C++ - Module 08

Templated containers, iterators, algorithms

Summary: *This document contains the subject for Module 08 of the C++ modules.*

Version: 7

Contents

Chapter I

General rules

For the C++ modules you will use and learn C++98 only. The goal is for you to learn the basic of an object oriented programming language. We know modern C++ is way different in a lot of aspects so if you want to become a proficient C++, developer you will need modern standard C++ later on. This will be the starting point of your C++ journey it's up to you to go further after the 42 Common Core!

- Any function implemented in a header (except in the case of templates), and any unprotected header means 0 to the exercise.

- Every output goes to the standard output and will be ended by a newline, unless specified otherwise.

- The imposed filenames must be followed to the letter, as well as class names, function names, and method names.

- Remember: You are coding in `C++` now, not in `C` anymore. Therefore:

  ○ The following functions are FORBIDDEN, and their use will be punished by a 0, no questions asked: `*alloc`, `*printf` and `free`.

  ○ You are allowed to use everything in the standard library. HOWEVER, it would be smart to try and use the C++-ish versions of the functions you are used to in C, instead of just keeping to what you know, this is a new language after all. And NO, you are not allowed to use the STL until you are supposed to (that is, until module 08). That means no vectors/lists/maps/etc... or anything that requires an include <algorithm> until then.

- Actually, the use of any explicitly forbidden function or mechanic will be punished by a 0, no questions asked.

- Also note that unless otherwise stated, the C++ keywords "using namespace" and "friend" are forbidden. Their use will be punished by a -42, no questions asked.

- Files associated with a class will always be `ClassName.hpp` and `ClassName.cpp`, unless specified otherwise.

- Turn-in directories are `ex00/`, `ex01/`, . . . , `exn/`.

- You must read the examples thoroughly. They can contain requirements that are not obvious in the exercise's description.

- Since you are allowed to use the `C++` tools you learned about since the beginning, you are not allowed to use any external library. And before you ask, that also means no `C++11` and derivatives, nor `Boost` or anything else.

- You may be required to turn in an important number of classes. This can seem tedious unless you're able to script your favorite text editor.

- Read each exercise FULLY before starting it! Do it.

- The compiler to use is `c++`.

- Your code has to be compiled with the following flags : `-Wall -Wextra -Werror`.

- Each of your includes must be able to be included independently from others. Includes must contain every other includes they are depending on.

- In case you're wondering, no coding style is enforced during in `C++`. You can use any style you like, no restrictions. But remember that a code your peer-evaluator can't read is a code they can't grade.

- Important stuff now: You will NOT be graded by a program, unless explicitly stated in the subject. Therefore, you are afforded a certain amount of freedom in how you choose to do the exercises. However, be mindful of the constraints of each exercise, and DO NOT be lazy, you would miss a LOT of what they have to offer

- It's not a problem to have some extraneous files in what you turn in, you may choose to separate your code in more files than what's asked of you. Feel free, as long as the result is not graded by a program.

- Even if the subject of an exercise is short, it's worth spending some time on it to be sure you understand what's expected of you, and that you did it in the best possible way.

- By Odin, by Thor! Use your brain!!!

Chapter II

Day-specific rules

- You will notice that in this particular subject, a lot of the problems you are asked to solve can be solved by NOT using standard containers and NOT using standard algorithms. However, using those is precisely the goal, and if you do not make every effort to use standard containers and algorithms wherever it's appropriate, you WILL get a very bad grade, however functional your work may be. Please don't be so lazy.

Chapter III

Exercise 00: Easy find

	Exercise : 00
	Easy find
Turn-in directory : *ex00/*	
Files to turn in : `easyfind.hpp main.cpp`	
Forbidden functions : `None`	

An easy one to start off on the right foot...

Make a template function called `easyfind`, templated on a type T, that takes a T and an int.

Assume the T is a container of int, and find the first occurrence of the second parameter in the first parameter.

If it can't be found, handle the error either using an exception or using an error return value. Take ideas from how standard containers work.

Of course, you will provide a `main` function that tests it thoroughly.

Chapter IV

Exercise 01: Span

	Exercise : 01
	Span
Turn-in directory : *ex01/*	
Files to turn in : `span.cpp span.hpp main.cpp`	
Forbidden functions : `None`	

Make a class in which you can store N `ints`. N will be an unsigned int, and will be passed to the constructor as its only parameter.

This class will have a function to store a single number (`addNumber`), that will be used to fill it. Attempting to add a new number if there are already N of them stored in the object is an error and should result in an exception.

You will now make two functions, `shortestSpan` and `longestSpan`, that will find out respectively the shortest and longest span between all the numbers contained in the object, and return it. If there are no numbers stored, or only one, there is no span to find, and you will throw an exception.

Below is a (way too short) example of a test `main` and its associated output. Of course, your `main` will be way more thorough than this. You have to test at the very least with 10000 numbers. More would be a good thing. It would also be very good if you could add numbers by passing a range of iterators, which would avoid the annoyance of making thousands of calls to `addNumber`...

```cpp
int main()
{
Span sp = Span(5);

sp.addNumber(5);
sp.addNumber(3);
sp.addNumber(17);
sp.addNumber(9);
sp.addNumber(11);

std::cout << sp.shortestSpan() << std::endl;
std::cout << sp.longestSpan() << std::endl;
}
```

```
$> ./ex01
2
14
$>
```

Chapter V

Exercise 02: Mutated abomination

	Exercise : 02
	Mutated abomination

Turn-in directory : *ex02/*
Files to turn in : `mutantstack.cpp mutantstack.hpp main.cpp`
Forbidden functions : `None`

Now that the appetizers are done, let's do some disgusting stuff.

The `std::stack` container is VERY cool, but it's one of the only STL containers that is NOT iterable. That's too bad. But why be okay with it, when we can simply play God and just butcher it to add some stuff we like?

You will splice this ability into the `std::stack` container, to repair this grave injustice.

Make a `MutantStack` class, that will be implemented in terms of a `std::stack`, and offer all of its member functions, only it will also offer an iterator.

Below is an example of code, the output of which should be the same as if we replaced the `MutantStack` with, for example, and `std::list`. You will of course provide tests for all of this in your `main` function.

```cpp
int main()
{
MutantStack<int>    mstack;

mstack.push(5);
mstack.push(17);

std::cout << mstack.top() << std::endl;

mstack.pop();

std::cout << mstack.size() << std::endl;

mstack.push(3);
mstack.push(5);
mstack.push(737);
//[...]
mstack.push(0);

MutantStack<int>::iterator it = mstack.begin();
MutantStack<int>::iterator ite = mstack.end();

++it;
--it;
while (it != ite)
{
    std::cout << *it << std::endl;
    ++it;
}
std::stack<int> s(mstack);
return 0;
}
```