mp_lists

Debugging and Lists

Extra credit: Feb 18, 23:59 PM
Due: Feb 25, 23:59 PM

Doxygen

Table of Contents

Direct links to Part 1 and Part 2

Solo MP This MP is to be completed without a partner.

You are welcome to get help on the MP from course staff, via open lab hours, or Piazza!

Goals

In this MP (machine problem) you will:

- learn to manipulate linked memory by writing functions to modify linked lists
- practice debugging more complex code
- practice using templates
- get familiar with iterators

Checking Out the Code

From your CS 225 git directory, run the following on EWS:

```
git fetch release
git merge release/mp_lists -m "Merging initial mp_lists files"
```

If you're on your own machine, you may need to run:

```
git fetch release
git merge --allow-unrelated-histories release/mp_lists -m "Merging initial mp_lists files"
```

Upon a successful merge, your mp_lists files are now in your `mp_lists` directory.

Background Information: Template Classes

Identical to what you saw in lecture, template classes provide the ability to create generic container classes. In this MP, you will be writing a `List` class.

```
template <typename T>
class List {
    // implementation
};
```

This simply says that our class `List` has a parametrized type that we will call `T`. Similarly, the constructor will look like this:

```
template <typename T>
List<T>::List() {
    // implementation
}
```

We need the `template <typename T>` or `template <class T>` above all of our functions—it becomes part of the function signature. The keywords `class` and `typename` can be interchanged.

Template classes need access to the implementation for compilation. Every time a different class is used as the template, the code must be compiled to support containing it. For example, if you want to make a `List<int>,` the compiler must take the generic `List<T>` implementation code and replace all the `T`s with `int`s inside it, and compile the result (this process is called **template instantiation**). Our solution to this is to `#include "List.hpp"` at the bottom of our `List.h` file, and not include `List.h` in our `List.hpp` file. This ensures that whenever a client includes our header file, he/she also gets the implementation as well for compilation purposes (there are other solutions, but this is how we will solve it in this course).

Background Information: Linked Lists

The interface of this `List` class is slightly different from what you have seen in lecture. This `List` has no sentinel nodes; the first node's `prev` pointer, and the last node's `next` pointer, are both `NULL`. In lieu of these sentinels, we keep a pointer `head` to the first node, and a pointer `tail` to the last node in the `List`. (In an empty list, both `head` and `tail` are `NULL`.) The `List` class also has an integer member variable, `length`, which represents the number of nodes in the `List`; you will need to maintain this variable.

Background Information: Iterators

We use iterators to figure out where we currently are in the list, what is the next/previous node, and to access the data. Iterator class has one member variable, namely a pointer to the node in the list. Some of the core functionality includes moving the pointer, getting current location, and checking the location of the iterator.

Background Information: GDB

We have a reference guide for GDB [here](). You should read through it to get an idea of how to start gdb, what commands you have available, etc.

To summarize a bit, you could run:

```
gdb --args ./test "[part=1]"
```

to debug the part one test cases with gdb,

```
gdb --args ./test "*insert*"
```

to debug all the tests with "insert" in their name.

Once you start gdb to execute the tests you want to run, you can set breakpoints to help you debug. For example, we could set a breakpoint at the beginning of a test case we want to debug, or breakpoint(s) at the beginning or end of functions which are not behaving as we expect. For example, if we are failing the insertFront test, we could add a breakpoint at the end of the `insertFront` function and see if the list looks how we expect. This will let us know whether to focus our debugging on exactly what the function is doing, or what the test case is doing after this call to `insertFront`.

For example, we can set a breakpoint in line 25 of our `List` class like so:

```
(gdb) b List.hpp:25
```

or a breakpoint in line 50 of the part 1 test cases like so:

```
(gdb) b tests/tests_part1.cpp:50
```

We can also set breakpoints using function names:

```
(gdb) b main
```

sets a breakpoint at the beginning of main, and

```
(gdb) b List<int>::sort()
```

sets a breakpoint at the beginning of the sort function. You can tab complete this, so for example after typing `b List<int>::ins` you can press tab to see a list of possible functions starting with `ins` in the `List` class, templatized with `int`.

Remember that Catch will print not only the name of a failing test case, but also what file the test was in and the line number of the failing assertion. You can use this to decide which tests to run and where you might want to set breakpoints.

Part 1: Debugging & Implementing Linked Lists

In your `mp_lists` folder, you will find that the `List` class is split into four `.h` or `.hpp` files:

- `List.h`
- `List.hpp`
- `List-ListIterator.hpp`
- `List-given.h`

We have provided a partial implementation of a few List functions for this part of the MP. Some functions are written, and some are unwritten. Those functions which are already coded may have a few bugs in them! This part of the MP is to help get you used to debugging certain kinds of logical and memory related bugs, as well as writing pointer manipulation code. All the functions are specified in `List.h`, and their (potentially empty) implementations are in `List.hpp` or `List-ListIterator.hpp` for you to write.

You should use gdb, valgrind, and any other debugging tools or techniques you're comfortable with to complete the first part of this MP (as well as general debugging in Part 2 and beyond).

See the Doxygen for details of the `List` class.

Notes on testing There are two ways to test this MP:

1. Using `make` to make `main.cpp` into `./mp_lists`, which allows you to write your own lists to test.
2. Using `make test` to make `./test`, which allows you to run the automated tests.

You're free to run Valgrind (or other tools) on the executables:

```
valgrind ./mp_lists
valgrind ./test [part=1]
```

You can also select test cases to run by their names, and run those under valgrind or gdb as well:

```
./test "List::reverse"
./test "*insert*"
valgrind ./test "*insert*"
gdb --args ./test "*insert*"
```

`List()`

This should default construct the list. Keep in mind everything mentioned in the background for the Linked List class.

`~List()` and `_destroy()`

Since the `List` class has dynamic memory associated with it, we need to define all of the Rule of Three. We have provided you with the Copy Constructor and overloaded `operator=`.

- You will need to implement the `_destroy()` helper function called by `operator=` (the assignment operator) and the destructor `~List()`
- The `_destroy()` function should free all memory allocated for `ListNode` objects.

Insertion

The `insertFront` Function

(See the Doxygen for `insertFront`.)

- This function takes a data element and prepends it to the beginning of the list.
- If the list is empty before `insertFront` is called, the list should have one element with the same value as the parameter.
- You may allocate new `ListNode`s.

Example For example, if `insertFront` is called on the list of integers

```
< 5 4 7 >
```

with the parameter `6`, then the resultant list should be

```
< 6 5 4 7 >
```

The `insertBack` Function

(See the Doxygen for `insertBack`.)

- This function takes a data element and appends it to the end of the list.
- If the list is empty before `insertBack` is called, the list should have one element with the same value as the parameter.
- You may allocate new `ListNode`s.

Example For example, if `insertBack` is called on the list of integers

```
< 5 4 7 >
```

with the parameter `6`, then the resultant list should be

```
< 5 4 7 6 >
```

Testing Your `insert` Functions

Once you have completed `insertFront` and `insertBack`, you should compile and test them. These tests do not rely on your iterator

```
make test
./test "List::insertFront*"
./test "List::insertBack*"
./test "List::insert*"
```

Iterator

In order to provide the client code with the ability to read the data from the list in a uniform way, we need to have an iterator. We have provided a list iterator class `List-ListIterator.hpp` which has some functionality implemented. However, there are a few functions yet to be written as well as some functions with buggy implementations! You will need to worry about all the functions with a `@TODO` comment:

- `ListIterator& operator++()`
- `ListIterator operator++(int)`
- `ListIterator& operator--()`
- `ListIterator operator--(int)`
- `bool operator!=(const ListIterator& rhs)`

You will also need to implement the `begin()` and `end()` functions in `List.hpp` to have a way of obtaining an iterator from a `List`.

Many of the more advanced functionality will be tested by using your iterator. So, you should make sure to debug and implement these after you have finished your insert functions but before you start working too much on the later functionality.

The `split` Helper Function

(See the [Doxygen for `split`].)

- This function takes in a pointer `start` and an integer `splitPoint` and splits the chain of `ListNode`s into two completely distinct chains of `ListNode`s after `splitPoint` many nodes.
- The split happens after `splitPoint` number of nodes, making that the `head` of the new sublist, which should be returned. In effect, there will be `splitPoint` number of nodes remaining in the current list.
- You may **NOT** allocate new `ListNode`s

Example For example, if `split` is called on the list of integers

```
list1 = < 1 2 3 4 5 >
```

then after calling `list2 = list1.split(2)` the lists will look like

```
list1 == < 1 2 >
list2 == < 3 4 5 >
```

Testing Your `split` Function

Once you have completed `split`, you should compile and test it.

```
make test
./test "List::split*"
```

You should see images `actual-split_*.png` created in the working directory (these are generated by repeatedly splitting `split.png`). Compare them against `expected-split_*.png`.

The `waterfall` Function

(See the [Doxygen for `waterfall`].)

- This function modifies the list in a cascading manner as follows.
- Every other node (starting from the second one) is removed from the list, but appended at the back, becoming the new `tail`.
- This continues until the next thing to be removed is either the `tail` (not necessarily the original `tail`!) or `NULL`.
- You may **NOT** allocate new `ListNode`s.
- Note that since the `tail` should be continuously updated, some nodes will be moved more than once.

Example For example, if `waterfall` is called on the list of integers

```
< 1 2 3 4 5 6 7 8 >
```

then the call to `waterfall()` should result in

```
< 1 3 5 7 2 6 4 8 >
```

(Do you see the pattern here?)

Step-by-Step Example We will look again at the list

```
< 1 2 3 4 5 6 7 8 >
```

When we call `waterfall`, this is how it should look step-by-step:

```
< 1 2 3 4 5 6 7 8 > - Skip the 1
  ^              ^
curr           tail

< 1 3 4 5 6 7 8 2 > - Remove the 2 and move it at the end
    ^            ^
  curr         tail

< 1 3 5 6 7 8 2 4 > - Skip the 3, and move the 4 to the end
      ^          ^
    curr       tail

< 1 3 5 7 8 2 4 6 > - Skip the 5 and move the 6 to the end
        ^        ^
      curr     tail

< 1 3 5 7 2 4 6 8 > - Skip the 7 and move the 8 to the end
          ^      ^
        curr   tail

< 1 3 5 7 2 6 8 4 > - We have moved past the original tail of the list.
            ^    ^      This is okay! Skip the 2 and move the 4 to the end,
          curr tail     now for the second time!

< 1 3 5 7 2 6 4 8 > Skip the 6 and move the 8 to the end, now for the second time!
              ^ ^
           curr tail
```

We are done now because we skip over the `4` and get to the `tail` of the list. The `8` stays in place, and we have finished. If you were keeping track of moves, you would notice that a number (they happen to be in order here for convenience) gets moved the same amount of times as it is divisible by 2! Technically this might not be true for the `8`, but we could have moved it that last time, it just would have stayed where it was (remove it from the `tail` and put it back to the `tail`). Kinda neat, huh?

Testing Your `waterfall` Function

Once you have completed `waterfall`, you should compile and test it.

```
make test
./test "List::waterfall"
```

Testing Part 1

Compile your code using the following command:

```
make test
```

After compiling, you can run all of the part one tests at once with the following command:

```
./test [part=1]
```

Notes

- These tests are **deliberately insufficient**. We strongly recommend augmenting these tests with your own.
- Be sure to think carefully about edge cases and reasonable behavior of each of the functions when called on an empty

list, or when given an empty list as a parameter.
- It is **highly advised** to test with lists of **integers** before testing with lists of `HSLAPixel`s.
- Printing out a list both forward and backwards (eg, using an iterator or a custom print function) is one way to check whether you have the double-linking correct, not just forward linking. Printing the size may also help debug other logical errors.

DOUBLE CHECK that you can confidently answer "no" to the following questions:

- Did I allocate new memory in functions that disallow it?
- Did I modify the data entry of any `ListNode`?
- Do I leak memory?

Extra Credit Submission

For extra credit, you can submit the code you have implemented and tested for part one of mp_lists. Follow the submission instructions section for handing in your code.

Part 2

The `reverse` Helper Function

(See the Doxygen for `reverse`.)

In `List.hpp` you will see that a public `reverse` method is already defined and given to you. You are to write the helper function that the method calls.

- This function will reverse a chain of linked memory beginning at `startPoint` and ending at `endPoint`.
- The `startPoint` and `endPoint` pointers should point at the new start and end of the chain of linked memory.
- The `next` member of the `ListNode` before the sequence should point at the new start, and the `prev` member of the `ListNode` after the sequence should point to the new end.
- You may **NOT** allocate new `ListNode`s.

Example For example, if we have a list of integers

```
< 1 2 3 4 5 6 7 >
```

(with `head` pointing at `1` and `tail` pointing at `7`) and call the public function `reverse()`

The resulting list should be

```
< 7 6 5 4 3 2 1 >
```

(with `head` pointing at `7` and `tail` pointing at `1`)

Your helper function should be as general as possible! In other words, **do not** assume your `reverse()` helper function is called only to reverse the entire list—**it may be called to reverse only parts of a given list**.

Additionally, the pointers `startPoint` and `endPoint` that are parameters to this function should at its completion point to the beginning and end of the new, reversed sublist.

We highly recommend you write this function iteratively. It is possible that you may run out of stack space if you write this function recursively.

The `reverseNth` Function

(See the [Doxygen for `reverseNth`].)

- This function accepts as a parameter an integer, n, and reverses blocks of n elements in the list.
- The order of the blocks should not be changed.
- If the final block (that is, the one containing the `tail`) is not long enough to have n elements, then just reverse what remains in the list. In particular, if n is larger than the length of the list, this will do the same thing as reverse.
- You may **NOT** allocate new `ListNode`s.

Example For example, if `reverseNth` is called on the list of integers

```
< 1 2 3 4 5 6 7 8 9 >
```

then the call to `reverseNth(3)` should result in

```
< 3 2 1 6 5 4 9 8 7 >
```

For the list of integers

```
< 1 2 3 4 5 6 >
```

the call to `reverseNth(4)` should result in

```
< 4 3 2 1 6 5 >
```

Hint You should try to use your `reverse()` helper function here.

Testing Your `reverse` Functions

Once you have completed `reverse` and `reverseNth`, you should compile and test them.

```
make test
./test "List::reverse"
./test "List::reverseNth #1"
./test "List::reverseNth #2"
```

Sorting

You will be implementing the helper functions for one more member function of the `List` template class: `sort`. This is designed to help you practice pointer manipulation and solve an interesting algorithm problem. In the process of solving this problem, you will implement several helper functions along the way—we have provided public interfaces for these helper functions to help you test your code.

The `merge` Helper Function

(See the [Doxygen for `merge`].)

- This function takes in two pointers to heads of sublists and merges the two lists into one in sorted order (increasing).
- You can assume both lists are sorted, and the final list should remain sorted.
- You should use `operator<` on the data fields of `ListNode` objects. This allows you to perform the comparisons necessary for maintaining the sorted order.
- You may **NOT** allocate new `ListNode`s!

Example For example, if we have the following lists

```
list1 = < 1 3 4 6 >
list2 = < 2 5 7 >
```

then after calling `list1.mergeWith(list2)` the lists will look like

```
list1 == < 1 2 3 4 5 6 7 >
list2 == < >
```

Testing Your `merge` Function

Once you have completed `merge`, you should compile and test it.

```
make test
./test "List::merge"
```

You should see the image `actual-merge.png` created in the working directory if your program terminates properly. This is generated by merging the images `tests/merge1.png` and `tests/merge2.png`. Compare this against `expected-merge.png`.

The `mergesort` Helper Function

(See the [Doxygen for `mergesort`].)

- This function sorts the list using the [merge sort] algorithm, explained below.
- You should use `operator<` on the data fields of `ListNode` objects. This allows you to perform the comparisons necessary for sorting.
- You should use the private helper functions you wrote above to help you solve this problem.
- You may **NOT** allocate new `ListNode`s
- This function's runtime will be graded for efficiency (correct Big-Oh runtime)

Example For example, if `sort` is called on the list of integers

```
< 6 1 5 8 4 3 7 2 9 >
```

the resulting list should be

```
< 1 2 3 4 5 6 7 8 9 >
```

Merge Sort — Algorithm Details

Merge Sort is a recursive sorting algorithm that behaves as follows:

- **Base Case**: A list of size 1 is sorted. Return.
- **Recursive Case**:

- Split the current list into two smaller, more manageable parts
- Sort the two halves (this should be a recursive call)
- Merge the two sorted halves back together into a single list

In other words, Merge Sort operates on the principle of breaking the problem into smaller and smaller pieces, and merging the sorted, smaller lists together to finally end up at a completely sorted list.

Testing Part 2

Compile your code using the following command:

```
make test
```

After compiling, you can run the part two tests at once with the following command:

```
./test [part=2]
```

Hint: Comparing similar images Occasionally `diff` may tell you that the 2 images differ, but you cannot easily tell the difference with the naked eye. In these scenarios, there is a great tool on ews machines called `compare` which can help you.

```
compare out.png out_01.png out_difference.png
```

This command will create a new image called out_difference.png where any differing pixels will be bright red.

Notes

- These tests are **deliberately insufficient**. We strongly recommend augmenting these tests with your own.
- Be sure to think carefully about reasonable behavior of each of the functions when called on an empty list, or when given an empty list as a parameter.
- It is **highly advised** to test with lists of **integers** before testing with lists of `HSLAPixel`s.
- Printing out a list both forward and backwards is one way to check whether you have the double-linking correct, not just forward linking. Printing the size may also help debug other logical errors.

DOUBLE CHECK that you can confidently answer “no” to the following questions:

- Did I allocate new memory in functions that disallow it?
- Did I modify the data entry of any `ListNode`?
- Do I leak memory?

Submission

Our grading system will checkout your most recent (**pre-deadline**) commit for grading. Therefore, to hand in your code, all you have to do is commit it to your Subversion repository.

Be sure your working directory is the `mp3` folder that was created when you checked out the code. To hand in your code, you first need to add the new files you created to the working copy of your repository by typing:

To commit your changes to the repository type:

```
git add -u
git commit -m "<your message>"
git push origin master
```

Guide: How to submit CS 225 work using git

Grading Information

You must submit your work to git for grading. We will use the following files for grading:

- `List.h`
- `List.hpp`
- `List-ListIterator.hpp`

All other files including any testing files you have added will not be used for grading.

Guide: How to submit CS 225 work using git

Good Luck!