Homework

26

A Flex Array

The heap allows you to create dynamic arrays, and wait until runtime to decide how many elements are needed, unlike the **static arrays** which are built into the C++ language. In fact, the heap is what the `vector` and `string` classes use to allow them to **expand** as the user adds new elements to the collection.

You'll find the definition for the `FlexArray` structure, as well as the prototypes for the functions you are going to write in the header file. Do not make any changes to the header file at all. You are going to write two functions. Here are the descriptions:

1. The *readData* Function

The ***readData()*** function reads integers from an input stream (such as `cin`) until the user terminates by either running out of data or by entering an invalid input such as `Q`.

- The function sets the `size_` data member to the number of numeric inputs.
- The member `data_` is a standard `unique_ptr` to a heap allocated array.
- At the end of the function, the array should have exactly `size_` elements.
- The function returns a reference to the modified `FlexArray` object.

At the outset, you won't know how many elements the user will enter. So, start with a capacity of `INITIAL_CAPACITY`. Do not change this from its current value of `2`.

Managing Memory in readData

In ***readData()***, whenever your allocated array fills up, create a new array of double the current capacity, copy the original elements to the new storage, free the original array and assign the new array to the member `data_`.

At the end of your function, you'll follow a similar pattern to **shrink the allocated memory** so that it is exactly the same size as the number of elements.

Be sure to delete any intermediate arrays. If you have any memory leaks displayed, fix them before submitting your assignment.

Remember: **`unique_ptr`** does not support ordinary copy or assignment, but you can create a reference to a **`unique_ptr`** and, you can transfer ownership from one **`unique_ptr`** to another by calling **`release()`** and/or **`reset()`**:

- **`release()`** returns the "raw" pointer stored in the **`unique_ptr`** and makes that **`unique_ptr`** null.
- **`reset()`** takes an optional raw pointer and repositions the **`unique_ptr`** to point to the given pointer. If the **`unique_ptr`** is not null, then the object to which the **`unique_ptr`** had pointed is deleted.

Calling **`release()`** breaks the connection between a **`unique_ptr`** and the object it manages. The pointer returned by **`release()`** is often used to initialize another smart pointer; responsibility for memory is transferred from one smart pointer to another.

2. The *`toString`* Function

For ***`toString()`***, the result should be:

- Delimited with braces **`"{"`** and **`"}"`**.
- Individual elements should be separated with a comma and a space.
- There should be no space before the first element or after the last.

You'll recognize this as the fencepost algorithm. You can convert the integer elements in the array by using the **`to_string()`** function in the **`<string>`** header.

When you add that and **`make test`**, all of the tests should pass.

Be sure to **`make submit`** to turn in your code for credit before the deadline. As always, if you run into problems, bring your questions to Piazza or come to my office hour.