Linköping University
Department of Science and Technology/ITN
Aida Nordman

TND004: Data structures
Lab 2

Goals

- To implement a dynamic data structure, doubly-linked list.
- To use pointers.
- To use big-Oh notation to estimate the running time of functions.
- To use C++ classes, overloaded operators, and move semantics.

Prologue

In the TNG033 course, there was a lab about implementing a singly-linked list to represent sets of integers. In this lab exercise, you go further and implement a class `Set` using instead **sorted doubly-linked lists**. All class `Set` member functions must have linear time complexity, in the worst-case. In addition, you will need to add **move semantics** to the class. By equipping the class `Set` with a move constructor and a move assignment operator, a performance boost can be provided to client code, as discussed in seminar 1 of the course.

Note that move semantics was not considered in previous C++ courses. Thus, you have now the opportunity to get acquainted with an important modern feature of C++.

On week 14, the move semantics will be introduced in the booked seminar. Move semantics is also nicely discussed in section 13.6 of the book "C++ Primer", 5th edition.

Section 3.5 of the course book presents a possible implementation for doubly-linked lists. Although in this lab you also need to implement doubly-linked lists, there are some important differences between the implementation required in this exercise and the book's implementation, as summarized below. Most of the differences are motivated by the fact that we are going to use doubly-linked lists to implement the concept of (mathematical) set. Nevertheless, studying the implementation presented in section 3.5 of the course book is recommended.

- The book presents an implementation of doubly-linked lists, while in this lab you are requested to implement the concept of set by using a doubly-linked list (though, sets can be implemented in other ways).
- In this lab exercise, the list's **nodes are placed in sorted order** and there are no repeated values stored in the list. On the contrary, the course book's implementation allows repeated values in lists and lists do not need to be sorted.
- The book uses a template class to implement a generic class `List` of objects. Template classes are not used in this lab.
- Iterator classes are not considered in this exercise, though they are implemented in the course book.

A similarity between the class `Set` and the course book's implementation of class `List` (see section 3.5) is that both support the move semantics.

This lab consists of three exercises.

- **Exercise1**: to implement all member functions of the class `Set` given in the file `set.h`. Some of these functions must be implemented before the **HA** lab session on week 15. File `set.h` indicates clearly which member functions must be implemend before the HA lab session on week 15.
- **Exercise 2**: to extend class `Set` to support move semantics.
- **Exercise 3**: to analyze the time complexity of some of the `Set` operations.

Preparation

You can find below a list of tasks that you need to do before the **HA** lab session on week 15.

1. Review lectures 2 to 4. Big-Oh notation was introduced in lecture 2 and lecture 3, while doubly linked lists where discussed in lecture 4.
2. Review the notes of seminar 1 about move semantics.
3. Download the zipped folder with the files needed for this lab and create a project with the files `set.h`, `set.cpp`, and `main.cpp`. It is then possible to compile and create an executable, though not all functions are fully implemented.
4. Study the class `Set` interface given in the file `set.h` and read also the description of class `Set` given below.
5. The member functions to be implemented before the **HA** lab session on week 15 are explicitly marked in the file `set.h`. Implement those functions. Note that all `Set` member functions must have a $O(n)$ time complexity, for a set with $n > 0$ elements. You can test your code with the program given in `main.cpp`. Your code should pass all test phases 1 to 5. The expected output is available in the file `lab2_out.txt`.
6. Test your code with e.g. DrMemory and make sure there are no memory leaks.
7. Review exercise 5 of lesson 1, in the TNG033 course[1]. This exercise of TNG033 course uses the same algorithm you'll need to implement union of two sets (i.e. member function `Set::operator+=`). Later on in the course, we will use this algorithm again.

Other member functions can be added to class `Set`, besides the given ones. In this case, the extra added functions should not belong to the public interface of the class (i.e. they should be private member functions).

In the beginning of the **HA** lab session on week 15, the lab assistant will give feedback on your code for class `Set`. Note that not preparing for the lab implies that the lab assistant has the right to down prioritize your questions during the lab session.

Exercise 1: class `Set`

In this exercise, you need to implement the class `Set` that represents sets of integers (`int`). **Sorted doubly-linked list** is the data structure used to implement sets in this lab. Notice that sets do not have repeated elements.

[1] Login is `TNG033` and password is `TNG033ht13`.

Every node of the list stores an integer value. To make it easier to remove and insert an element from the list, the list's implementation uses "dummy" nodes at the head and tail of the list, as discussed in lecture 4 and in the course book. Thus, an empty doubly-linked list consists of two nodes pointing at each other (see figures in slide 3 of lecture 4)

The class `Set` provides the usual set operations like union, difference, subset test, and so on. The class definition is given in the file `set.h`. Extra information about the meaning of the overloaded operators is given below.

- Overloaded `operator+=` such that `R+= S;` should be equivalent to $R = R \cup S$, i.e. the union of R and S is assigned to set R. Recall that the union $R \cup S$ is the set of elements in set R or in set S (without repeated elements). For instance, if $R = \{1, 3, 4\}$ and $S = \{1, 2, 4\}$ then $R \cup S = \{1, 2, 3, 4\}$.

  Union of sets is conceptually similar to the problem of merging two sorted sequences. An algorithm to merge sorted sequences efficiently was discussed in the TNG033 course[2] (see solution for lesson 1, exercise 5).

- Overloaded `operator*=` such that `R*= S;` should be equivalent to $R = R \cap S$, i.e. the intersection of R and S is assigned to set R. Recall that the intersection $R \cap S$ is the set of elements in **both** R and S. For instance, if $R = \{1, 3, 4\}$ and $S = \{1, 2, 4\}$ then $R \cap S = \{1, 4\}$.

- Overloaded `operator-=` such that `R-= S;` should be equivalent to $R = R - S$, i.e. the set difference of R and S is assigned to set R. Recall that the set difference $R - S$ is the set of elements that belong to R but do not belong to S. For instance, if $R = \{1, 3, 4\}$ and $S = \{1, 2, 4\}$ then $R - S = \{3\}$[3].

- Overloaded `operator<=` such that `R <= S` returns true if R is a subset of S. Otherwise, false is returned. Set R is a subset of S if and only if every member of R is a member of S. For instance, if $R = \{1, 8\}$ and $S = \{1, 2, 8, 10\}$ then R is a subset of S (i.e. `R <= S` is true), while S is not a subset of R, (i.e. `R <= S` is false).

- Overloaded `operator<` such that `R < S` returns true, if R is a proper subset of S. A set R is a proper subset of a set S if R is strictly contained in S and so necessarily excludes at least one member of S. For instance, `S < S` always evaluates to false, for any set S. **Hint:** use `operator<=` to implement this function.

- Overloaded `operator==` such that `R == S` returns true, if R is a subset of S and S is a subset of R (i.e. both sets have the same elements). Otherwise, false is returned. **Hint**: use `operator<=` to implement this function.

- Overloaded `operator!=` such that `R != S` returns true, if R (S) has an element that does not belong to set S (R), i.e. R and S have different elements. **Hint:** use `operator<=` to implement this function.

Note that all `Set` member functions should have a linear time complexity. The member functions to be implemented before the **HA** lab session on week 15 are explicitly marked in the file `set.h`.

[2] Login is `TNG033` and password is `TNG033ht13`.

[3] The set difference $R - S$ is also known in the literature as the "*relative complement of S in R*".

The file `main.cpp` contains a test program for class `Set` and it can be downloaded from the course website. Feel free to add other tests to this file, though the original file must be used when presenting the lab.

Other member functions can be added to class `Set`, besides the ones described above. In this case, the extra added functions should not belong to the public interface of the class.

Exercise 2: move semantics

Exercise **2** in this lab consists in adding move semantics to the class `Set`. Thus, it should be possible to "steal" the list of nodes hold by a R-value `Set` (e.g. a temporary `Set` variable) during execution of a copy-assignment or a copy constructor from the R-value.

Once you have added move semantics to class `Set` do the following. Consider `Test Phase 10`, in the file `main.cpp` and the line of code below. Then, use the debugger to trace the execution of the code and see which functions are called when this line of code is executed.

```
cout << "std::move(*ptr_S) * S5 = " << std::move(*ptr_S) * S5 << endl;
```

In `Test Phase 10`, if you replace the line above by the following line of code, which functions are called? Use again the debugger to trace the execution of the code.

```
out << "S5 * std::move(*ptr_S) = " << S5 * std::move(*ptr_S) << endl;
```

Exercise 3: time complexity analysis

Exercise 3 consists in analysing the time complexity for each of the following statements. Use Big-Oh notation and **motivate clearly** your answers. Assume that `S1` and `S2` are two `Set` variables and that `k` is an `int` variable.

- `S1 = S2;`
- `S1 * S2`
- `std::move(S1) * S2`
- `k + S1`

Deliver your answers in paper, preferably computer typed, when you present the lab on week 16. Do not forget to indicate the name plus LiU login of each group member.

Presenting solutions and deadline

You must demonstrate your solution orally during your **RE** lab session on **week 16**. Note that this lab has a strict deadline. Failing to present the lab on week 16 implies that you cannot be awarded 3p for the labs in the end of the course.

Necessary requirements for approving your lab are given below.

- Use of global variables is not allowed, but global constants are accepted.
- Readable and well-indented code. Note that complicated functions and over-repeated code make code quite unreadable and prone to bugs. Always check your code and perform code refactoring whenever possible.
- There are no memory leaks. We strongly suggest that you use one of the tools listed in the appendix to check whether your program has memory leaks.

- The code generates no compilation warnings.
- All `Set` member functions must be executed in linear time, in the worst-case.
- STL containers cannot be used in this exercise.
- Bring a written answer to exercise 3, with indication of the name plus LiU login of each group member. You'll need to discuss your answers with the lab assistant who in turn will give you feedback. Hand written answers that we cannot understand are simply ignored.

If you have any specific question about the exercises, then send us an e-mail. Be short and concrete, otherwise you won't get a quick answer. You can write your e-mail in Swedish. Add the course code to the e-mail's subject, i.e. "`TND004:` …".

Finally, presenting your solution to the lab exercises can only be done in the **RE** session of week 16.

Appendix

Checking for memory leaks

Memory leaks are a serious problem threat in programs that allocate memory dynamically. Moreover, it is often difficult to discover whether a program is leaking memory.

Specific memory monitoring software tools can help the programmers to find out if their programs leak memory. Most of these tools are commercial, but there are a few which you can install and try for free.

- Dr.Memory available for Windows, Linus, and Mac. My experience of using Dr.Memory is that it is easy to install[4] and easy to use. Just make sure that you have added the path to the directory where Dr. Memory is installed to the **PATH** environment variable.
- Valgrind only available for Linux. I have heard good reviews about it.
- Visual leak detector for Visual C++ (guess … for Visual Studio!)[5]. Dr.Memory detects a larger range of memory-related problems when compared to “visual leak detector”.

Lycka till !

[4] Dr. Memory is installed in most of the computers of the lab rooms.

[5] Visual leak detector is not installed in the lab rooms. But, you can easily install it in a Windows 10 machine.