IB9N7 C++ for Quantitative Finance
Worksheet 5

Vectors, random numbers
Hints and solutions

09 November 2015
(Week 6)

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.

Objectives for this lab session

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

- Know how to create vectors and to manipulate elements within those vectors.
- Reinforced your understanding of variables, operators, control structures, etc.
- Be able to generate random numbers (albeit with questionable statistical properties) between a certain range.

Exercises

Exercise 1: Vectors

(a) Try some of the examples from the lecture.

In particular, try the Fibonacci example.

Note that it is not actually necessary to use a vector to *calculate* Fibonacci numbers, but it is necessary if you want to *store* them.

Add code that outputs the indices and corresponding values of the fibonacci vector (just like was shown for `FX_rates`).

Calculate the 20th Fibonacci number.

```
#include <iostream>
#include <vector>
#include <cstdlib>

int main()
{
    std::vector<unsigned int> fibonacci(21u);
    // the first two elements get special treatment
    fibonacci.at(0u) = 0u; // already initialised, but make this clear
```

```cpp
    fibonacci.at(1u) = 1u;
    for (std::vector<unsigned int>::size_type i = 2u; i != fibonacci.size(); ++i)
    {
        fibonacci.at(i) = fibonacci.at(i - 2u) + fibonacci.at(i - 1u);
    }

    for (std::vector<unsigned int>::size_type i = 0u; i != fibonacci.size(); ++i)
    {
        std::cout << "fibonacci.at(" << i << ")␣=␣" << fibonacci.at(i) << std::
 endl;
    }

    return EXIT_SUCCESS;
}
```

Note: as usual, modular arithmetic issues may occur if you want to calculate large Fibonacci numbers. You do not need to work around these issues for this session.

(b) Saving your old version, rewrite your code so that the vector is initially empty, and pushes each number to the back once calculated.

See answer to next part.

(c) Can you tell when the vector's capacity has changed?

If so, can you identify any patterns?

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

int main()
{
    std::vector<unsigned int> fibonacci;
    std::vector<unsigned int>::size_type fib_last_capacity = fibonacci.capacity();
    // the first two elements get special treatment
    fibonacci.push_back(0u);
    fib_last_capacity = fibonacci.capacity();
    fibonacci.push_back(1u);
    fib_last_capacity = fibonacci.capacity();
    for (std::vector<unsigned int>::size_type i = 2u; i != 21u; ++i)
    {
        std::cout << "Inserting␣element␣with␣index␣" << i << "..." << std::endl;
        fibonacci.push_back(fibonacci.at(i - 2u) + fibonacci.at(i - 1u));

        if (fibonacci.capacity() != fib_last_capacity)
        {
            std::cout << "Capacity␣grown␣from␣" << fib_last_capacity << "␣to␣" <<
 fibonacci.capacity() << std::endl;
            fib_last_capacity = fibonacci.capacity();
        }
    }

    /*
    for (std::vector<unsigned int>::size_type i = 0u; i != fibonacci.size(); ++i)
    {
        std::cout << "fibonacci.at(" << i << ") = " << fibonacci.at(i) << std::endl;
    }
    */

    return EXIT_SUCCESS;
}
```

Note that when the vector needs to grow, the capacity always doubles.

(d) Suppose that you want to calculate *N* Fibonacci numbers, where *N* is very large (you can ignore modular arithmetic issues for this part).

Which version would be quicker and why?

Which version would be more likely to succeed and why?

The form that reserves enough memory upfront is quicker (no reallocation, copying, destroying) and more likely to succeed (other version effectively needs twice as much memory while reallocating and copying).

(e) If v is a vector<int>, comment on the following attempts to print its elements in reverse order.

```
for(auto i = v.size(); i>0; --i){ //(1)
    cout << v.at(i-1) << "\n";
}

for(auto s = v.size(), i=s-1; i<s ; --i){ //(2)
    cout << v.at(i) << "\n";
}

for(auto s = v.size(), i=0*s; i<s ; ++i){ //(3)
    cout << v.at(s-1-i) << "\n";
}

for(auto i = v.size()-1; i>=0 ; --i){ //(4)
    cout << v.at(i) << "\n";
}

for(int i = v.size(); --i>=0 ;){ //(5)
    cout << v.at(i) << "\n";
}

for(int i = v.size()-1; i>=0 ; --i){ //(6)
    cout << v.at(i) << "\n";
}
```

(1) works. (2) works but may confuse people because it relies on wrap-around of unsigned arithmetic. It would be better to set i=s-1u so that there is no confusion with the signed type. (3) works, and it uses the usual pattern of for loops. In both (1) and (3), the user has a formula every time they call at(), which may be a pain.

(4) is probably an infinite loop: the condition i>=0u is always true if i is an unsigned type. Using i>=0 is a comparison between signed and unsigned types which is always a bad idea; in this case it will probably also always be true. You will get an exception from a bad use of the at function.

(5) works. (6) works unless v is empty. There are platforms where vectors can grow larger than will fit in an int. The type std::ptrdiff_t is provided as an alias for a signed integer type which is always big enough.

Note that none of these options is very nice. Some of the comments on this blogpost get into this issue. http://herbsutter.com/2015/01/14/.

Exercise 2: Random numbers

One important use of vectors is for storing random, or pseudo-random, numbers. Pseudo-random numbers are not truly random, but are produced as part of a sequence by a deterministic algorithm. Good algorithms for this purpose produce sequences with good statistical properties.

C++ natively supports powerful capabilities to generate pseudo-random numbers (and if the implementation provides it, true random numbers too). We need #include<random>. There is a family of sources of randomness, and a family of distributions which we can generate from. As a source of randomness, we will rely on std::mt19937, which is the most common form of a Mersenne-Twister generator, which provides random numbers fast enough and well enough for most tasks.

A distribution object which generates int variables uniformly from a range of integers is std::uniform_int_distribution<int> and one which generates doubles from a range of reals is `std::uniform_real_distribution<double>`. The ranges are given as values when creating the distribution. For multiple random numbers, you need to use the same random generator objects each time, so that the pseudorandom sequence can progress. To generate a number you call the distribution as a function with the random generator object as an argument.

As an example, here is a program which prints three random integers between 1 and 39 inclusive, and one random number between 0 and 1.

```
#include <iostream>
#include <random>
using namespace std;
int main(){
    mt19937 mt;
    uniform_int_distribution<int> dist (1,39);
    uniform_real_distribution<double> dist2 (0,1);
    cout<<dist(mt)<<"\n";
    cout<<dist(mt)<<"\n";
    cout<<dist(mt)<<"\n";
    cout<<dist2(mt)<<"\n";
}
```

(a) Write a program which creates a vector of *N* random integer numbers (between 0 and 1000 inclusive), for some value *N* input by the user. Output these numbers to the console.

What do you notice when you run your program many times?

```
#include <iostream>
#include <random>
#include <vector>

int main()
{
    // get input from the user
    std::cout << "How␣many␣random␣numbers␣would␣you␣like?␣";
    std::vector<int>::size_type N = 0u;
    std::cin >> N;

    // declare the vector
    std::vector<int> v(N);

    // declare the random generator objects
    std::mt19937 mt;
    std::uniform_int_distribution<int> dist(0,1000);

    // fill up with random numbers
    for (auto s = v.size(), i=s*0; i != s; ++i)
    {
        v.at(i) = dist(mt);
    }

    // output the vector contents
    // note that these two loops need not be separate
    for (auto s = v.size(), i=s*0; i != s; ++i)
    {
        std::cout << "Element␣v.at(" << i << ")␣=␣" << v.at(i) << std::endl;
    }

}
```

On every run, the output is always the same:

```
Element v.at(0) = 815
Element v.at(1) = 135
Element v.at(2) = 906
Element v.at(3) = 835
Element v.at(4) = 127
Element v.at(5) = 969
Element v.at(6) = 914
Element v.at(7) = 221
Element v.at(8) = 632
Element v.at(9) = 308
```

(b) Adapt your program to compute — and display — the minimum and maximum of the values in the vector you generated in the previous part.

```
#include <iostream>
#include <random>
#include <vector>

int main()
{
    // get input from the user
    std::cout << "How␣many␣random␣numbers␣would␣you␣like?␣";
    std::vector<int>::size_type N = 0u;
    std::cin >> N;

    // declare the vector
    std::vector<int> v(N);

    // declare the random generator objects
    std::mt19937 mt;
    std::uniform_int_distribution<int> dist(0,1000);

    // fill up with random numbers
    for (auto s = v.size(), i=s*0; i != s; ++i)
    {
        v.at(i) = dist(mt);
    }

    // output the vector contents
    // note that these two loops need not be separate
    for (auto s = v.size(), i=s*0; i != s; ++i)
    {
        std::cout << "Element␣v.at(" << i << ")␣=␣" << v.at(i) << std::endl;
    }

    int mini = 10001, maxi = -1;
    for(int i : v){
        if(i<mini){
            mini=i;
        }
        if(i>maxi){
            maxi = i;
        }
    }

    std::cout<<"The␣minimum␣is␣"<<mini<<"␣and␣the␣maximum␣is␣"<<maxi<<".\n";
}
```

(c) You can send an integer when you create the `mt19937` which will be used as a seed to the generator. You could try inputting an unsigned int from the user and use this. This lets the user control whether they get the same numbers every time. For example

```
std::cout << "What␣seed?␣";
unsigned int x = 0;
std::cin >> x;
std::mt19937 mt(x);
```

(d) Sometimes you just want to get a different set of random numbers every time. One simple way to achieve this is to base a seed on the current time in the following way. You will need to `#include <ctime>`.

```
std::mt19937 mt(std::time(0));
```

This makes it harder to reproduce the results from the previous execution of the program, which may in turn make debugging harder.