

C++ - Module 08

Templated containers, iterators, algorithms

Summary:

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

Version: 8

- 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!!!



Regarding the Makefile for C++ projects, the same rules as in C apply (see the Norm chapter about the Makefile).




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 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.

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.

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()`).

