Lab: Singly-Linked List Class `ListInt`

Learning Outcomes

- Gain experience in class templates by implementing containers that are independent from types of stored data
- Gain experience in developing software that follows data abstraction and encapsulation principles
- Reading, understanding, and interpreting C++ unit tests written by others

Introduction

We applied the interface/implementation methodology in $\text{Lab } 4$ to create a singly-linked list ADT library. There were two problems with the ADT from $\text{Lab } 4$. The first problem is that the ADT was implemented using C-style opaque pointers. The second problem is that the ADT only supports a singly-linked list of `int` values. We have learned that the C++ class mechanism provides improved data encapsulation and abstraction facilities. Further, we learned about C++'s template mechanism that allows classes to be parameterized with one or more types. Container classes such as `std::vector<T>`, `std::stack<T>`, and `std::forward_list<T>` which are used to manage elements of a certain type, are a typical example of this feature. By using class templates, we can implement such container classes while the element type is still open.

Our aim is to use C++'s class and template mechanism to convert the singly-list ADT from $\text{Lab } 4$ into a class template `List<T>`. However, templates are a blessing and a curse. They have many useful properties, such as great flexibility and near-optimal performance, but unfortunately they're not perfect. As usual, the benefits have corresponding weakness. For templates, the main problem is that their flexibility and performance come at the cost of poor separation between the "inside" of a template [its definition] and its interface [its declaration]. Very often, these error messages come much later in the compilation process than we would prefer. This manifests itself in poor error diagnostics causing pages of spectacularly poor error messages to be printed to the console. The good news is that it is possible to reduce the generation of such error messages by following certain basic recommendations:

1. First develop and test the class using one or more concrete types.
2. Once that works, replace the concrete types with template parameters.
3. Next, it is time to test the class template with a template argument equivalent to a concrete type from step 1.

Today's Task

Today's task is to apply C++'s class mechanism and the design principles from lectures to define and implement a class `ListInt` that encapsulates a singly-linked list of `int` values. The node in the singly-linked list is similar to the one from $\text{Lab } 4$:

```
struct Node {
  int data;    // the actual data in the node
  Node *next;  // pointer to the next Node
};
```

We ensure data encapsulation by declaring `Node` in the private section of class `ListInt`.

```
class ListInt {
public:
  // type aliases
public:
  // interface
private:
  struct Node {
    int data;    // the actual data in the node
    Node *next;  // pointer to the next Node
  };
};
```

The ADT from Lab 4 requires us to iterate over the entire linked list to reach the last element. We alleviate this inefficiency by adding a data member `tail` that will point to the last element of the linked list. Data member `head` will point to the first element of the linked list. To provide practice with static data members and functions, we add a static data member `object_counter` to keep track of the number of instantiations of type `ListInt` and a public static member function `object_count` that allow clients to access the value in `object_counter`. Many member functions in the `ListInt` interface will dynamically allocate objects of `Node` that are appropriately initialized. This commonly used operation can be implemented as a private member function `new_node`. An incomplete view of class `ListInt` would look like this:

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

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

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

You must define the following type aliases: `size_type`, `value_type`, `reference`, `const_reference`, `pointer`, and `const_pointer`.

You must define the following member functions as the interface for class `ListInt`:

```
class ListInt {
public:
  // type aliases
public:
  // three ctors:
    // default ctor
    // copy ctor
    // single-argument ctor that takes a std::initializer_list<value_type>
  // destructor [that will call member function clear]

  // copy assignment with another ListInt object
  // copy assignment with std::initializer_list<value_type>
  // both copy assignment overloads should use copy-and-swap idiom

  // operator += overload: concatenates with nodes of another ListInt container
  // operator += overload: concatenates with std::initializer_list<value_type>

  // operator [] overload: returns int data of node at position i in linked list
  // operator [] overload: const overload of above function

  // push_front: adds value_type to front of the list
  // push_back:  uses tail data member to efficiently add
  //             value_type to back of the list
  // pop_front:  returns the value of node at front of the list
  //             and then destroys this front node
  // clear:      erases all nodes in linked list
  // swap:       exchanges contents of container with another ListInt container

  // size:       returns number of nodes
  // empty:      returns true if linked list is empty; otherwise false
private:
 // data encapsulated here ...
};

// three overloads of operator +:
   // add two ListInt objects
   // add ListInt and std::initializer_list<ListInt::value_tye>
   // add std::initializer_list<ListInt::value_tye> and ListInt
```

This exercise will provide you with a self-evaluation of your current knowledge of linked lists and classes. Further, it will serve as an excellent review of a substantial portion of the final test. Carefully review your implementation of $\text{Lab } 4$, linked lists, and classes [constructors, destructors, rule of 3, operator overloading, static data members and member functions].

Implementation details

Do not use the tests implemented in listint-driver.cpp - these tests will be useful only after you've implemented and tested the necessary interface. Instead, implement and test each member function using your own tests - this will allow you to strategize the order in which you wish to define the member functions. The following is my recommended strategy:

1. Begin by writing the definition of class `ListInt` in header file listint.hpp. Examine the use cases in driver listint-driver.cpp if you're unsure about declarations of member functions.
2. Provide definitions of static data member and member function in source file listint.cpp.
3. Both header and source files must not include headers `<forward_list>` and `<list>`.
4. Define private member function `new_node`.
5. The default constructor and member functions `size` and `empty` can be trivially implemented and must therefore be defined next.
6. Member function `push_back` is a critical function that can be used to implement the other two constructors and must therefore be implemented next. Use your driver to carefully test the behavior of `push_back` and the previously defined functions.
7. Using `push_back`, it is straightforward to define the two [non-default] constructors.
8. You should next define member function `clear`.
9. Using `clear`, the destructor can be trivially defined. Use valgrind to ensure your current implementation has no memory related errors.
10. Define member function `swap`.
11. Using `swap` and the copy-swap idiom, the two copy assignment overloads can be trivially defined.
12. Next, define the two overloads of operator `[]`.
13. Using these operator `[]` overloads, it is fairly straightforward to define both overloads of operator `+=`.
14. Having defined both overloads of operator `+=`, the three non-member overloads of operator `+` can be trivially defined.
15. You can complete class `ListInt`'s interface by defining member functions `push_front` and `pop_front`. These functions are not reused by other parts of the interface and must therefore be implemented at the end.
16. Finally, after thoroughly testing your implementation, you should perform one final set of checks by using the tests provided in file listint-driver.cpp.
17. You can now leverage the knowledge and expertise gained in defining class `ListInt` to define a class template `List<T>`. This topic will be considered in a forthcoming exercise.

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 listint.hpp and listint.cpp.

Compiling, executing, and testing

Download $\text{listint-driver.cpp}$ and output files containing the correct output for the 6 unit tests in the driver. Follow the steps from the previous lab to refactor a `makefile` that can compile, link, and test your program.

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.