussed so far) are called *free* or *non-member functions*
- When we write a class we tell to the compiler what any objects of the class have or can do
  - Example of data: a customer has a name, address, email, phone number
  - Example of member functions: we can withdraw or debit the account, we can show the balances and transactions
- Keeping the data and what we can do with it together inside a class makes the code easier to change and use
- Let us design a class of Bank account
- Account:
  - Contains account's ID number
  - Contains Current balance
  - Contains list of transaction objects
  - Can withdraw and debit some amount of cash
  - Can report the balances and the latest transactions
- Transaction:
  - Holds the date and time of the transaction
  - Holds amount and transaction type
  - Can report the amount and transaction type
- Member function Deposit:
  - Create a transaction object
  - Add the object to the vector of a transactions
  - Update the balances

Design into the code

- Generally, member variables are *private* inside the class
- The *private* members are not accessible outside of the class
- This is idea of *encapsulation* - the idea of bundling of data and what we can do with it within one unit
- Functions, you think are important, are normally *public* - the functions are *services* that a class can offer
- *Public* functions are accessible from outside of a class
- There are special member functions called *constructors*:
  - Constructors initialize variables
  - Name of the contractors is the same as of a class
  - Constructors do not have return type (in particular void)
  - Constructors can take parameters
- The use case for constructors: you should have the transaction amount at the initialization of the transaction and should not be able to change it
- Let us take a look at demo ”Classes”

Files

- So far we were using only one file for the source code with extension .cpp
- Imagine the big organization with code that has several thousands lines
  - Requires compiling it all when making small changes
  - Difficult to coordinate the work of the developers
  - Difficult to find anything in one single file
- In practise the code organised in multiple files to resolve those issues
- In case of multiple files one has to compile each of the files in to object file and then link them all using the linker
- If we use the other files in our code we need to explicitly tell the compiler about it

Header Files

- In previous implementation we had to declare the functions and classes before we can use them
- Consider the project that has tens of functions and classes, this may become frustrating
- We can put all of the declaration in a separate files and then include the file into the code we would like to compile
- The file is called *Header* file and has extension .h
- The directive to include the file would ne: *#include*
- Everything that starts with # will be included into the code by the *preprocessor* and after that the whole file will be compiled
- As a result your code would look nicer, be more maintainable and easier to understand

Organising the code

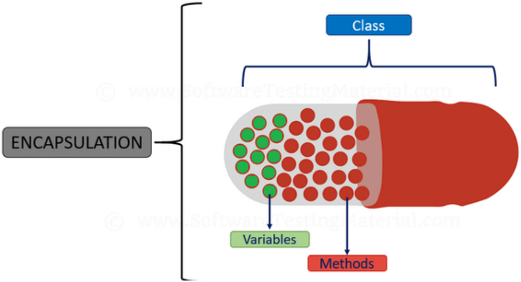
- Typical approach:
  - Have one header file per class to explain and declare what is in the class
  - Have one implementation file .cpp per class that implements all the functions of a class
- Any code that uses the class should include the header file
- We should also include the header file in .cpp file that implements the class
- We will have a look at demo ”Classes”

Inline Functions

- In one file (header file .h) we declare the functions and in another one (implementation file .cpp) we implement the functions of classes
- Some functions in a class are obvious
- Often the function work with private variables
- For such functions it makes sense to implement them in a header file
- These functions are called *inline*
- The advantages of using the inline functions is that it can speed up your application
- For example we can make *current_balance* function in Account class to be inline

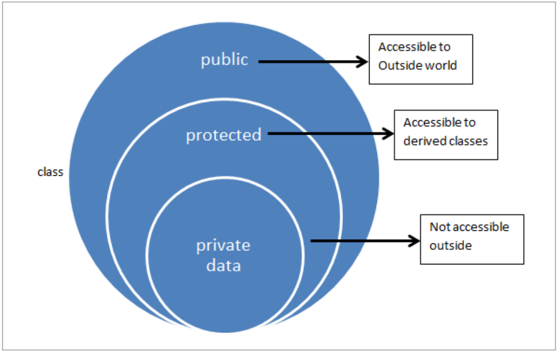
Encapsulation

- Encapsulation is about how changeable a class is
- By default the member variables should be private
- We can add public member functions that work with private variables
- In some sense such public functions are gatekeepers
- Add as few public functions as possible
- As a rule of thumb: the more encapsulated, the better



Access specifiers

- Any member of a class can have it is own access specifier that defines how the member can be accessed:



Creating Instances of a class

- A constructor that takes no arguments is called a *default constructor*
- Declare objects with default constructors the same as built in types:

```
Account    a1;
```

- Declare objects with some parameters using ():

```
Transaction    t1(10, "Deposit");
```

- Do not use $=$ when declaring an object and instantiating it

Constructors

- The objects are initialized by a special function called *constructor*
- The constructor takes parameters that can be used to initialize the member variables
- We also can initialize the member values to a default values
- The colon introduces initializers
- Do not forget the empty braces in case we use lazy initialization
- As a reminder from previous slides:
  - Name of the contractors is the same as of a class
  - Constructors do not have return type (in particular void)
- Demo: ”Constructors”

Scope

- So far we have seen the object created locally
- Each object has a lifetime:
  - When the line is reached the *constructor* is called and memory is allocated
  - Normally we say that the object is created on the *stack* (*stack* semantics)
  - The object is then has a *scope* - it lasts until the close the brace is encountered
  - When the close brace is reached, the memory for the object is freed and the another special functions is called - a *destructor*
- The resources are acquired when the constructors is called and are released when the destructors are called
- In case we do not specify the destructors - there are default destructors
- Demo: ”Scope”

Struct

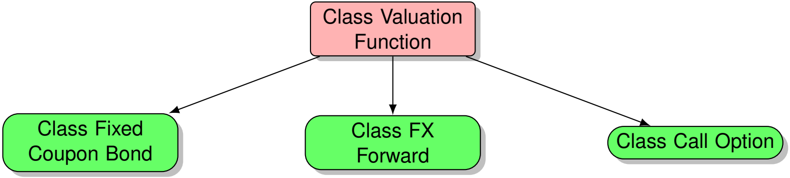
- One keyword to define your own type is *class*
- There is another keyword to define types - *struct*
- Historically the keyword was used to define plain data
- However the struct can have member functions, constructors and destructors and everything else that a class has
- The only difference is that if we do not specify the access specifiers, by default it is *public*
- However when we need some business logic and some member functions people use classes

Namespaces

- When you have a lot of code it is impossible to code it without *namespaces*
- They are designed to prevent name collisions
- You separate the namespaces using ::
- Example *std* :: *string*
- It enables me to have my own *string* class which is different from the standard string class
- We can code full name of the class we specified however there is a simpler syntax for this
- Demo: Namespaces

Inheritance

- Inheritance is key to OOP design
- In C++ we can have *Base* classes and *Derived* from them
- Derived classes can add or override member variables or functions
- As an example we can consider different valuation function:



Inheritance

- We are indicating with colon that we are inheriting from something else:

```
class Account : public Customer{}
```

- Inside the brackets we declare only what we are adding or overriding from the Base class
- If we override we provide a special implementation of something that is in a Base class
- In general the implementation of Derived classes is of no difference as compared to Base classes
- However the constructors are different: we need to pass some parameters to the base class for initialization

```
Account(string first, string last, int ID, string type)
Customer(first, last, ID), acctype(tp){}
```

- Demo: Inheritance

Enumerations

- Enumerations have keyword *enum* and give names to a set of constants
- For example you can have a function that return a status:
  - 1 if approved
  - 0 if cancelled
  - 2 if pending
- It is easy to forget what do these numbers mean
- We can give names to the numbers using *enumerations*
- Demo: Enum

PreProcessor

- Include statements starts with #
- This is a command for the another part of the building system, which is called *preprocessor*
- The preprocessor control what is compiled
  - Controls that a header s compiled inside the source files
  - Controls part of standard library to be compiled
- There a special command #*pragmaonce* used by the preprocessor
- If we include several header files, we can redefine some of the classes causing the compiler errors
- We do not want the thing to be included multiple times
- One of the way to get rid of this problem just to include #*pragmeonce* in every header filer and compile will cope with a problem for you

Free Functions

- So far we have seen different functions: free functions, constructors, member functions
- The following function takes the parameter by values

```
double total_balance(Account a1, Account a2)
```

- It creates its own copy of the variable to work with it inside the function
- If something is changed inside the function the variable itself is not changed in this case
- If is possible to take a parameter by reference

```
double total_balance(Account & a1, Account & a2)
```

- If we have a line of the parameter that changes it the function also changes the variable
- Demo: Free function

Member Functions

- We declare the member function in a class, we can also declare it in a header file
- If we implement the member function in cpp we use the full name:

$$Account :: GetName()$$

- We cam implement the function in a header file to make it inline
- Member functions should be mark as *const* unless it is going to change a member variable
- *const* member functions mean that they do not change any variable in a class
- This is useful information for both compilers and developers
- Demo: Member Function

Interview Questions

- Describe the concept of encapsulation?
- What is inline function?
- What is the difference between free and member functions?
- What is a class and object?

Interview Questions

- What are the constructors?
- What is the use of Inheritance in OOP?
- Can a class inherit from multiple classes in C++?

Interview Questions

- What is the output of the following program:

```
#include <iostream>
using namespace std;
class Rect{
int x, y;
public:
void set_values (int,int);
int area (){
return (x * y);
}
};
void Rect::set_values (int a, int b) {
x = a;
y = b;
}
int main (){
Rect recta, rectb;
recta.set_values (5, 6);
rectb.set_values (7, 6);
cout << "recta area: " << recta.area();
cout << "rectb area: " << rectb.area();
return 0;
}
```

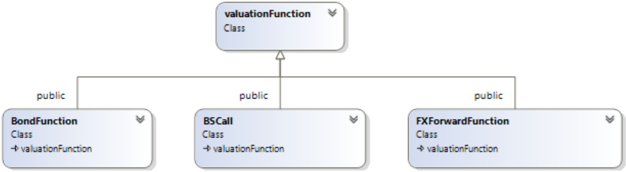
Interview Questions

- What is the output of the following program:

```
#include<iostream>
using namespace std;
class BaseClass1 {
public:
Base1()
{ cout << "␣BaseClass1␣constructor" << endl;  }
};
class BaseClass2 {
public:
Base2()
{ cout << "BaseClass2␣constructor" << endl;  }
};
class DerivedClass: public BaseClass1, public BaseClass
public:
Derived()
{  cout << "DerivedClass␣constructor" << endl;  }
};
int main()
{
Derived d;
return 0; }
```

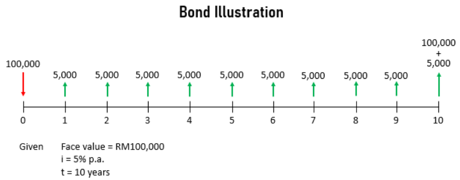
Closed Form Pricing

- Various methods can be applied to price instruments:
  - Monte-Carlo Simulation: suitable for exotic payoffs
  - Closed form: suitable for simple payoffs
- Closed form pricers are pricers that produce price from a function that implements a certain formula for pricing
- For scalability we can create class ”ValuationFunction” which serves as a template for all of the pricing functions



Implementing Bond Valuation Function

- The bond can be defined by the following attributes that can be taken into constructor:
  - Identifier
  - Nominal
  - Yield
  - FaceValue
  - Coupon Rate (Fixed)
  - Coupon Frequency
  - TTM



- Dirty Price includes accrued interest rate:

$$P_{dirty} = \sum_{i=0}^{i=9} e^{-yield * TimeToCoupon[i]} * coupon * faceValue$$

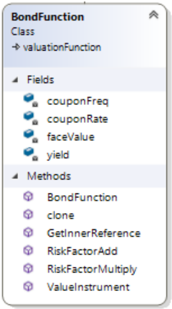
$$+ e^{-yield * TimeToCoupon[10]} * (1 + coupon) * faceValue$$

- $accruedInterest = timeToNextCoupon * coupon * faceValue$
- Clean Price does not include accrued interest rate

$$P_{clean} = P_{dirty} - accruedInterest$$

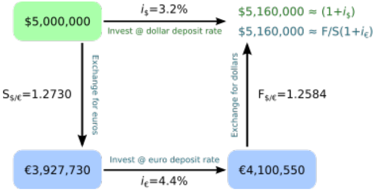
Implementing Bond Valuation Function

- Bond Valuation class structure



Implementing FX Forward Valuation Function

- The bond can be defined by the following attributes that can be taken into constructor:
  - Identifier
  - Nominal
  - SpotFXrate
  - $rate_{domestic}$
  - $rate_{foreign}$
  - Strike (ForwardFXrate at time zero)
  - TTM
- FXForward Valuation is implemented in ValueInstrument method



- The relationship between *SpotFXrate* and *ForwardFXrate* at the start of FX forward is:

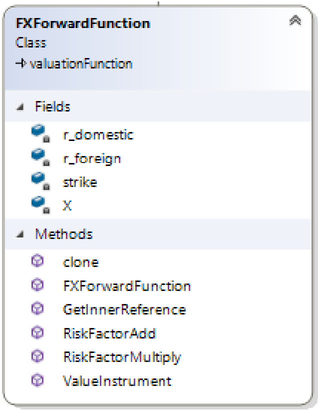
$$ForwardFXrate_0 = SpotFXrate_0 * e^{(rate_{domestic} - rate_{foreign})TTM_{factor}}$$

- That is the price of FX forward at time t is given by [John C. Hull "Options, Futures and Other Derivatives" for reference]:

$$SpotFXrate * e^{-rate_{domestic}TTM_{factor}} - Strike * e^{-rate_{foreign}TTM_{factor}}$$

Implementing FX Forward Valuation Function

- FX Forward Valuation class structure



Implementing Call Option Valuation Function

- The Call Option can be defined by the following attributes that can be taken into constructor:
  - Identifier
  - Nominal (N)
  - SpotPrice (S)
  - *interestRate* (r)
  - *dividendRate* (d)
  - impliedVol (σ)
  - Strike (K)
  - TTM (t)
- Call Option Valuation is implemented in ValueInstrument method
- The option price is derived from Black-Scholes model as follows:

$$P_{call-option} = N(Se^{-dt}N(d_1) - Ke^{-rt}N(d_2))$$

Where

- $N(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} \mathrm{e}^{-\frac{1}{2}z^2} dz$
- $d_1 = \frac{1}{\sigma\sqrt{t}} \left[\ln\left(\frac{S}{K}\right) + t\left(r + \frac{\sigma^2}{2}\right) \right]$
- $d_2 = \frac{1}{\sigma\sqrt{t}} \left[\ln\left(\frac{S}{K}\right) + t\left(r - \frac{\sigma^2}{2}\right) \right]$

Implementing Call Option Valuation Function

- Call Option Valuation class structure