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 is very fast to access to random data, optimizing operator[]. Unfortunately, using a static array makes very heavy 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 insertion and deletion instead of random access, two operations 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 containing the MySet class has *.hpp* as extension because MySet is a template class.

2 Template parameters

The class is a collection of template elements and, to specify the type of the elements, it requires as 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 used a functor implementing *operator*(). For this reason, I need another template parameter and I called it *E*. So, MySet class must take as 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 containing a value *T* and a Node pointer pointing to the next node of the collection. This Node struct is private, so that it is not visible and it has an extra constructor taking as input a value *T* by reference. To keep a collection of elements, a Node pointer member variable _head is enough. However, I put into MySet class another Node pointer member variable, _tail, which is used during the execution of the *add* method to immediately find the end of the collection. I have introduced another member variable, called _size, used to store the current number of elements that are in a MySet object. It is useful to check, for example, if the index passed to the operator[] method is not greater then the number of elements that currently are in the collection, as well as to check if insertion and deletion operations have been successfully completed. _size, like _head and _tail, is updated when a new element is added and when an existing one is removed. Since the comparison between two elements is possible only through a functor passed as template parameter, I created an _equal member variable used to check whether two elements are equal.

4 Implemented methods

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

4.1 Member methods

4.1.1 operator[]

It takes as input an *unsigned int* as index. I used an *unsigned int* because, being an index, it can not be negative. In this way, if someone tries to call this method with a negative number, the compiler immediately warns him. First of all, this method checks whether the index passed as input is not greater then the size of the collection, represented by the _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 stopping 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 returned element, I used the *const* modifier at the beginning of the signature. This method returns the 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 points, using *operator**. This is the only way to locally store an element and then to return his reference.

4.1.2 add

It takes as 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 that takes a reference to an element T as input, 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 the _tail member variable. If the element is found I delete the created node before throwing the exception, in order to avoid memory leaks.

4.1.3 remove

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

4.1.4 constructor from two const_iterator

It is the last member method I have implemented. Since MySet is a template class, I want this method to take any type of iterators as input and, to do this, I used a template type Q for the iterators. In this way it is possible to pass any type of iterator you want. Using the iterators, it is possible to create a new MySet object from them. I left the type-check task to the compiler, using *static_cast*. As all the constructors I have implemented, all their code are in a *try/catch* clause so that, if something goes wrong, *removeAll* method is called in the *catch* clause allowing to keep the object in a coherent state or, more precisely, it deletes all nodes created by it in the heap. In all the constructors, after the *removeAll* method is called, I broadcast the exception with the *throw* statement to warn who has called the method that something 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 to the *std::ostream* taken as input, using the previously defined *operator**. 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 the MySet passed as parameter, and I used it to iterate through its elements. I create an helper local variable of the same type of the MySet passed as input and I fill it using the *add* method only if the current element does not satisfy the predicate functor. After having iterated among all the elements, I return 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. However, this required an explicit *delete* by the user to avoid memory leaks at the end of the program and, for this reason, I decided to return it by value, avoiding explicit *delete*.

4.2.2 operator+

As *add* method works properly, I am sure that, at the beginning, both the two MySet objects have not duplicate elements, so I can start the concatenation from the first one without checking whether they have duplicate elements. If the method tries to add an element that already exists, the *add* method will throw an exception, as specified in the *add* description. Finally, I return the resulting MySet by copy, for the same reason that I have explained in the *filter_out* description.

4.3 Helper methods

All the following helper methods are private.

4.3.1 findValue

To help *add* method check whether an element exists or not, I implemented *findValue* that takes as input a constant reference to an element and iterates among all the nodes of the collection. It return true only if it finds this element inside the collection, using the _equal attribute, and otherwise false. I implemented this method to reduce the complexity of the *add* method. At the beginning, I used this helper method also in the *remove* method, for the same reason I used it for *add* method. But I realized that it is not necessary iterate among all the nodes of the collection, but until the element is found, and this could be happen immediately. I decided to keep this method private because it is not required from the 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*, which allows to remove all the elements from the collection. It is used by all constructors of the MySet class. This method iterates among all the nodes and delete them, one by one. After it has finished, the _size, the _head and the _tail are set to 0 and all the elements are removed from the collection and from the heap. I decided to keep this method private because I do not want to allow to delete all the elements with one method, but if someone wants to do this, he can use a loop using the _size attribute.

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 through the elements of the collection and I chose the *forward_iterator* type because I have used a linked list data structure. Moreover, since MySet is not an ordered collection, I think that 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 the MySet class. At the beginning, it was composed only by a simple *main* method, in order to compile *myset.hpp*. After I have finished the MySet class, I have filled it with a series of tests written first using *int*, then *std::string* and then using Person struct, created with the goal of testing MySet with a custom type. To check the correctness of the methods that I have implemented, I used assertions.

5.1 Person struct

For testing the MySet class with custom types, I chose to implement a simple Person struct that has two attributes of *std::string* type and one of *int* type. In addition to the fundamental methods, it also has the *operator*() method to check whether two persons are equal. They are equal if they have the same name, the same surname and the same age.

5.2 Methods and tests

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

5.3 Typedef

For avoiding defining every time different 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 is a list of the functors I created for the tests of the MySet class. All the functors I have defined implement *operator*(), which is a boolean method.

- equal_int, used to check whether two *int* are equal;
- equal_string, used to check whether two *std::string* are equal;
- equal_person, used to check whether two *Person* are equal;
- is_odd, used to test the method *filter_out* with *int* type, which returns true only if the number passed as parameter is odd.
- even_length, used to test the method *filter_out* with *std::string* type, which 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, which returns true only if the age of the person passed is greater than 25.

Since the compiler suggested me not to use == with *std::string*, I used the *compare* method to check whether two *std::string* are equals or not. For the same reason, during the tests, I preferred using 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.