Project exam report

Simone Paolo Mottadelli 820786

s.mottadelli2@campus.unimib.it

Contents

1 Introduction

The project requires to design and implement a dynamic set of generic elements of type **T**, where no duplicated elements are allowed.

2 Implementation of the set

In order to implement the aforementioned data structure, I've decided to use a singly linked list template, where each element of the list is identified by its value, because it is a simple way to implement a set in which it is not required to keep elements in an orderly manner.

More specifically, the dynamic set that I've implemented has two template parameters:

- **T:** it is the type of the elements stored in the set.
- **Eql:** it is the functor used to determine whether two elements are equal or not.

Users of this container class can't insert an element of type **T** in the set, if it already belongs to it. This event happens if and only if the element to be inserted is equal to another element of the set, based on how the functor **Eql** has been defined.
A custom exception *duplicated_element_exception*, which extends *std::runtime_error*, has been used to represent the event of a duplicated element in the set.

As a generic set is composed by elements, inside the *set class* there is an inner class called *element*, where an element has a value of type **T** and a pointer to the next element of the set.
The first element to be inserted in the set is called the *head*, because it is the first link in the chain from which each element in the set can be reached, dereferencing their pointers. As it is very important to know which element is the *head*, there is an instance attribute inside the class, *_head*, which is nothing but a pointer of type *element*.
The *set class* also has an attribute storing the size, or cardinality, of the set.

3 Implementation of instance methods

The only main method I'd spend a word on is the destructor, which relies on a helper function *clear*, that deletes and removes from the memory all the elements of the set recursively.
But, apart from the four main methods (default constructor, copy constructor, assignment operator and destructor), the first method that has been implemented is the *add* method, which is used to add an element to the set, increasing the cardinality of the set accordingly. If the set doesn't have any elements yet, then the element to be inserted becomes the *_head* of the set, otherwise a helper function is called to complete the insertion. The helper function is defined recursively and it appends that element at the end of the set, because it has to compare the value to be inserted in input with all the values of the other elements that are already in the set. Of course, if an element already exists in the set, a *duplicated_element_exception* is thrown and this happens when the functor **Eql** returns true.

The second method that has been implemented is the *remove* function, which obviously removes an element to the set, decreasing the cardinality of the set accordingly. If the element doesn't exist, another custom exception, *non_existent_element_exception*, which also extends *std::runtime_error* and represents the event of a non existent element in the set, is thrown. This function relies on a helper function too; in fact, if the element to be removed is the *_head*, then the second element of the set becomes the *_head*, otherwise the helper function, which is defined recursively, scans all the elements of the set and if it finds one of them with the same value of the value in input to the function, using the functor **Eql**, it removes that element. In that case, the previous element of the one to be removed points to the next element of the one that is being removed.

The size, or cardinality, of a set can be obtained using the *size* method. This function wasn't a functional requirement of the project, but I think it is useful to be able to get this information. I used it a lot during the testing phase.

The fourth method is the redefinition of the *operator[ ]*, used to access an element of the set. This function receives in input an index of type *unsigned int* to an element, because I've imagined that indexes must be positive. Note

that the *_head* is at position 0, while the last element of the set is at position $size - 1$. This function uses an assertions mechanism to avoid indexes out of bound: the value of the index must be less than the cardinality of the set. I've preferred to use the assertions mechanism over exceptions because, as it is a logical error made by the user of this container class, the execution can't continue. This function relies on a helper function, also defined recursively, which scans the elements of the set, decreasing by one unit the index in input for each element scanned, until the index becomes 0 and, in that case, the value of the element is returned.

To facilitate access to the elements of the data structure, the set provides *forward const iterators*. I've decided to use this kind of iterator because the set is not ordered, so there is no need to access to elements randomly. The method *begin* returns an iterator at the beginning of the data sequence, while the method *end* returns an iterator at the end of the data sequence.

A secondary constructor has been implemented to give the user the possibility to create a new set starting from the data sequence defined by the pair of iterators of type **Q** received in input. This function creates a new set; scans all the elements in the data sequence and adds them to the set newly created. Note that there could be exceptions thrown (e.g. *duplicated_element_exception* if there is any attempt of adding an element to the set twice). The compiler automatically resolves compatibility issues between types, thanks to a static cast.

4 Implementation of global methods

The first global method implemented is the redefinition of the *operator<<*, which sends all the elements of a set to an output stream passed as reference in input. Note that it is a template method with parameters **T** and **Eql**, as the set is template.

The second one is the *filter_out* method, which has three template parameters: **T**, **Eql** and **Pred**.
This function receives a reference to a set of elements of **T** type and with equality functor **Eql** in input and creates a new set with all the elements of the set, except the ones that do NOT satisfy the predicate **Pred**. So if an element doesn't satisfy the predicate, the function *add* is called.
I've decided to return the set newly created by copy, so that the users don't have to remember to delete it at the end of their program.

The last global function implemented is the redefinition of the *operator+* function. This function has two template parameters, **T** and **Eql**, as it receives two references to two sets in input. This function creates a new set containing the elements of both sets, using the instance method *add* repeatedly. The newly created set is returned by copy for the same implementation choices mentioned above.
Note that, just for consistency, if an element appears both in the first set and the second set, then a *duplicated_element_exception* is thrown.

5 Testing

The *set class* has been tested with *integers*, *std::strings* and a custom *student class*, with *name* and *age* as fields.
For each type, instance methods have been tested in a unique test function, while global functions have been tested separately.
Assertions mechanism has been used to automatize the tests.
Of course, three functors for **Eql** have had to be created using *structures*:

- **equal_int:** it is a functor with the redefinition of *operator()*, that takes two *integers* in input and returns true if and only if they have the same value.

- **equal_string:** it is a functor with the redefinition of *operator()*, that takes two *std::string* in input and returns true if and only if they have the same characters.

- **equal_student:** it is a functor with the redefinition of *operator()*, that takes two *student* in input and returns true if and only if they have the same *name* and *age*.

and some predicates for **Pred** using *structures* too:

- **hasnt_six_characters:** it is a predicate with the redefinition of *operator()*, which takes a *std::string* in input and returns true if and only if hasn't got six characters.

- **over_18:** it is a predicate with the redefinition of *operator()*, which takes a *student* in input and returns true if and only if it is eighteen years old or older.

- **is_even:** it is a predicate with the redefinition of *operator()*, which takes an *integer* in input and returns true if and only if it is even.

- **is_odd:** it is a predicate with the redefinition of *operator()*, which takes an *integer* in input and returns true if and only if it is odd.