Linköping university
ITN

TNG033: Programming in C++

Lab 1

Course goals

- To write programs using pointers and dynamic memory allocation/deallocation.
- To implement a dynamic data structure: singly linked list class.
- To create classes with overloaded operators and use deep copying of objects.

Make more clear that the lists are sorted and that the constructor that build a list from an array should build a sorted list. This should be tested by doctest.

Preparation

You must perform the tasks listed below before the start of the lab session on week 46. In each **HA** lab session, your lab assistant has the possibility to discuss about three questions with each group. Thus, it is important that you read this lab description and do the indicated preparation steps so that you can make the best use of your time during the lab session.

To solve the exercises in this lab, you need to understand concepts such as pointers, dynamic memory allocation and deallocation, and how singly linked lists can be implemented. These topics were introduced in lectures 1 to 3, though for simplicity classes were not yet used in these lectures.

Classes, constructors, destructors, `const` member functions, deep copying of objects, and friend functions are also used in the exercises of this lab. These concepts were discussed in lectures 4 to 6.

- Download the files for this exercise from the course website and create a project with the files `set.hpp`, `doctest.hpp`, `set.cpp`, and `main.cpp`. You should be able to compile, link, and execute the program. The file `main.cpp` contains specific tests for the `Set` member functions, called unit tests. Since `set.cpp` contains only stubs[1], the unit tests fail.
- Read the information given in the appendix.
- The lists in this lab are implemented with a **dummy node** and the **lists' nodes are sorted** increasingly. Sorting the nodes has the advantage of speeding up unsuccessful searches.
- Exercise 3, of self-study exercises set 1, is about merging two sorted sequences and you should follow a similar **algorithm** for implementing union of two sets (`operator+`). Simple modifications in the algorithm can also help you in the implementation of sets difference operation (`operator-`).

[1] A stub contains just enough code to allow the program to be compiled and linked. Thus, must often a stub is a (member) function with dummy code.

- Read the exercise and implement all functions explicitly marked in the list of member functions for class `Set`, before your **HA** session on week 46. Your code should successfully pass the tests in "`Phase 1`" and "`Phase 2`".
- Note down the main questions that you want to discuss with your lab assistant.

Presenting solutions and deadline

The exercises are compulsory and you should demonstrate your solution orally during your **RE** lab session on **week 47** After week 47, if your solution for lab 1 has not been approved then it is considered a late lab. Note that a late lab can be presented provided there is time in a **RE** lab session. Note that a fixed time slot is assigned to you group for presenting the lab. You can find your time slot here.

Three necessary requirements for approving your lab are given below.

- Use of global variables is **not** allowed, but global constants are accepted.
- The code must be readable, well-indented, and use good programming practices.
- Your solution must not have pieces of code that are near duplicates of each other. Duplicated code is considered a bad programming practice, since it decreases the code readability and increases maintenance/debugging time. A possible way to address this problem is to create private member functions and move the duplicate code into these functions. Another way to tackle the problem can be to implement a public member function by calling other public member functions.
- Your code must pass all unit tests in `main.cpp`.
- The compiler must not issue warnings when compiling your code. In Visual Studio, you should set the compiler's warning level to **level4** (`\W4`).
- There are no memory leaks. We strongly suggest that you use one of the tools suggested in the appendix to check whether your code generates memory leaks, or other memory-related programming errors, before "redovisning".

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 English or Swedish. Add the course code to the e-mail's subject, i.e. "`TNG033: …`".

Exercise

Implement a class, named `Set`, representing a set of integers. A set is internally implemented by a **sorted singly linked list**. Every node of the list stores an element in the set (an integer) and a pointer to the next node. To simplify the implementation of the operations that insert (remove) an element in (from) the list, the first node of each set is a dummy node. Thus, an empty set consists just of one dummy node.

Note that sets must not have repeated elements (according to the mathematical definition of set).

A brief description of the files distribute with this lab is given below.

- The header file `set.hpp` contains the interface of class `Set`, i.e. the class definition. The public interface of the class cannot be modified, i.e. you cannot add new public member functions neither change the ones already given. Private (auxiliary) member functions can be added.

  The class `Set` definition contains a private class `Node`. Class `Node` defines a list's node that stores an integer and a pointer to the next node. Note that all `Node` members are public within class `Set` and, therefore, can be accessed from `Set` member functions.

- You should add the implementation of each member function of class `Set` to the source file `set.cpp`.

- The source file `main.cpp` contains unit tests to test all member functions of class `Set`. The tests are organized in five test suits, named "`Phase 1: …`" to "`Phase 5: …`". Though you can add extra tests to the file `main.cpp`, you must use the original file distributed with the lab, when presenting your solution.

  Starting from "`Phase 1`", you should implement those member functions needed by each test phase, then compile, link and run the program. If it passes the tests for the current test phase then you can proceed to the next phase. The file `main.cpp` clearly indicates which member functions are required for each test phase.

- Note that your code should pass the tests in "`Phase 1`" and "`Phase 2`", before the **HA** lab session on week 46.

- Do not spread the use of `new` and `delete` all over the code. Instead, define two private member functions.

  ```
  // Insert a new node after node pointed by p
  // the new node should store value
  void insert(Node* p, int value) const;

  // Remove the node pointed by p
  void remove(Node* p);
  ```

  Then, `Set` member functions should call the functions above, whenever it's needed to allocate memory for a new node or deallocate a node's memory.

Description of class `Set` member functions

A brief description of the `Set` member functions is given below. It's also indicated in which test suits the functions are tested.

Test: **Phase 1**

- `Set ();` -- implement before **HA** session
  Constructor for an empty set,
  e.g. `Set S{};`
- `Set (int v);` -- implement before **HA** session
  Constructor for the singleton $\{v\}$,
  e.g. `Set S{5};`
- `Set (const Set& S);` -- implement before **HA** session
  Copy constructor initializing `R` with set `S`,
  e.g. `Set R{S};`
- `Set (int a[], int n);` -- implement before **HA** session
  Constructor creating a set `S` from `n` integers in a **non-sorted** array `a`,
  e.g. `Set S{a, 10};`
- `~Set();` -- implement before **HA** session
  Destructor deallocating the nodes in the list, including the dummy node.
- `bool member (int x) const;` -- implement before **HA** session
  `S.member(x)` returns `true` if the element `x` is in the set `S`.
  Otherwise, `false` is returned.

  `int cardinality() const;` -- implement before **HA** session
  `S.cardinality()` returns the number of elements in the set `S`.
- `bool empty() const;` -- implement before **HA** session
  `S.empty()` returns `true` if the set `S` is empty.
  Otherwise, `false` is returned.

Test: **Phase 2**

- `Set& operator=(Set S);` -- implement before **HA** session
  Overloaded assignment operator,
  e.g. `R = S;`

- `std::ostream& operator<<(std::ostream &os, const Set& S);`
  -- implementation is already given in the definition's file `set.cpp`
  Stream insertion operator `operator<<` outputs all the elements in set `S` to the output stream `os`,
  e.g. `cout << S;`

Test: **Phases 3** and **5**

- `Set operator+(const Set& S) const;`
  `R+S` returns a new set with the set **union** of `R` and `S`. The union is the set of elements in set `R` or in set `S` (without repeated elements). For instance, if $R = \{1, 3, 4\}$ and $S = \{1, 2, 4, 9, 11\}$ then $R + S = \{1, 2, 3, 4, 9, 11\}$.
- `Set operator*(const Set& S) const;`
  `R*S` returns a new set with the **intersection** of `R` and `S`. The intersection is the set of elements in **both** `R` and `S`. For instance, if $R = \{1, 3, 4\}$ and $S = \{1, 2, 4\}$ then $R * S = \{1, 4\}$.
- `Set operator-(const Set& S) const;`
  `R-S` returns a new set with the **difference** of `R` and `S`. The difference 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 $R - S = \{3\}$.
- `Set operator+(int x) const;`
  `R + x` returns a new set with the **union** of `R` and set **`{x}`.**
- `Set operator-(int x) const;`
  `R - x` returns a new set with the **difference** of `R` and set `{x}`.

Test: **Phase 4**

- `bool operator<=(const Set& S) const;`
  `S <= R` returns `true` if `R` is a **subset** of `S`. Otherwise, `false` is returned. `S` is a subset of `R` if and only if every member of `S` is a member of `R`. For instance, if $S = \{1, 8\}$ and $R = \{1, 2, 8, 10\}$ then R is a subset of S (i.e. `S <= R` is `true`), while R is not a subset of S, (i.e. `S <= R` is `false`).
- `bool operator<(const Set& S) const;`
  `S < R` returns `true` if `S` is a **proper subset** of `R`. Otherwise, `false` is returned. A proper subset `S` of a set `R` is a subset that is strictly contained in `R` and so necessarily excludes at least one member of **`R`**. For instance, `S < S` always evaluates to `false`, for any set `S`.
- `bool operator==(const Set& S) const;`
  `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.
- `bool operator!=(const Set& S) const;`
  `R != S` returns `true` if `R == S` is `false` (i.e. the sets differ at least in one element). Otherwise, `false` is returned.

Note that the tests in "`Phase 3`", "`Phase 4`", and "`Phase 5`" are commented in the given file `main.cpp`. Remember to uncomment them when you have coded the corresponding functions.

Lycka till!!

Appendix

Sets

You can find information about mathematical sets and their operations here.

Advised compiler settings: Visual Studio

Compilers can issue **errors** and **warnings** about your code.

An error implies that your program is not written according the rules of the C++ language (in other words, your program is not a C++ program). Consequently, the program can neither be translated to binary code nor be executed.

Compilers will always warn you about things that might be difficult to find during testing. For example, your compiler can warn you that you are using an uninitialized variable.

A program can be executed, even if the compiler issues warnings about the code. However, it is very important that you do not ignore the compiler warnings. Although the code may compile and run, compiler warnings are usually a strong indication of a serious problem in the code that may affect badly the program execution at any time.

You can find here how to set the compiler warning's level in Visual Studio.

Debugger

A debugger is a very useful tool that most of the IDEs, like Visual Studio, have to help programmers to find bugs in the code.

Debuggers can execute the program step-by-step, stop (pause) at a particular instruction indicated by the programmer by means of a breakpoint, and trace the values of variables.

An introduction to the use of the debugger in Visual Studio can be found here.

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 or have other memory-related programming errors. You can find below some tools 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[2] and easy to use. Just make sure that you have added the path to the folder where Dr. Memory is installed to the `PATH` environment variable. This tool is installed in the lab rooms.

[2] I have tested it on a Windows machine.

To make things easier create a shortcut to Dr.Memory on your desktop. Then, all you need to do is to drag and drop the executable of your program (`.exe` file) on the Dr.Memory's shortcut.

- Valgrind only available for Linux.
- Visual leak detector for Visual C++ (only for Visual Studio!).

Unit testing

Unit testing is a software testing method by which individual units of source code, e.g. member functions of a class, are tested to determine whether they are fit for use.

There are several frameworks that can be used for unit testing in C++, e.g. *Google test*, *CppUnit*, *doctest*. In this course, we have chosen *doctest* because it's a light framework consisting of a single header file.

In a simplified way, unit tests consist of a collection of test cases. A **test case** is a specification of a testing procedure, and expected results, that define a single test to be executed to achieve a particular software testing objective, such as to exercise a particular path of execution or to verify compliance of a function with a specific requirement. An example of a test case, used in this lab, for testing the single element constructor is given below. The tests are written in the `main` function (file `main.cpp`). Note that test cases have names, like "`Single Element Constructor`", and can have sub-cases.

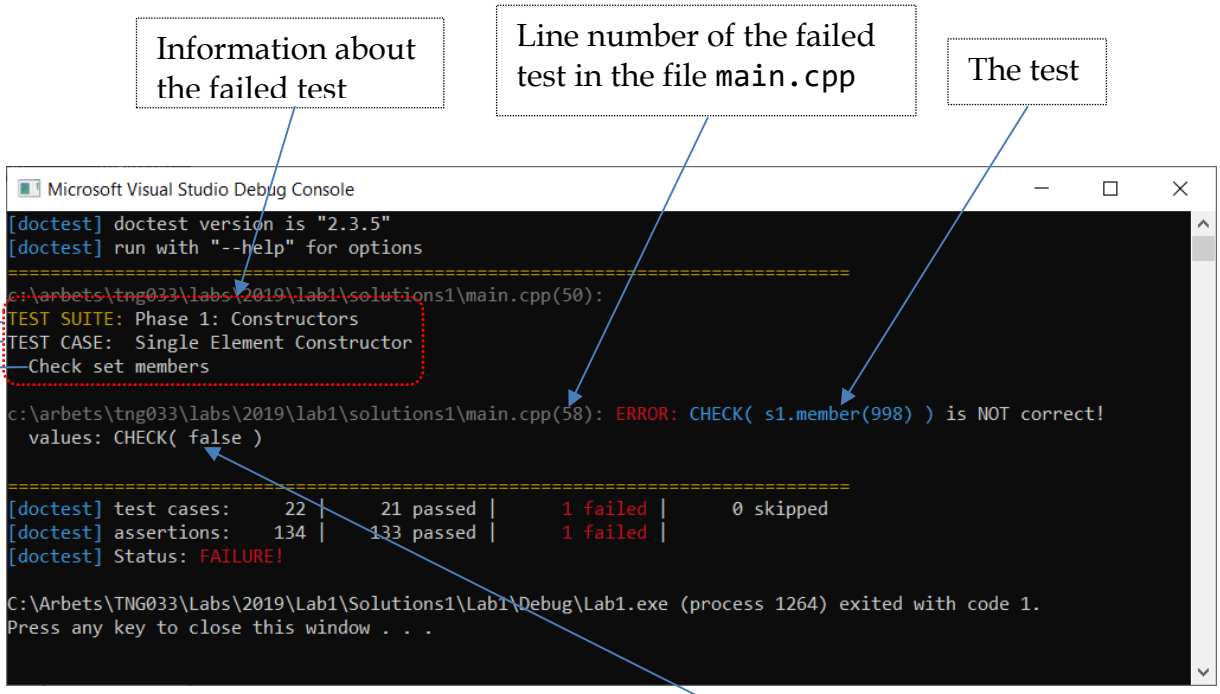
```
TEST_CASE("Single Element Constructor") {
        const Set s1{999};

        SUBCASE("Check set cardinality") {
            CHECK(s1.cardinality() == 1);
        }

        SUBCASE("Check set members") {
            CHECK(s1.member(999));
            CHECK_FALSE(s1.member(998));
        }
    }
```

Check if condition `s1.cardinality() == 1` is `true`

Check if condition `s1.member(999)` is `true`

Check if condition `s1.member(998)` is `false`

For instance, if `CHECK(s1.member(999));` fails then the failed test, with file name and line number is shown in the console window.



Test cases are often grouped into **test suits**. For instance, all test cases to test the constructors of class `Set` can be grouped under a test suit, named "`Phase 1: Constructors`". An excerpt of this test suit is shown below (see also `main.cpp`)

Test `s1.member(998)` evaluated to `false`

```
TEST_SUITE("Phase 1: Constructors") {

    TEST_CASE("Default Constructor") {
        const Set empty{};

        SUBCASE("Test whether empty set") {
            CHECK(empty.empty());
        }

        SUBCASE("Check set members") {
            CHECK_FALSE(empty.member(0));
        }
    }

    TEST_CASE("Single Element Constructor") {
        const Set s1{999};

        SUBCASE("Check set cardinality") {
            CHECK(s1.cardinality() == 1);
        }

        SUBCASE("Check set members") {
            CHECK(s1.member(998));
            CHECK_FALSE(s1.member(998));
        }
    }
    …
}
```

In Visual Studio, it's possible to use the debugger when a test fails. To this end do the following.

1. Set a break point on the (first) test that fails.
2. `Step Into` (`F11`). Yes, the debugger lands on the *doctest* code!
3. `Step Out` (`shift+F11`) to get out of `doctest.hpp` file.
4. `Step Into` (`F11`) again. This time the debugger should land on the `Set` member function called in the failed test, e.g. `Set::member(int)`.

It's also possible to set breakpoints in any other C++ instructions used in the tests and use `Step Into` (`F11`), `Step Over` (`F10`), etc.