

# Software Design (C++) Course Project

Eric Andrews

013594545

Autumn 2010

Department of Computer Science, University of Helsinki

## ***Task: implement a singly linked list of string values in C++***

The given task was to implement a singly linked list which stores string values. The list itself should contain standard data-type operations, ability to access, pop and push to front of list, iterator-support for traversing the list, the ability to modify the list in an arbitrary position via iterators, and finally, special operators for reversing and swapping lists.

Furthermore, technical requirements were set: comprehensive automated unit testing and verification of preconditions, postconditions along with class invariants in a liberal manner.

## ***Implementation***

I divided the list and the iterators into two separate modules to keep the code less cluttered. In the list file I also defined a struct named `SNode` to represent a single node in the list, and an exception specific to the list-class. Similarly I defined an exception for iterators in the iterator file. Because we weren't allowed to use existing testing frameworks, I wrote my own mini testing framework.

## **SList**

Contains a single data member, namely the pointer to the first node, which is `NULL` when the list is empty. Member functions throw an `SListException` if a precondition fails.

Postconditions are checked with the `assert`-facility when applicable. Class invariants are checked before and after performing computations that may alter the state of the object.

*Special or nontrivial decisions and solutions:*

- `insert_after` on `begin()` -iterator of empty list throws an exception.
- One cannot input strings containing whitespace via input operator because whitespace are interpreted as delimiters. Example: { hello world } can't be inputted as "hello world".
- There must be whitespace between the closing curly bracket and the last value when using input operator. Otherwise the closing bracket will be interpreted as part of the last value. Example: { hello world} } is interpreted as "hello" and "world}".
- Member functions that take iterators as parameters are tested extensively. Basically we check whether the given iterator points to the list by going through to whole list. This operation costs  $O(n)$ . In real world usage these checks should be turned off.
- Class invariant is checked by member function `check()`. The function verifies that the list doesn't contains loops by applying the *"The Tortoise and the Hare Algorithm"*. It

also checks that *empty()* works as expected. The function costs  $O(n)$  and should be removed from use in real world code.

## **SListIterator and SListConstIterator**

Iterators' data consist of a single pointer to a node. To avoid repeating myself, I set up a class hierarchy between the const iterator and its non-const counterpart so that the latter derives from the first. To enhance encapsulation I gave `SList` friend-access to privates, and made the derived iterator (non-const) use the base class through protected members.

*Special or nontrivial decisions and solutions:*

- An `end()`-iterator is defined so that its node pointer is a `NULL`.
- Similarly `begin()` for an empty list points to `NULL`. Thus `begin == end`.
- There is no “pure” zero-argument constructor. To achieve the same functionality I use `NULL` as a default value in the default constructor of both iterators.
- Because of a compiler problem called *circular dependency*, I am unable to include the list header directly into the iterator header via `#include`. Instead I use forward declaration to notify the compiler of the existence of the list and node.

## **test\_helpers.cpp and test\_helpers.hpp**

Contains helper (template) functions for asserting that two objects are equal (`==`), not equal (`!=`), same (same pointer address?) and not same. Also allows asserting whether an expression evaluates to true or false, and contains a function to signal failure when an exception should have been thrown but wasn't.

Tests are run through `TestRunner`. It is first given pointers to the test functions along with verbal explanations for each test, and then the tests are run. After completing each test it will indicate whether the test passed (OK) or failed. A test function will fail if an assert evaluates to the opposite of what was expected, or if an exception is thrown by something in the code. If a 3rd-party exception is thrown (`NotAssertionFalseException`) the whole test suite will stop there and allow the exception to propagate upwards. Otherwise the test suite will continue normally.

## **s\_list\_tests.cpp, s\_list\_iterator\_tests.cpp**

Both contain several tests for testing out the list and the iterators along with a function that tie tests to verbal descriptions and passes the whole thing to a test runner. *main.cpp* combines list tests and iterator tests, and provides means to run them both.

## **Compiling and running (Linux)**

Use provided makefile for easier compilation. Compile simply by executing `make` in terminal and run by executing `./tests`.