CSE333 Exercise 9

Out: Wednesday, February 16
Due: Monday, February 21 by **11 am PST**
Rating: 3 (note)

Goals

- Define a class that inherits from and implements an abstract class
- Evaluate a situation to figure out what smart pointers to use
- Utilize smart pointers to manage complex memory structures

Problem Description

In this exercise, you will create a **simplified C++ linked list class which can only store integers**. Note the major differences between this and the C-style linked list module from Homework 1:

- `LinkedList` was a public-facing `typedef` -ed struct with the implementation hidden in `LinkedList_priv.h`. Now it is a class with private data members for `num_elements` , `head` , and `tail` .
- `LinkedListNode` was defined in `LinkedList_priv.h` but is now a private data member of the class.
- `LinkedList` functions took pointers to `LinkedList` structs as parameters but now these functions are public class members.
- `LinkedList_Allocate` is handled by the class constructor(s) and `LinkedList_Free` is handled by the class destructor.
- `LinkedList` memory was manually managed by you as the programmer but now will be managed via smart pointers.

Provided Files

We have provided you with the following **five** source files, which can be downloaded from this web directory (./ex9_files) or with the commands:

```
bash$ wget https://courses.cs.washington.edu/courses/cse333/22wi/exercises/ex9_files/<filename
```

- **`IntList.h`** — Defines an abstract class `IntList` , which can store integers. Integers can be added to or removed from either the front or the end of the list. Users can also get the size of the list. Note that since this is

an abstract class, you cannot construct an instance of `IntList` (*i.e.*, `IntList list();` would not compile).

- **`test_list.cc`** — Tests your implementation of the `LinkedIntList` files described below. Note that these tests are not exhaustive, and you may want to more thoroughly check your solution.
- **`Makefile`** — Provided for your convenience in compiling the executable.
- **`LinkedIntList.h`** — Contains the class definition of `LinkedIntList` that you will need to complete.
- **`LinkedIntList.cc`** — Contains member function definitions of `LinkedIntList` that you will need to complete.

Note: You will only submit `LinkedIntList.h` and `LinkedIntList.cc`.

Requirements

- `LinkedIntList` should derive from `IntList`, and have implementations of all the pure virtual functions declared in `IntList`.
- `LinkedIntList` should have an empty destructor since smart pointers should be used to clean everything up.
- `LinkedIntList` should have a default (0-argument) constructor.
- You should explicitly disable (`= delete`) the copy constructor and assignment operator for `LinkedIntList`.

We ***highly recommend*** that you start by implementing `LinkedIntList` using normal ("raw") pointers, *i.e.*, no smart pointers. Your code should still pass the provided test file, but will leak memory. Afterwards, you should edit your implementation to use smart pointers and make sure no memory errors are generated.

Implementation Notes

Linked List

When implementing a linked list, consider the following cases:

- **Insertion:** (1) empty list, (2) one or more elements.
- **Deletion:** (1) empty list, (2) one element, (3) more than one element.

Smart Pointers

This exercise makes use of smart pointers to manage memory; as a result, you are forbidden from using `delete` in `LinkedIntList` (other than for disabling the cctor and op=). Be sure to use the most appropriate smart pointer type(s). Refer to lecture slides and C++ documentation (*e.g.*, the `memory` header (http://www.cplusplus.com/reference/memory/)) for details on how to use smart pointers and their methods.

Abstract Classes

We have provided the abstract class `IntList`. An abstract class is one that has at least one pure virtual function. Your `LinkedIntList` should inherit from `IntList`, but should not be an abstract class itself. To avoid this, you should make sure that `LinkedIntList` has no pure virtual functions.

Style Focus

Don't forget that good practices from previous exercises still apply!

Classes

This is yet another exercise with C++ classes so you should be sure to follow the best standards with commenting and organizing your classes as laid out in the lecture and the ex6 and ex7 sample solutions.

Const

Functions, parameters, and variables should be labeled with `const`, where appropriate, throughout your program.

Documentation

Make sure that you have descriptive comments with function and class declarations.

Submission

You will submit: `LinkedIntList.h` and `LinkedIntList.cc`.

Your code must:

- Compile without errors or warnings on CSE Linux machines (lab workstations, `attu`, or CSE home VM).
- Have no runtime errors, memory leaks, or memory errors (`g++` and `valgrind`).
- Be contained in the files listed above and compile with the provided Makefile.
- Have a comment at the top of your `.cc` and `.h` files with your name(s) and CSE or UW email address(es).
- Be pretty: the formatting, modularization, variable and function names, commenting, and so on should be consistent with class style guidelines. Additionally, the linter shouldn't have any complaints about your code (`cpplint.py`).
- Be robust: your code should deal with hard-to-handle/edge cases and bogus user input (if there are any) gracefully.

Submit your code on Gradescope (https://www.gradescope.com). Don't forget to add your partner if you have one.