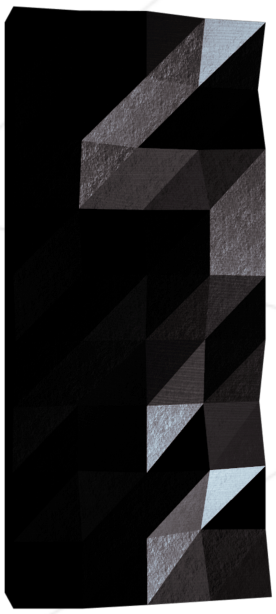

Abstract_data

C++ containers, hard mode

*Summary: The standard C++ containers have all a specific usage.
To make sure you understand them, let's re-implement them!*

Version: 1.1

Contents

Chapter I

Forewords

Even now C++ inheritance is not of much use for generic programming. Let's discuss why.
Many people have attempted to use inheritance to implement data structures and container classes. As we know now, there were few if any successful attempts. C++ inheritance, and the programming style associated with it are dramatically limited. It is impossible to implement a design which includes as trivial a thing as equality using it. If you start with a base class X at the root of your hierarchy and define a virtual equality operator on this class which takes an argument of the type X, then derive class Y from class X. What is the interface of the equality? It has equality which compares Y with X. Using animals as an example (OO people love animals), define mammal and derive giraffe from mammal. Then define a member function mate, where animal mates with animal and returns an animal. Then you derive giraffe from animal and, of course, it has a function mate where giraffe mates with animal and returns an animal. It's definitely not what you want.
While mating may not be very important for C++ programmers, equality is. I do not know a single algorithm where equality of some kind is not used.

Alexander Stepanov (designer of the STL) in this interview

Chapter II

Objectives

In this project, you will implement a part of the C++ containers from the standard template library.

You have to take the structure of each standard container as reference. If a part of the Orthodox Canonical form is missing in it, do not implement it.

As a reminder, you have to comply with the C++98 standard, so you don't have to implement any later feature of the containers, but every C++98 feature (even deprecated ones) is expected.

Chapter III

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`
- You have to turn in a `Makefile` which will compile your source test. It must not relink.
- Your `Makefile` must at least contain the rules:
  `$(NAME)`, `all`, `clean`, `fclean` and `re`.

Formatting and naming conventions

- Your classes must be usable by adding `-I <root of your project>/include` and including the sames header than for the std lib, for example `map` and `multimap` are included in `map.hpp`, `pair` in `utility.hpp`...

Allowed/Forbidden

- You are only allowed to use the c++ standard library and only in the cases explicitly allowed in this document.

A few design requirements

- Memory leakage occurs in C++ too. When you allocate memory, you must avoid **memory leaks**.
- 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) and organize your work as you wish as long as you turn in the mandatory files.
- By Odin, by Thor! Use your brain!!!

Chapter IV

Mandatory part

Implement the following containers:

IV.1 Sequence containers

- `list`
- `deque`
- `vector`
  You don't have to do the `vector<bool>` specialization.

IV.2 Associative containers

- `map`
- `set`
- `multimap`
- `multiset`

`If you're smart, you shouldn't have to write your binary search tree's code four times. Once you've done one of the associative containers, it shouldn't take a lot of time to complete the three others.`

IV.3 Container adaptors

- `stack`
- `queue`
- `priority_queue`

Your container adaptors will use your `vector` and `deque` classes as default underlying containers. But they must still be compatible with other containers, the STL ones included.

IV.4 Requirements

- The namespace must be `ft` and each function or type that you reimplement must be implemented in the `ft` namespace.
- Each inner data structure used in your containers must be logical and justified (this means using a simple array for `map` is not ok).
- You cannot implement more public methods in your containers than the ones offered in the standard containers. Everything else must be private or protected. Each public function or variable must be **justified**.
- All the member functions, non-member functions and overloads of the standard containers are expected.
- You must follow the original naming. Take care of details.
- If the container has an **iterator** system, you must implement it.
- You must use `std::allocator`.
- You are not allowed to make reference to the `std` namespace except for the following types:
  - `std::allocator`
  - `std::string`
  - `std::numeric_limits`
  - iterator_tags (you must define all your iterator tags as typedefs of the `std`'s ones)

  you must reimplement every other function, constant or type that you might need (yes, even exceptions).
- The non-member functions and classes (for example `swap()`) must be implemented in the `ft` namespace.
- Each use of `friend` must be justified and will be checked during evaluation.
- You **must** follow the **complexities** requested by the ISO.

As our good old Alexander Stepanov would say:

"We need to separate the implementation from the interface but not at the cost of totally ignoring complexity. Complexity has to be and is a part of the unwritten contract between the module and its user. The reason for introducing the notion of abstract data types was to allow interchangeable software modules. You cannot have interchangeable modules unless these modules share similar complexity behavior. If I replace one module with another module with the same functional behavior but with different complexity tradeoffs, the user of this code will be unpleasantly surprised. I could tell him anything I like about data abstraction, and he still would not want to use the code. Complexity assertions have to be part of the interface."

You can use https://www.cplusplus.com/
and https://cppreference.com/ as references
but in case of doubt you must refer to the iso 14882-1998.

IV.5 Testing

You will find some testers attached to this subject that will be tested during the evaluation.

- In addition you must also provide your tests, at least a `main.cpp`, for your defense.
- You must produce two binaries that run the same tests: one with your containers only, and the other one with the STL containers.
- Compare **outputs** and **performance / timing** (your containers can be up to 20 times slower).
- Test your containers with: `ft::<container>`.

Since the max_size() value is different in each implementations, you don't need to have the same return values as the STL for this method.

Chapter V

Bonus part

V.1 More!

Implement the following containers:

- `unordered_map`
- `unordered_set`
- `unordered_multimap`
- `unordered_multiset`

Since these containers don't exist in C++98, you will take the C++11 version as reference, remove all the methods and types that are not feasible in C++98, and use the allocators the way they are in C++98.

V.2 Your own container

Create a containers that doesn't exist in any version of the STL but provide a real plus compared to existing containers.
Your container must follow the conventions of the STL. For example:

- The naming conventions (If your container has a function `push_back()`, it must be named identically. Not `pushback()`, and not `back_push()`.) This is crucial for C++ templating.
- The containers signatures (All your containers must have an allocator template argument and this allocator must be used. No type must be missing from the class (`value_type`, `pointer_type`, `iterator`, ...), unless it makes sense to not have any like in the `stack`.)
- Your containers' methods (If your container is a sequence container, it should have a function `insert()` as specified here.).

You will document your methods in a comment above. The documentation must include:

- a description of the function
- a description of the parameters and return value

- the exceptions that can be throwed by the function
- the iterator and reference invalidations
- the complexity of the function

For example, a fibonnacci heap would be a good container to implement, it provide different complexities than a binary heap and cannot be implemented as an adaptor.

The points of this last part will be applied only if you have completed the first bonus part.

The bonus part will only be assessed if the mandatory part is PERFECT. Perfect means the mandatory part has been integrally done and works without malfunctioning. If you have not passed ALL the mandatory requirements, your bonus part will not be evaluated at all.

Chapter VI

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 files to ensure they are correct.