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.

Preparation

You must perform the tasks listed below before the start of the lab session *Lab1 HA*. 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 the lab descriptions and do the indicated preparation steps so that you can make the best use of your time during the lab sessions.

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. Then, you can use use CMake to create a project with the files `set.hpp`, `set.cpp`, and `main.cpp`. For how to use CMake, you can see this short guide. Make sure you have installed CMake.
- You should be able to compile, link, and execute the program. The file `main.cpp` contains specific tests for the `Set` member functions. Since `set.cpp` contains only stubs[1], the tests fail.
- Assertions are used to test your code. See the appendix, for more information about how assertions can be used to test code.
- Read the exercise.
- 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-`) and set intersection (`operator*`).

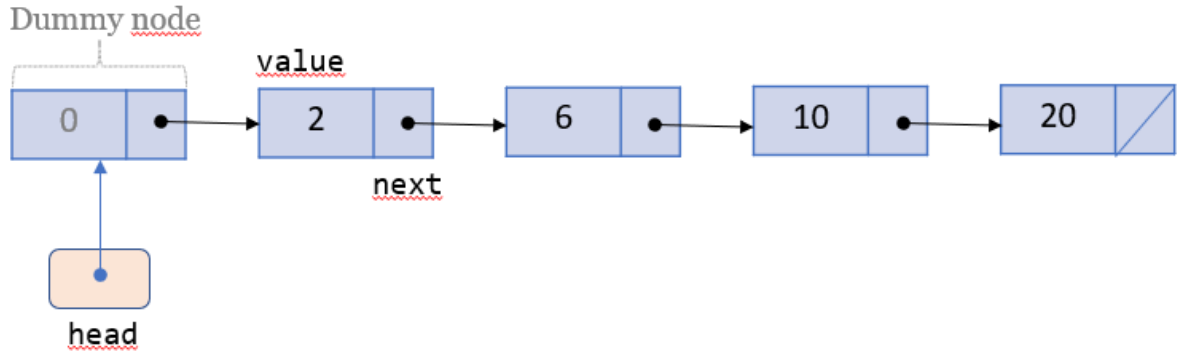
[1] A stub contains just enough code to allow the program to be compiled and linked. Thus, must often a stub is a (member) function with dummy code.

- Implement all functions explicitly marked in the list of member functions for class `Set`, before your lab session *Lab1 HA*. Your code should successfully pass the tests from "`Phase 1`" to "`Phase 5`".
- Note down the main questions that you want to discuss with your lab assistant.

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. Sorting the nodes has the advantage of speeding up unsuccessful searches. To simplify the implementation of the operations that insert (remove) an element in (from) the list, the first node of each list 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).

Files description

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.
- You should add the implementation of each member function of class `Set` to the source file `set.cpp`.
- The source file `main.cpp` tests the member functions of class `Set`. The tests are organized into 11 groups, named "`TEST PHASE 1: …`" to "`TEST PHASE 11: …`". You can add extra tests to the file `main.cpp`. However, the tests you add must be clearly indicated in the code. You cannot remove any of the given tests, though.

Starting from "`TEST 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 from "`TEST PHASE 1`" to "`TEST PHASE 5`", before the lab session *Lab1 HA*.

Description of class `Set`

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

A (static) member function named `Set::get_count_nodes()` is given already implemented and it returns the total number of existing nodes. This function is used in the test code to help detecting possible memory leaks through the use of assertions. The counter of the total number of existing nodes in the program (member variable `Set::Node::count_nodes`)[2] is updated by the given `Set::Node` constructor and destructor.

For instance, in the given `main`, you can find assertions such as the one below to test whether the total number of existing nodes is equal to *e.g.* 2.

```
assert(Set::get_count_nodes() == 2);
```

If the test fails then the program stops running and a message is displayed with information about the failed assertion (see appendix, section "*Testing code: assertions*").

A brief description of the `Set` member functions that you need to implement is given below.

TEST PHASE 0 & 5

- `Set();` -- implement before *Lab1 HA*
  Constructor for an empty set,
  e.g. `Set S{};`
- `Set(int v);` -- implement before *Lab1 HA*
  Constructor for the singleton $\{v\}$,
  e.g. `Set S{5};`
- `~Set();` -- implement before *Lab1 HA*
  Destructor deallocating the nodes in the list, including the dummy node.
- `size_t cardinality() const;` -- implement before *Lab1 HA*
  `S.cardinality()` returns the number of elements in the set `S`.
- `bool empty() const;` -- implement before *Lab1 HA*
  `S.empty()` returns `true` if the set `S` is empty.
  Otherwise, `false` is returned.

[2] Note that `Set::Node::count_nodes` does not count the number of nodes of any particular list. It's just a counter of the total number of nodes existing in the program, at any execution point.

TEST PHASE 1

- `Set(const std::vector<int>& V);` -- implement before *Lab1 HA*
  Constructor creating a set `S` from `integers` in a **non-sorted** vector `V`,
  e.g. `Set S{V};`

TEST PHASE 2

- `Set(const Set& S);` -- implement before *Lab1 HA*
  Copy constructor initializing `R` with set `S`,
  e.g. `Set R{S};`

TEST PHASE 3

- `Set& operator=(Set S);` -- implement before *Lab1 HA*
  Overloaded assignment operator,
  e.g. `R = S;`

TEST PHASE 4

- `bool member(int x) const;` -- implement before *Lab1 HA*
  `S.member(x)` returns `true` if the element `x` is in the set `S`.
  Otherwise, `false` is returned.

- `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. `std::cout << S;`

TEST PHASE 6

- `bool operator<=(const Set& S) const;`
  `R <= S` returns `true` if `R` is a **subset** of `S`. Otherwise, `false` is returned. `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. `S <= R` is `false`).
- `bool operator<(const Set& S) const;`
  `R < S` returns `true` if `R` is a **proper subset** of `S`. Otherwise, `false` is returned. A proper subset `R` of a set `S` is a subset that is strictly contained in `S` and so necessarily `R` excludes at least one member of **`S`**. 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.

TEST PHASE 7

- `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 $R + S = \{1, 2, 3, 4, 9, 11\}$.

TEST PHASE 8

- `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 $R * S = \{1, 4\}$.

TEST PHASE 9

- `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 $R - S = \{3\}$.

TEST PHASE 10

- `Set operator-(int x) const;`
  `R - x` returns a new set with the **difference** of `R` and set `{x}`.

"`TEST PHASE 11`" in the `main` tests the use of more complex set expressions involving union, intersection, and set difference such as `S1 - 5 * (S1 + S2) - 99999`.

Code requirements

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);

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

Thus, operations `new` and `delete` must only appear in the functions above. `Set` member functions should call the private member functions `insert` and `remove`, whenever it's needed to allocate memory for a new node or deallocate a node's memory.

Remember that you can add extra tests to the file `main.cpp`. However, the tests you add must be clearly indicated in the code. You cannot remove any of the given tests, though.

Presenting solutions and deadline

The exercises in this lab are compulsory and you should demonstrate your solutions during the lab session *Lab1 RE*. Read the instructions given in the labs web page and consult the course schedule. We also remind you that your code for the lab exercises cannot be sent by email to the staff.

Before presenting your code make sure that it is readable, well-indented, and that the compiler issues no warnings.

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 tests in `main.cpp`.
- Your code must satisfy the given code requirements.
- The compiler must not issue warnings when compiling your code. If have used CMake to create the project then the warning level is set, automatically.
    - If you have manually created a project in Visual Studio then you should set the compiler's warning level to **Level4** (`\W4`) ") and make sure the compiler is compiling C++17.
    - If you have manually created a project in Xcode then set the pedantic warning flag of the compiler (go to "*Build Settings*") and make sure the compiler is compiling C++17.
- There are no memory leaks or other memory related bugs. 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*".

The given test program (file `main.cpp`) output is given below.

```
TEST PHASE 0: default constructor and constructor int -> Set

TEST PHASE 1: constructor from a vector

TEST PHASE 2: copy constructor

TEST PHASE 3: operator=

TEST PHASE 4: member

TEST PHASE 5: cardinality and empty

TEST PHASE 6: equality, subset, strict subset

TEST PHASE 7: union

TEST PHASE 8: intersection

TEST PHASE 9: difference

TEST PHASE 10: mixed-mode set difference

TEST PHASE 11: union, intersection, and difference

Success!!
```

Lycka till!!

Appendix

Sets

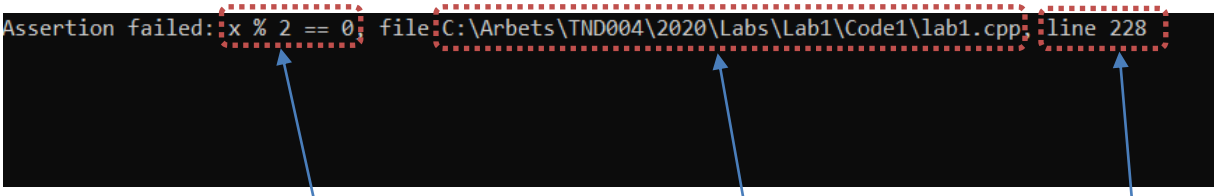
You can find information about mathematical sets and their operations here.

Testing the code: assertions

In C/C++ programming language, assertions can be used to express that a given condition must be true, at a certain point in the code execution. For instance, consider the following code.

```
int main() {
    int x{7};

    /*  Some code in between and let's say x
       is accidentally changed to 9  */
    x = 9;

    // Programmer assumes x is even in rest of the code
    assert(x % 2 == 0);

    /* Rest of the code */

}
```

The expression `assert(x % 2 == 0);` tests, during execution time, whether the condition `x % 2 == 0` evaluates to `true`. If the evaluated condition is not true – an **assertion failure** –, then the program typically crashes and information about the failed assertion is shown. Note that a program stops executing at the first assertion that fails.



Assertion that failed

Source file containing the assertion `x % 2 == 0`

Line number of the failed assertion

Assertions can be useful to identify bugs in a program and they are used in each test phase of the provided `main` function. Finally, to use assertions in C/C++ language, it is needed to include the library `cassert.h`.

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 and other memory bugs

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 suffer from memory corruption bugs such as memory leaks. There is a number of tools which you can install and try for free. Note that no tool will detect all memory bugs in the code (i.e. all tools have limitations). Thus, using several tools and inspecting carefully the code is the best strategy. Some of these tools are listed below.

- DrMemory available for Windows, Linux, and Mac. This tool does not produce reliable memory diagnostics with executables created by Visual Studio compiler. Instead, you can compile your code with Clang compiler and then use DrMemory.
- Address sanitizer Asan and Clang (cannot detect memory leaks on Windows).
- Visual leak detector for Visual C++ is a library, named vld.h, that can be included in C++ programs. It's easy to use and install. The downside is that it only detects memory leaks and it can only be used with programs built in Visual Studio.
- Valgrind only available for Linux.

Clang compiler and DrMemory are also installed in the computers of the lab rooms.