

Exporting C++ functions to Excel

Version 3, Antoine Savine, September 2018

Introduction

To expose C++ functions in Excel is a powerful tool in use today in most if not all financial institutions.

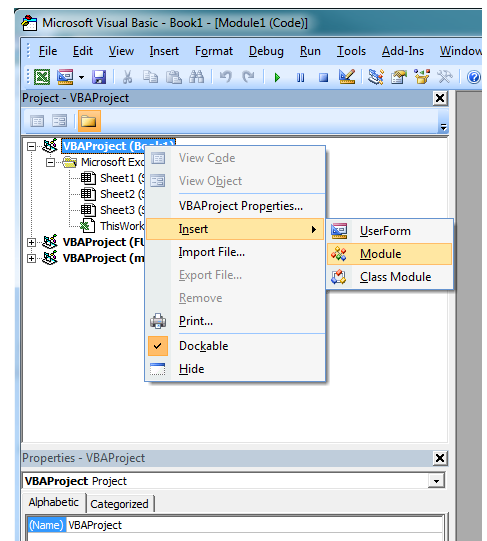
C++ is a strong standard for the development of financial derivatives libraries. C++ is a high performance language, “close to the metal”, ideal for the efficient implementation of complex models and numerical methods. C++ is also object oriented and offers a powerful template environment, ideal for the design of large, consistent, versatile financial libraries. C++ ships with standard libraries that provide efficient implementation of standard data structures and algorithms, as well as, since C++11, libraries for memory management, functional programming and parallel computing. It is ideally suited for the implementation of financial calculations.

Excel, on the other hand, provides an ideal front-end for the calculation and the visualization of results. It ships with standard functions, such as basic linear algebra and statistics, but nothing related to derivatives, not even Black & Scholes’ formula. Fortunately, Excel ships with a programming language for the development of custom functions, Visual Basic for Applications, or VBA. VBA allows Excel users to implement their own functions and then use them in Excel just like native Excel functions. For instance, the Black-Scholes formula may be implemented in VBA as follows:

1. From Excel, press Alt+F11 to open a VBA window.
2. In the upper left section of the VBA window, right-click on your Excel workbook and select insert/module.
3. In the central section of the VBA window, code your Black-Scholes function:

```
Function blackScholes(spot, vol, mat, strike)
    std = Sqr(mat) * vol
    halfVar = 0.5 * std * std
    d1 = (Log(spot / strike) + halfVar) / std
    d2 = d1 - std
    blackScholes = spot * Application.NormSDist(d1) - strike * Application.NormSDist(d2)
End Function
```

4. Go back to Excel. You should now see your function under the category “User Defined” and you may use it just like any other



Excel function. You may call it with various sets of inputs and chart Black-Scholes values against spot and volatility, for instance. (To see all available functions, click the *fx* symbol on the right of the formula bar in Excel).

5. Say we want to compute an implied volatility, the volatility that must be input in Black-Scholes to hit the price of some call option knowing the underlying spot price. One solution is Excel's built-in goal seek or solver in conjunction with the VBA implementation of Black-Scholes. Another solution is code our own implied volatility function, for instance with a basic bisection algorithm:

```
Function iVol(spot, price, mat, strike)
    ub = 2
    lb = 0.0001
    While ub - lb > 0.0001
        mb = 0.5 * (ub + lb)
        bs = blackScholes(spot, mb, mat, strike)
        If bs > price Then
            ub = mb
        Else
            lb = mb
        End If
    Wend
    iVol = 0.5 * (ub + lb)
End Function
```

6. Now we have another function that we may use in Excel. We can download option price data for different strikes into Excel and compute and visualize a “smile”: the implied volatility for different quoted strikes. We may even use Excel's linear regression tool to estimate the skew: the slope of implied volatility as a function of the strike.

This is all very nice, but VBA is not a well suited programming language for financial calculations. It may be enough for simple analytical formulas and basic algorithms, but it is far too slow for more complex algorithms like Monte-Carlo Simulations, Finite Difference Methods or calibration. It is not robust enough for the design of large financial libraries. And it is not well enough structured and not close to the metal enough for the development of advanced risk management features like scripted payoffs or adjoint differentiation. For all of these, we need C++.

To calculate 5,000 implied volatilities in VBA takes over 2sec on a standard workstation, which may seem acceptable at first sight. If we try a (slightly) more advanced algorithm, like a (very) basic Monte-Carlo simulation in Black-Scholes:

```
Function bsMonteCarlo(spot, vol, mat, strike, nPaths, nSteps)

    Rnd (-1)
    Randomize (12345)

    dt = mat / nSteps
    sdt = vol * Sqr(dt)
    halfVar = 0.5 * sdt * sdt
    logS0 = Log(spot)
    result = 0

    For n = 1 To nPaths

        logS = logS0

        For i = 1 To nSteps
            logS = logS - halfVar + sdt * Application.NormSInv(Rnd())
        Next i

        spotAtT = Exp(logS)
        If spotAtT > strike Then
            payoff = spotAtT - strike
        Else
            payoff = 0
        End If

        result = result + payoff

    Next n

    bsMonteCarlo = result / nPaths

End Function
```

(where *Rnd()* is a built-in VBA function that returns random numbers in (0,1), and *Application.NormSInv()* is Excel's implementation of the inverse cumulative Normal distribution, so repeated calls to *Application.NormSInv(Rnd())* return independent standard Gaussian numbers.)

We find that a calculation with 10,000 paths and 60 steps takes over 15sec, which is unacceptable by production standards. Such a simple simulation should complete in a few hundredths of a second.

What we want is write custom Excel functions in C++, not VBA. Fortunately, this is doable, if somewhat more involved.

We can even do much more than just return a scalar result out of a number of scalar inputs. We can export functions that take as arguments and return as results strings, vectors or matrices, and even ranges of numbers mixed with strings. We can call C++ functions that create a data structure and store it in Excel's memory for subsequent use by other functions. The possibilities are vast. The whole story is

told in Steve Dalton's 1997 book *Financial Applications Using Excel Add-in Development in C/C++*. In this tutorial, we will be barely scratching the surface of possibilities. We will learn to export C++ functions to Excel and use them as Excel's native functions, like we just did with VBA. We start with functions that take a number of scalar arguments and return a scalar result. Then, we will learn to communicate whole ranges of numbers: vectors and matrices, both ways, between C++ and excel. Finally, we will learn to communicate ranges that contain not only numbers, but also strings.

You will need Excel 2007+ 32bit and Visual Studio to complete the tutorial and exercises. We recommend Visual Studio 2017 "Community Edition", which may be downloaded for free from Microsoft's web site. If you already have another version installed, both can work side by side without interference. Make sure to install all C++ tools during installation, and configure VS2017 for a C++ development environment when the installer asks.

Our first Excel C++ function

It is customary to start programming tutorials with something that displays "Hello World", however, to communicate strings between C++ and Excel is somewhat more complicated than numbers. For this reason, we begin with a sample function that multiplies two numbers and returns the result.

You will need a number of files collected in the folder "cppFiles", which you normally downloaded along with the tutorial. If not, they may be downloaded from my public dropBox folder: tinyurl.com/ant1savinePub, Password = 54v1n3, folder /Vol/xlCpp/cppFiles/. This folder contains the following files that you will need to export your functions to Excel, reproduce the examples and complete the exercises:

- `xlcall.cpp`, `xlcall.h`, `framework.h`, `MemoryManager.cpp`, `memorymanager.h`, `MemoryPool.cpp` and `MemoryPool.h` are part of Microsoft's Excel SDK, which can be downloaded separately from

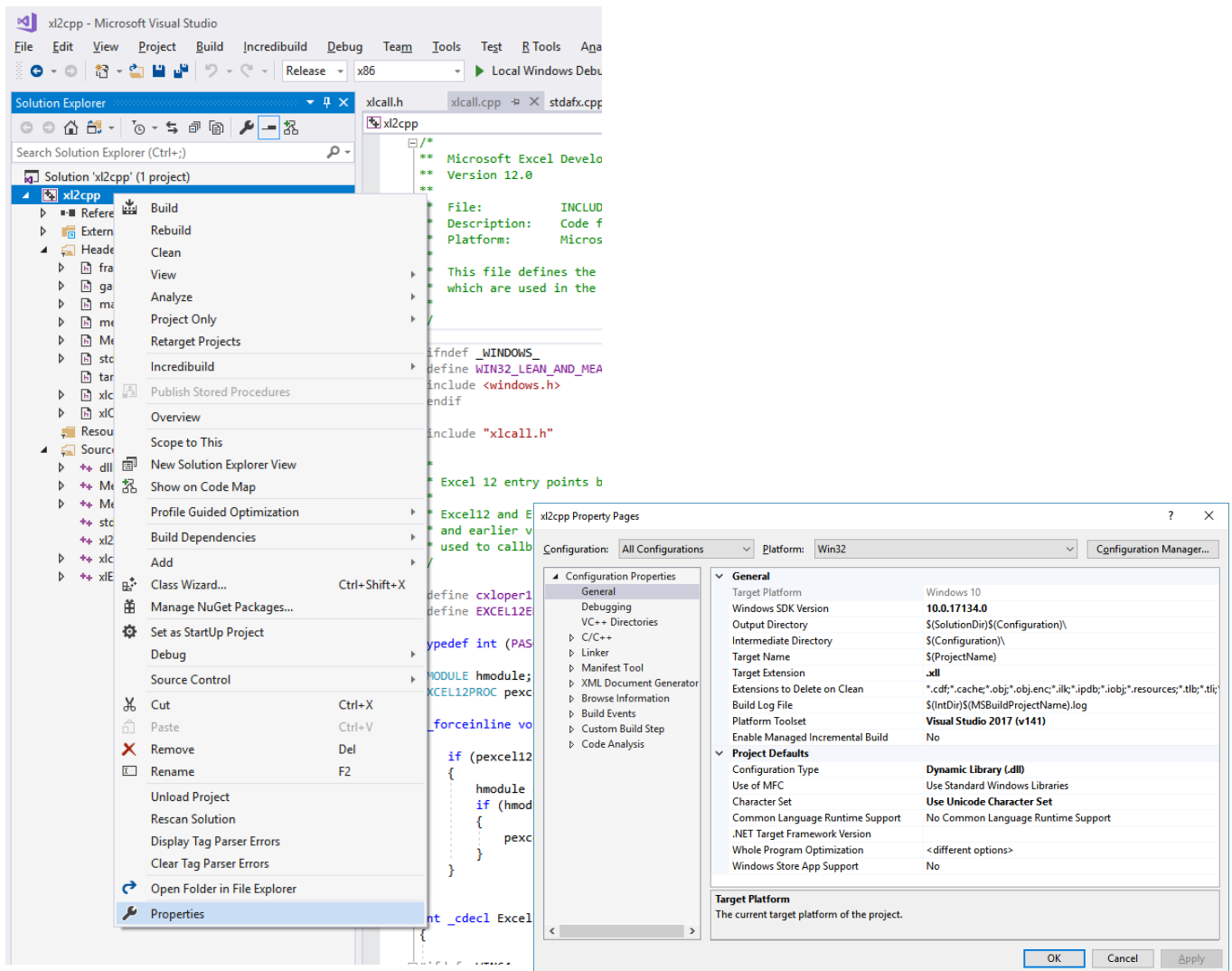
<https://www.microsoft.com/en-us/download/details.aspx?id=20168>

As a convenience, I copied and somewhat simplified the seven necessary files in the `cppFiles` folder.

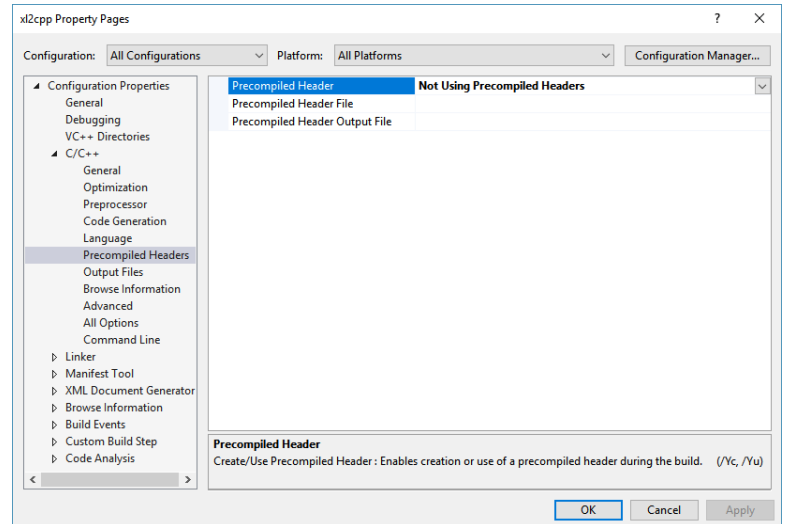
- `gaussians.h` provides an implementation for the cumulative normal distribution (for Black-Scholes) and its inverse (for the generation of Gaussian random numbers for Monte-Carlo simulations). It is necessary for the examples and exercises.
- `xlOper.h` contains functions to simplify the communication of ranges with strings between Excel and C++ code.
- `xlExport.cpp` is where we code functions to be exported to Excel.

In what follows, we articulate a series of step by step instructions to export C++ functions to Excel. We give precise instructions for Visual Studio 2017. Manipulations are similar with older versions.

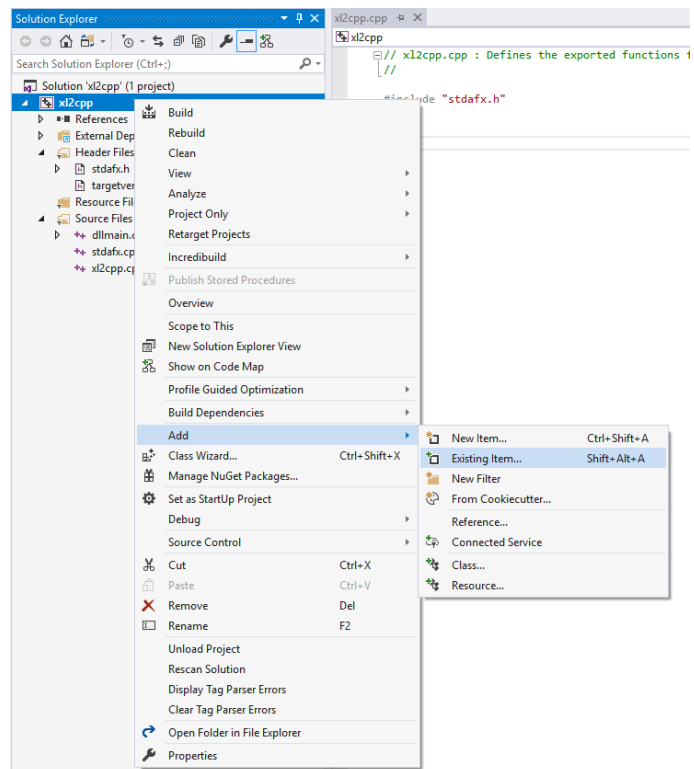
1. In Visual Studio, create a new “Win32 DLL” project. Go to the menu File/New/Project then Installed/Visual C++/Windows Desktop/Dynamic-Link Library (DLL) Project, give it a name and a location on your hard drive, click OK.
2. By default, VS builds an application with the extension “.dll”. We would rather build it with the extension “.xll” so it opens in Excel. Right click on your project (not the solution) in the solution explorer, select “properties”, then select “all configurations” in the “configuration” drop down, and “Win32” or “x86” in the “platform” drop down. On the Configuration Properties/General page, under the General/Target extension field, replace “.dll” by “.xll” and click OK.



3. Disable precompiled headers. Right click the project in the solution explorer and select “properties” again, then select “all configurations” in the “configuration” drop down, and “all platforms” in the “platform” drop down. On the Configuration C/C++ page, under “Precompiled Headers”, set the “Precompiled Header field to “Not Using Precompiled Headers”.



4. Copy the contents of the folder “cppFiles” into your VS solution folder. Then, add the 10 copied files to your project: right click the project in the solution explorer again, select add/existing items and add the 10 files to the project. Check that the files appear in the solution explorer, the .cpp files under “Source Files” and the headers under “Header files”.



5. Double click to open the xlExport.cpp file under your project source files in the solution explorer. This is where we define and register the functions we want to export to Excel. A sample function “xMultiply2Numbers” is already implemented, as you can see, in 2 stages:

- a. First we have the implementation itself

```
extern "C" __declspec(dllexport)
double xMultiply2Numbers(double x, double y)
{
    return x * y;
}
```

The function must be declared as `extern "C" __declspec(dllexport)` in order to be exported. The rest is standard.

- b. Next, we have the registration of the function as an Excel function. All registrations occur in the special function `xlAutoOpen()`, which, as the name indicates, is executed when excel opens the DLL. You will need to register all functions that you want to export to Excel. Registration of a single function consists of the block:

```
Excel12f(xlfRegister, 0, 11, (LPXLOPER12)&xDLL,
    (LPXLOPER12)TempStr12(L"xMultiply2Numbers"), // 1
    (LPXLOPER12)TempStr12(L"BBB"), // 2
    (LPXLOPER12)TempStr12(L"xMultiply2Numbers"), // 3
    (LPXLOPER12)TempStr12(L"x, y"), // 4
    (LPXLOPER12)TempStr12(L"1"), // 5
    (LPXLOPER12)TempStr12(L"myOwnCppFunctions"), // 6
    (LPXLOPER12)TempStr12(L""), // 7
    (LPXLOPER12)TempStr12(L""), // 8
    (LPXLOPER12)TempStr12(L"Multiplies 2 numbers"), // 9
    (LPXLOPER12)TempStr12(L"")); // 10
```

This block must be repeated for every function to be registered. The contents of the strings arguments to all the TempStr12 being:

Line 1: The name of the function, exactly as in its definition.

Line 2: A number of Bs corresponding to the number of arguments plus 1. Our sample function takes 2 arguments, so we have 3 Bs.

Line 3: A repeat of Line 1.

Line 4: The name of the arguments as they appear in the function wizard in Excel, separated by commas.

Line 5: “1”

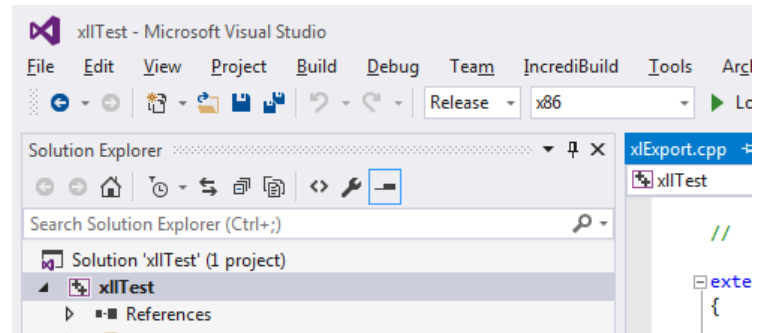
Line 6: The category, either new or existing, where the function is meant to be registered in Excel.

Lines 7 and 8: Empty strings.

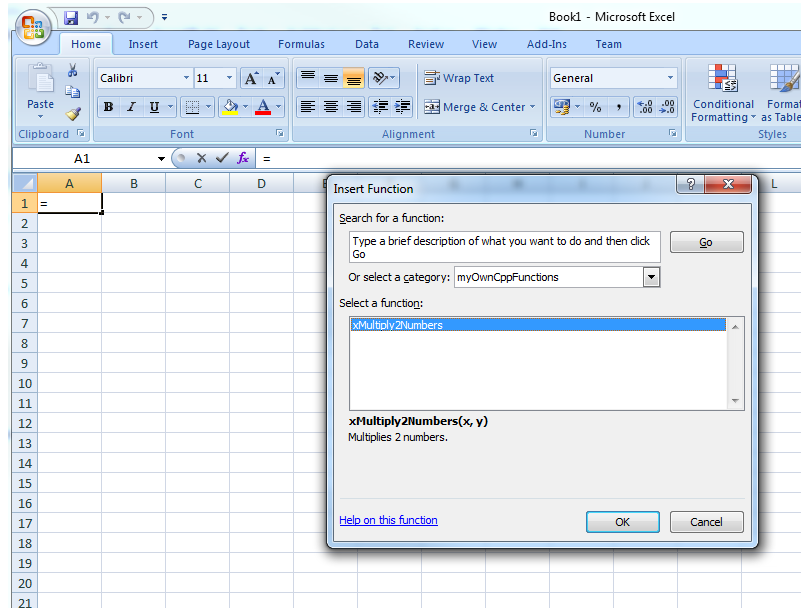
Line 9: A basic description of the function for the function wizard in Excel.

Line 10: Empty string.

6. Build the project in release32 mode. Select Release in the configuration drop down that probably shows “Debug” on the toolbar located below the main menu, and x86 or Win32 on the drop down on its right. Go to the menu build/build solution to build the xll.



7. Look in your solution folder. VS should have created a “Release” folder with a number of files inside it. The one of interest is named *yourProject.xll*. Open it from Excel like you would open a workbook. Now, launch a function wizard (click on the *fx* symbol on the right of the formula bar). You should see a brand new category “myOwnCppFunctions” with a function “xMultiply2Numbers” under it. Use this function and confirm its behavior is correct.
8. Congratulations, you just exported your first C++ function to Excel.



Our second Excel C++ function: xBlackScholes

Now let us export something more useful, like an implementation of Black-Scholes. Close your Excel session and go back to your VS project. We are going to implement Black & Scholes’ formula and export it to Excel.

It is best practice to define the function in standard C++ in a separate file and export a wrapper function to Excel. Add a new C++ file your project and call it blackShcoles.cpp. Right click your project, select Add/New item, select “C++ File”, give it the name blackScholes and press “Add”. Check that the file appears under your project’s source files in the solution explorer.

In your new blackScholes.cpp file, write the function:

```
#include <math.h>
#include "gaussians.h"

double blackScholes(double spot, double vol, double mat, double strike)
{
    double std = sqrt(mat) * vol;
    double halfVar = 0.5 * std * std;
    double d1 = (log(spot / strike) + halfVar) / std;
    double d2 = d1 - std;
    return spot * normalCdf(d1) - strike * normalCdf(d2);
}
```

We also need to create a header file for it. Add a new C++ header file to the project, call it blackScholes.h and copy the function's signature.

```
double blackScholes(double spot, double vol, double mat, double strike);
```

Now we have coded our function, we must export it and register it in Excel. Open xlExport.cpp. First, we wrap our function for export:

```
#include <windows.h>
#include "xlcall.h"
#include "framework.h"

#include "blackScholes.h"

// Wrappers

extern "C" __declspec(dllexport)
double xMultiply2Numbers(double x, double y)
{
    return x * y;
}

extern "C" __declspec(dllexport)
double xBlackScholes(double spot, double vol, double mat, double strike)
{
    return blackScholes(spot, vol, mat, strike);
}
```

Next we register it in Excel:

```
// Registers

extern "C" __declspec(dllexport) int xlAutoOpen(void)
{
    XLOPER12 xDLL;

    Excel12f(xlGetName, &xDLL, 0);

    Excel12f(xlfRegister, 0, 11, (LPXLOPER12)&xDLL,
        (LPXLOPER12)TempStr12(L"xMultiply2Numbers"),
        (LPXLOPER12)TempStr12(L"BBB"),
        (LPXLOPER12)TempStr12(L"xMultiply2Numbers"),
        (LPXLOPER12)TempStr12(L"x, y"),
        (LPXLOPER12)TempStr12(L"1"),
        (LPXLOPER12)TempStr12(L"myOwnCppFunctions"),
        (LPXLOPER12)TempStr12(L""),
        (LPXLOPER12)TempStr12(L""),
        (LPXLOPER12)TempStr12(L"Multiplies 2 numbers"),
        (LPXLOPER12)TempStr12(L""));

    Excel12f(xlfRegister, 0, 11, (LPXLOPER12)&xDLL,
        (LPXLOPER12)TempStr12(L"xBlackScholes"),
        (LPXLOPER12)TempStr12(L"BBBBB"),
        (LPXLOPER12)TempStr12(L"xBlackScholes"),
        (LPXLOPER12)TempStr12(L"spot, vol, mat, strike"),
        (LPXLOPER12)TempStr12(L"1"),
        (LPXLOPER12)TempStr12(L"myOwnCppFunctions"),
        (LPXLOPER12)TempStr12(L""),
        (LPXLOPER12)TempStr12(L""),
        (LPXLOPER12)TempStr12(L"Implements Black & Scholes formula"),
        (LPXLOPER12)TempStr12(L""));

    /* Free the XLL filename */
    Excel12f(xlFree, 0, 1, (LPXLOPER12)&xDLL);

    return 1;
}
```

Build the solution, open the xll from Excel. Your xBlackScholes function is available in Excel.

Debugging

We can debug xlls. Debug builds are substantially slower at run time but they allow to execute the code line by line and visualize the variables values.

First, configure the debugger to attach to Excel and open the xll. Open your project properties: right click the project, select “properties”, select Debug on the configuration drop down and Win32 or x86 on the platform drop down. Go to Configuration Properties/Debugging. Edit the “command” field to Excel’s path, usually something like “C:\Program Files (x86)\Microsoft Office\Office12\excel.exe”. Set the field “command arguments” to “\$(TargetPath)” (without the quotes). Click OK.

Now you can build the Debug xll. Select Debug on the toolbar where you previously selected Release, and build the solution. Then launch the debugger (menu Debug/Start debugging or F5). This should launch Excel and open the (debug) xll. Your functions must be immediately available.

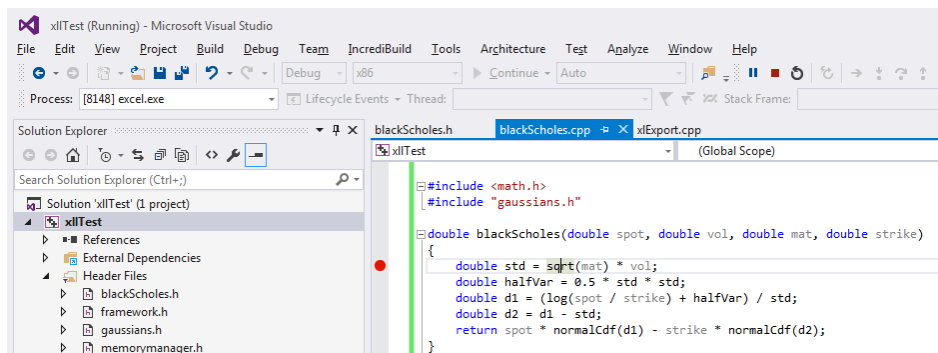
Set a breakpoint on the 1st line of your *blackScholes()* function: click on the line and press F9. A red breakpoint appears on the left.

Go back to Excel, and call *xBlackScholes()*.

Execution stops on the breakpoint. The window “Call stack” shows the sequence of functions being called. The window “Auto” shows the current value of scoped variables. The Debug menu allows to navigate the session: “step over” to execute the next line, “step into” to debug inside the function being called on the current line, “continue” to run to the next breakpoint, you may also activate or deactivate breakpoints, etc.

The window “Watch1” allows users to specify variables and expressions on the left (column “Name”) and visualize their current values on the right (column “Value”).

Run the function line by line, inspecting the value of the variables and familiarizing yourself with the debugger. You will most probably need it to complete the exercises.



Exercises

1. Implement the implied volatility routine in C++ and export it to Excel. How long does it take now to compute 5,000 implied volatilities with 5,000 calls to the function from Excel (in Release)? Compare with VBA.
2. Implement the basic Monte-Carlo simulation in Black-Scholes. How long does it take to complete one simulation with 10,000 paths and 60 steps? What speed-up factor does C++ give us compared to VBA?

You will need to generate random numbers in C++.

C++ provides native means of generating uniform random numbers. This is not particularly efficient but good enough for this exercise.

You must include `<stdlib.h>` to use this facility.

First, call the function `srand()` with a positive integer, say 54321, to initialize the generator.

Then, repeatedly call `double(rand() + 1) / (RAND_MAX + 2)` to generate uniform numbers between 0 and 1.

Turn each number into a Gaussian with a call to our `invNormalCdf()`, defined in the header "gaussians.h".

3. Implement the Black-Scholes risk sensitivities: delta, gamma, vega, theta, ... and start building and maintaining your own C++ library for financial mathematics.

Importing and exporting ranges of numbers

So far, we learned to write excel functions that take scalar numbers as arguments and return a scalar number as a result.

We can also communicate entire ranges of numbers, vectors and matrices, as arguments and results from and to excel. Financial applications use this facilities all the time: yield curves, volatility and correlation matrices, cash-flow schedules, are all ranges of numbers, and their size (number of rows and columns) is dynamically specified at run time.

This section shows how to export to Excel functions that take ranges as arguments and return ranges as results.

Ranges as arguments

First, we show how to pass ranges as arguments from excel to custom functions written in C++. To illustrate this, we will write a rather useless function that takes a matrix $M = \left[x_{ij} \right]_{\substack{0 \leq i \leq n-1 \\ 0 \leq j \leq m-1}}$ of size $n \times m$, computes the norms of its m columns vectors $v_j = (x_{ij})_{0 \leq i \leq n-1}$, $\|v_j\|^2 = \sum_{i=0}^{n-1} x_{ij}^2$, adds them together with a weight of $j+1$ for column j , and multiplies the end result by a scalar y . Hence, the final result to be returned is:

$$f\left(\left[x_{ij} \right]_{\substack{0 \leq i \leq n-1 \\ 0 \leq j \leq m-1}}, y\right) = y \sum_{j=0}^{m-1} (j+1) \|v_j\|^2$$

This function has no particular utility or meaning, its only purpose is to illustrate how to access, from C++ code, the size (number of rows and columns) and the individual elements of a range of numbers passed from excel.

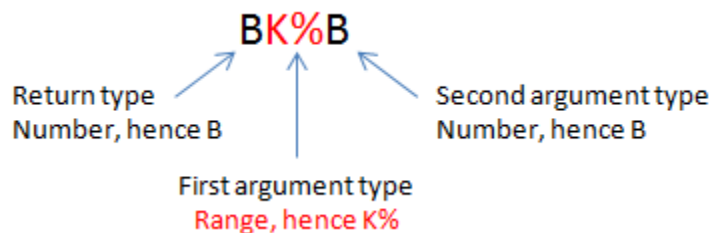
Let us start with the registration of the function in xlExport.cpp. Write the following registration block below your last existing registration block:

```
Excel12f(xlfRegister, 0, 11, (LPXLOPER12)&xDLL,
    (LPXLOPER12)TempStr12(L"xRangeFunc"),
    (LPXLOPER12)TempStr12(L"BK%B"),
    (LPXLOPER12)TempStr12(L"xRangeFunc"),
    (LPXLOPER12)TempStr12(L"matrix, y"),
    (LPXLOPER12)TempStr12(L"1"),
    (LPXLOPER12)TempStr12(L"myOwnCppFunctions"),
    (LPXLOPER12)TempStr12(L""),
    (LPXLOPER12)TempStr12(L""),
    (LPXLOPER12)TempStr12(L"Range function"),
    (LPXLOPER12)TempStr12(L""));
```

The only new syntax here is the outlined registration of the result and argument types in line 2:

```
(LPXLOPER12)TempStr12(L"BK%B")
```

Instead of the usual (number of arguments + 1) Bs we have:



We write the code for the return type first. The return type is a number in this case, the code for a number is B. Then we move on to the type of the 1st argument, in this case a range of numbers. The code for the type “range of numbers” is K%. Finally, we have the type of the 2nd argument, a number, hence B. The entire code for the return and argument types is therefore “BK%B”.

Now we have registered the function, we can write its definition below our previously exported function:

```
extern "C" __declspec(dllexport)
double xRangeFunc(FP12* matrix, double scalar)
```

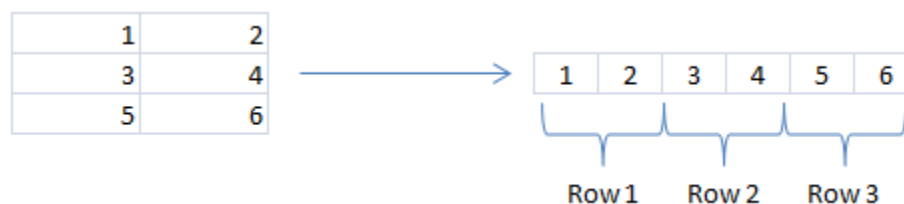
B
K%
B

The return type is double (code B in the registration). The type of the 1st argument (code K% in the registration) is a range of numbers, which excel passes as a pointer on a struct defined in xllcall.h under the name FP12. This struct contains the necessary information to access the number of rows, columns and individual elements. The type of the second argument (code B) is double.

The first thing we must do is access the size (number and rows and columns) and elements of the range from the data in the FP12 struct we got from excel:

```
extern "C" __declspec(dllexport)
double xRangeFunc(FP12* matrix, double scalar)
{
    // Unpack matrix
    size_t rows = matrix->rows;
    size_t cols = matrix->columns;
    double* numbers = matrix->array;
```

Now the local variable *rows* contains the number of rows (n), the variable *cols* contains the number of columns (m), and the pointer *numbers* points to a section of memory where the elements of the matrix have been stored by excel row by row:



Hence, we access the individual element x_{ij} with the syntax:

numbers[i*cols+j]

Important: in this syntax, the indices are zero-based, that is $0 \leq i \leq \text{rows} - 1$ and $0 \leq j \leq \text{cols} - 1$.

We can now write the rest of the function. Note how we access the individual elements in the computation of the result, in accordance with the rule above.

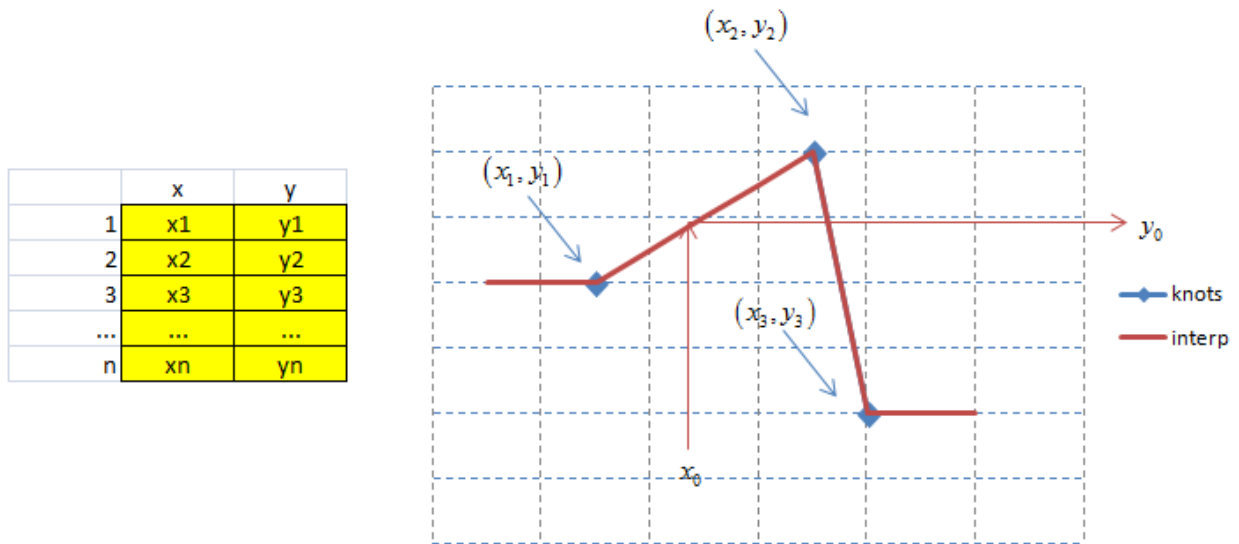
```
extern "C" __declspec(dllexport)
double xRangeFunc(FP12* matrix, double scalar)
{
    // Unpack matrix
    size_t rows = matrix->rows;
    size_t cols = matrix->columns;
    double* numbers = matrix->array;

    // Compute result
    double result = 0.0;
    for (size_t j = 0; j < cols; ++j)
    {
        double norm = 0.0;
        for (size_t i = 0; i < rows; ++i)
        {
            // To access matrix[i][j] = numbers[i*cols+j]
            double xij = numbers[i * cols + j];
            norm += xij * xij;
        }
        norm = sqrt(norm);
        result += (j + 1) * norm;
    }
    result *= scalar;
    return result;
}
```

Also note that you must `#include <math.h>` on the top of your file in order to use `sqrt`.

Exercise





Write and export to excel a linear interpolation / flat extrapolation function, taking a 2 column range with a knot point on each row, and a number x_0 , and returns its interpolated value y_0 .



Make sure that the exported function defined in xlExport.cpp only unpacks and transforms data, and then calls an interpolation function defined in a different file that does not refer to excel in any way. This is best practice in C++, this way your function may be called by non-excel clients too.

Ranges as results

Excel also allows functions to return ranges of numbers. For instance, its built-in function for matrix product, *mmult()*, returns a range with the same number of rows as its first argument, and the same number of columns as its second argument. To use it, first select the output range, type the *mmult* formula and press ctrl+shift+enter. This fills the selected area with the result of the product of the 2 argument matrices.

EOMONTH		   		=mmult(A1:D3,F1:G4)						
	A	B	C	D	E	F	G	H	I	J
1	1	2	3	4		13	14		=mmult(A1:D3,F1:G4)	
2	5	6	7	8		15	16			
3	9	10	11	12		17	18			
4						19	20			
5										

We will learn to return ranges in this way from custom functions written in C++, and to illustrate our purpose, we will code our own matrix product function.

The additional difficulty here comes from memory management: when we read a range from excel, excel has ownership of the memory and takes care of allocation and de-allocation. We only read memory to access the contents of the range.

On the contrary, when we return a range as a result to excel, it is our responsibility to allocate memory for the range, correctly set its size and contents, and de-allocate memory *after* we return it to excel. How can we de-allocate memory after our function returned and execution continues outside of our code? Many solutions have been implemented, we introduce the simplest solution, with the mechanisms provided in framework.h.

This will all become clearer as we work through the example. First, let us register our function, below the last existing registration block in xlExport.cpp:

```
Excel12f(xlfRegister, 0, 11, (LPXLOPER12)&xDLL,
    (LPXLOPER12)TempStr12(L"xMyOwnMmult"),
    (LPXLOPER12)TempStr12(L"K%K%K%"),
    (LPXLOPER12)TempStr12(L"xMyOwnMmult"),
    (LPXLOPER12)TempStr12(L"matrix1, matrix2"),
    (LPXLOPER12)TempStr12(L"1"),
    (LPXLOPER12)TempStr12(L"myOwnCppFunctions"),
    (LPXLOPER12)TempStr12(L""),
    (LPXLOPER12)TempStr12(L""),
    (LPXLOPER12)TempStr12(L"Matrix product"),
    (LPXLOPER12)TempStr12(L""));
```

The syntax on line 2:

```
(LPXLOPER12)TempStr12(L"K%K%K%")
```

should be clear by now: the return type is a range, hence K%, and we have 2 arguments, also of type range, hence K%K%. In all, K%K%K%.

Let us move to the definition of the exported function, below the definition of the previously exported function in xlExport.cpp:

```
extern "C" __declspec(dllexport)
FP12* xMyOwnMmult(FP12* matrix1, FP12* matrix2)
```

The return type, as well as the type of the arguments, are ranges, therefore, as we know by now, pointers on FP12 structs. Note that the return type is not the struct FP12, but a pointer on such struct FP12*.

The first thing to do is unpack the arguments into rows, columns and elements for both arguments, as we did previously, except now we are repeating the process for both arguments:

```
{
    // Unpack matrices

    size_t rows1 = matrix1->rows;
    size_t cols1 = matrix1->columns;
    double* numbers1 = matrix1->array;

    size_t rows2 = matrix2->rows;
    size_t cols2 = matrix2->columns;
    double* numbers2 = matrix2->array;
```

The next thing to do is allocate memory for the result. Hopefully the comments help make sense of the code below:

```
    // Allocate result

    // Calculate size
    size_t resultRows = rows1, resultCols = cols2, resultSize = resultRows * resultCols;
    // Return an error if size is 0
    if (resultSize <= 0) return nullptr;

    // First, free all memory previously allocated
    // the function is defined in framework.h
    FreeAllTempMemory();

    // Then, allocate the memory for this result
    // We don't need to de-allocate it, that will be done by the next call
    // to this function or another function calling FreeAllTempMemory()

    // Memory size required, details in Dalton's book, section 6.2.2
    size_t memSize = sizeof(FP12) + (resultSize - 1) * sizeof(double);
```

```
// Finally allocate, function definition in framework.h
FP12* result = (FP12*)GetTempMemory(memSize);
```

The size (number of elements) of the result matrix is the number of rows multiplied by the number of columns. We compute it in the variable *resultSize*. The size of the memory needed for the corresponding FP12 struct is:

```
sizeof(FP12) + (resultSize - 1) * sizeof(double)
```

(see details in Dalton, section 6.2.2). If we classically allocated this memory with `malloc` or `::operator new`, we wouldn't know how to free it. Instead, we use pre-allocated memory provided by *GetTempMemory()*, defined in `framework.h`. And here is the trick: we free all the allocated temporary memory *before* we make any new allocation, in this function, and all functions that return ranges. Hence, the allocated memory will be freed on the next call to such a function. This effectively resolves the issue of freeing memory allocated for return ranges in a particularly simple way.

To free previously allocated memory by a call to *FreeAllTempMemory()* is a crucial step to perform before allocation. Every wrapper function that allocates memory for the range returned to Excel must start by freeing previously allocated memory. Otherwise, the memory allocated by successive calls from Excel is never released and repeated function calls from Excel rapidly exhaust the available RAM.

Finally, we can calculate the result and return it:

```
// Compute result
result->rows = resultRows;
result->columns = resultCols;
matrixProduct(rows1, cols1, numbers1, rows2, cols2, numbers2, result->array);

// Return it
return result;
}
```

Here is the complete function:

```
extern "C" __declspec(dllexport)
FP12* xMyOwnMmult(FP12* matrix1, FP12* matrix2)
{
    // Unpack matrices

    size_t rows1 = matrix1->rows;
    size_t cols1 = matrix1->columns;
    double* numbers1 = matrix1->array;

    size_t rows2 = matrix2->rows;
    size_t cols2 = matrix2->columns;
    double* numbers2 = matrix2->array;

    // Allocate result
```

```

// Calculate size
size_t resultRows = rows1, resultCols = cols2, resultSize = resultRows * resultCols;
// Return an error if size is 0
if (resultSize <= 0) return nullptr;

// First, free all memory previously allocated
// the function is defined in framework.h
FreeAllTempMemory();

// Then, allocate the memory for this result
// We don't need to de-allocate it, that will be done by the next call
// to this function or another function calling FreeAllTempMemory()

// Memory size required, details in Dalton's book, section 6.2.2
size_t memSize = sizeof(FP12) + (resultSize - 1) * sizeof(double);

// Finally allocate, function definition in framework.h
FP12* result = (FP12*)GetTempMemory(memSize);

// Compute result
result->rows = resultRows;
result->columns = resultCols;
matrixProduct(rows1, cols1, numbers1, rows2, cols2, numbers2, result->array);

// Return it
return result;
}

```

Note that this function unpacks the arguments, packs the result and takes care of its memory management, but it does not actually compute the matrix product. In accordance with C++ best practice, it delegates linear algebra to a dedicated function, which could be defined as:

```

void matrixProduct(
    size_t rows1, size_t cols1, double* numbers1,
    size_t rows2, size_t cols2, double* numbers2,
    double* results)
{
    // Sizes
    size_t resultRows = rows1, resultCols = cols2;
    // We assume that cols1 == rows2, we could throw otherwise

    // Loop on results
    for (size_t i = 0; i < resultRows; ++i) for (size_t j = 0; j < resultCols; ++j)
    {
        // Calculate RESij
        double tempRes = 0.0;
        for (size_t k = 0; k < cols1; ++k)
            tempRes += numbers1[i*cols1 + k] * numbers2[k*cols2 + j];
        // Commit it to memory
        results[i*resultCols + j] = tempRes;
    }
}

```

Exercise

Write a function that takes a column of maturities, a row of strikes, a spot and a volatility number, and returns the range of corresponding Black-Scholes call prices.

Importing and exporting mixed ranges containing numbers and strings

In this final section, we show how to communicate mixed ranges that contain both strings and numbers.

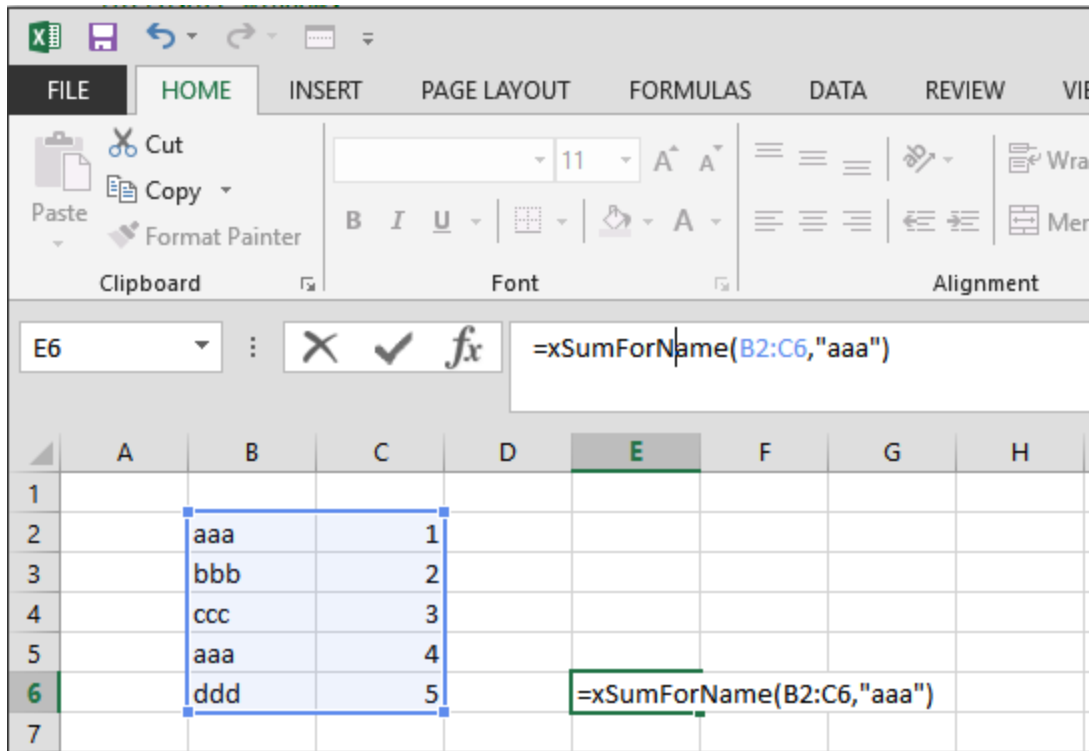
To communicate strings, ranges that contain strings, or mixed ranges that contain strings and numbers, are somewhat complicated processes in the Excel SDK. In the interest of simplicity, I encapsulated the difficulties into a small number of simple functions in `xlOper.h`. In what follows, we use these simple functions to implement communication between Excel and C++ code. We don't explain how these functions were implemented, or how to use the lower level Excel SDK constructs. Interested readers are referred to Dalton's book.

We only demonstrate the communication of ranges that contain a mix of strings and numbers. We don't separately cover the communication of single strings or ranges of strings only. Both are considered a particular case. For example, to communicate a single string, we communicate a range with one row and one column, which first (and only) element is the communicated string.

Finally, when a mixed range is passed as an argument from Excel, we assume that the C++ code knows what elements are supposed to be strings and what elements are supposed to be numbers. It is possible to detect the type of individual elements at run time, but this is not covered here.

Mixed ranges as arguments

Once again, we illustrate the method with a simple example instead of describing the general methodology. We code a simple function `xSumForName()` that takes a range with two columns as an argument: a column of names as strings and a column of corresponding values. The names are not necessarily unique. The function also takes a name as a second string argument, and returns the sum of the values associated to that name. For example:



returns 5. This function has no other utility than to demonstrate how to pass mixed ranges and strings as arguments.


We start with the registration in xlExport.cpp:

```
Excel12f(xlfRegister, 0, 11, (LPXLOPER12)&xDLL,
    (LPXLOPER12)TempStr12(L"xSumForName"),
    (LPXLOPER12)TempStr12(L"BQQ"),
    (LPXLOPER12)TempStr12(L"xSumForName"),
    (LPXLOPER12)TempStr12(L"Table, Name"),
    (LPXLOPER12)TempStr12(L"1"),
    (LPXLOPER12)TempStr12(L"myOwnCppFunctions"),
    (LPXLOPER12)TempStr12(L""),
    (LPXLOPER12)TempStr12(L""),
    (LPXLOPER12)TempStr12(L"Computes the sum a values for a given name"),
    (LPXLOPER12)TempStr12(L""));
```

The function returns a number, code **B**. The first argument is a mixed range. The code for mixed ranges is **Q**. Recall that we consider a single string as a mixed range with one element. Hence, the second argument is also a mixed range with code **Q**. The code for the function is therefore **BQQ**.

Moving on to the function declaration, also in `xlExport.cpp`, we already know that the type corresponding to code **B** is *double*. The type corresponding to code **Q** is *LPXLOPER12*, defined in `xlcall.h`. This is the most general type defined in the Excel SDK, quite literally “(Long) Pointer on an Excel Operator (with Excel version 12 up)”:

```
extern "C" __declspec(dllexport)
double xSumForName(LPXLOPER12 tab, LPXLOPER12 xName)
```



B
Q
Q

First, we read the *LPXLOPER12 tab*, using functions defined in `xlOper.h`. Recall that *tab* is supposed to be a range with two columns, a first column of names as strings, and a second column of corresponding values as numbers.

```
// Utility functions
#include "xlOper.h"

extern "C" __declspec(dllexport)
double xSumForName(LPXLOPER12 tab, LPXLOPER12 xName)
{
    // Read table
    // -----

    // Dimensions

    size_t cols = getCols(tab);
    // Must be 2
    if (cols != 2) return -1;

    size_t rows = getRows(tab);

    // Read strings and numbers
    vector<string> strings(rows);
    vector<double> numbers(rows);
    for (size_t i = 0; i < rows; ++i)
    {
        strings[i] = getString(tab, i, 0);
        // Could not get a string
        if (strings[i].empty()) return -1;

        numbers[i] = getNum(tab, i, 1);
        // Could not get a number
        if (numbers[i] == numeric_limits<double>::infinity()) return -1;
    }
}
```

The functions *getCols()* and *getRows()*, defined in *xlOper.h*, return the number of columns and rows in their (*LPXLOPER12*) argument. We use them to read the number of rows and columns in *tab*, checking that we have two columns, returning the error code -1 otherwise (since the function returns a number, we can't return an explicit error message, we do this next when we learn to return strings).

After the number of columns and rows in the *LPXLOPER12 tab* are known and correct, we read its elements, in a loop, and copy them in the vectors *strings* and *numbers*.

The function *getString(LPXLOPER12, row, col)*, defined in *xlOper.h*, returns the string read in the cell on the given *row* and *column* (both with index starting at 0) on the given *LPXLOPER12*. For example, the line:

```
strings[i] = getString(tab, i, 0);
```

reads the string on the 1st column, row *i*, of the *LPXLOPER12 tab*, and copies it into *strings[i]*. When *getString()* fails to read a string (for instance, because the element is not a string, maybe a number), it returns an empty string. This is checked and the error code -1 is returned in this case.

Similarly, *getNum(LPXLOPER12, row, col)* returns the number read on the given *row* and *column* of the *LPXLOPER12*. The line:

```
numbers[i] = getNum(tab, i, 1);
```

reads the number on the 2nd column, row *i*, of the *LPXLOPER tab*, and copies it into *numbers[i]*. When *getNum()* fails to read a number (for instance, when the element is not a number, maybe a string), it returns *infinity*, or in standard C++: `numeric_limits<double>::infinity()`.

Mixed ranges are therefore passed from Excel to C++ code in 3 steps:

1. Register argument types **Q**.
2. Pass arguments as *LPXLOPER12*.
3. Read sizes with *getRows()* and *GetCols()*, read elements with *getNum()* and *getString()*.

We read the 2nd argument, the string *xName*, in the same manner:

```
// Read name
// -----

// Must be single cell
if (getCols(xName) != 1 || getRows(xName) != 1) return -1;

string name = getString(xName, 0, 0);
if (name.empty()) return -1;
```

xName is a string passed as a mixed range, so we check that the range is of size 1, and that the unique element is a string, in which case it is copied into the string *name*. Otherwise, -1 is returned to notify an error.

Finally, we compute and return the result. Since the result is a number, there is nothing new here:

```
// Calculate and return result
// -----

double res = 0;

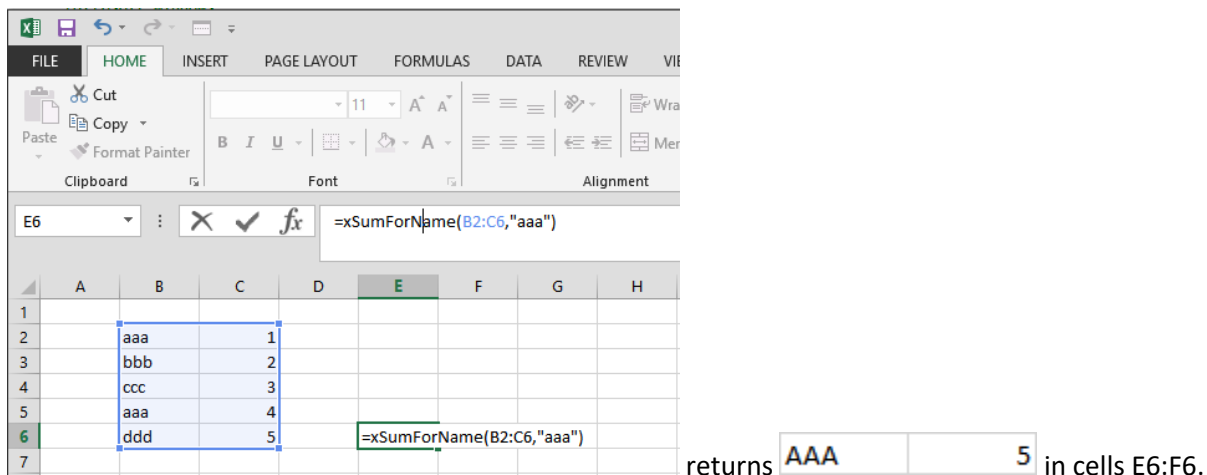
for (size_t i = 0; i < rows; ++i)
{
    if (strings[i] == name) res += numbers[i];
}

return res;
}
```

It is best practice to apply STL algorithms in place of hand crafted loops for this type of operation, however in the context of this tutorial, we keep things simple and focus on communication with Excel.

Mixed ranges as results

Finally, we demonstrate (once again, with a simple example) how to return mixed ranges (including single strings) from C++ to Excel. We reuse the previous function `xSumForName()`, except we now return a range with two cells: first the name given as an argument, but in uppercase, and secondly, the number previously calculated. For example:



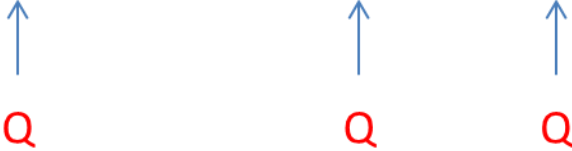
The screenshot shows the Microsoft Excel interface. The formula bar at the top displays the function call `=xSumForName(B2:C6,"aaa")`. Below the formula bar, a table of data is visible:

	A	B	C	D	E	F	G	H
1								
2		aaa	1					
3		bbb	2					
4		ccc	3					
5		aaa	4					
6		ddd	5		=xSumForName(B2:C6,"aaa")			
7								

Below the table, the text "returns" is followed by a visual representation of the function's output: a cell containing the uppercase string "AAA" and a cell containing the number "5". This indicates that the function returns a mixed range of a string and a number, which are placed in cells E6 and F6 respectively.

This changes registration. The return type is no longer a number, but a mixed range. The code is no longer **B** but **Q**:

```
extern "C" __declspec(dllexport)
LPXLOPER12 xSumForName(LPXLOPER12 tab, LPXLOPER12 xName)
```



Q Q Q

```
Excel12f(xlfRegister, 0, 11, (LPXLOPER12)&xDLL,
  (LPXLOPER12)TempStr12(L"xSumForName"),
  (LPXLOPER12)TempStr12(L"QQQ"),
  (LPXLOPER12)TempStr12(L"xSumForName"),
  (LPXLOPER12)TempStr12(L"Table, Name"),
  (LPXLOPER12)TempStr12(L"1"),
  (LPXLOPER12)TempStr12(L"myOwnCppFunctions"),
  (LPXLOPER12)TempStr12(L""),
  (LPXLOPER12)TempStr12(L""),
  (LPXLOPER12)TempStr12(L"Computes the sum a values for a given name"),
  (LPXLOPER12)TempStr12(L""));
```

The part of the code that reads the arguments is essentially unchanged. The only thing is we now return a string, so we may as well return explicit error messages in place of the -1 code. In addition, like for ranges of numbers, to return mixed ranges consumes memory that must be released. For this reason:

Any function that returns a range must free all temporary memory first.

This step is crucial, and worth repeating. Any function that returns a range (either as FP12* or LPXLOPER12, even if the range consists of a single number or string) must always start by releasing temporary memory:

```
extern "C" __declspec(dllexport)
LPXLOPER12 xSumForName(LPXLOPER12 tab, LPXLOPER12 xName)
{
    // Release memory
    FreeAllTempMemory();
```

Next, we read the arguments as previously, except that we now return an explicit error message in place of a cryptic code:

```
// Read table
// -----

// Dimensions

size_t cols = getCols(tab);
// Must be 2
if (cols != 2) return TempStr12("Table must be two columns wide");

size_t rows = getRows(tab);

// Read strings and numbers
vector<string> strings(rows);
vector<double> numbers(rows);
for (size_t i = 0; i < rows; ++i)
{
    strings[i] = getString(tab, i, 0);
    // Could not get a string
    if (strings[i].empty())
        return TempStr12("All elements in first column must be strings");

    numbers[i] = getNum(tab, i, 1);
    // Could not get a number
    if (numbers[i] == numeric_limits<double>::infinity())
        return TempStr12("All elements in second column must be numbers");
}

// Read name
// -----

// Must be single cell
if (getCols(xName) != 1 || getRows(xName) != 1)
    TempStr12("Name must be a single cell");

string name = getString(xName, 0, 0);
if (name.empty()) return TempStr12("Name must be a string");
```

TempStr12(), defined in *xlOper.h*, takes a string and wraps it into a *LPXLOPER12* (of single cell type), allowing to return explicit error messages as strings from any function with return type *LPXLOPER12*.

Next, we compute the two results: the name in uppercase (implemented in a simplistic manner, a proper implementation should use SLT algorithms instead) and the sum of values corresponding to the name (as previously):

```
// Calculate result
// -----

// name in uppercase
string upperName;
for (size_t j = 0; j < name.size(); ++j)
{
    upperName += toupper(name[j]);
}

// sum
double sum = 0;
for (size_t i = 0; i < rows; ++i)
{
    if (strings[i] == name) sum += numbers[i];
}
```

Finally, we wrap results into a LPXLOPER12 and return it:

```
// LPXLOPER12
LPXLOPER12 result = TempXLOPER12();

// Size
resize(result, 1, 2);

// Set
setString(result, upperName, 0, 0);
setNum(result, sum, 0, 1);

return result;
}
```

TempXLOPER12(), defined in *xOper.h*, constructs an XLOPER12 in temporary memory and returns a LPXLOPER12 on it. Think of it as an Excel SDK equivalent of:

```
XLOPER12* result = new XLOPER12;
```

(recall that a LPXLOPER12 is a pointer on a XLOPER12).

The function *resize(LPXLOPER12, rows, columns)*, also defined in *xOper.h*, sets the size (number of rows and columns) of a LPXLOPER12. A new LPXLOPER12 must be resized before any of its elements are set. The initial size of a LPXLOPER12 is zero rows, zero columns, so to write anything there would result in an out of bounds exception. The line:

```
resize(result, 1, 2);
```

sizes the LPXLOPER12 *result* with one row, two columns. The two cells are implicitly initialized to empty strings, which can be overwritten by strings or numbers with the functions *setString()* and *setNum()*.

These functions *setString()* and *setNum()* are dual to *getString()* and *getNum()*: they write strings and numbers in a cell on a given row and a given column (both indexes starting with 0) of a given LPXLOPER12 (inside the bounds specified by *resize()*). Hence,

```
setString(result, upperName, 0, 0);
```

writes the string *upperName* in the first row, first column of the LPXLOPER12 *result*. The next line:

```
setNum(result, sum, 0, 1);
```

writes the number *sum* in the first row, second column of *result*.

Exercise

1. Modify the function implementing the Black-Scholes formula so it takes an additional string argument *callOrPut*. Return the price of the call if the user passes “call”, the price of the put if she passes “put”, or an explicit error message otherwise.
2. Further improve the implementation of the Black-Scholes function to optionally compute risk sensitivities. Implement an additional string argument *computeRisk*, which the user may pass as “yes” or “no”. If it is “no”, return the price in a single cell. If it is yes, return the price, along with delta, gamma, vega and theta in a range with two columns, five rows. Return the 5 labels “price”, “delta”, “gamma”, “vega”, “theta” in the first column, and the corresponding values in the second column.

Questions and comments: antoine@asavine.com