Programmazione e Amministrazione di Sistema

C++ Project report
February 2019

Rota Claudio
816050
c.rota30@campus.unimib.it

Contents

1 Introduction

MySet class is a class that contains template data and it does not allow to have duplicate elements. At the beginning, i thought to implement this class with a static array because it's very fast to access to a random data, optimizing operator[]. Using a static array, unfortunately, makes very heavy the insertion and deletion operations. For this reason, i decided to implement this class with a simple linked list data structure. I chose it because i preferred optimizing the execution time of the insertion and deletion instead of random access, two operation against one. Moreover, in almost all cases, i have to iterate among the elements of the collection from the first one to the last one. The file that contains MySet class has extension *.hpp* because the class is a template class.

2 Template parameters

The class is a collection of template elements, and to specify the type of the elements, it needs to take in input a template parameter, called T, like *int*, *std::string* and so on. Since i can't establish when two T values are equals, i need to let that something does it for me. The best way is using a functor that implements *operator*(). For this reason, i need another template parameter and i called it E. So, MySet class must take in input two template parameters, T, which represents the type of the elements, and E, which represents the functor to check the equality.

3 Member variables

Since i chose a linked list data structure, the element in the collection is wrapped into a Node struct that contains a value T and a Node pointer to the next node of the collection. MySet class contains this Node struct as private, keeping it invisible. Node struct has an extra constructor that takes in input a value T by reference. To keep a collection of elements like MySet does, a Node pointer member variable _head is enough. However, i put into MySet class another Node pointer member variable, _tail, that is used during the execution of *add* method to find immediately the end of the collection. I have introduced another member variable, called _size, used to store the current number of elements that there are in MySet object. It is useful to

check, for example, if the index passed to an operator[] method is not greater then the number of elements that there are in the collection, as well as to check if the insertion operation and deletion operation have been successfully done. _size, like _head and _tail, is updated when a new element is added and when an existing one is removed. Since that the comparison between two element is possible only through a functor passed as template parameter, i created an _equal member variable used to check if two elements are equals.

4 Implemented methods

According to the requests of the text of the project, i summarize briefly here the methods i implemented and the reasons for why i chose to implement them in that way.

4.1 Member methods

4.1.1 operator[]

It takes in input an *unsigned int* as index. I used an *unsigned int* because, being an index, it can't be negative. In this way, if someone tries to call this method with a negative number, the compiler warns him immediately. This method checks as first thing if the index in input is not greater then the size of the collection, represented by _size member variable. To check this, i used an assertion instead of throwing an exception because i think *range-out-of-bound* is a serious error and i preferred to stop the program if it happens. *Operator*[] iterates among the nodes of the collection until it reaches the position specified as input parameter. Since the text of the project specifies that this operator must not allow to modify the element returned, i used a *const* modifier at the beginning of the signature. This method returns a reference to the element situated in the position specified but it is read-only. I used a local T pointer variable to store the element, and then i return the content of the memory to which the pointer pointed, using the *operator**. This is the only way to store locally an element and then return his reference.

4.1.2 add

It takes in input a constant reference to an element of type T. The method creates a new Node in the heap with the element passed, thanks to his constructor which tanks in input a reference to an element T, and tries to insert it in the collection. If the collection is empty, the method inserts the node created as first element, otherwise it iterates from the first to the last node of the collection. The method uses the private helper method *findValue* to iterate among the nodes of the collection. If the element is found, *add* throws an exception with an error message to warn that the element already exists, otherwise it inserts the node created at the end of the collection, using _tail member variable. If the element will be found, before throwing the exception, i delete the node created to avoid memory leaks.

4.1.3 remove

Like *add* method, *remove* takes in input a constant reference to an element of type T. This method starts to iterate from the first node of the MySet object until the element will be found. If this element is not found, *remove* throws an *std::runtime_error* with an error message to warn that the element is not found inside of the collection.

4.1.4 constructor from two const_iterator

It is the last member method that i have implemented. Since MySet class is a template class, i want that this method takes in input any type of iterators and to do this, i used a template type Q for the iterators. In this way it's possible to pass any type of iterator you want. Using the iterators, it's possible to create a new MySet object from them. I left the type-check to the compiler, using *static_cast*. As all constructor i have implemented, all their code are in a *try/catch* clause so, if something goes wrong, *removeAll* method is called in a *catch* clause and this keeps the object in a coherent state or, better, it deletes all nodes created by him in the heap. In all constructor, after the calls to *removeAll* method, i broadcast the exception with *throw* statement to warn who has called the method that something is happened.

4.1.5 operator<<

I have implemented this method after the constant iterator, because i would to use them into it. As usual, i used the constant iterators in a *while* loop and i have sent the values on the *std::ostream* took in input, using *operator** previously defined. For a pure aesthetic goal, i have printed curly brackets at the beginning and at the end.

4.2 Global methods

4.2.1 filter_out

This method is located among the final methods in file *myset.hpp*, and it takes as parameters a constant reference to a template MySet, with T and E template parameters, and a constant reference to a template type P predicate functor, according to the text of the project. In this method, i create a constant iterator from MySet passed as parameter, and i used it to iterate on his elements. I create an helper local variable of the same type of MySet passed and i fill it using *add* methods, only if the current element does not satisfy the predicate functor. After having iterate among all the elements, i returned the helper local variable filled by copy. The reason why i chose to return it by copy is because, initially, i returned it by reference, creating it in the heap. This, however, required an explicit *delete* by the user to avoid memory leaks at the end of the program, and so i decided to return it by value, avoiding explicit *delete*.

4.2.2 operator+

This is the final method implemented. It takes in input two constants reference to a template MySet, with T and E template parameters. To do the concatenation between the first and the second MySet object, i created a constant iterator from the second MySet object, and so i used it to add each of his element to the first one, using *add* method. As *add* method works properly, i am sure that, at the beginning, both two MySet objects haven't duplicate elements, so i can start the concatenation from the first one without checking if it has duplicate elements. If the method tries to add an element that already exists, *add* method will throw an exception, as specified in *add* description. Finally, i return the resulting MySet by copy, for the same reason that i have explained in *filter_out* description.

4.3 Helper methods

All the following helper methods are private.

4.3.1 findValue

To help *add* method to check if an element exists or not, i implemented *findValue* that takes in input a constant reference to an element and iterates among all nodes of the collection. It return true only if finds this element inside the collection, using _equal attribute, and otherwise false. I implemented this method to reduce the complexity of *add* method. At the beginning, i used this helper method also in *remove* method, for the same reason i used it for *add* method. But i realized that it is not necessary iterate among all nodes of the collection, but until the element will be found, and this could be happen immediately. I decided to keep this method private because is not required from text of the project, but i am sure that it works properly because i use this method whenever i use *add* method.

4.3.2 removeAll

Another helper method is *removeAll*, that allows to remove all elements from the collection. It is used by all constructor of MySet class. This method iterates among all nodes and delete them, one by one. After it has finished, the _size, the _head and the _tail are set to 0 and all elements have been removed from the collection and from the heap. I decided to keep this method private because i don't want to allow to delete all elements with one method, but if someone wants to do this, he can use a loop using the _size.

4.4 Exceptions

Since i have to throw an exception when an element already exists during an insertion or when an element does not exist during a deletion, i chose to throw an *std::runtime_error* because i think it is the most indicated type of *std::exception*.

4.5 Iterators

According to the text of the project, i have implemented a const_iterator to allow to iterate over the elements, and i chose *forward_iterator* type because

i have used a linked list data structure. Moreover i think that, since is not an ordered collection, it does not make any sense to iterate in a random mode.

5 Main

I created *main.cpp* file in order to compile *myset.hpp* as well as to test MySet class. At the beginning, it was composed only by a simple *main* method, in order to compile *myset.hpp*. After i have finished MySet class, i have fill it with a series of tests written first using *int*, then *std::string* and then using Person struct, created with the goal to test MySet with a custom type. To check the correctness of the methods that i have implemented i used the assertions.

5.1 Person struct

For testing MySet class with custom types, i chose to implement a simple Person struct which has two attributes of *std::string* type and one of *int* type. Over then the fundamental methods, it has also an *operator*() method to check if two person are equals. They are equals if they have the same name, the same surname and the same age.

5.2 Methods and tests

I created *mySetInt*, *mySetString* and *mySetPerson* which help to create a specific MySet used in the tests. Each test checks every methods of MySet class, and, in particular, *remove* method is tested trying to remove two elements on the head, two elements in the middle and two elements at the end.

5.3 Typedef

For avoiding to define every times MySet objects, i defined three types of data:

- MySetInt used with *int* type and *equal_int* as functor to check equality;
- MySetString used with *std::string* type and *equal_string* as functor to check equality;

- MySetPerson used with *Person* type and *equal_person* as functor to check equality.

5.4 Functors

Here there are a list of the functors that i have created for the tests of MySet class. All functors that i have defined implement a *operator*(), that is a boolean method.

- equal_int, used to check if two *int* are equals;
- equal_string, used to check if two *std::string* are equals;
- equal_person, used to check if two *Person* are equals;
- is_odd, used to test the method *filter_out* with *int* type, that returns true only if the number passed as parameter is odd.
- even_length, used to test the method *filter_out* with *std::string* type, that returns true only if the length of the string passed as parameter has even length;
- is_over_25, used to test the method *filter_out* with *Person* type, that returns true only if the age of the person passed is greater then 25.

Since the compiler suggested me to don't use == with *std::string*, i used *compare* method to check if two *std::string* are equals or not. For the same reason, during the tests, i preferred to use a functor *equal_string* and *equal_person* with *std::string* and *Person* types.

6 Execution times

- Access time:
  - $\Omega(1)$ if you want to access to the first element.
  - $\mathcal{O}(n)$ if you want to access to the last one.
- Insertion time: $\Theta(n)$
- Deletion time:
  - $\Omega(1)$ if the element to delete is the first element of the collection.
  - $\mathcal{O}(n)$ if the element to delete does not exist or it is the last one.