Homework. 4: The C++ STL Library

Ignacio Vizzo, Rodrigo Marcuzzi, E-Mail ivizzo@uni-bonn.de, rmarcuzzi@uni-bonn.de

Handout : 08.11.2021
Handin: 21.11.2021 at 23:59:59 (CET)

General rules

1. You need to provide the build system for this homework. This means, you need to provide as many **CMakeLists.txt** files as you think is needed.
2. Your code will be checked using `clang-format`. If your code in not properly formatted, then the tests will not run.
3. Your code will be static-analyzed by `clang-tidy`, if your code does not comply with the analysis, the tests will not run. If you are using VSCode with the reccomended setup, most of the errors will be visible while you program.
4. **ALL** tasks should be solved within the `homework_4` folder, no need to create `task_x` folders.
5. The design of how to solve this exercise is up to you. The **ONLY** requirement to pass the tests are:
   - You should provide one header file(put it where you preffer) and call it `homework_4.h`.
   - All the functions of this exercise must be accessible form this header file
   - You should guarantee this header file can be included thorugh your build system.

A The C++ STL Library (10 points)

You learnt how to use STL containers and STL algorithms,and you probably love it, but for this exercise let's pretend that you don't like quite much the `std::vector` STL container. You want to use all the functionality provided in `std::vector` but you also want to "augment" it naming ALL the vectors of your applications.

You also love to use the algorithms defined in the STL library, but for your particular application, you want to avoid the explicit usage of iterators. You know that this will impact the flexibility when using this algorithms, but you still want to take the risk.

Therefore, your task is twofold:

1. Design an `ipb::named_vector<T>` that behaves **exactly** as a `std::vector`. This means, that the user of this container should not know ANYTHING about the implementation, you should be able to use it as if it was a standard vector.
2. Design a small utility library(`ipb_algorithm`) to wrap-up some STL algorithms of interest to your brand new container.

A.1 `ipb::named_vector` container

To provide a similar interface for your container you will need to use template types. It's not the focus of this exercise, and we will cover this topic later on during the lectures.

A template type is virtually "any" type in C++, and user-defined types. It's a common practice to call this types "T". `T` can be `int`, `std::string`, `AnyType`, and so on ... This is the mechanism that is used all across the STL library to provide generic functionality for any kind of data.

When you create a `std::vector`, you are basically using templates:

```cpp
std::vector<int> v1; // T == int
std::vector<std::string> v2; // T == std::string
std::vector<Bananas> v3; // T == Bananas
```

With this in mind, design a new container, that behaves as an std::vector, but has also a name. You will need to use template parameters for this.

This sounds more complicated than it actually is, as an example, you can start from something like this.

```cpp
template <typename T>
struct my_container {
  T some_variable_inside_this_container_;
  T some_other;
  int one_int_value_;
  float one_float_value_;
  // ...
};
```

A.1.1 Requirements to pass the tests

- You should provide access to the data members through a member function called **vector()** and access to the name of the container thorugh a member function called **name()**;
- Your `ipb::named_vector` implementation should be accessible by only including the homework API(`homework_4.h`).
- You need to make your container behave as an standard vector, the interface should be the same. For this, you must guarantee that:
  1. The `size()` of the container is equal to the size of the data elements plus the size of the name.
  2. Your container it's considered to be empty if the name is empty **OR** there are no data elements on it.
- Your container must provide the same resizing functionality as an `std::vector`, this means that the **reserve(), resize(), capacity()** methods should be also implemented.

A.1.2 *Tips*

1. The solution for this particular task is actually extremely simple if you know how to do it properly.
2. The solution might be given in less than 20 lines of code(depending on how you actually solve the task.)

A.2 The ipb_algorithm library

Let's prettend that you need to write a small application and for this you plan to use some algorithms from the STL library(NOTE: this application is not part of this exercise).

You are also working with other developers, but they don't like iterators at all. You are in charge of designing and writing the library that implements some basic functionality. You have **complete** freedom on how to design and implement this library, BUT, the only requirement is that the library interfaces should not take as input arguments any sort of iterator.

To keep it simple, write this library just to use your `ipb::named_vector`. And from here you can pick one of theese two options(no difference in the points):

1. Implement the library just using `ipb::named_vector<int>`.
2. Implement the library to use any kind of `ipb::named_vector`.

A.2.1 ipb_algorithm library specification

Name	**Description**	**Input Example**	**Output Example**	Extra Arguments
accumulate	returns the sum over all the elements in the data container	”name”, 4, 2, 1, 3	10	-
count	return how many elements are stored in the container.	”name”, 4, 2, 1, 3,	1	count for 1
all_even	returns true if all values in the container are even, false otherwise.	”other name”, 4, 2, 6, 8	TRUE	-
clamp	clamp all the values in the container to a given range [min, max]	”Grandote”, 42, 12, 6, -18	”Grandote”, 15, 12, 6, 0	clamp to [0, 15]
fill	Fill all the elements in the container with a given value	”Grandote”, 42, 12, 6, -18	”Grandote”, -99, -99, -99, -99	fill with -99
find	returns true if a given value is in the container	”name”, 4, 2, 1, 3	TRUE	find 1
print	print everything inside the container to the standard output	”Jose Luis” , 1, 2, 3, 4	”Jose Luis” : 1, 2, 3, 4	-
toupper	converts the name(if any) of the container to upper case	”Jose Luis” , 1, 2, 3, 4	”JOSE LUIS” , 1, 2, 3, 4	-
sort	Sort the values in the container	”any name”, 15, 12, 6, 0	”any name”, 0, 6, 12, 15	-
rotate	shift rotate the values in the container a given number of posissions	”any name”, 0, 6, 12, 15	”any name”, 12, 15, 0, 6	rotate by 2
reverse	reverse(mirror) the values in the container	”JOSE LUIS” , 1, 2, 3, 4	JOSE LUIS , 4, 3, 2, 1	-

A.2.2 Requirements to pass the tests

- Provide the `ipb_algorithm` library from your build system, the tests will link against this library,
- To avoid conflicts, place all your algorithms implementation inside the namespace **ipb**.
- The types used for the function prototypes must be consistent, this means that you shall not count for a floating point for a given integer type container(using the count algorithm).

NOTE: In the files provided with this homework sheet and inspect the **tests** folder. Tests are written in such a way that you can actually infer the desired functionality by just reading the sources files, no matter if you know the unit test framework or not. If you find this step confusing, just skip it for the moment.