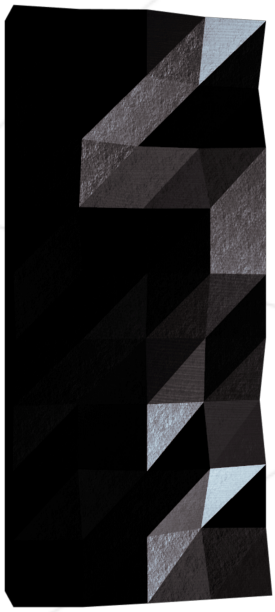

C++ - Module 09

STL

Summary:
This document contains the exercises of Module 09 from C++ modules.

Version: 2.2

Contents

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 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 explicitely 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

It is mandatory to use the standard containers to perform each exercise in this module.

Once a container is used you cannot use it for the rest of the module.

`It is advisable to read the subject in its entirety before doing the exercises.`

`You must use at least one container for each exercise with the exception of exercise 02 which requires the use of two containers.`

You must submit a `Makefile` for each program which will compile your source files to the required output with the flags `-Wall`, `-Wextra` and `-Werror`.

You must use c++, and your Makefile must not relink.

Your `Makefile` must at least contain the rules `$(NAME)`, `all`, `clean`, `fclean` and `re`.

Chapter IV

Exercise 00: Bitcoin Exchange

	Exercise : 00
Bitcoin Exchange	
Turn-in directory : *ex*00/	
Files to turn in : `Makefile, main.cpp, BitcoinExchange.{cpp, hpp}`	
Forbidden functions : `None`	

You have to create a program which outputs the value of a certain amount of bitcoin on a certain date.

This program must use a database in csv format which will represent bitcoin price over time. This database is provided with this subject.

The program will take as input a second database, storing the different prices/dates to evaluate.

Your program must respect these rules:

- The program name is btc.
- Your program must take a file as argument.
- Each line in this file must use the following format: "date | value".
- A valid date will always be in the following format: Year-Month-Day.
- A valid value must be either a float or a positive integer, between 0 and 1000.

`You must use at least one container in your code to validate this exercise. You should handle possible errors with an appropriate error message.`

Here is an example of an input.txt file:

```
$> head input.txt
date | value
2011-01-03 | 3
2011-01-03 | 2
2011-01-03 | 1
2011-01-03 | 1.2
2011-01-09 | 1
2012-01-11 | -1
2001-42-42
2012-01-11 | 1
2012-01-11 | 2147483648
$>
```

Your program will use the value in your input file.

Your program should display on the standard output the result of the value multiplied by the exchange rate according to the date indicated in your database.

If the date used in the input does not exist in your DB then you must use the closest date contained in your DB. Be careful to use the lower date and not the upper one.

The following is an example of the program's use.

```
$> ./btc
Error: could not open file.
$> ./btc input.txt
2011-01-03 => 3 = 0.9
2011-01-03 => 2 = 0.6
2011-01-03 => 1 = 0.3
2011-01-03 => 1.2 = 0.36
2011-01-09 => 1 = 0.32
Error: not a positive number.
Error: bad input => 2001-42-42
2012-01-11 => 1 = 7.1
Error: too large a number.
$>
```

Warning: The container(s) you use to validate this exercise will no longer be usable for the rest of this module.

Chapter V

Exercise 01: Reverse Polish Notation

	Exercise : 01
RPN	
Turn-in directory : $ex01/$	
Files to turn in : `Makefile, main.cpp, RPN.{cpp, hpp}`	
Forbidden functions : `None`	

You must create a program with these constraints:

- The program name is RPN.
- Your program must take an inverted Polish mathematical expression as an argument.
- The numbers used in this operation and passed as arguments will always be less than 10. The calculation itself but also the result do not take into account this rule.
- Your program must process this expression and output the correct result on the standard output.
- If an error occurs during the execution of the program an error message should be displayed on the standard error.
- Your program must be able to handle operations with these tokens: `"+ - / *"`.

`You must use at least one container in your code to validate this exercise.`

You don't need to manage the brackets or decimal numbers.

Here is an example of a standard use:

```
$> ./RPN "8 9 * 9 - 9 - 9 - 4 - 1 +"
42
$> ./RPN "7 7 * 7 -"
42
$> ./RPN "1 2 * 2 / 2 * 2 4 - +"
0
$> ./RPN "(1 + 1)"
Error
$>
```

Warning: The container(s) you used in the previous exercise are forbidden here. The container(s) you used to validate this exercise will not be usable for the rest of this module.

Chapter VI

Exercise 02: PmergeMe

	Exercise : 02
PmergeMe	
Turn-in directory : *ex*02/	
Files to turn in : `Makefile, main.cpp, PmergeMe.{cpp, hpp}`	
Forbidden functions : `None`	

You must create a program with these constraints:

- The name of the program is PmergeMe.
- Your program must be able to use a positive integer sequence as argument.
- Your program must use the merge-insert sort algorithm to sort the positive integer sequence.

To clarify, yes, you need to use the Ford-Johnson algorithm. (source: Art Of Computer Programming, Vol.3. Merge Insertion, Page 184.)

- If an error occurs during program execution, an error message should be displayed on the standard error.

You must use at least two different containers in your code to validate this exercise. Your program must be able to handle at least 3000 different integers.

It is strongly advised to implement your algorithm for each container and thus to avoid using a generic function.

Here are some additional guidelines on the information you should display line by line on the standard output:

- On the first line you must display an explicit text followed by the unsorted positive integer sequence.
- On the second line you must display an explicit text followed by the sorted positive integer sequence.
- On the third line you must display an explicit text indicating the time used by your algorithm by specifying the first container used to sort the positive integer sequence.
- On the last line you must display an explicit text indicating the time used by your algorithm by specifying the second container used to sort the positive integer sequence.

The format for the display of the time used to carry out your sorting is free but the precision chosen must allow to clearly see the difference between the two containers used.

Here is an **example** of a standard use:

```
$> ./PmergeMe 3 5 9 7 4
Before:  3 5 9 7 4
After:   3 4 5 7 9
Time to process a range of   5 elements with std::[..] :  0.00031 us
Time to process a range of   5 elements with std::[..] :  0.00014 us
$> ./PmergeMe `shuf -i 1-100000 -n 3000 | tr "\n" " "`
Before:  141 79 526 321 [...]
After:   79 141 321 526 [...]
Time to process a range of 3000 elements with std::[..] : 62.14389 us
Time to process a range of 3000 elements with std::[..] : 69.27212 us
$> ./PmergeMe "-1" "2"
Error
$> # For OSX USER:
$> ./PmergeMe `jot -r 3000 1 100000 | tr '\n' ' '`
[...]
$>
```

The indication of the time is deliberately strange in this example. Of course you have to indicate the time used to perform all your operations, both the sorting part and the data management part.

Warning: The container(s) you used in the previous exercises are forbidden here.

The management of errors related to duplicates is left to your discretion.

Chapter VII

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.

16D85ACC441674FBA2DF65190663F33F793984B142405F56715D5225FBAB6E3D6A4F
167020A16827E1B16612137E59ECD492E47AB764CB10B45D979615AC9FC74D521D9
20A778A5E