p4-calculator

EECS 280 Project 4: Linked List and Calculator

Project Due Monday, 19 Nov 2018, 8pm

Table of Contents

Project Roadmap

This is a big picture view of what you'll need to do to complete this project. Most of the pieces listed here also have a corresponding section later on in the spec that goes into more detail.

This project will be autograded for correctness, comprehensiveness of your test cases, and programming style. See the style checking tutorial for the criteria and how to check your style automatically on CAEN.

You may work alone or with a partner. Please see the syllabus for partnership rules.

Download the starter code

Use the tutorial from project 1 to get your visual debugger set up. Use this `wget` link `https://eecs280staff.github.io/p4-calculator/starter-files.tar.gz`.

Before setting up your visual debugger, you'll need to rename each `.h.starter` file to a `.h` file.

```
$ mv List.h.starter List.h
$ mv Stack.h.starter Stack.h
```

You'll also need to create these new files and add function stubs.

```
$ touch calc.cpp
```

These are the executables you'll use in this project:

- `List_compile_check.exe`
- `List_public_test.exe`
- `List_tests.exe`
- `Stack_public_test.exe`
- `calc.exe`

If you're working in a partnership, set up version control for a team.

Familiarize yourself with the code structure

The code structure is *templated* and *object-oriented*, with classes representing a doubly-linked list and a stack.

Test and implement the ADTs

You are provided interfaces for the `List` and `Stack` classes. Test and implement these.

- `List` : This container class is similar to the linked list discussed in the lecture, but with a few differences: it is doubly-linked to allow efficient inserts and deletes anywhere in the list, and it supports an iterator. We will also evaluate your test cases for `List` to see how well they expose bugs.
- `Stack` : You will then use your `List` to implement a `Stack` , which only allows push and pop operations from one end. This `Stack` class makes use of the `List` class, so you should implement the `List` class first.

Test and implement the postfix calculator

Write and test a `main()` function that runs an interactive calculator program. You will use the `Stack` class to implement a postfix (also known as RPN) calculator. In a postfix calculator, operators appear after their operands, rather than in between them. For example, to compute (2 + 3) * 5, you would type `2 3 + 5 *`

Submit

Submit the following files to the autograder.

- `List.h`
- `Stack.h`
- `calc.cpp`
- `List_tests.cpp`

List Class

The member functions you have to implement are given in `List.h.starter`. You should copy that file to `List.h` and then implement each member function. The main difference from the version in lecture is that it is doubly-linked. It also allows you to create an iterator and then use the iterator to search, insert, or delete at any position in the list. Note that this is a class template, so that it can hold data of any type. For class templates, it is necessary to give the code for member functions inside the header file (it turns out that the compiler requires that in order to instantiate the class, given a specific type). Therefore, there will **not** be a `List.cpp`. See the lecture slides on how to add member functions in the header file for a class template.

You must not change the public interface of the `List` class, and you must use a doubly-linked list (i.e., nodes chained using pointers) implementation (no arrays or vectors, etc.). The basic member functions that `List` provides are in `List.h.starter`.

You must manage memory allocation so that there are no memory leaks, etc. For example, when adding an item, you will need to dynamically allocate the memory for a node to hold the item's value and the pointers to the next and previous nodes in the linked list. When removing items, you will need to delete that previously allocated memory. The `List` destructor needs to ensure that all the nodes in the linked list are deleted.

To compile and run your `List` tests, run the following commands:

```
$ make List_tests.exe
$ ./List_tests.exe
```

Since C++ only instantiates templates that are needed, we have included a simple program that attempts to instantiate every member of the `List` template to make sure they compile. To compile and run the public `List` compilation test, run the following commands:

```
$ make List_compile_check.exe
$ ./List_compile_check.exe
```

Writing unit tests for List

You must write and submit tests for the `List` class. Your test cases MUST use the unit test framework, otherwise the autograder will not be able to evaluate them. Since unit tests should be small and run quickly, you are limited to **50** `TEST()` items per file and your whole test suite must finish running in less than 5 seconds. Please bear in mind that you **DO NOT** need 50 unit tests to catch all the bugs. Writing targeted test cases and avoiding redundant tests can help catch more bugs in fewer tests.

How we grade your tests

We will autograde your `List` unit tests by running them against a number of implementations of those modules. If a test of yours fails for one of those implementations, that is considered a report of a bug in that implementation.

We grade your tests by the following procedure:

1. We compile and run your test cases with a **correct** solution. Test cases that pass are considered **valid**. Tests that fail (i.e. falsely report a bug in the solution) are invalid. The autograder gives you feedback about which test cases are valid/invalid. Since unit tests should be small and run quickly, your whole test suite must finish running in **less than 5 seconds**.
2. We have a set of intentionally **incorrect** implementations that contain bugs. You get points for each of these "buggy" implementations that your **valid** tests can catch.
3. How do you catch the bugs? We compile and run all of your **valid** test cases against each buggy implementation. If **any** of these test cases fail (i.e. report a bug), we consider that you have caught the bug and you earn the points for that bug.

Stack Class

You should complete the implementation of `List` (and test it) before working on `Stack`. The skeleton code for `Stack` is given in `Stack.h.starter`. Copy `Stack.h.starter` to `Stack.h`. You must only use the public interface of the `List` class to implement the stack. The `List` class should not have any friends.

The core functions of the `Stack` class are `push(item)`, `pop()`, and `top()`. See the RMEs for the description of these operations. Given the `List` type, the basic operations on a stack, `push()` and `pop()`, are straightforward to implement. To push, you can simply add an element at one end of the list (either end will do). To pop, you simply remove the element from the same end

of the list. Another function is `top()` that simply returns the top element (as a reference to eliminate an unnecessary copy and to allow it to be modified) without modifying the stack.

Though you will not be turning them in, you should write your own test cases for the `Stack` class since the public tests are not comprehensive. Write them in a file called `Stack_tests.cpp`.

To compile and run your `Stack` tests, run the following commands:

```
$ make Stack_tests.exe
$ ./Stack_tests.exe
```

Postfix (RPN) Calculator

You will now use your `Stack` template to develop a Reverse-Polish Notation calculator in `calc.cpp`. The calculator must support integers and floating-point values (doubles).

Important: In order for your program to produce the correct output you must set the floating-point precision of cout to 4 using the following line of code at the beginning of your main function:

```
cout.precision(4);
```

An RPN calculator is one in which the operators appear **after** their respective operands, rather than in between them. So, instead of computing the following:

((2 - 3) * 4) / (-6)

an RPN calculator would compute this equivalent expression (note that "n" means negate):

```
2 3 - 4 * 6 n /
```

RPN notation is convenient for several reasons. First, no parentheses are necessary since the computation is always unambiguous. Second, such a calculator is convenient to implement with a stack. In the case above, when you see a number, you simply push it on the stack. When you see an operator, you pop the top two values (or just one for a unary operator), apply the operator on them, and then push the result back on the stack. In the case above, the stack would change as follows **(top value shown first)**:

- Stack after seeing `2` : [2]
- Stack after seeing `3` : **[3, 2]**
- Stack after seeing `-` operator: [-1]

- Stack after seeing `4` : **[4, -1]**
- Stack after seeing `*` operator: [-4]
- Stack after seeing **`6` : [6, -4]**
- **Stack after seeing `n` operator: [-6, -4]**
- Stack after seeing `/` operator: [0.6667]

Notice that the stack only contains numbers at all times. The operators **never** go on the stack.

The calculator program is invoked with no arguments, and it starts out with an empty stack. It takes its input from the standard input stream and writes its output to the standard output stream. Here are the commands your calculator must respond to and what you must do for each. **Each command is separated from the next one by one or more whitespace characters (including possibly newlines).**

Input	Action
`<some number>`	a number can be in any of the following forms: • one or more digits [0 – 9] (i.e. 2, 42, 900) • one or more digits followed by a decimal point (i.e. 2., 42., 900.) • zero or more digits, followed by a decimal point, followed by one or more digits (i.e. 3.5, 2.333333, .5) Notice that all these are non-negative values. Push the value on the stack. The following are examples of things that are not valid numbers for user input: -2, -0, 1,234. Only non-negative numbers are entered (to simplify your project).
`+`	pop the top two numbers off the stack, add them together, and push the result onto the top of the stack. This requires a stack with at least two operands. **Note:** You should avoid making multiple calls to `Stack` member functions within one statement, since the order in which operands are evaluated is undefined in C++. For example, in the expression `expr1 + expr2` it is possible for `expr2` to be evaluated before `expr1` . This can result in undefined behavior when `expr1` and `expr2` have side effects.
–	pop the top two numbers off the stack, subtract the first number popped from the second, and push the result onto the top of the stack. This requires a stack with at least two operands.
*	pop the top two numbers off the stack, multiply them together, and push the result onto the top of the stack. This requires a stack with at least two operands.

Input	Action
`/`	pop the top two numbers off the stack, divide the second value popped by the first number, and push the result onto the top of the stack. This requires a stack with at least two operands. **Note:** Your calculator must check for division by zero. If the user attempts to divide by zero, do the following before continuing the program as normal: • Put the two popped elements back on the stack in their original order • Print an error message using exactly the following line of code: `cout << "Error: Division by zero" << endl;`
`d`	duplicate: pop the top item off the stack and push two copies of the number onto the top of the stack. This requires a stack with at least one operand.
`r`	reverse: pop the top two items off the stack, push the first popped item onto the top of the stack and then the push the second item onto the top of the stack (this just reverses the order of the top two items on the stack). This requires a stack with at least two operands.
`p`	print: print the top item on the stack to standard output, followed by a newline. This requires a stack with at least one operand and leaves the stack unchanged.
`c`	clear: pop all items from the stack. This input is always valid.
`a`	print-all: print all items on the stack in one line, from top-most to bottom-most, each value followed by a single space. The end of the output must be followed by exactly one newline. This input is always valid and leaves the stack unchanged. For an empty stack, for example, only the newline will be printed. For a stack with two elements, say with stack contents being **[47, 42] (top value shown first)**, the following will be printed: `47 42 <NEWLINE>` (Where `<NEWLINE>` corresponds to the newline character produced by `endl` or `"\n"`).
`n`	negate: negate the top item on the stack. This requires a stack with at least one operand.
`q`	quit: exit the calculator with a 0 exit value. This input is always valid. End-of-file (e.g., typing control-D on Linux) must also cause the calculator to exit with status 0. Note: **do not** call `exit(0)`, because it will cause memory leaks!

Each command is separated by whitespace.

For simplicity, you can assume that you are given valid input in our tests. No error checking on inputs to the calculator is required.

Implement your calculator in a file called `calc.cpp`.

To compile your calculator, you can use:

```
$ make calc.exe
```

To run your calculator interactively, you can use:

```
$ ./calc.exe
```

And then start typing the commands.

Negative zero

The C++ `double` format distinguishes between positive and negative zero. The following example illustrates negative zero:

```
$ ./calc.exe
1 n 0 * p
-0
```

Your program should not do anything special for negative zero.

Requirements and Restrictions

It is our goal for you to gain practice with good C++ code, classes, and dynamic memory.

DO	DO NOT
Modify .cpp files, `List.h` and `Stack.h`	Modify other `.h` files
For `List` and Stack, make helper member functions private	Modify the public interface of `List` or `Stack`
Use these libraries: `<iostream>`, `<string>`, `<cassert>`, `<sstream>`, `<utility>`	Use other libraries

DO	DO NOT
`#include` a library to use its functions	Assume that the compiler will find the library for you (some do, some don't)
Use C++ strings	Use C-strings
Send all output to standard out (AKA stdout) by using `cout`	Send any output to standard error (AKA stderr) by using `cerr`
`const` global variables	Global or static variables
Pass large structs or classes by reference	Pass large structs or classes by value
Pass by const reference when appropriate	"I don't think I'll modify it ..."
Use Valgrind to check for memory errors	"It's probably fine..."

Starter Code

You can find the starter files on the course website.

File(s)	Description
`List.h.starter`	Skeleton `List` class template header file without function implementations. Rename this file to `List.h` and then add your function implementations.
`Stack.h.starter`	Skeleton `Stack` class template. Rename it to `Stack.h` and then add your function implementations.
`List_tests.cpp`	Add your `List` unit tests to this file.
`List_compile_check.cpp`	A "does my code compile" test for `List.h`
`List_public_test.cpp`	A very small test case for `List.h` .
`Stack_public_test.cpp`	A few basic test cases for `Stack.h` .

`calc_test00.in`
`calc_test00.out.correct`
`calc_test01.in`
`calc_test01.out.correct`
	Simple test cases for the calculator program.

File(s)	Description
`Makefile`	A `Makefile` that has targets for compiling the published test cases and your own tests. Use `$ make test` to compile and run all tests.
`unit_test_framework.h` `unit_test_framework.cpp`	The unit test framework you must use to write your test cases.

Appendix A: What's in a typename?

You saw the use of `typename` for declaring templates. When compiling your project, you may get the following kind of error:

```
./Stack.h:94:8: error: missing 'typename' prior to dependent type name 'List<T>::Iter
List<T>::Iterator i;
^~~~~~~
```

If you see an error message that talks about missing 'typename' prior to dependent type name, simply stick in the keyword " `typename` " before the type declaration. In the instance above, it would become:

```
typename List<T>::Iterator i;
```

The same thing would apply if you declared a loop variable. For example:

```
for (List<T>::Iterator i; /*...*/)
```

may need to become (if you get an error from the compiler):

```
for (typename List<T>::Iterator i; /*...*/)
```

Discussion of dependent types and why we have to insert the keyword `typename` is beyond the scope of this course (the reason is quite subtle and deep). If you want to see some explanation, see this article:

http://pages.cs.wisc.edu/~driscoll/typename.html

Appendix B: Project 4 Coding Practices Checklist

The following are coding practices you should adhere to when implementing the project. Adhering to these guidelines will make your life easier and improve the staff's ability to help you in office hours. You **do not** have to submit this checklist.

General code quality:

- ☐ Helper functions used if and where appropriate. Helper functions are designed to perform one meaningful task, not more
- ☐ Lines are not too long
- ☐ Descriptive variable and function names (i.e. `int radius` instead of `int x`)
- ☐ Effective, consistent, and readable line indentation
- ☐ Code is not too deeply nested in loops and conditionals
- ☐ Main function is reasonably short
- ☐ Avoids redundant use of `this` keyword

Test case quality:

- ☐ Test cases are small and test one behavior each.
- ☐ Test case names are descriptive, or test cases are commented with a short description of what they test.
- ☐ Test cases are written using the unit testing framework.

Project-specific quality:

- ☐ Calculator operations (+, -, etc.) in main are implemented as helper functions.
- ☐ The big three are only implemented when required.