Homework 6: Parity II

CSCI 60

Due 11:59pm Wednesday, November 5, 2025

Krehbiel

Overview. This program will have you build on your `Parity` class from Homework 5, adding functions enabling the dynamic memory management discussed in Week 5 of class. You should also replace your original insert definition (and possibly constructor) consistent with a new requirement to keep entries in sorted order, and you should add a remove function to efficiently remove all of a particular value at once.

The updated `parity.h` file reflects the new requirements; there are no starter driver or implementation files for you this week. Instead, make a copy of your `parity.cpp` from last week as a starting point for this week, and create and your own `main.cpp` to test your code as you go. You will only upload your completed $\boxed{\texttt{parity.cpp}}$.

Milestone 1: Prepare, run, and diagram a stub program. This step could be very fast or take a substantial amount of time as you get up to speed on our Week 5-style diagrams for dynamic memory and the role of assignment, copy-constructors, and destructors.

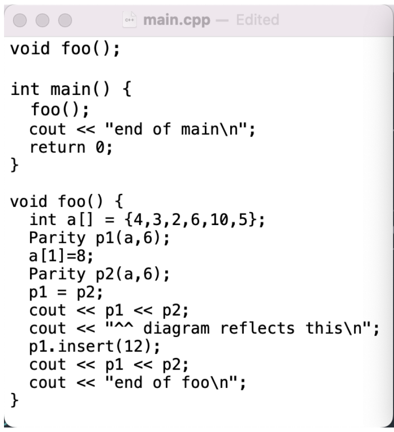
1. Make sure your code for Homework 5 *compiles*. You will not be re-graded on `min`, `max`, or `odd`, so it is okay if your implementations have logic errors or are stubs, as long as they compile. Write a stub for the new `remove` function.

2. Write compilable stub functions for the three essential functions for dynamic memory management: a copy-constructor, a destructor, and an overloaded assignment operator. These are declared in the updated header file. As practice, write your stub functions to emulate the copy-constructor, destructor, and assignment that would be automatically generated for you if you did not declare them in your class: the default copy-constructor and assignment operator make shallow-copies of the object passed in and the default destructor does nothing.

3. Upload your code to the autograder to make sure it compiles.

4. *Temporarily* add a single `cout` statement to each of your new memory management functions so you can see when they are called. Write a driver program to test *one at a time* that they automatically get called when you expect them to. You should remove these `cout` statements before you submit.

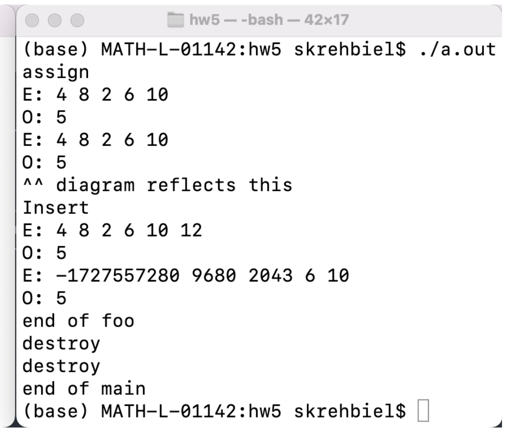
You may have weird output or memory-related runtime errors when you start since your memory management functions aren't implemented yet. Draw diagrams to make sense of when and why these errors occur. An example main program with associated output and memory diagram is below; note that I skipped Step 4 above when I ran this. Use the driver program from class Week 5 as further inspiration.



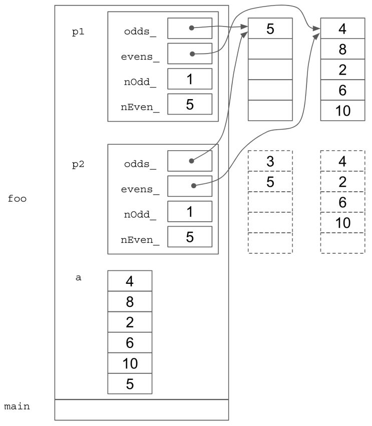
```
void foo();

int main() {
  foo();
  cout << "end of main\n";
  return 0;
}

void foo() {
  int a[] = {4,3,2,6,10,5};
  Parity p1(a,6);
  a[1]=8;
  Parity p2(a,6);
  p1 = p2;
  cout << p1 << p2;
  cout << "^^ diagram reflects this\n";
  p1.insert(12);
  cout << p1 << p2;
  cout << "end of foo\n";
}
```



```
(base) MATH-L-01142:hw5 skrehbiel$ ./a.out
assign
E: 4 8 2 6 10
O: 5
E: 4 8 2 6 10
O: 5
^^ diagram reflects this
Insert
E: 4 8 2 6 10 12
O: 5
E: -1727557280 9680 2043 6 10
O: 5
end of foo
destroy
destroy
end of main
(base) MATH-L-01142:hw5 skrehbiel$
```



Note: These diagrams reflect stub functions built on top of last week's implementation; after you complete Milestones 3 and 4, the order of contents will be different.

Milestone 2: Implement your memory management functions. Once you are confident in your diagrams, actually implementing your memory management functions should be straightforward. Some things to consider:

- The assignment operator is overloaded as a member function and should make a deep copy of the contents of the righthand argument into newly allocated space for the calling object so that subsequent mutations to either object only affects that object. Make sure you free rather than orphan the existing contents of the calling object. The autograder can't tell when you orphan memory, but you will lose credit for your source code if you do.
- The copy-constructor is called explicitly when a new object is declared with an existing object passed in as a single parameter, and it's called automatically whenever an object is passed into or returned from a function by value. Like the assignment operator, member variables for the object being declared should be given their own memory and a deep copy of the contents of the existing object.
- The destructor is called automatically whenever an object passes out of scope. Its main purpose is to avoid orphaning memory only accessible by its member variables. Note that the destructor can also be called explicitly at any time, so make sure that it results in a sensible object *without* any associated dynamic memory that can be used normally after destruction.

As usual, you will lose credit for submitting code that does not compile or code that compiles but has runtime errors, including memory errors. For this assignment, you may also lose credit for code that orphans memory, even if its output is correct. It is your job in Milestones 1 and 2 to diagram your code effectively to ensure that orphaning memory is not an invisible side effect of your implementation!

Milestone 3: Revise your old insert (and possibly constructor) definition. Your new implementations should be consistent with the new invariant that the elements be stored in sorted, non-decreasing order, rather than in the order they were provided. Do not use any C++ sorting libraries.

Your new insert function should only require a single loop, making the algorithm *linear* time, to find the right spot for the entry and shift the contents down. Note that this process should be done *in place* if the currently allocated memory has excess capacity or it should include deep copying contents to a *new array* if more space is required. In the latter case, you may handle this with two sequential (not nested) loops and still maintain linear runtime.

If your constructor from last week simply called insert, you might not need to modify your constructor at all after redefining insert. Otherwise, you could write a more efficient constructor that uses this sorting logic directly to avoid unnecessary allocation and deallocation steps.

Milestone 4: Implement a remove function. Add some new functionality to your class by implementing a function to remove every occurrence of a particular value, returning the number of entries removed. For full credit, this should also be done in *linear* time, meaning you should have a single loop that runs through the array one time to count all entries of a particular value and shift down the subsequent values. Like every member function in your class, the remove function should maintain the storage invariant; this means that remove may result in downsizing the array containing the value to be removed.

Finishing touches: As always, before you submit make sure to remove code from debugging, and generally review the style guidelines from HW1. Some of your member functions may be longer than is usually good practice, and that's okay for this assignment as long as you use good comments to orient the reader.