



C++ - Module 08

Templated containers, iterators, algorithms

Summary:

This document contains the exercises of Module 08 from C++ modules.

Version: 7

Contents

I	Introduction	2
II	General rules	3
III	Module-specific rules	5
IV	Exercise 00: Easy find	6
V	Exercise 01: Span	7
VI	Exercise 02: Mutated abomination	9

Chapter I

Introduction

C++ is a general-purpose programming language created by Bjarne Stroustrup as an extension of the C programming language, or "C with Classes" (source: [Wikipedia](#)).

The goal of these modules is to introduce you to **Object-Oriented Programming**. This will be the starting point of your C++ journey. Many languages are recommended to learn OOP. We decided to choose C++ since it's derived from your old friend C. Because this is a complex language, and in order to keep things simple, your code will comply with the C++98 standard.

We are aware modern C++ is way different in a lot of aspects. So if you want to become a proficient C++ developer, it's up to you to go further after the 42 Common Core!

Chapter II

General rules

Compiling

- Compile your code with `c++` and the flags `-Wall -Wextra -Werror`
- Your code should still compile if you add the flag `-std=c++98`

Formatting and naming conventions

- The exercise directories will be named this way: `ex00`, `ex01`, ... , `exn`
- Name your files, classes, functions, member functions and attributes as required in the guidelines.
- Write class names in **UpperCamelCase** format. Files containing class code will always be named according to the class name. For instance: `ClassName.hpp/ClassName.h`, `ClassName.cpp`, or `ClassName.hpp`. Then, if you have a header file containing the definition of a class "BrickWall" standing for a brick wall, its name will be `BrickWall.hpp`.
- Unless specified otherwise, every output messages must be ended by a new-line character and displayed to the standard output.
- *Goodbye Norminette!* No coding style is enforced in the C++ modules. You can follow your favorite one. But keep in mind that a code your peer-evaluators can't understand is a code they can't grade. Do your best to write a clean and readable code.

Allowed/Forbidden

You are not coding in C anymore. Time to C++! Therefore:

- You are allowed to use almost everything from the standard library. Thus, instead of sticking to what you already know, it would be smart to use as much as possible the C++-ish versions of the C functions you are used to.
- However, you can't use any other external library. It means C++11 (and derived forms) and Boost libraries are forbidden. The following functions are forbidden too: `*printf()`, `*alloc()` and `free()`. If you use them, your grade will be 0 and that's it.

- Note that unless explicitly stated otherwise, the `using namespace <ns_name>` and `friend` keywords are forbidden. Otherwise, your grade will be -42.
- **You are allowed to use the STL in the Module 08 and 09 only.** That means: no **Containers** (vector/list/map/and so forth) and no **Algorithms** (anything that requires to include the `<algorithm>` header) until then. Otherwise, your grade will be -42.

A few design requirements

- Memory leakage occurs in C++ too. When you allocate memory (by using the `new` keyword), you must avoid **memory leaks**.
- From Module 02 to Module 09, your classes must be designed in the **Orthodox Canonical Form, except when explicitly stated otherwise**.
- Any function implementation put in a header file (except for function templates) means 0 to the exercise.
- You should be able to use each of your headers independently from others. Thus, they must include all the dependencies they need. However, you must avoid the problem of double inclusion by adding **include guards**. Otherwise, your grade will be 0.

Read me

- You can add some additional files if you need to (i.e., to split your code). As these assignments are not verified by a program, feel free to do so as long as you turn in the mandatory files.
- Sometimes, the guidelines of an exercise look short but the examples can show requirements that are not explicitly written in the instructions.
- Read each module completely before starting! Really, do it.
- By Odin, by Thor! Use your brain!!!



You will have to implement a lot of classes. This can seem tedious, unless you're able to script your favorite text editor.



You are given a certain amount of freedom to complete the exercises. However, follow the mandatory rules and don't be lazy. You would miss a lot of useful information! Do not hesitate to read about theoretical concepts.

Chapter III

Module-specific rules

You will notice that, in this module, the exercises can be solved WITHOUT the standard Containers and WITHOUT the standard Algorithms.

However, **using them is precisely the goal of this Module**. You are allowed to use the STL. Yes, you can use the **Containers** (vector/list/map/and so forth) and the **Algorithms** (defined in header `<algorithm>`). Moreover, you should use them as much as you can. Thus, do your best to apply them wherever it's appropriate.

You will get a very bad grade if you don't, even if your code works as expected. Please don't be lazy.

You can define your templates in the header files as usual. Or, if you want to, you can write your template declarations in the header files and write their implementations in .tpp files. In any case, the header files are mandatory while the .tpp files are optional.

Chapter IV

Exercise 00: Easy find

	Exercise : 00
Easy find	
Turn-in directory : <i>ex00/</i>	
Files to turn in : <code>Makefile</code> , <code>main.cpp</code> , <code>easyfind.{h, hpp}</code> and optional file: <code>easyfind.hpp</code>	
Forbidden functions : None	

A first easy exercise is the way to start off on the right foot.

Write a function template `easyfind` that accepts a type `T`. It takes two parameters. The first one has type `T` and the second one is an integer.

Assuming `T` is a container of integers, this function has to find the first occurrence of the second parameter in the first parameter.

If no occurrence is found, you can either throw an exception or return an error value of your choice. If you need some inspiration, analyze how `standard containers behave`.

Of course, implement and turn in your own tests to ensure everything works as expected.



You don't have to handle associative containers.

Chapter V

Exercise 01: Span

	Exercise : 01
Span	
Turn-in directory : <i>ex01/</i>	
Files to turn in : <code>Makefile</code> , <code>main.cpp</code> , <code>Span.{h, hpp}</code> , <code>Span.cpp</code>	
Forbidden functions : None	

Develop a **Span** class that can store a maximum of **N** integers. **N** is an unsigned int variable and will be the only parameter passed to the constructor.

This class will have a member function called `addNumber()` to add a single number to the Span. It will be used in order to fill it. Any attempt to add a new element if there are already **N** elements stored should throw an exception.

Next, implement two member functions: `shortestSpan()` and `longestSpan()`

They will respectively find out the shortest span or the longest span (or distance, if you prefer) between all the numbers stored, and return it. If there are no numbers stored, or only one, no span can be found. Thus, throw an exception.

Of course, you will write your own tests and they will be way more thorough than the ones below. Test your Span at least with a minimum of 10 000 numbers. More would be even better.

Running this code:

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

    sp.addNumber(6);
    sp.addNumber(3);
    sp.addNumber(17);
    sp.addNumber(9);
    sp.addNumber(11);

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

    return 0;
}
```

Should output:

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

Last but not least, it would be wonderful to fill your Span using a **range of iterators**. Making thousands calls to `addNumber()` is so annoying. Implement a member function to add many numbers to your Span in one call.



If you don't have a clue, study the Containers. Some member functions take a range of iterators in order to add a sequence of elements to the container.

Chapter VI

Exercise 02: Mutated abomination

	Exercise : 02
Mutated abomination	
Turn-in directory : <i>ex02/</i>	
Files to turn in : <code>Makefile</code> , <code>main.cpp</code> , <code>MutantStack.{h, hpp}</code> and optional file: <code>MutantStack.hpp</code>	
Forbidden functions : None	

Now, time to move on more serious things. Let's develop something weird.

The `std::stack` container is very nice. Unfortunately, it is one of the only STL Containers that is NOT iterable. That's too bad.

But why would we accept this? Especially if we can take the liberty of butchering the original stack to create missing features.

To repair this injustice, you have to make the `std::stack` container iterable.

Write a **MutantStack** class. It will **be implemented in terms of** a `std::stack`. It will offer all its member functions, plus an additional feature: **iterators**.

Of course, you will write and turn in your own tests to ensure everything works as expected.

Find a test example below.

```
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;
}
```

If you run it a first time with your `MutantStack`, and a second time replacing the `MutantStack` with, for example, a `std::list`, the two outputs should be the same. Of course, when testing another container, update the code below with the corresponding member functions (`push()` can become `push_back()`).