Laboratory 3: Introduction to Templates in C++

The aims of today's lab are:

- Finish Lab 2 (the `StringInverter` class);
- Introduce the `vector` class of the Standard Template Library (STL);
- Understand the concept of template functions.

Task 0: Finish Last Week's Lab

Make sure you finish the `StringInverter` class from Lab 2!

Task 1: Using CMake

Same as before, use CMake to generate the project files for XCode, MS VC++, or Makefiles.

Task 2: STL Vector

To store X elements of the same type, C developers would tend to use an array. C++ programmers would favour the `vector` class provided by the STL. The first solution is not practical when X varies and would require dynamic allocation and recopies of the data, which makes it slow and inefficient.

For the first task of the week, you are given one C++ file (`src/TestVector.cpp`), which corresponds to a short program to get familiar with this `vector` class. At the moment it only contains some C code, your task is to improve it.

1. To use the `vector` class, add at the top of the file:

   ```
   #include <vector>
   ```

   The documentation is in `http://www.cplusplus.com/reference/vector/vector/`

2. a) In the main, create an empty vector of double precision floating point numbers. See below for an example (but of integers):

   ```
       // Create an empty vector of integers
       vector<int> vec;
   ```

   b) Double check the size of the vector, i.e. how many numbers are currently stored in the vector:

```
// Display the original size of vec
cout << "vector␣size␣=␣" << vec.size() << endl;
```

c) Add 50 random numbers between 0.0 and 1.0. To produce random numbers, you can use the function `randd()` that the C language provides. The header file you need is `<cstdlib>` (for `srand`, `randd()` and `time(0)`). It is already included (see L. 23 of `TestVector.cpp`). The code below shows how to add 5 values into the vector:

```
// Push 5 values into the vector
for(unsigned int i = 0; i < 5; ++i)
{
    vec.push_back(i);
}
```

d) Double check the size of the vector. Yes again, just to be sure it is 50.

3. a) Display every element of the vector. Vectors work a bit like C arrays. You can recycle the same `for` loop as L. 65 to L. 70. The problem is that you have to know the size of the array. Here, with the vector, do not hard code `50`. Use the `size()` method of the `vector` class.

   b) Display every element of the vector again. This time as a C++ developer: using a `const_iterator`, a `for` loop, and `std::cout <<`. The loop below shows how to sequentially access all the elements of the array using an iterator:

```
for (vector<int>::const_iterator ite = vec.begin();
        vec != vec.end();
        ++ite)
{
    int temp = *ite;
}
```

   The iterator (here `ite`) behaves like a pointer. To retrieve the value pointed by the iterator, use the `*` operator. To move to the next element, use the `++` operator (similarly, to move to the previous element, use the `--` operator).

4. In the previous `for` loop, replace `.begin()` by `.rbegin()`, and `.end()` by `.rend()`. What happened?

5. Instead of the `for` loop, use `std::copy` to display every element of the vector. To do so, you need to include another 2 header files: `#include <algorithm>` for `std::copy`, and `#include <iterator>` for `std::ostream_iterator`. In `http://www.cplusplus.com/reference/iterator/ostream_iterator/`, you can see an example. Two further examples are given below for your convenience:

```
// ostream_iterator example
#include <iostream>     // std::cout
#include <iterator>     // std::ostream_iterator
#include <vector>       // std::vector
#include <algorithm>    // std::copy

    ...
    ...
    std::ostream_iterator<int> out_it (std::cout,",␣");
    std::copy ( vec.begin(), vec.end(), out_it );
    // but most people would write:
    std::copy ( vec.begin(),
                vec.end(),
                std::ostream_iterator<int>( std::cout,", " )
              );
    ...
    ...
```

Adapt this code to your own problem. The main difference is the template argument. You have a vector of `doubles`, in the example it is an array of `ints`.

6. Remove the last N elements of the vector with:

$$N = 10 \times \text{randd}()$$

   You can use the method `vector::erase` or `vector::pop_back`. See `http://www.cplusplus.com/reference/vector/vector/erase/` and `http://www.cplusplus.com/reference/vector/vector/pop_back/`.

7. Now, display the

   a) smallest and

   b) largest values contained in the vector.

   To do so, use `std::min_element` and `std::max_element` provided in the `<algorithm>` header. Note that these functions return an iterator. Again, iterators are a bit like pointers. To display the value pointed by an iterator, add a $\star$ before it, e.g. `*ite`. See `http://en.cppreference.com/w/cpp/algorithm/min_element` for an example.

8. Finally, display the average value.

   a) To get the sum of all the elements of the vector, use `std::accumulate()` function. See `http://en.cppreference.com/w/cpp/algorithm/accumulate` for an example. It is provided by the `<numeric>` header. The last parameter is: `0` if you are summing integer numbers, `0.0` for double-precision floating-point numbers, or `0.0f` for single-precision floating-point numbers.

   b) To get the number of elements in the vector, use its `size()` method. You can't hard code 50 as the number of element because you just delete N elements...

Note that the STL vector class implements other functionalities, such as copy constructor, operator=+, etc. It can be used as an array (like in C) or as a FILO. See `http://www.cplusplus.com/reference/vector/vector/` for more information.

Task 3: Advanced C++ Programming: Create your own Template Functions

For this task, you are given three C++ files:

1. `include/utils.h`, a header file with the declarations of some functions (i.e. the functions' signature).
2. `include/utils.inl`, a header file with the definitions of these functions (i.e. the functions' implementation).
3. `src/TestUtils.cpp`, a test program to try your template functions.

Modify `include/utils.h` and `include/utils.inl` to convert every function as a template function. To implement every function, you need to write the code directly in `include/utils.inl`, which is included in the header file. This is due to the way C++ compilers handle inline and template functions. In `src/TestUtils.cpp`, test every template function with different data types.

For each function to convert from inline to template, there are 3 steps to follow:

1. Remove `inline` (in utils.inl). In C, we used macros as an optimized technique for compilers to reduce the execution time. In C++, we used inline functions (see `http://www.cplusplus.com/articles/2LywvCM9/` for an online article on this topic). The keyword `inline` needs to be remove for template functions.
2. Add `template <typename T>` at the front of every function declaration (in utils.h and in utils.inl). `template <typename T>` means that the corresponding function is template, and that the template type is `T`. It is a sort of virtual type that the compiler replaces during the compilation with the corresponding type (`int`, `double`, `char`, etc.).
3. Replaces occurrences of `int` by `T` (in `utils.inl`).

To call a template function, you can explicit the template type:

```
    std::cout << getMinValue<int>(1, 2) << std::endl;
```

or the compiler can try to guess (if it can't, a compilation error will be generated):

```
std::cout << getMinValue(1, 2) << std::endl;
```

Note: There are also template classes. The STL vector class is one of them.