Roxana Dumitrescu

C++ in Financial Mathematics

Presentation of the course

Which programming languages are being asked for in major financial centres?

C++ is considered to be "the hottest "programming language to know in the financial sector.

If you want a job (investment bank, hedge fund etc), you have to know C++!

Banks (and not only) are using C++...

“C++ is used in situations where speed is everything,” says the head of one bank.

Presentation of the course

Why C++ is so fast? Because it is a compiled language.

- Compiled languages are translated to the target machine's native language by a program called a compiler. This results in very fast code.
- Interpreted languages (R, Matlab, Python) are read by a program called an interpreter and are executed by that program. Interpreted languages are usually much slower than a compiled program.

“We've seen more requirements not just from the large banks, but also from high frequency trading...”, says the managing director of a financial centre.

Much of the code in banking is in C++, and **there is a perception that if you can do C++ you can do anything!**

Presentation of the course

Aim and course design

The aim of this course is to give an introduction to C++ programming and object oriented design with a special focus on applications in computational finance. The course is a combination of lectures and practical programming sessions.

Presentation of the course

This lecture is primarily designed for people who have never programmed before. It is mainly structured into two parts:

- The first part consists in a solid introduction to C++ programming, including object-oriented programming and generic programming. The theoretical notions will be accompanied by very intituitive examples!
- The second part will focus only on financial examples! We'll show how the features of C++ (learnt during the first part) can be used to solve real financial problems. We'll study different models and create programs which will answer the following important financial problem: How do you compute the price of complex financial products?

Presentation of the course

You'll mainly learn how to design programs in financial mathematics (for example, how to price call/put options in the Black-Scholes model either using the analytical formulas, either using Monte-Carlo methods, how to price a portfolio of derivatives etc.).

Presentation of the course

At the end of the lecture:

- You’ll have a deep knowledge of C++
- You’ll be able to create high quality programs in C++ in financial mathematics. What does high quality mean?
    - Easy updating, maintenance
    - Testing
    - Facility to reuse the code and to extend it rapidly
    - Flexibility
    - Ease of understanding and readability (even for large programs)
    - Scalability (software which continues to work with exponentially increasing data volume)

Presentation of the course

Suggestions

- Try to understand the lectures and try to do the exercises which are proposed during the practicals by yourself! It is impossible to learn a language without attempting to speak it!
- If there is something you don’t understand, don’t panic! You can contact me whenever you need a help!

Some References

- J. Armstrong, C++ in Financial Mathematics
- D. Brown, G. Satir, C++ The Core Language: A Foundation for C Programmers, Cambridge
- M. Capinscki, T. Zastawniak, Numerical Methods in Finance with C++, Cambridge.
- Daniel Duffy, Introduction to C++ for Financial Engineers: An Object-Oriented Approach, Wiley Finance.
- M. Joschi, C++ Design Patterns and Derivatives Pricing, Cambridge
- B. Stroustrup (Creator of C++), Programming: Principles and Practice Using C++. Pearson Education, 2014.
- B. Eckel, Thinking in C++, Vol.I and II, Prentice Hall.

One useful way to view C++ is as a collection of languages:

- the C language (the basic language, and pointers),
- an object-oriented programming language,
- A template programming language (generic programming).

Plan of the lecture

- Your first program in C++
- Basic Data types, Variables and Constants
- Operators
- Flow of control
- Functions

Your first program in C++

The first program is a program called "Hello world", which simply prints "Hello world" to your computer screen. Although it is very simple, it contains all the fundamental components that C++ programs have.

```
# include<iostream>
using namespace std;
int main()
{
cout<<"Hello world!";
return 0;
}
```

Your first program in C++

```
# include<iostream>
```

This statement is called an include directive. It tells the compiler and the linker that the program will need to be linked to a library of routines that handle input from the keyboard and output to the screen (specifically the cin and cout statements that appear later). The header file "iostream" contains basic information about this library. You will learn much more about libraries of code later in this course.

Your first program in C++

```
using namespace std;
```

This statement is called a using directive. The latest versions of the C++ standard divide names (e.g. cin and cout) into subcollections of names called namespaces. This particular using directive says the program will be using names that have a meaning defined for them in the std namespace (in this case the iostream header defines meanings for cout and cin in the std namespace).

Your first program in C++

```
int main();
```

The function named *main()* is a special function in all C++ programs; it is the function called when the program is run. The execution of all C++ programs begins with the *main* function!

Lines 3 and 6: { and }
The open brace ({) at line 3 indicates the beginning of *main*'s function definition, and the closing brace (}) at line 6 indicate its end. Everything between these braces is the function's body that defines what happens when *main* is called.

The "return 0;" statement in main indicates the program worked as expected without any error during its execution.

Your first program in C++

Line 4: cout<<"Hello World!";

This line is a C++ statement. Statements are executed in the same order that they appear within a function's body.

This statement has three parts:

- cout, which identifies the standard character output device (usually, this is the computer screen). If you don't write the using directive, you have to write std::cout.
- the insertion operator $<<$ which indicates that what follows is inserted into cout.
- Finally, a sentence within quotes ("Hello world!") is the content inserted into the standard output.

In C++, the separation between statements is specified with an ending semicolon (;).

Basic data types and variables

When you use a variable in C++, you must specify the type of data that will be stored in that variable. Once you have chosen the type of data to be stored in a given variable you can't change it. The jargon phrase is that it is a *statically typed language*.

Why is it important to specify the type of data?

Because the data is stored in the computer data as strings of 1s and 0s. The integers are encoded as binary numbers, characters are represented as numbers written in binary etc.

Basic data types and variables

Memory terminology

Because data is stored on computers using 1s and 0s, it is natural to store integers using binary.

- A single binary digit is called *a bit* (truncation of binary digit).
- 8 binary digits are called *a byte*.
- $1024 (= 2^{10})$ bytes make a kilobyte.
- $1048576 (= 2^{20})$ bytes make a megabyte.
- Numbers representing memory locations are often written in hexadecimal. The base is 16. The 16 characters are $0, 1, 2, 3..., 9, A, B, ..., F$.

Basic data types and variables

Variables

- We can assign symbolic names, known as **variables**, for storing information in the memory. Variables can be used to store floating-point numbers, characters and even pointers to other locations in memory.
- The variables must be declared before using them.

Declaration:

```
Type_of_data Name_Variable;
```

Basic data types and variables

Build-in Data Types: Types and keywords

- **Boolean** $\mapsto$ **bool**
- **Character** $\mapsto$ **char**
- **Integer** $\mapsto$ **int**
- **Floating point** $\mapsto$ **float**
- **Double floating point** $\mapsto$ **double**

- The **float**, **double** are used to store real numbers. Note that C++ considers the numbers 1.0 and 1 to have different types and so to be different.
- **Bool** is a variable which can take the value either false or true.

Basic Data Types and variables

- A **char** is a data type which is intended to store a character.

  There is an important technical point concerning data of type "char". In memory a **char** is stored as a number between 0 and 255 and, consequently, takes up exactly one byte. Hence the data of type "char" is simply a subset of the data type "int". We can even do arithmetic with characters. For example, the following expression is evaluated as true on any computer using the ASCII character set:

```
'9'-'0'==57-48==9
```

The ASCII code for the character ’9’ is decimal 57 and the ASCII code for the character ’0’ is decimal 48.

Basic data types and variables

However, declaring a variable to be of type "char" rather than type "int" makes an important difference as regards the type of input the program expects, and the format of the output it produces.

```
int main()
{       int number;
char character;
cout<<"Type in a character: \n";
cin>>character;
        number=(int)character;
cout<<"The character is "<<character;
cout<<"is represented as the number ";
cout<<number<<" in the computer. \n";
return 0;}
```

Basic data types and variables

C++ gives you a number of choices for storing integer data depending on the potential range of values your variable might take. We give below other data-type specifiers that are available, which all mean an integer of one form or another:

- signed
- unsigned
- short
- long
- long long

For the ploating-point types: long double.

Basic data types and variables

You can store smaller numbers in shorts than you can in long longs.
You can be sure that you'll be able to store a value between -2^{31} and $2^{31} - 1$ in an **int** variable and a value between -2^{63} and $2^{63} - 1$ in a long long variable.
By specifying that a variable is **unsigned**, you are saying that it is nonnegative. Specifying the sign of a number takes up one bit of memory.

Basic data types and variables

In order to get the size of various data types, use the **sizeof()** operator.

```
int main() { cout << "Size of char : " <<
   sizeof(char) << endl;
       cout << "Size of int : " << sizeof(int)
   << endl;
       cout << "Size of short int : " <<
   sizeof(short int) << endl;
   cout << "Size of long int : " <<
   sizeof(long int) << endl;
   cout << "Size of float : " << sizeof(float)
   << endl;
   cout << "Size of double : " <<
   sizeof(double) << endl;
   return 0;}
```

Basic data types and variables

An example

```
int main()
{cout<<"Please enter your age (followed by
   "<<" 'enter'):\n";
int your_age;
cin>>your_age;}
// note how several values can be output by a
   single statement\\
// a statement that introduces a variable is
   called a declaration\\
// a variable holds a value of a specified
   type\\
```

Basic data types and variables

Comments

- We read into a variable
  - Here, your_age
- A variable has a type
  - Here, int
- The type of a variable determine what operations we can do on it

Basic data types and variables

Casting

- Sometimes it is important to guarantee that a value is stored as a real number, even if it is in fact a whole number. A common example is where an arithmetic expression involves division. When applied to two values of type **int**, the division operator "/" signifies integer division, so that (for example) $7/2$ evaluates to 3. In this case, if we want an answer of 3.5, we can simply add a decimal point and zero to one or both numbers - "7.0/2", "7/2.0" and "7.0/2.0" all give the desired result.
- If both the numerator and the divisor are variables, this trick is not possible. Instead, we have to use a type **cast (= convert between one data type and another).** For example, we can convert "7" to a value of type *double* using the expression "static_cast<double>(7)".

Basic data types and variables

Casting

Hence in the expression

```
answer=static_cast<double>(numerator)/denominator
```

the "/" will always be interpreted as real-number division, even when both "numerator" and "denominator" have integer values.

Other type names can also be used for type casting. For example, "static_cast<int>(14.35)" has an integer value of 14.

Basic data types and variables

Some tips on formatting real number output When program

output contains values of type **float**, **double**, **long double**, we may wish to restrict the precision with which these values are displayed on the screen, or specify whether the value should be displayed in fixed or floating point form.

```
int main()
{   float number;
     cout<<"Type in a real number. \n";
     cin>>number;
     cout.setf(ios::fixed);
     cout.precision(2);
     cout<< "The square root of "<< number<<
   "is approximately";
     cout<<sqrt(number)<< "\.n";
     return 0;}
```

Basic data types and variables

Some tips on formatting real number output

Output

Type in a real number.

200

The square root of 200.00 is aproximately 14.14.

whereas replacing line 5 with "cout.setf(ios::scientific)" produces the output:

Type in a real number.

200

The square root of $2.00e+02$ is approximately $1.41e+01$.

In scientific notation all numbers are written in the form $p * 10^n$, where p is called the significand or mantissa. In the above example, $p = 1.41$ and $n = 1$.

Basic data types and variables

You can assign an alternative name to a data type, using **typedef**.

```
typedef int Integer;
```

We can now declare variables to be of type "Integer":

```
typedef int Integer;
Integer i,j,k;
```

It is often used in order to simplify the declaration of compound types such as the struct type (which we'll see later).

Basic data types and variables

Later in the course we will study the topic of data types in much more details. We will see how the programmer may define his or her own data type. This facility provides a powerful programming tool when complex structures of data need to be represented and manipulated by a C++ program.

Basic data types and variables

- We have seen that **variables have to be declared before they can be used in a program**, using statements such as:

```
float number;
```

- Between this statement and the first statement which assigns "number" an explicit value, the value contained in the variable "number" is arbitrary. **In C++ is possible to initialise variables with a particular value at the same time as declaring them.** We can thus write:

```
double PI= 3.1415926535;
```

Constants

- We can specify that a variable's value cannot be altered during the execution of a program using the reserved word **const**.

```
const double PI= 3.1415926535;
const char tab = '\t';
const zip = 12440;
```

Enumerations

Constants of type "int" may also be declared with an enumeration statement.

```
enum{MON, TUES, WED, THURS, FRI, SAT, SUN};
```

can be seen as

```
const int MON=0;
const int TUES=1;
const int WED=2;
const int THURS=3;
const int FRI=4;
const int SAT=5;
const int SUN=6;
```

Enumerations

By default, members of an "enum" list are given the values $0, 1, 2$ but when "enum" members are explicitly initialised, uninitialised members of the list have values that are one more than the previous value on the list:

```
enum{MON=1, TUES, WED, THURS, FRI, SAT=-1,
    SUN};
```

In this case, the value of "FRI" is 5, and the value of "SUN" is 0.

Operators

- Assignement operator
- Arithmetic operators
- Compound operators
- Relational operators
- Logic operators

Operators

We can operate on variables and constants using **operators.**

- **Assignment operator (=)**
  - The assignement operator assigns a value to a variable.

```
x=5;
```

The assignement operator assigns to a variable *x* the value contained in variable *y*.

```
x=y;
```

The value of *x* at the moment this statement is executed is lost and replaced by the value of *y*.

Operators

- If we assign the value of *y* to *x* and *y* changes at a later moment, it will not affect the new value taken by *x*.
- The following assignement

```
y=2+(x=5);
```

is equivalent to:

```
x=5;
y=2+x;
```

Operators

- **Arithmetic operators (+,-,*,/, %)** The arithmetical operations in C++ are:
  - Operator $+$: addition
  - Operator $-$: subtraction
  - Operator $*$: multiplication
  - Operator $/$: division
  - Operator %: modulo

  The last operator gives the remainder of a division of two values.

Operators

- **Compound assignment (+=,-=,*=,/=,%=)** Compound assignment operators modify the current value of a variable. They are equivalent to assigning the result of an operation to the first operand:
  - The *expression* $y+=x$; is *equivalent to* $y=y+x$;
  - The *expression* $y-=5$; is *equivalent to* $x=x-5$;
  - The *expression* $x/=y$; is *equivalent to* $x=x/y$;
  - The *expression* $x*=y$; is *equivalent to* $x=x*y$;

Operators

- **Increment and decrement (++,–)**
  The increase operator $(++)$ and the decrease operator $(--)$ increase or reduce by one the value stored in a variable. The following three statements are equivalent:

```
++x;
x+=1;
x=x+1;
```

  The following three statements are also equivalent:

```
x++;
x+=1;
x=x+1;
```

Operators

- **Increment and decrement (++,–)**
  These operators can be used both as a prefix (++x) and as a postfix (x++)!

  When an increment or decrement is used as part of an expression, there is an important difference between prefix and postfix forms. If you are using prefix form then increment or decrement will be done before rest of the expression, and if you are using postfix form, then increment or decrement will be done after the complete expression is evaluated.

Operators

Increment operator $++$ as prefix/postfix

```
x=3;
y=++x;
// x contains 4, y contains 4
```

```
x=3;
y=x++;
// x contains 4, y contains 3
```

In *Example 1*, the value assigned to *y* is the value of *x* after being increased. In *Example 2*, *y* has (after assignment) the value that *x* had before being increased.

Relational/logical Operators and boolean expressions

- Intuitively, we think of expressions such as $''2 < 7'', ''1.2! = 3.7''$ and $''6 \geq 9''$ as evaluating to "**true**" or "**false**" ($! =$ means not equal to).
- Such expressions can be written using **the relational and comparison operators** ("==", "!=", ">", ">=", <, "<=") and combined using the **logical operators** "&&" (**"and"**), "||" (**"or"**) and "!" (**"not"**).

Relational/logical Operators and boolean expressions

Logical operators

- *Operator "&&"*: Called Logical AND operator. If both operands are non-zero, then condition becomes true.

  If A holds 1 and B holds 0, then

```
(A&&B) is false.
```

- *Operator "||"*: Called Logical OR operator. If any of the two operands is non-zero, then condition becomes true.

  If A holds 1 and B holds 0, then

```
(A||B) is true.
```

Relational/logical Operators and boolean expressions

Logical operators

- *Operator "!"*: Called Logical NOT operator. Use to reverse the logical state of its operand. If a condition is true, then logical operator NOT will make it false.

  If A holds 1 and B holds 0, then

```
!(A||B) is false.
```

Relational/logical Operators and boolean expressions

Examples

Expression	**True or False**
$(6 \leq 6)$ && $(5 < 3)$	**False**
$(6 \leq 6)$ || $(5 < 3)$	**True**
$(5! = 6)$	**True**
$(5 < 3)$ && $(6 \leq 6)$ || $(5! = 6)$	**True**
$(5 < 3)$ && $((6 \leq 6)$ || $(5! = 6))$	**False**
$!((5 < 3)$ && $((6 \leq 6)$ || $(5! = 6)))$	**True**

Relational/logical Operators and boolean expressions

- The forth of these expressions is true because the operator && has a higher precedence than the operator || as it has a behavior on booleans close to * and +. If is doubt, use ( ) parentheses.
- Compound Boolean expressions are typically used as the condition in "if statements" and "for loops".

Example

```
if  ((total_test_score >= 50) &&
   (total_test_score < 65))
cout<<"You have just scraped through the
   test.\n";
```

Relational/logical Operators and boolean expressions

- **Conditional ternary operator (?)** The conditional operator evaluates an expression, returning one value if that expression evaluates to true, and a different one if the expression evaluates as false. Its syntax is:

```
condition ? result1 : result2
```

  If condition is true, the entire expression evaluates to result1, and otherwise to result2.

Summing up

- General structure of a C++ program
- Basic types of data, casting
- Variables, constants: need to be declared before they are used

```
int x;
x=5;
```

A const can't be changed!

```
const double y=5;
y=7; // ERROR
```

- Boolean expressions and operators (assignment, arithmetic operators, compound operators, logic operators (AND, OR, NOT)), conditional ternary operator.