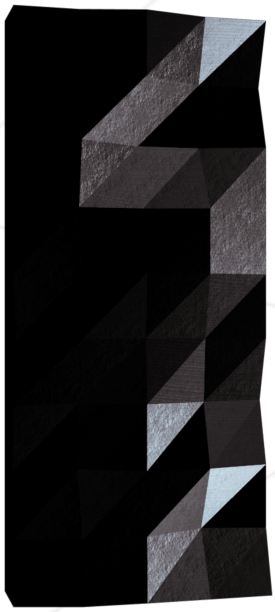

C++ - Module 08

Templated containers, iterators, algorithms

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

Version: 10.1

Contents

I	Introduction	2
II	General rules	3
III	Module-specific rules	6
IV	AI Instructions	7
V	Exercise 00: Easy find	9
VI	Exercise 01: Span	10
VII	Exercise 02: Mutated abomination	12
VIII	Submission and peer-evaluation	14

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.tpp`. 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 message must end with a newline character and be 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 code your peer evaluators can't understand is code they can't grade. Do your best to write 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 the C++-ish versions of the C functions you are used to as much as possible.
- 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 only in Modules 08 and 09.** That means: no **Containers** (vector/list/map, and so forth) and no **Algorithms** (anything that requires including 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 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 must use the STL — especially the **Containers** (vector/list/map/and so forth) and the **Algorithms** (defined in header `<algorithm>`) — whenever they are appropriate. 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

AI Instructions

Context

This project is designed to help you discover the fundamental building blocks of your 42 training.

To properly anchor key knowledge and skills, it's essential to adopt a thoughtful approach to using AI tools and support.

True foundational learning requires genuine intellectual effort — through challenge, repetition, and peer-learning exchanges.

For a more complete overview of our stance on AI — as a learning tool, as part of the 42 training, and as an expectation in the job market — please refer to the dedicated FAQ on the intranet.

Main message

- Build strong foundations without shortcuts.
- Really develop tech & power skills.
- Experience real peer-learning, start learning how to learn and solve new problems.
- The learning journey is more important than the result.
- Learn about the risks associated with AI, and develop effective control practices and countermeasures to avoid common pitfalls.

Learner rules:

- You should apply reasoning to your assigned tasks, especially before turning to AI.

- You should not ask for direct answers to the AI.
- You should learn about 42 global approach on AI.

Phase outcomes:

Within this foundational phase, you will get the following outcomes:

- Get proper tech and coding foundations.
- Know why and how AI can be dangerous during this phase.

Comments and example:

- Yes, we know AI exists — and yes, it can solve your projects. But you're here to learn, not to prove that AI has learned. Don't waste your time (or ours) just to demonstrate that AI can solve the given problem.
- Learning at 42 isn't about knowing the answer — it's about developing the ability to find one. AI gives you the answer directly, but that prevents you from building your own reasoning. And reasoning takes time, effort, and involves failure. The path to success is not supposed to be easy.
- Keep in mind that during exams, AI is not available — no internet, no smartphones, etc. You'll quickly realise if you've relied too heavily on AI in your learning process.
- Peer learning exposes you to different ideas and approaches, improving your interpersonal skills and your ability to think divergently. That's far more valuable than just chatting with a bot. So don't be shy — talk, ask questions, and learn together!
- Yes, AI will be part of the curriculum — both as a learning tool and as a topic in itself. You'll even have the chance to build your own AI software. In order to learn more about our crescendo approach you'll go through in the documentation available on the intranet.

✓ Good practice:

> I'm stuck on a new concept. I ask someone nearby how they approached it. We talk for 10 minutes — and suddenly it clicks. I get it.

✗ Bad practice:

> I secretly use AI, copy some code that looks right. During peer evaluation, I can't explain anything. I fail. During the exam — no AI — I'm stuck again. I fail.

Chapter V

Exercise 00: Easy find

	Exercise: 00
Easy find	
Directory: *ex*00/	
Files to Submit: `Makefile, main.cpp, easyfind.{h, hpp}` and optional file: `easyfind.tpp`	
Forbidden: `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 is of 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 VI

Exercise 01: Span

	Exercise: 01
Span	
Directory: *ex*01/	
Files to Submit: `Makefile, main.cpp, Span.{h, hpp}, Span.cpp`	
Forbidden: `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 far more thorough than the ones below. Test your `Span` with at least 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 of calls to `addNumber()` is so annoying. Implement a member function to add multiple numbers to your `Span` in a single 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 VII

Exercise 02: Mutated abomination

	Exercise: 02
Mutated abomination	
Directory: *ex*02/	
Files to Submit: `Makefile, main.cpp, MutantStack.{h, hpp}` and optional file: `MutantStack.tpp`	
Forbidden: `None`	

Now, it's time to move on to 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()`).

Chapter VIII

Submission and peer-evaluation

Turn in your assignment in your `Git` repository as usual. Only the work inside your repository will be evaluated during the defense. Don't hesitate to double check the names of your folders and files to ensure they are correct.

During the evaluation, a brief **modification of the project** may occasionally be requested. This could involve a minor behavior change, a few lines of code to write or rewrite, or an easy-to-add feature.

While this step may **not be applicable to every project**, you must be prepared for it if it is mentioned in the evaluation guidelines.

This step is meant to verify your actual understanding of a specific part of the project. The modification can be performed in any development environment you choose (e.g., your usual setup), and it should be feasible within a few minutes — unless a specific timeframe is defined as part of the evaluation.
You can, for example, be asked to make a small update to a function or script, modify a display, or adjust a data structure to store new information, etc.

The details (scope, target, etc.) will be specified in the **evaluation guidelines** and may vary from one evaluation to another for the same project.