# C++ Pool - d08

## Templated containers, iterators, algorithms

Staff 42 bocal@staff.42.fr

*Abstract:   This document contains the subject for day 08 of 42's C++ pool and, also, the game.*

# Contents

# Chapter I

# General rules

- 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 -42, no questions asked: *alloc, *printf and free.

  - You are allowed to use basically 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 actually are supposed to (that is, until d08). 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 -42, 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. If something seems ambiguous, you don't understand C++ enough.

- Since you are allowed to use the `C++` tools you learned about since the beginning of the pool, you are not allowed to use any external library. And before you ask, that also means no `C++11` and derivates, nor `Boost` or anything your awesomely skilled friend told you `C++` can't exist without.

- 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 ! Really, do it.

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

- 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 contains every other includes they are depending on, obviously.

- The subject can be modified up to 4h before the final turn-in time.

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

- Important stuff now : You will NOT be graded by a program, unless explictly 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 day 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 absolutely 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 !!!
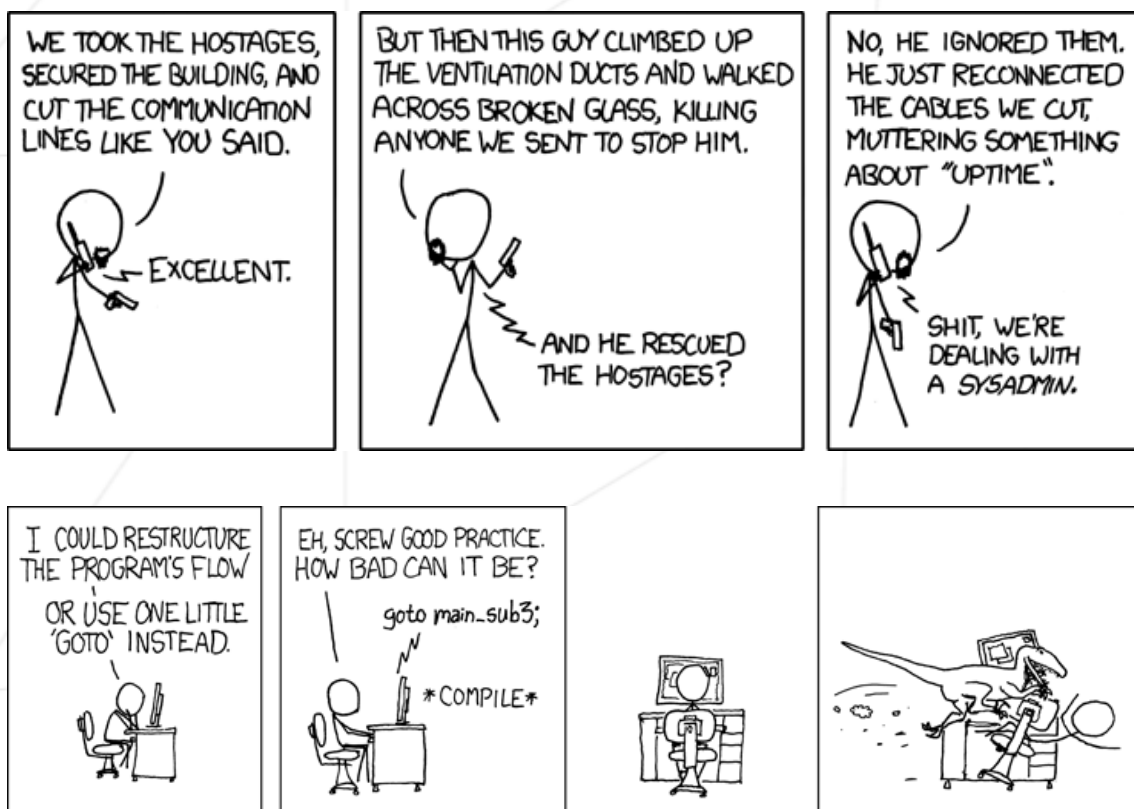
# Chapter II

# Day-specific rules

- Classes will have to be in Coplien's form, without us asking. It's a pre-requisite to have a positive grade. Be thorough.

- C-style casts are forbidden. You have to use C++-style casts.

- 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

# Foreword

Wisdom of the day :

# Chapter IV

# Exercise 00: Easy find

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

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 occurence 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 V

# Exercise 01: Span

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

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's 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` ...

```
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;
}
```

```
zaz@blackjack ex01 $ ./ex01
2
14
zaz@blackjack ex01 $
```

# Chapter VI

# 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` |
| Remarks : `n/a` |

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;
}
```

# Chapter VII

# Exercise 03: Open your mind, but please don't fuck it

| | Exercise 03 |
|---|---|
| | Open your mind, but please don't fuck it |
| Turn-in directory : *ex03/* | |
| Files to turn in : `main.cpp + Whatever you need` | |
| Forbidden functions : `None` | |
| Remarks : `n/a` | |

Brainfuck is a very cool programming language. Contrary to popular belief, it doesn't really mean "Open your mind", though. More like "have some intimate relations with your cerebellum".

We would like you to make a Brainfuck interpreter, however it would imply having you write "fuck" a lot of times, and since we don't like profanity, we would prefer that you do not write "fuck" too much. Because, you know, "fuck" is considered a bit vulgar, so it wouldn't be very professional to write "fuck" in your code. Additionally, if we write "fuck" in a subject, we would stoop to the lowest form of humor in an attempt to make you a tiny bit more interested in what you're doing here. So, no, we will not actually refer to it as Brainfuck in this subject, because that would mean writing "fuck", right here, in this very sentence. And that would be bad. I mean, come on, imagine writing "fuck" a lot of times in a school assignment just for shock value. Who would do that ?

So instead of coding a Brainfuck interpreter, you will code a Mindopen interpreter. What's Mindopen, you ask ? It's a way of writing Brainfuck without ever writing "fuck".

First, you will read up on Brainfuck (Google it). Then, you will define your Mindopen language by taking the very same instructions that exist in Brainfuck, and affecting other new symbols to them.

Then you will write a program that does the following :

- Open a file that contains Mindopen code

- Read this file, and for each instruction deciphered, create an Instruction derived object that represents the actual instruction to run, and queue it in a ... well, a queue of instructions in memory

- Close the file

- Execute each instruction in queue

In case it isn't obvious, this means you must also create a set of Instruction classes, one for each actual instruction in the language, that all have a method like `execute` or something, that will execute the actual instruction. You will also probably need an interface to manipulate all those instructions and store them all in the same container ...

A thorough and detailed `main` function will be expected, as well as some test files that are actual Mindopen programs to use.

# Chapter VIII

# Exercise 04: In Poland, expression evaluates you

| Exercise 04 | |
|---|---|
| In Poland, expression evaluates you | |
| Turn-in directory : *ex*04/ | |
| Files to turn in : `main.cpp + Whatever you need` | |
| Forbidden functions : `None` | |
| Remarks : `n/a` | |

In this last exercise, you will make a program that takes a mathematical expression as argument. In this expression, you will only find parentheses, integers (that will fit in an `int`), and the $+ - /*$ operators.

You must first tokenize this expression, i.e. convert it to a set of `Token` derived objects, then convert it to postfix (a.k.a Reverse Polish) notation.

When this is done, you must evaluate the expression, outputting every step to the standard output. By "each step", we mean the current input, resulting operation, and resulting stack state.

Of course, errors must be handled appropriately. Your `main` function must be detailed and easy to read.

Below is an example of what the output should look like:

```
zaz@blackjack ex04 $ ./ex04 "3 + ((1 + 4) * 2) - 1"
Tokens: Num(3) Op(+) ParOpen ParOpen Num(1) Op(+) Num(4) ParClose Op(*) Num(2) ParClose Op(-) Num(1)
Postfix : Num(3) Num(1) Num(4) Op(+) Num(2) Op(*) Op(+) Num(1) Op(-)
I Num(3) | OP Push      | ST 3]
I Num(1) | OP Push      | ST 1 3]
I Num(4) | OP Push      | ST 4 1 3]
I Op(+)  | OP Add       | ST 5 3]
I Num(2) | OP Push      | ST 2 5 3]
I Op(*)  | OP Multiply  | ST 10 3]
I Op(+)  | OP Add       | ST 13]
I Num(1) | OP Push      | ST 1 13]
I Op(-)  | OP Substract | ST 12]
Result : 12
```