IB9N7 C++ for Quantitative Finance
Worksheet 6

Functions
Hints and solutions

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.

If you do not understand (or you disagree with) the hints and solutions, please ask us at the start of the next lab session. Remember that you are not necessarily expected to be able to solve everything without help.

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.

Prototype: `double std::pow(double base, double exponent)`

Return type: `double`
Identifier: `std::pow` (i.e. `pow` in namespace `std`)
Parameters: `double base, double exponent`
Signature: `std::pow(double, double)`
Header file: `<cmath>`
Implementation: Standard library

(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.

If you are having problems with the falling factorial, check that your function corresponds to a valid mathematical definition of the function first.

See answer to next part.

(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.)

If you are having problems with the binomial function, check that your function corresponds to the correct mathematical definition of the function first (in particular that the arguments to the falling factorial function are correct, and that the correct division is done).

```cpp
#include <iostream>
#include <cmath>
#include <cstdlib>

// Provide a typedef to make it easier to change data types
// for the return values of fatorial and binom_coeff etc,
// in case unsigned int isn't large enough.
// Could also change n, k to be of type myint,
// but in general, unsigned int will be large enough for those.
typedef unsigned int myint;

myint falling_factorial(unsigned int n, unsigned int k)
{
    // Handle some special cases to avoid modular arithmetic
    // in the loop counter.
    if (k == n + 1u)
    {
        // The falling factorial is well—defined and has value 0u
        return 0u;
    }
    // Handle error cases
    if (k > n + 1u)
    {
        // The falling factorial is not well—defined.
        // We on't know how to signal an error yet, so just send 0u for now.
        return 0u;
    }

    // Do the actual calculation!
    myint fact = 1u;
    // Note: if n == 0u then we return 1u, which is correct.
    // Don't need to provide an explicit branch for that case.
    for (unsigned int i = (n - k + 1u); i <= n; ++i)
    {
        fact *= i;
    }
    return fact;
}

myint factorial(unsigned int n)
{
    return falling_factorial(n, n);
}

void main_factorial_test()
{
    unsigned int n = 0u;
    std::cout << "Enter␣the␣value␣of␣n,␣and␣I␣will␣calculate␣(n!):␣";
    std::cin >> n;
    std::cout << n << "!␣=␣" << factorial(n) << std::endl;
}

myint binom_coeff(unsigned int n, unsigned int k)
{
    if (n >= k)
    {
        // could also make this slightly more efficient by replacing
        // k with n — k when k > n — k
        return falling_factorial(n, k) / factorial(k);
    }
    else
    {
        // not an error, but the binomial coefficient is 0u
        // and the expression for the other case does not apply
        return 0u;
    }
}

void main_binom_test()
{
    unsigned int n = 0u, k = 0u;
    std::cout << "Enter␣the␣values␣of␣n␣and␣k,␣and␣I␣will␣calculate␣'n␣choose␣k':␣";
    std::cin >> n >> k;
    /*
    // no need to (and not possible to) check this since n, k are unsigned
    if ((n < 0) || (k < 0))
    {
        std::cout << "Please enter non—negative n and k." << std::endl;
        return;
    }
    */
    std::cout << n << "␣choose␣" << k << "␣=␣" << binom_coeff(n, k) << std::endl;
}

int main()
{
    main_factorial_test();
    main_binom_test();
    return EXIT_SUCCESS;
}
```

You can also add extra tests for `falling_factorial` if desired.

Exercise 2: Vector arguments

It is assumed in the following that you have chosen to provide separate translation units for some of these functions as appropriate.

(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"`.

```
// in the following, have used a const reference to a vector
// instead of passing a copy of the vector. this is faster and still safe.
void print_vector_head(const std::vector<int> & v, std::vector<int>::size_type k)
{
    std::vector<int>::size_type n = v.size();
    // print header message
    if (k > n)
    {
        std::cout << "There␣are␣only␣" << n << "␣<␣" << k << "␣elements␣in␣the␣vector,␣so␣I'll␣print␣all␣of␣them!\n";
        k = n;
    }
    else if (k == n)
    {
        std::cout << "All␣" << n << "␣elements␣of␣the␣vector␣follow:\n";
    }
    else
    {
        std::cout << "First␣" << k << "␣elements␣of␣the␣vector␣follow:\n";
    }

    // now print the array elements themselves
    for (std::vector<int>::size_type i = 0u; i != k; ++i)
    {
        std::cout << "Element␣[" << i << "]␣=␣" << v.at(i) << "\n";
        //std::cout << "Element at index " << i << " = " << v.at(i) << "\n";
    }
}
```

(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`.

```
void print_vector_head(const std::vector<int> & v, std::vector<int>::size_type k,
        bool verbose)
{
    std::vector<int>::size_type n = v.size();
    // print header message
    if (k > n)
    {
        std::cout << "There␣are␣only␣" << n << "␣<␣" << k << "␣elements␣in␣the␣vector,␣so␣I'll␣print␣all␣of␣them!\n";
        k = n;
    }
    else if (k == n)
    {
        std::cout << "All␣" << n << "␣elements␣of␣the␣vector␣follow:\n";
    }
    else
    {
        std::cout << "First␣" << k << "␣elements␣of␣the␣vector␣follow:\n";
    }

```

```cpp
    // now print the array elements themselves
    for (std::vector<int>::size_type i = 0u; i != k; ++i)
    {
        if (verbose)
        {
            std::cout << "Element␣[" << i << "]␣=␣";
            //std::cout << "Element at index " << i << " = ";
        }
        std::cout << v.at(i) << "\n";
    }
}
```

(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.

```cpp
void print_vector_all(const std::vector<int> & v, bool verbose)
{
    print_vector_head(v, v.size(), verbose);
}
```

Arguably it's slightly slower to call the existing function than to have a devoted function.

(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.

Note that you could also write this function so that it takes an existing vector as a non-const reference argument, and populates that vector with random numbers, instead of returning a new vector, but then you would have to decide what to do if the input vector was not initially empty at the start of the call to `make_vector_rand_range`, and it would be less tidy.

If the `mt19937` were taken by value, then the state of the random generator will be copied into the function, and so the state of the original one will not change through the function. Multiple successive calls to the function with the same arguments would return the same answer. Note that a const reference will not work, because generating random numbers modifies the generator object.

```cpp
std::vector<int> make_vector_rand_range(std::mt19937& rng, std::size_t n, int max
    ){
    std::vector<int> answer(n);
    std::uniform_int_distribution<int> uid(0,max);
    for(int& i : answer){
        i = uid(rng);
    }
    return answer;
}
```

(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`.

```cpp
std::vector<int>::size_type count_in_vec(const std::vector<int>& v, int i)
{
    std::vector<int>::size_type count = 0u;
    for (auto j : v)
    {
        if (j == i)
        {
            ++count;
        }
    }
    return count;
}
```

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?

I also defined a function `print_count` as follows:

```
void print_count(const std::vector<int> & v, int i)
{
    std::cout << "The␣vector␣contains␣" << count_in_vec(v, i) << "␣instances␣of␣"
      << i << ".\n";
}
```

My `main` and auxilliary main function were:

```
void main_vector_rand()
{
    std::size_t n;
    std::cout << "How␣big␣would␣you␣like␣the␣vector?␣";
    std::cin >> n;
    //n = 100000000;

    // generate the random numbers
    std::mt19937 rng(std::time(0));
    std::vector<int> v2(make_vector_rand_range(rng, n, 1000));
    //print_vector_all(v2, true);

    // check the frequencies of some of the random numbers
    print_count(v2, 22);
    print_count(v2, 0);
    print_count(v2, 1);
    print_count(v2, 2);
    print_count(v2, 998);
    print_count(v2, 999);
    print_count(v2, 1000);
    print_count(v2, 1001);
}

int main()
{
    main_vector_rand();

    return EXIT_SUCCESS;
}
```

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:␣";
        for(int i:input){
            cout<<i<<",";
        }
        cout<<endl;
    }else{
        cout<<"working"<<endl;
    }
}
```

Some possibilities

```
void repeat_vector_elements(vector<int>& v){
    vector<int> spare;
    for(int i : v){
        spare.push_back(i);
        spare.push_back(i);
    }
    v=spare; //or v.swap(spare);
}

void repeat_vector_elements(vector<int>& v){
    v.reserve(v.size()*2u); //optional
    for(auto i=v.begin();i!=v.end();++i){
        i=v.insert(i+1, *i);
    }
}
void repeat_vector_elements(vector<int>& v){
    //v.reserve(v.size()*2u); //optional
    for(auto i=v.size();i!=0;--i){
        v.insert(v.begin()+i, v.at(i-1));
    }
}
void repeat_vector_elements(vector<int>& v){
    const auto s = v.size();
    v.resize(s*2u);
    for(auto i=s-1;i<s;--i){
        v.at(i*2)=v.at(i*2+1)=v.at(i);
    }
}
```

The second suggestion is an example of a common idiom in C++ when inserting elements in an iterator loop. `insert` can invalidate iterators (although, in this case, if reserve is used it won't). We use the fact that `insert` returns a new iterator to the new location to keep our loop variable valid. Also note that we cannot write the loop as `for(auto i=v.begin(), e=v.end();i=e;++i)`! because e may become invalid.

(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.erase(v.begin()+i)`. When using the `erase` 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:␣";
        for(int i:input){
            cout<<i<<",";
        }
        cout<<endl;
    }else{
        cout<<"working"<<endl;
    }
}

```

When using the `erase` function, it is easiest to work backwards through the vector. This also means that to-be-removed elements won't get moved.

```
void remove_odd_elements(vector<int>& v){
    for(size_t i=v.size(); i>0; --i){
        if(v.at(i-1)%2 == 1){
            v.erase(v.begin()+i-1);
        }
    }
}

void remove_odd_elements(vector<int>& v){
    for(size_t i=0; i<v.size();){
        if(v.at(i)%2 == 1){
            v.erase(v.begin()+i);
        }else{
            ++i;
        }
    }
}

```

(The previous two examples also have versions with iterators.) Better:

```
void remove_odd_elements(vector<int>& v){
    size_t number_skipped = 0u;
    for(size_t source=0;source<v.size();++source){
        if(v.at(source)%2 == 1){
            ++number_skipped;
        }else{
            v.at(source-number_skipped) = v.at(source);
        }
    }
    v.resize(v.size()-number_skipped);
}

```

Note that, in this example, we are looking at two places in the vector at any time. It is easier to do a `for` loop over the source than the destination of the copies.

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`.

```cpp
#include<iostream>
#include <vector>
#include <functional>
double product(const vector<double>& v){
    double out = 1;
    for(double d : v){
        out *= d;
    }
    return out;
}
double summation(const vector<double>& v){
    double out = 0;
    for(double d : v){
        out += d;
    }
    return out;
}
void applyFunction(const vector<double>& v,
        std::function<double(const vector<double>&)> f){
    std::cout<<f(v)<<"\n";
}

int main(){
    std::vector<double> vec{1.4,2,2.8,8.34987,439};
    applyFunction(vec,product);
    applyFunction(vec,summation);
}

```

Remember to talk through your answers with the lab assistants!