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 using 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 are introduced in lectures 1 to 3, though for simplicity classes were not yet used in these lectures.

Classes, constructors, destructors, `const` member functions and deep copying of objects are also concepts important for the exercises of this lab. These concepts are discussed in lectures 4 to 5.

1. 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 `test.cpp`. For how to use CMake, you can see this short guide.
2. Compile, link, and execute the program (use `ctrl-F5` in Visual Studio). The file `test.cpp` contains specific tests for the class `Set` member functions. Since `set.cpp` contains only stubs[1], the tests fail.
3. Read the exercise description (section "Exercise" with three sub-sections).
4. Read and understand the class `Set` definition given in the file `set.hpp`.
5. Read the appendix, for information about how assertions can be used to test code, memory debuggers, etc. Note that assertions are used in the `main` function to test your code.
6. 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 "`TEST PHASE 0`" to "`TEST PHASE 4`".

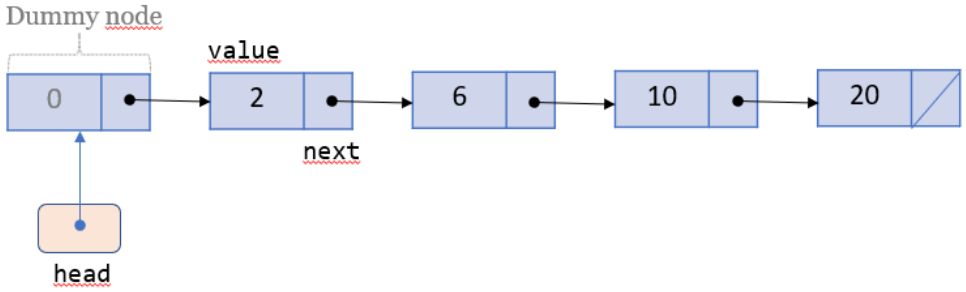
[1] A stub contains just enough code to allow the program to be compiled and linked. Thus, must often a stub is a (member) function with dummy code.

7. Understand the solution proposed for exercise 3 in set 1 of self-study exercises. The exercise is about merging two sorted sequences. A similar **algorithm** can be used for implementing union of two sets (`Set::set_union`) in this lab.

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 sets of integers. In this lab, a set is implemented as 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 list is a **dummy node**, as depicted in the figure below.



Thus, an empty set consists just of one dummy node.

A `Set` object stores a pointer (named `Set::head`) to its dummy node and a counter of the number of elements in the set (named `Set::counter`).

Note that sets cannot 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.

- A file named `set.hpp` contains the interface of class `Set`, i.e. the class definition. This file is called **header file** and header files typically have the extension (or suffix) `.hpp` or `.h`. The file `set.hpp` does not contain function definitions (implementation) and, therefore, it cannot be compiled.
- A file named `set.cpp` is a **source file** where the implementation of the member functions of class `Set` should be placed. As usual, a source code file has the extension `.cpp` and it can be compiled separately from other source files belonging to the program (use `ctrl-F7` in Visual Studio).

  So that the class definition given in the header file is visible to the compiler when compiling the source file `set.cpp`, it is needed to have `#include "set.hpp"` at the start of the file `set.cpp`.
- The source file `test.cpp` contains the `main` function and it tests the member functions of class `Set`. The tests are organized into 10 groups, named "`TEST PHASE 0: …`" to "`TEST PHASE 9: …`".

So that the compiler can find class `Set`, when compiling the source file `test.cpp`, it is needed to have `#include "set.hpp"` at the start of `test.cpp`.

Starting from "`TEST PHASE 0`", you should implement those member functions needed by each test phase, then compile, link and, run the program (use `ctrl-F5` in Visual Studio). If it passes the tests for the current test phase, then you can proceed to the next phase. The `main` in the file `test.cpp` clearly indicates which member functions are required for each test phase.

Description of class `Set`

The class `Set` definition contains a private class `Node` (see file `set.hpp`). 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 only used in assertions appearing in the test code (`main` function) to help detecting possible memory bugs. 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. **Your code should not modify this counter in any way.**

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 is given below.

TEST PHASE 0

- `Set();` -- implementation given
  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 set. 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 a **non-sorted** vector `V` of **non-unique** integers
  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 `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<<` writes all the elements in set `S` to the output stream `os`,
  e.g. `std::cout << S;`

TEST PHASE 5

- `bool is_subset(const Set& S) const;`
  `R.is_subset(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.is_subset(S)` returns `true`), while S is not a subset of R, (i.e. `S.is_subset(R)` returns `false`).

TEST PHASE 6

- `Set set_union(const Set& S) const;`
  `R.set_union(S)` returns a new set with the set **union** of `R` and `S` ($R \cup S$). 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.set_union(S)` returns the set $\{1, 2, 3, 4, 9, 11\}$.

TEST PHASE 7

- `Set set_intersection(const Set& S) const;`
  `R.set_intersection(S)` returns a new set with the **intersection** of `R` and `S` ($R \cap S$). 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.set_intersection(S)` returns the set $\{1, 4\}$.

TEST PHASE 8

- `Set set_difference(const Set& S) const;`
  `R.set_difference(S)` returns a new set with the **difference** of `R` and `S` ($R - S$). 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.set_intersection(S)` returns the set $\{3\}$.

Requirements

- The public interface of the class (given in the header file) 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, though.
- The nodes in the singly linked list must be sorted.
- Members functions of class `Set` should not modify the variable `Set::Node::count_nodes`.
- The implementation of the constructor creating a set from a non-sorted vector of non-unique integers cannot rely on sorting a vector. Thus, the algorithm `std::sort` nor `std::unique` should not be used .

  [**Hint**: for each item, consider inserting it in a sorted list.]
- The implementation of the member function `Set::set_union` should use an **algorithm** similar to the one implemented for exercise 3 of set 1 of exercises. Note, however, that you should not use vectors to implement `Set::set_union` (i.e. your code should not copy the sets elements into vectors, then merge the sequences in the vectors, and finally create a new set from the merged sequence). Instead, your solution should re-implement the algorithm used in the solution of exercise 3, set 1, but using solely singly linked lists (instead of vectors).
- It is not a good programming practice to spread the use of `new` and `delete` all over the code. Instead, define private member functions, `insert` and `remove`, to insert and remove nodes, respectively. For instance, a member function `insert` can be declared as follows.

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

  These functions should also update the `Set::counter` data member which keeps track of the number of elements in the set (see class definition in file `set.hpp`).

  Thus, `Set` member functions should call the private member functions `insert` and `remove`, whenever it's needed to allocate memory for a new node (with `new`) or deallocate a node's memory (with `delete`).
- You can add extra tests to the `main` function. 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.

Necessary requirements for approving your lab are given below.

- The code must be readable, well-indented, and use good programming practices.
- Your code must pass all tests in `test.cpp`.
- Your code must satisfy the requirements listed above.
- The compiler must not issue warnings when compiling your code.
- There are no memory leaks or other memory related bugs. You can use some 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 `test.cpp`) output is shown below.

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

TEST PHASE 1: constructor from a vector
TEST PHASE 1: cardinality and empty

TEST PHASE 2: copy constructor

TEST PHASE 3: operator=

TEST PHASE 4: member

TEST PHASE 5: is_subset

TEST PHASE 6: union

TEST PHASE 7: set_intersection

TEST PHASE 8: difference

TEST PHASE 10: union, intersection, and difference

Success!!
```

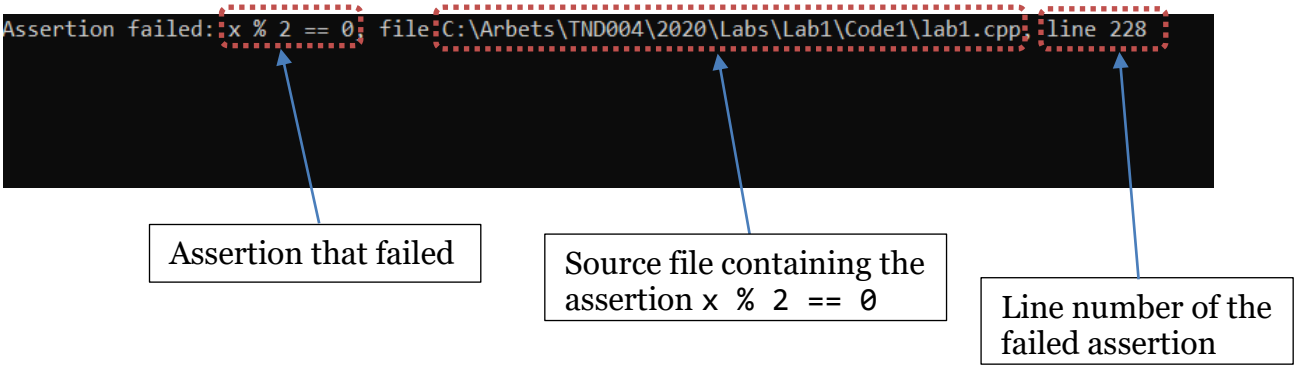
Appendix

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.



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 are several 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. Visual leak detector is not installed in the computers of the lab rooms.