Programming Assignment: 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

Setup

Download archive `ass8.zip` from the assessment web page and unzip the archive into directory `ass8`. The directory contains the following:

- Driver `sllist-driver-int.cpp` to test your definition of class template `hlp2::sllist<T>` when `T` is `int`. This driver contains five tests and the corresponding correct output files named `test?.txt` are located in directory `output/int`. A *makefile* named `makefile-int` is provided to automate the building and testing of both the non-debug and debug versions of the executable.
- Driver `sllist-driver-person.cpp` to test your definition of `hlp2::sllist<T>` when `T` is a C-style structure `Person` [that is defined in the driver]. This driver contains fourteen tests and the corresponding correct output files named `test?.txt` are located in directory `output/person`. A *makefile* named `makefile-person` is provided to automate the building and testing of both the non-debug and debug versions of the executable.
- Driver `sllist-driver-hlp2str.cpp` to test your definition of `hlp2::sllist<T>` when `T` is `hlp2::Str`. Header file `str.hpp` [containing definition of class `hlp2::Str`] and source file `str.cpp` [containing definitions of both member functions of class `hlp2::Str` and related non-member functions] are also provided. The driver contains five tests and the corresponding correct output files named `test?.txt` are located in directory `output/hlp2str`. A *makefile* named `makefile-hlp2str` is provided to automate the building and testing of `hlp2::sllist<T>` when `T` is `hlp2::Str`.

You'll submit two files `sllist.hpp` and `sllist.tpp`. Header file `sllist.hpp` contains definition of class template `hlp2::sllist<T>`. Define static data members, static member functions, class template member functions, and non-member functions in file `sllist.tpp`. You must include file `sllist.tpp` at the bottom of file `sllist.hpp`.

> ***Both submitted files must not include standard library header `<forward_list>` nor standard library header `<list>`. In addition, both submitted files must not contain the text `<forward_list>` nor the text `<list>`.***

Task

The goal is to exercise the topic of *class templates* by converting class `hlp2::list_int` [from the previous programming assignment] into a class template `hlp2::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 other existing types [such as `hlp2::Str`] in your programs and surprisingly types that you've yet to invent.

1. Begin by refactoring structure `Node` nested in class `list_int`. The previous assignment defined structure `Node` like this:

```
class list_int {
public:
  // type aliases
public:
  // interface
  static size_type object_count();
private:
  struct Node {
    int data;     // actual data in the node
    Node *next;  // pointer to next Node
  };

  Node *head {nullptr};  // pointer to first node of list
  Node *tail {nullptr};  // pointer to last node of list
  size_type counter {0}; // number of nodes in list

  Node* new_node(value_type data) const;
};
```

   The refactored structure `Node` will now look like this:

```
class list_int {
public:
  // type aliases
public:
  // interface
  static size_type object_count();
private:
  struct Node {
    value_type data;      // actual data of type int in node
    Node* next{nullptr}; // pointer to next Node
    Node(value_type const&); // conversion ctor to initialize
                             // Node object with value of type int
    ~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 `hlp2::list_int`.

2. In class `hlp2::list_int`, add static member function `node_count` that returns the value in `Node::node_counter`.
3. Add a constructor that satisfies the following use cases:

```
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
hlp2::list_int li(std::begin(ai), std::end(ai));

int ai2[]{-1,-2,-3,-4,-5,-6,-7,-8,-9,-10};
// same as above
std::list_int li2(std::begin(ai2), std::end(ai2));
// same as above
std::list_int 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. In the previous assignment, member function `pop_front` returned the value of node at the front of the list and then destroyed this front node. Now, refactor member function `pop_front` so that it destroys the front node and returns nothing.
6. In namespace `hlp2`, add non-member function `swap` to efficiently swap two objects of type `list_int`.
7. Test your implementation by amending the driver from the previous assignment.
8. Use `std::list_int` as a recipe to define a class `std::list_str` for nodes with values of type `hlp2::Str` [header file **str.hpp** contains the definition of class `hlp2::Str` while source file **str.cpp** contains definitions of class `hlp2::Str`'s member functions and related non-member functions].
9. You must ensure your definition of `list_str` is efficient by replacing any pass-by-value semantics in `list_int` with pass-by-reference semantics. Of course, any changes to class `list_str` must be mirrored in class `list_int`.
10. Test your implementation by amending the driver that you used to test class `std::list_int` from step 7. You can get more detailed debug information about class `hlp2::Str` and its use in class `std::list_str` by turning on macro DEBUG in **str.cpp**. This can be done using option -D with g++ [as in -D=DEBUG] when compiling source file **str.cpp**.
11. After thoroughly testing classes `std::list_int` and `std::list_str`, define class template `hlp2::sllist<T>` 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 these files.
12. ***File-level documentation is required for both files. Function-level documentation of data members, member and non-member functions is required only in* sllist.hpp*.***
13. ***Your submissions must not include standard library headers* `<forward_list>` *and* `<list>`*.***

> ***Both submitted files must not include standard library header* `<forward_list>` *nor standard library header* `<list>`*. In addition, both submitted files must not contain the text* `<forward_list>` *nor the text* `<list>`*. Therefore, remove all such occurrences before submitting your submissions to the online grader.***

14. Use driver `sllist-driver-int.cpp` and *makefile* `makefile-int` to test your definition of class template `sllist<T>` when `T` is `int`. The 5 correct output files named `test?.txt` are located in directory `output/int`.
15. Use driver `sllist-driver-person.cpp` and *makefile* `makefile-person` to test your definition of class template `sllist<T>` when `T` is `Person`. The 14 correct output files named `test?.txt` are located in folder `output/person`.
16. Use driver `sllist-driver-hlp2str.cpp` to test your definition of class template `sllist<T>` when `T` is `hlp2::Str`. The 5 correct output files named `test?.txt` are located in directory `output/hlp2str`. The 5 correct output files named `debug-test?.txt` with DEBUG macro turned on are also located in directory `output/hlp2str`.
17. 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.

Compiling, executing, and testing

Let's suppose you've defined class template `hlp2::sllist<T>` in file `sllist.hpp` and defined member functions of this class template and related non-member functions in `sllist.tpp`. Use driver source file `sllist-driver-int.cpp` to test your definition of class template `hlp2::sllist<T>` by instantiating and exercising objects of type `hlp2::sllist<int>`. The driver contains five tests and the corresponding correct output files are located in directory `output/int`. First, you create a pair of directories to store the non-debug and debug versions of the executables and output files generated by these executables:

```
$ mkdir release-int debug-int
```

You create executable `sllist-int.out` in directory `release-int` like this:

```
$ g++ -std=c++17 -pedantic -Wall -Wextra -Werror sllist-driver-int.cpp -o ./release-int/sllist-int.out
```

You execute each unit test and store the corresponding output in directory `release-int`. For example, you'd do this to execute unit test 1:

```
$ ./release-int/sllist-int.out 1 > ./release-int/your-test1.txt
```

Next, you compare your output from the unit test against the correct output like this:

```
$ diff -y --strip-trailing-cr --suppress-common-lines ./release-int/your-test1.txt ./output/int/test1.txt
```

If **`diff`** doesn't generate any diagnostic output, your class `hlp2::sllist<int>` has passed the first unit test and will have to be further tested again the remaining four unit tests.

After successfully passing the five unit tests, you'll have to perform diagnostic checks using Valgrind to check for memory leaks and other memory related errors. You begin by using option $-g$ to create a debug executable **`sllist-debug-int.out`** in directory **`debug-int`**:

```
$ g++ -std=c++17 -pedantic -Wall -Wextra -Werror -g sllist-driver-int.cpp -o ./debug-int/sllist-debug-int.out
```

Use Valgrind to check for memory-related errors and then further compare your output to the correct output like this:

```
$ valgrind --tool=memcheck --track-origins=yes ./debug-int/sllist-debug-int.out 1 > ./debug-int/your-test1.txt
$ diff -y --strip-trailing-cr --suppress-common-lines ./debug-int/your-test1.txt ./output/int/test1.txt
```

If **`diff`** doesn't generate any diagnostic output, Valgrind did not find any memory-related errors and further your class `hlp2::sllist<int>` generated the correct output. You'll have to test against the remaining four unit tests to confirm that objects of type `hlp2::sllist<int>` instantiated by your class template `hlp2::sllist<T>` behave correctly.

Testing your output and checking for memory leaks and errors is doable by explicitly typing out commands in the Linux shell for one or two tests. However, it can become overly cumbersome when having to repeat for more than a couple of tests and for a number of different drivers. This is where an automation tool like $make$ shines.

When you run $make$, by default, it will look for a $makefile$ called **`makefile`**. The $makefile$ to automate the testing of `hlp2::sllist<T>` when `T` is `int` is named **`makefile-int`**. Use option $-f$ to tell $make$ that the $makefile$ is called **`makefile-int`** and then specify the various rules you'd like $make$ to run. Thus, we can run $make$ with rule prep in $makefile$ **`makefile-int`** to create directories **`release-int`** and **`debug-int`** directories in the current working directory:

```
$ make -f makefile-int prep
```

Run $make$ with rule release to create an up to date non-debug program executable **`sllist-int.out`** in **`./release-int`**:

```
$ make -f makefile-int release
```

There are five tests specified by command-line parameters 1, 2, 3, 4, and 5. To run test 1 using $make$, you'd run the command:

```
$ make -f makefile-int test1
```

The output file **your-test1.txt** is created in directory **./release-int** directory and the diff command is given this file and my [correct file] **test1.txt** [located in directory **output/int**]. If the diff command is not silent, your output is incorrect and your code must be debugged.

To run all five tests sequentially, run the command:

```
$ make -f makefile-int test-all
```

To ensure there are no memory leaks or errors, you must use Valgrind to analyze the runtime behavior of your program in relation to the memory allocated from the free store. Since Valgrind requires option -g, you need a different target. The **makefile** contains a target called debug. Run the *make* command with target debug like this to create the debug version of the executable in directory **./debug-int**:

```
$ make -f makefile-int debug
```

To run test 1 with Valgrind, you'd run the command:

```
$ make -f makefile-int debug-test1
```

In this case, the output file **your-debug-test1.txt** is created in directory **./debug-int** and the diff command is given this file and my correct output file **test1.txt** [located in directory **output/int**]. If the diff command is not silent, your output is incorrect and your code must be debugged.

To run all tests, run the command:

```
$ make -f makefile-int debug-test-all
```

If you want to get rid of all object, executable, and output text files, run *make* with the target clean:

```
$ make -f makefile-int clean
```

You can similarly use *makefile* **makefile-person** and driver source file **sllist-driver-person** to test objects of type `hlp2::sllist<Person>` instantiated from class template `hlp2::sllist<T>`.

Likewise, you can use *makefile* **makefile-hlp2str**, driver source file **sllist-driver-hlp2str**, files **sllist.hpp** and **sllist.tpp** to test your implementation of class template `hlp2::sllist<T>` by instantiating and exercising objects of type `hlp2::sllist<hlp2::Str>`.

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.