IB9N7 C++ for Quantitative Finance
Worksheet 6

Functions

16 November 2015
(Week 7)

Use the opportunity to ask questions to the assistants during the lab! Keep records of your solutions and show them to us when you have finished.

To help us judge the timing, please let us know how long it takes you to do the exercises. If you are unable to finish the exercises during the lab session, please finish them before next time.

For this worksheet, you are strongly encouraged to adapt code you have already written in previous exercises, rather than starting from scratch.

Ensure that your functions follow the best practice guidelines as mentioned in the lecture. Where requirements about functions are not specified (e.g. name, number and type of parameters, default values, behaviour under invalid input, etc), you are free to make your own choices (but of course you should be able to justify your choices).

Objectives for this lab session

By the end of this session, you should have completed the following:

- Know how to define a function and specify parameters and return values.
- Appreciate how functions make life easier.

Exercises

For each question that involves writing code, decide on an appropriate translation unit in which to place that code.

Exercise 1: Simple functions

The previous worksheets required writing small programs performing summations, factorials and random number generation. Instead of having separate *programs* for each of these, it is better to have separate *functions*.

(a) You may have previously met a function in C++ for calculating powers. It is declared in the header file <cmath>. Its prototype is `double std::pow(double base, double exponent)`. Write down the return type, identifier, parameters and signature for this function. You should already have enough information to be able to do this.

(b) Write a function that calculates n-factorial ($n!$) for a given number n. (Compare Exercise 4(a) in Worksheet 4.)

Your function should **not** invoke itself (such functions are called **recursive functions** and require extra care).

In anticipation of the next question, you might like to generalise this to a falling factorial function, and to define your factorial function in terms of the falling factorial.

(c) Write a function that that takes two arguments n and k and calculates the binomial coefficient "n choose k" $\binom{n}{k}$, the number of ways choosing k objects from n. (Compare Exercise 4(b) in Worksheet 4.)

(Hint: use the function you wrote for the previous part. The advantages of functions are now obvious. In fact, when you solved this question originally, you should have been thinking to yourself that there must be a better way than repeating the same kind of calculations by duplicating code.)

Exercise 2: Vector arguments

(a) Write a void function `print_vector_head` that prints out the first k elements of a vector of integers, where k is a parameter to the function. (A "void function" means that the return type is void.)

If there are less than k elements in the vector, it should print out a message to that effect, and the whole vector.

Your function should print one line for each of the elements it prints, with each line of the form `"Element [i] = value"`.

(b) Adapt your function so that it takes another parameter, `verbose`, a `bool`, where it prints the text `"Element [i] = value"` on each line if `verbose` is `true` and prints just the value on each line if `verbose` is `false`.

(c) Write a void function `print_vector_all` that prints out all the elements of a vector of integers.

Hint: you can do this simply by invoking the `print_vector_head` function with an appropriate value of k.

(d) Write a function `make_vector_rand_range` that takes as its parameters a `std::mt19937`, a natural number n and an int `max`, and returns a vector of `int`s of size n consisting of random integer numbers between the range 0 and `max` inclusive. (Compare Exercise 2(a) in Worksheet 5.)

In a simple calculation using random numbers, we will want to only have one `mt19937` for all of them, so that they all behave like they are independent. It is important that the `mt19937` is taken by reference. Check you understand why.

(e) Write a function that takes as its parameters an integer `i` and a vector (of integers) `v` and counts the number of times that `i` appears in `v`.

Apply this function, and the function from the previous part with `max` = 1000, to determine the frequency of 0 and of `max` within your random vector. Do they look reasonable?

Exercise 3: Misc

Go back through your solutions to the other lab exercises. Can you think of any other places where your code would have been better as a function?

Exercise 4: More examples of functions with vectors (extra exercise)

(a) Write a function `repeat_vector_elements` which takes a `vector<int>` and modifies it so that each element now appears twice next to each other. It might be tested like this. Note that assignment (=), and comparison (`==`, `!=`) operators work on vectors in the logical way: they look at the whole data.

```
void test_repeat_vector_elements(){
    vector<int> input{2,3,3,7};
    vector<int> wantedOutput{2,2,3,3,3,3,7,7};
    repeat_vector_elements(input);
    if(input!=wantedOutput){
        cout<<"not working"<<endl;
    }
}
```

(b) Write a function `remove_odd_elements` which takes a `vector<int>` and modifies it so that it only contains its former even elements (in the original order). You can remove the *i*th element of a vector with `v.remove(v.begin()+i)`. When using the `remove` function, all subsequent elements must be moved up by 1, so you may be able to come up with a better implementation which doesn't use it — hopefully only moving each element once. It might be tested like this.

```
void test_remove_odd_elements(){
    vector<int> input{2,3,-18,3,7,736};
    vector<int> wantedOutput{2,-18,736};
    remove_odd_elements(input);
    if(input!=wantedOutput){
        cout<<"not working"<<endl;
    }
}

```

Exercise 5: function objects (extra exercise)

The `std::function` object, which is declared in the <`functional`> header, makes a variable whose value is a function, with specified return type and argument types. For example, the following program outputs the number 6.

```
int a(const int& h, int i){
    return h + i;
}
int main(){
    std::function<int(const int&, int)> f;
    f=a;
    std::cout<<f(4,2)<<"\n";
}
```

Using this facility, it is easy to create functions which take functions as parameters. (There is in fact a more fundamental type, called a *function pointer*, which can store a function as a value. `std::function` is a newer, more powerful and easier to use way to achieve this. It is fine to reassign a `std::function` to a new function - it is just a value. It is also fine to put `std::function` objects in a vector. Using an unassigned `std::function` object causes an exception - which means, for us, the program will crash.)

- Write two small functions `product` and `summation` that respectively return the product and the sum of a `vector<double>` Y.

  Write a third function `applyFunction` that accepts a `vector<double>` Y and a function object F and prints $F(Y)$. Test it by using the functions `product` and `summation`.

Remember to talk through your answers with the lab assistants!