List my Reasons

PROGRAM TWO

A test of templates and the first use of list classes and structures alongside sorting functions

John Grzegorczyk

Outline

This time around we want you focusing on building using template classes and structures, so that the code can be reused!

Description

Another layered program with data storage using lists, queues, and/or stacks to store information the user specifies. Students will be focused on learning how template classes work, and how to best utilize their function without trying to limit the functionality of the class. This will require them to also learn how to overload some operator functions for their classes so that they can utilize their template classes as a true template. Additionally, since these classes should all be templates (aside from the data-class), the students should be able to work on each part in isolation, integrating even better with the Git workflow we previously introduced.

Requirements

The program will again consist of at least four parts and (around) six files. Which will be subdivided as follows:

Parts

Driver

- Contains a full program flow.
  - All classes are used.
  - All necessary functions are accessible tested.
  - User has some kind of inputs.
- Has a natural way to end the program, other than crashing it.

Linked list class

- Private data:
  - Head (and tail) list node pointers (tail not required, but helps)
  - Size (optional)
- Public functions:
  - Add to the list (append, prepend, insert)
  - Get from the list (back, front, at)
  - Remove from the list (pop, remove, etc.)
  - You do not need to do all these functions
    - You need to have at least one of each type
    - Consider how that will change the class functionality
- Public “friend” functions:
  - Some sorting function (quicksort, merge sort)
    - Please don’t do bubble sort!
    - Try to use a type of partitioning sorting algorithm
    - If list node is not inside of linked list, this function does not have to be inside of linked list!
  - Stream `operator <<` (for printing data)
- Need at least 10 functions for this class!

Data class

If you decide to use the same data class as your previous assignment and update it with operator function (which we will allow), you MUST

come up with at least one additional class to use for your program, that class needs at least 5 functions and minimum 3 sub-data.

- Stores some kind of information
- Data is private but modifiable through function calls (getters/setters)
- Must have comparison and stream operator overloads:
    - `operator <<`
    - `operator <`
    - `operator >`
    - `operator ==`
    - `operator <=` (optional)
    - `operator >=` (optional)

Other class

You have two options for this class, a template helper class (or struct) to linked list (basically list node, but outside of the linked list class), or a smart pointer implementation.
If you have a group of 5, you MUST do both!

List Node

- Private data:
    - Stores data of the template type
    - Holds a pointer to another list node (or two for double linked)
- Public functions:
    - Can get the data at this location
    - Can iterate forward (and backwards if doubly-linked)
    - Can set data, next (and previous if doubly-linked)

WARNING! Be careful with your destructor! IF the destructor deletes attached nodes, you will need to detach the node when removing it from the list (but list clean-up is basically automatic). IF the destructor does NOT delete attached nodes, you can freely delete from the list (but the clean-up is manual).

Smart pointer

- Private data:
    - Pointer to the data of the template type
- Public functions:
    - Assignment of the pointer
        - This can be done via the assignment `operator =`
    - Use of the pointer
        - Can be done through a retrieve function.
        - Can by done via overloading operators:
            - `*` (dereference operator, returns template type)
            - `->` (arrow operator, returns the stored pointer for recursion)
            - `_Ty*` (casting operator, allows implicit casting to the pointer, should return the pointer inside)
    - Comparison operator
        - Checks for nullptr (returns true for both, false for 1)
        - Calls comparison operator of stored data (if neither is null)
    - Destructor should clean up the stored memory

WARNING! How you write your assignment and constructor determines whether this class breaks. If you can pass pointers directly to this class, there is a risk of multiple smart pointer's pointing at the same data, which in turn will cause segmentation faults whenever one goes out of scope. The recommended way to build this class is to copy data but instead create a new pointer for each smart pointer.

Files

- Driver.cpp
- Classes
  - LinkedList.h(pp)
  - [data class].cpp & [data class].h(pp)
  - SmartPointer.h(pp)
    - SmartPointer needs its own file
    - List Node can be in the same LinkedList file
  - NOTE: The classes' names and files should match
  - NOTE: The processor can use .h and .hpp files the same
  - NOTE: Template classes MUST be defined in the header file
- Makefile
  - NOTE: The Makefile is for your and our convenience
- TEST_CASE.txt
  - One extensive test case is required!
  - Should test:
    - Every possible path of the program
    - Edge cases of classes
      - Consider especially:
        - Empty list (to start)
        - Emptying the list (after adding)
        - Out of range calls (insert/remove)
        - Printing (especially for empty cases)
- Other files:
  - Save files (if you made save data or had a file to read data in for your test case)
  - Other cpp and h/hpp helper files (if you write extras)

It is also acceptable to just use the .h(pp) files without the .cpp files, HOWEVER Make will not be able to speed up the compile time if that is the case.

Extra Notes

You need to be able to sort the list ascending AND descending with whatever sorting function you use, this can be done in quite a few ways, the easiest is to write 2 different sort functions, the next is adding a Boolean value for sorting direction as a variable, for 5 bonus points, you can look into function pointers and pass a pointer to a comparison operation to allow you to sort in a unique way every time! (Meaning you can call the sort algorithm with a function that doesn't necessarily use either of the data class's < or > operators and instead maybe sorts based on a different datatype in the class) For example: Vehicle might compare based on license plate, but I also write a function that takes a vehicle and compares based on make and model.

Timeline

Week Four

If it is desired, the roles in the group can be switched around if anyone wants to try a different role, if you did not use the roles (other than group leader), feel free to switch the group leader and/or try out the other roles!

- Program outline
- Group decisions
  - Project ideation and selection
  - Program breakdown
  - Project name
- Initial commit
  - Project README.md
    - Contains a description of the program design
    - Outline of what the program will do
    - Classes that are needed
  - Class definitions (won't be too important this time with templates)
- Write-up (same idea as last time)
  - Needs to explain what the program idea is
  - What classes will be used
  - Functions that will be needed
  - Program flow
  - Individual role breakdown
  - This does not need to have everything
  - Just rough-draft first-thoughts
  - Must be submitted by the end of the night!

Week Five

(workday/check-in)

- Progress check
  - What classes are done
  - What functions need work
  - What functions need to be added for the program to run
- Writing Makefiles (we will go through this with you)
- First bug testing (if far enough along)
  - Tests individual classes (where possible)
    - For templates, we can test with basic datatypes (like int)
  - Write drivers in unique branches for testing
  - Note down and create issues for any bugs found
- Change any classes as needed
  - Functions that would help
  - Variables that make your life easier
- Building an initial test case for the whole program (if there is time)
  - Fix all minor class bugs first
  - Then try to merge and test the whole program

Week Six

(Fall break)
Labs will not be meeting, but you should still meet with your group or at least discuss and figure out what you need to test before week

seven submissions! This should follow the same idea as Week 3 when you were bug testing, writing issues on GitHub, fixing issues, iterating tests, and writing any last functionality you may need before final submissions. (Here are the Week Three notes for you to follow if you need)

- Intensive bug testing
  - Whole program is tested
  - What breaks
  - What doesn't behave as expected
  - What looks wrong
- Writing issues for bugs
  - Open GitHub and go to your repository
  - Go to "issues" and create new
  - Label it based on the bug you found
- Fixing (A good bit of this will also be done out of class)
  - After thorough testing
  - Open an issue and add yourself to it
  - Create a new branch off the dev branch
  - Attempt to figure out what causes the issue
  - Write a fix
  - Verify it is fixed
  - Create a merge request
  - Repeat
- Further testing iterations
  - After fixing every issue or when testing your fix
  - Look for other potential bugs/issues
  - Note them down and repeat this cycle
- New functionality
  - If you have no bugs or very few
  - Add new functions and feature
  - Create new branches to test ideas
  - Try to add a couple new pointer types
  - Extra work won't be merged if you do not have time to finish

Week Seven

Same exact thing as last week!
(final testing and program due at midnight with a write-up)

- Progress check/bug testing
  - Is everything done?
  - Is it all functional?
- Cleaning code
  - Messy code is tidied and simplified
  - Broken code is fixed/removed
  - Program is finalized and merged/rebased into development
- Group leader writes a finalized program write-up
  - Contains all information from the first write-up (polished)
  - What did your group try to do?
    - What was successful?
    - What was unsuccessful?
  - What was your process?
  - What would you do differently?
  - What did you learn?
  - What is the expected outcome of the program?
  - How will the TA run/test your program

    - Additional information for us:
        - Thoughts
        - Comments
        - Concerns
        - Questions
        - Everyone should write these down when you think them
        - Put them in a central location for your Leader
        - Your leader will compile all of them
            - Take out duplicates
            - Organize and order
            - Anonymize names/information
    - It doesn't need to be perfect, but please make it neat.
- Make sure you have a test file for your TA!

Examples

County List

Previous years required program; we would use a template linked list to store a series of counties in the area.

Classes

This example only used 2 classes in prior years, but we can also consider how it might be good to use a smart pointer where in previous years it was not utilized (and a TON of memory leaks occurred because of it).

County

Stores data on the index, name, state, and population of a county.

List

The linked list that is doubly-linked and has functions for selection and merge sort.

Smart Pointer

(Added to the previous years program to help with memory leak issues). This is just a pointer wrapper that holds the memory for us and deletes itself when the time comes.

Structure

County List

Stores the data for a set of counties `List< SmartPointer< County > >` this was `List< County* >`, but that is way messier to clean up.

Flow

Program reads in a dataset from a file given by the user and inputs the counties into the list, then calls a sort on the data and prints the sorted data.

Parking Lot Manager

I am lazy, and reformatted my parking lot because I didn't have a Lot (ba-dum-tiss) of time to work on things.

Classes

Vehicle

Same exact class from before, but now it has comparison operators!

John Grzegorczyk

Program Two

Linked List

Template class that stores data in a series of nodes.

List Node

Template class used by linked list that stores a datatype and relative nodes.

Storage

Brought as an idea from P1 that is a template storage array.

Structure

Parking Lot

Now is a bonus template class called storage `Storage< Vehicle >`

Parking Garage

Is a linked list of Parking Lots, `LinkedList< Storage< Vehicle > >`

Parking Lot Manager

Stores two linked Lists for Parking Lot and Parking Garage that track all the lots and garages you own!

Flow

Like our P1 example, user is able to purchase/build lots, destroy lots, edit lots, and visit other player's lots. Still working on multithreading so the gameplay loop can have NPCs to visit and leave the lot.

Rubric

Driver

10 points

Classes

50 points

Linked List Class

20 points

Functions

18 points

Definitions

(1 point for each of the required 10, declarations do not count)
10 points

Functionality

(Every broken function is -1, up to -8 points)
8 points

Members

(properly uses a pointer to the first [and last] node[s], and uses list nodes)
2

Data Class

(If you added to your old class, that is at most worth 10 points, then the remaining 10 would be attributed to whatever extra class you did)
20 points

Functions

18 points

Definitions

(declarations do not count)
10 points

Getters/Setters

5 points

John Grzegorczyk
Program Two

Comparison Operators

5 points

Functionality

(Every broken function is -1, up to -5 points, you are unlikely to break these)
5 points

Members

(Contains at least 3 sub-data)
5 points

Extra Class

(If you need to do both, they are both worth 5 points, as they are rather small classes)
10 points

Functions

8 points

Definitions

(Has at least 5 functions, since both list node and smart pointer will need at least 5 functions to be usable)
5 points

Functionality

(Every broken function is -1, up to -3 points)
3 points

Members

(properly uses a pointer to the first [and last] node[s], and uses list nodes)
2

Sorting Function

(This could be a part of the list class)
5 points

Ascending Sort

2 points

Descending Sort

2 points

Works

(At least one of the sorting algorithms works)
1 point

Memory Management

(Same as last program, points are deducted for the following reasons)
20 points

Leaked Memory

Up to 10 points lost

Overwriting Pointers

-5 points

Failure to Clean

-5 points

Segmentation Faults

Up to 10 points lost

Out of Bounds Calls

-6 points

Uninitialized Pointer Calls

-2 points

Null Pointer Calls

-2 points

Other

15 points

Included All Files

1 point

Test Case

3 points

John Grzegorczyk
Program Two

Makefile

1 point

Formal Write-up

5 points

Good comments

5 points

Class Comments

(Above each class should have information on what it does)
1 point

Function Comments

(Each function should have information on it's purpose and what it does)
3 points

Inline Comments

(Various comments in your code explaining convoluted code)
1 point