Homework 5: Parity I

CSCI 60

Due 11:59pm Wednesday, October 30, 2024

Krehbiel

Overview. This is the first of two assignments that will have you implement a class called `Parity`, which uses dynamic memory management to keep track of two arrays, one of even ints and one of odd ints. A small driver program is provided for you in `main.cpp`. The interface for the class is given in `parity.h`. Declarations for an overloaded copy constructor, assignment operator, and destructor are commented out; that's for next week, but it's also good preparation for your midterm, so Homework 6 will be made available to you early! This week you will implement the other declared functions in the Homework 5 starter implementation file $\boxed{\texttt{parity.cpp}}$, which currently contains only a stub function for an overloaded insertion operator. This implementation file is the only one you will upload.

This handout details how to implement each of the functions declared in the Homework 5 interface file, organized into milestones with accompanying suggestions for how to use the driver program as you develop. Do not alter the boilerplate code at the top of each file or use functions defined in libraries not already included. Look over the submission checklist from HW1 for usual submission instructions and style guidelines, and read announcements about common technical problems encountered throughout the week.

Milestone 0: Run a stub program. This involves three quick steps that should take under ten minutes if you are solid on syntax for defining member functions:

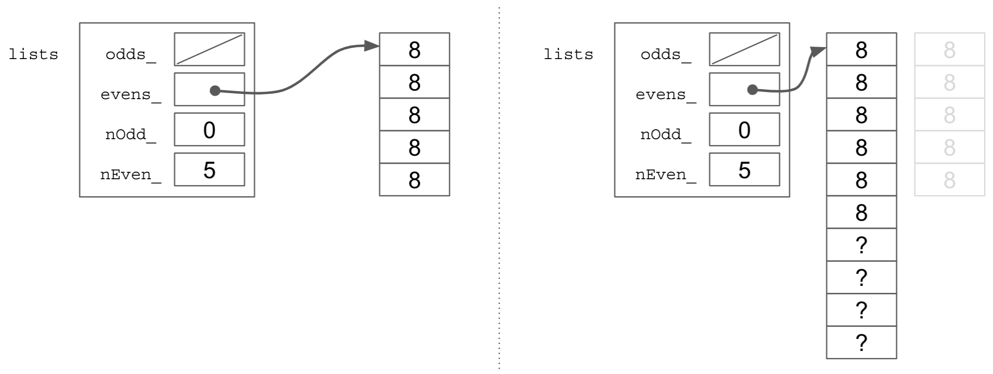
1. Write stub functions for `insert`, `odd`, and the 2-argument constructor in the implementation file. Stubs for `min`, `max`, and the overloaded insertion operator << are already there for you.
2. Compile (`$ g++ main.cpp parity.cpp -std=c++11`) and run to see the stub output.
3. Upload to gradescope to confirm that the autograder runs (but doesn't pass the tests).

Milestone 1: Implement and test the insert member function. This is the main task for the week, and you'll want to do a lot of memory diagrams and code tracing as well as running the driver program as you develop. As the only mutator function, `insert` is responsible for maintaining the invariants outlined under the private member variable declarations as it changes them.

Specifically, it should check whether its parameter `val` is even or odd and add it to the end of the corresponding dynamic array associated with the object. However, if that array is at its current capacity, `insert` should create a new array with an additional `CAP` elements before inserting the new value, consistent with the invariant written in comments in the class definition in `parity.h`. This will involve similar steps to those used to implement `insert` for our dynamic array class `GArray` from lecture this week:

1. Allocate a new larger dynamic array and store its address in a local pointer variable,
2. copy the contents of the old array into the new one,
3. release memory associated with the old array,
4. reassign the pointer member variable so that it points to the new array, and
5. add the new value and update the appropriate size member variable.

The diagram on the next sheet provides before and after memory diagrams illustrating the effect of a sixth call to `lists.insert(8)`, assuming `CAP=5`.



There are a few important things to note about managing array size and capacity for this assignment:

1. The array capacity will always be a *multiple* of `CAP` and not necessarily `CAP` itself.
2. `CAP` is a *private* static constant because this is an implementation detail that the class clients don't need to know about. Your implementation code should refer to `CAP` and not its currently initialized value of 5 – never hardcode values when a program is designed to be more general!
3. Finally, note that `Parity` member variables only keep track of the number of entries *used* in each array, not how much space is allocated for it. The problem-solving part of this assignment involves thinking about how to use the `%` operator to determine whether all cells in the array are used.

You can test your `insert` function as you write it using `cout` statements; just be sure to remove them before you submit or this will break the autograder.

Milestone 2: Implement and test your insertion operator overload. Overload << so that, for example, after inserting a sequence of numbers `3 5 -1 2 6 8 10 7 -2 11 15` into a `Parity` object called `lists`, the command `cout << lists;` in `main.cpp` would output the following:

```
E: 2 6 8 10 -2
O: 3 5 -1 7 11 15
```

Note that the `Parity` class declares << as a `friend`, so you have access to all the private member variables associated with the `Parity` object whose contents are being inserted into the `ostream` object. As always, make sure you pipe your outputs to the `ostream` object provided as a parameter and *not* `cout`, which will look right on your computer but it will break the autograder.

Milestone 3: Implement and test your 2-argument constructor. The 2-argument constructor should take in an array and its size, and it should initialize a `Parity` object consistent with one that resulted from a sequence of calls to `insert`, passing in the values inside the array in order. Note that you may use your `insert` function to do this so you don't have to rethink all the same logic! This is less efficient than determining how much space to allocate for all the contents at once, but you won't lose points for instead using the function you've already written. You can test your constructor with a 3-line main function that:

1. initializes an array (on the stack), say, {`3,5,-1,2,6,8,10,7,-2,11,15`},
2. constructs a local `Parity` object using that array and its size, and
3. uses >> to output the object to the console to get a display like in Milestone 2.

Milestone 4: Implement accessor functions. Finish implementing `min`, `max`, and `odd`. The `min` and `max` functions should scan through the arrays keeping track of the smallest and largest values, respectively, across both arrays. You should use the predefined constants `INT_MAX` and `INT_MIN` from the stub functions to initialize these local variables so you have something to return if both arrays are empty.

The boolean function `odd` should return `true` if the sum of all values in both arrays is odd and `false` if the sum is even. Note however that summing up all these values is not the most efficient way to do this. Can you write a 1-line function body that returns the correct boolean with a single expression that doesn't examine the contents of either array?

Finishing touches: Write test code that makes sure your functions work with empty arrays and different values for `CAP`. Make sure your implementation file doesn't have any remaining `cout` statements used for debugging, remove any commented out code, and generally review the style guidelines from HW1, including the rule of thumb to have a brief comment describing what each function does and how. It is also helpful to put your implementations in the order they are declared in the class definition. When you are satisfied, upload only your complete $\boxed{\texttt{parity.cpp}}$ to Gradescope, without changing the filename.