

# 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 [Lab 4](#) to create a singly-linked list ADT library. There were two problems with the ADT from [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 [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 and thereby complete this assignment in a short time period by following certain basic recommendations:

1. First develop and test the class using specific types.
2. Once that works, replace the specific types with template parameters.
3. Next, it is time to test the class template with a template argument equivalent to a specific type from step 1. **Note that for class templates, member functions are instantiated only if they're used.** You can use this fact to save debugging time by *exercising only a small number of member functions at a time*. This means you want to eschew the use of my driver file until you're close to submitting the solution. Since my driver file exercises all of the class templates' member functions, even a minor error in a single member function could cause lengthy error messages to be generated by all the instantiations of member functions.
4. If you run into a lengthy error message, don't panic!!! First, look at the error itself and take out what is useful for you: e.g., missing operator, or something not assignable, or something that is `const` that should not be. Then find in the call stack, the innermost code in your driver where you call a member function of a class template. This will generally be the first part of the lengthy error message. Stare for a while at this code and its preceding lines

because this is the most likely place where the error occurred [in my experience]. Then ask yourself, does the template argument meet the requirements and the concepts of the template to which it is applied. For example, does this type allow default construction? Does it allow its objects to be copied, to have their address, and so on.

5. Do not get scared to the point that you decide not to use templates for the rest of your life. In most cases, the problem is much simpler than the never-ending error messages make us believe. In my experience, most errors in templates can be found faster than run-time errors - with some training. And, to obtain that training you'll be implementing a class template that clones the singly-linked list.

## 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 Lab 4:

```
1 struct Node {
2     int data;    // the actual data in the node
3     Node *next; // pointer to the next Node
4 };
```

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

```
1 class ListInt {
2 public:
3     // type aliases
4 public:
5     // interface
6 private:
7     struct Node {
8         int data;    // the actual data in the node
9         Node *next; // pointer to the next Node
10    };
11 };
```

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:

```
1 class ListInt {
2 public:
3     // type aliases
4 public:
5     // interface
6     static size_type object_count();
7 private:
```

```

8   struct Node {
9       int data;    // actual data in the node
10      Node *next;  // pointer to next Node
11  };
12
13  Node *head {nullptr}; // pointer to first node of list
14  Node *tail {nullptr}; // pointer to last node of list
15  size_type counter {0}; // number of nodes in list
16
17  Node* new_node(value_type data) const;
18  };

```

You must declare 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`:

```

1   class ListInt {
2   public:
3       // type aliases
4   public:
5       // three ctors:
6           // default ctor
7           // copy ctor
8           // single-argument ctor that takes std::initializer_list<value_type>
9       // destructor [that will call member function clear]
10
11      // copy assignment with another ListInt object
12      // copy assignment with std::initializer_list<value_type>
13      // both copy assignment overloads should use copy-and-swap idiom
14
15      // operator += overload: concatenates with nodes of another ListInt
16      // operator += overload: concatenates with
17      // std::initializer_list<value_type>
18
19      // operator [] overload: returns int data of node at position i in linked
20      // list
21      // operator [] overload: const overload of above function
22
23      // push_front: adds value_type to front of the list
24      // push_back:  uses tail data member to efficiently add
25      //              value_type to back of the list
26      // pop_front:  returns the value of node at front of the list
27      //              and then destroys this front node
28      // clear:      erases all nodes in linked list
29      // swap:       exchanges contents of container with another ListInt
30      //              container
31
32      // size:       returns number of nodes
33      // empty:      returns true if linked list is empty; otherwise false
34  private:
35      // data encapsulated here ...
36  };

```

```

35 // three overloads of operator +=:
36 // add two ListInt objects
37 // add ListInt and std::initializer_list<ListInt::value_tye>
38 // 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 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 [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*.