PA: Singly-Linked List Class `sllist<T>`

Learning Outcomes

- Gain experience in the design and implementation of class templates
- Learn about how to implement nested classes
- Gain experience in developing software that follows data abstraction and encapsulation principles
- Reading, understanding, and interpreting C++ unit tests written by others

Task

The goal is to exercise the topic of *class templates* by modifying class `ListInt` from the previous programming assignment to design and implement a class template `sllist<T>`. The generic representation using class templates will now allow users to construct lists of different types such as `int`s, `double`s, `Student`s, and many other existing types in your programs and types yet to be invented.

1. Begin by refactoring the nested `Node` structure:

```
struct Node {
  Node* next{nullptr}; // pointer to next Node
  value_type data;     // actual data of type int in node
  Node(T const&);      // conversion ctor to initialize Node object
  ~Node();             // dtor
  // count of Nodes created [by currently instantiated lists]
  static size_type node_counter;
};
```

   Static data member `node_counter` will keep track of the total number of instantiations of objects of type `Node` across the many instantiations of objects of type `ListInt`.

2. In class `ListInt`, add static member function `node_count` that returns the value in `Node::node_counter`.

3. Add a constructor that satisfies the following use case:

```
std::array<int, size10> ai{-1,-2,-3,-4,-5,-6,-7,-8,-9,-10};
// linked list of 10 nodes with head pointing to node with
// data value -1 and tail pointing to node with data value -10
ListInt li(std::begin(ai), std::end(ai));

int ai2[]{-1,-2,-3,-4,-5,-6,-7,-8,-9,-10};
// same as above
ListInt li2(std::begin(ai2), std::end(ai2));
// same as above
ListInt li3(ai2, ai2+sizeof(ai2)/sizeof(int));
```

4. Add member function `front` that returns a reference to the node data at the front of the list. Add a `const` overload of `front` that returns a read-only reference.
5. Refactor member function `pop_front` so that it returns nothing.
6. In namespace `hlp2`, add non-member function `swap` to swap two objects of type `ListInt`.
7. Test your implementation by amending the driver from the previous assignment.
8. Use `ListInt` to define a class `ListStr` for nodes with values of type `hlp2::Str` [which is defined in the files str.hpp and str.cpp].
9. Test `ListStr` by turning on the `DEBUG` macro in `str.cpp` [you can supply `DEBUG` macro to the compiler by using the `-D` flag of g++ [as in `-D=DEBUG`]. You must ensure your definition of `ListStr` is efficient by replacing any pass-by-value semantics in `ListInt` with pass-by-reference semantics. Of course, any changes to class `ListStr` must be mirrored in class `ListInt`.
10. After thoroughly testing classes `ListInt` and `ListStr`, define class template `sllist` in file sllist.hpp. Define static data members, static member functions, class template member functions, and non-member functions in file sllist.tpp. Of course, you must not forget to include file sllist.tpp at the bottom of file sllist.hpp. You will submit both files.
11. ***File-level documentation is required for both files. Function-level documentation of data members, member and non-member functions is required in*** sllist.hpp***.***
12. ***Your submissions must not include standard library headers*** `<forward_list>` ***and*** `<list>`***.***
13. Use driver sllist-int-driver.cpp to test your definition of class template `sllist<T>` where `T` is `int`. The correct output files are located in directory int+output.
14. Use driver sllist-hlp2str-driver.cpp to test your definition of class template `sllist<T>` where `T` is `hlp2::Str`. The correct output files with `DEBUG` macro turned on and off are located in folders hlp2str+output+debug and hlp2str+output, respectively.
15. Use driver sllist-person-driver.cpp to test your definition of class template `sllist<T>` where `T` is `Person`. The correct output files are located in folder person+output.
16. In addition to lecture presentations and source code, use the following references from the text book:
    1. Section 16.1 for an introduction to class templates.
    2. Page 667 for an explanation of `static` members of class templates.
    3. Pages 669-670 for using keyword `typename` [and not keyword `class`] for accessing class template members that are types.
    4. Section 19.6 for an introduction to nested classes.

Submission Details

Please read the following details carefully and adhere to all requirements to avoid unnecessary deductions.

Header file and documentation details

You will be submitting files $\mathrm{sllist.hpp}$ and $\mathrm{sllist.tpp}$.

Compiling, executing, and testing

Download the three drivers and corresponding output files containing the correct output. To enhance your compiling, linking, and testing experience, I recommend refactoring a `makefile` that can automate these tasks.

File-level and function-level documentation

Every source and header file *must* begin with a *file-level* documentation block. This module will use Doxygen to tag source and header files for generating html-based documentation. In addition, every function that you declare and define and submit for assessment must contain *function-level documentation*. This documentation should consist of a description of the function, the inputs, and return value.

Submission and automatic evaluation

1. In the course web page, click on the appropriate submission page to submit the necessary files.
2. Please read the following rubrics to maximize your grade. Your submission will receive:
    - F grade if your submission doesn't compile with the full suite of `g++` options.
    - F grade if your submission doesn't link to create an executable.
    - Your implementation's output doesn't match correct output of the grader (you can see the inputs and outputs of the auto grader's tests). The auto grader will assign 50% of the grade based on the input and output files given to you. The remaining 50% of the grade will be awarded based on the additional tests implemented by the auto grader.
    - The auto grade will provide a proportional grade based on how many incorrect results were generated by your submission. $A+$ grade if your output matches correct output of auto grader.
    - A deduction of one letter grade for each missing documentation block in your submissions. Each source file must have **one** file-level documentation block and a function-level documentation block for each defined function. A teaching assistant will physically read submitted source files to ensure that these documentation blocks are authored correctly. Each missing or incomplete or copy-pasted (with irrelevant information from some previous assessment) block will result in a deduction of a letter grade. For example, if the automatic grader gave your submission an $A+$ grade and one documentation block is missing, your grade will be later reduced from $A+$ to $B+$. Another example: if the automatic grade gave your submission a C grade and the two documentation blocks are missing, your grade will be later reduced from C to F.