Lab 5: Dynamic Array Wrapper

Information

Topics: Dynamic arrays, basic data structure functionality, unit tests

Turn in: All source files (.cpp and .hpp).

Starter files: Download on GitHub or D2L.

```
Lab 05 - Dynamic Array Wrapper/
├── lab5_main.cpp ...  Contains main()
├── lab5_SmartDynamicArray.hpp ...  Class declaration
├── lab5_SmartDynamicArray.cpp ...  Class function
│                                   definitions
├── lab5_Tester.hpp ...  Unit test functions
├── cuTEST/ ...  Unit test framework
│   ├── Menu.hpp
│   ├── StringUtil.hpp
│   ├── TesterBase.hpp
│   └── TesterBase.cpp
└── CodeBlocks Project/
    └── Lab 05 - Dynamic Array Wrapper.cbp ...  Code::Blocks
                                                project
                                                file
```

Your project needs to be compiling as a C++11 project.

Getting started

This lab will be an extension of the Smart Static Array lab, with some changes...:

- There is no longer a MAX_SIZE; the array can be resized.
- Our `m_data` member is now a pointer, to be used to allocate memory for a dynamic array.
- We have to keep track of `m_itemCount` as well as `m_arraySize` as two integer variables now, as the array can resize.
- The class needs a destructor to ensure we free memory before it is destroyed.

A lot of the functionality will be the same, or similar to, the SmartStaticArray, and this lab will walk through the updates to make it work as a Dynamic Array wrapper.

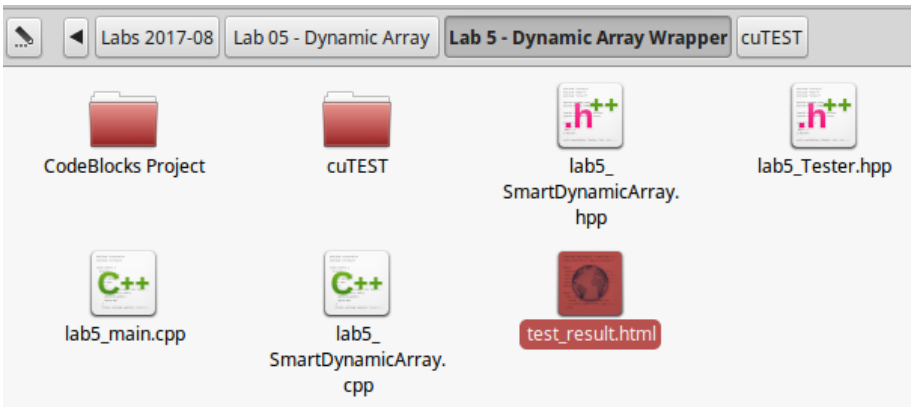
cuTEST updates

cuTEST has been updated so that it is not run as a console program anymore. Instead, it will run and immediately close after it is done, with the results of the unit tests written out to an html file.

The unit test output is color coded, with failed tests highlighted in red, and passed tests highlighted in green. The page also contains descriptions for what the test tests, as well as expected and actual output values.

Test set	Test	Prerequisite functions Functions that need to be implemented for these tests to work right	Pass/fail	Expected output The output expected from the function's return
Test_Initialize	Check to make sure new SmartDynamicArrays initialize its m_data pointer to nullptr	SmartDynamicArray constructor	failed	m_data = 0
Test_Initialize	Check to make sure new SmartDynamicArrays initialize its m_itemCount value to 0	SmartDynamicArray constructor	failed	m_itemCount = 0
Test_Initialize	Check to make sure new SmartDynamicArrays initialize its m_arraySize value to 0	SmartDynamicArray constructor	passed	m_arraySize = 0
SUMMARY FOR Test_Initialize: 1 out of 3 tests passed				

Make sure to check your project folder for the `test_result.html` file. It may not be in the same location as mine, depending on what IDE you're using. Use your Operating System's SEARCH tool if you can't find it.



The SmartDynamicArray class declaration

There have been some changes to the class, so let's highlight some:

```
1  class SmartDynamicArray
2  {
3      public:
4      SmartDynamicArray();
5      ~SmartDynamicArray();
6
7      void Push( const string& newItem );
8      void Insert( int index, const string& newItem );
9      void Extend( const SmartDynamicArray& other );
10     void Pop();
11     void Remove( int index );
12     string Get( int index ) const;
13     void Resize();
14     void Resize( int newSize );
15
16     int Size() const;
17     bool IsFull() const;
18     bool IsEmpty() const;
19
20     string operator[]( int index );
21     SmartDynamicArray& operator=( const
       SmartDynamicArray& other );
22     bool operator==( const SmartDynamicArray& other );
23     bool operator!=( const SmartDynamicArray& other );
24
25     private:
26     void ShiftRight( int index );
27     void ShiftLeft( int index );
28     bool IsInvalidIndex( int index ) const;
29     bool IsNonContiguousIndex( int index ) const;
30     void AllocateMemory();
31     void AllocateMemory( int newSize );
32     void DeallocateMemory();
33
34     string* m_data;
35     int m_itemCount;
36     int m_arraySize;
37
38     friend class Tester;
39 };
```

Now for the private member variables, we have...

```
string* m_data
int m_itemCount
int m_arraySize
```

We will be using pointers to allocate memory for a dynamic array, and resize it as-needed. This means we also need to take care to protect against memory errors.

> **Need to review?**
> If you're not feeling too confident with pointers, dynamic arrays, and memory management, watch the video lectures for CS 200 here:
> http://edu.moosader.com/course/cs200/viewbyassignment.php

Additionally, some of the functions added to the class are just additional helpers (to reduce duplicate code), or functions needed for working with memory management.

What could possibly go wrong?

Some memory errors you might encounter are...

Invalid Memory Address - usually occurs when accessing unallocated memory, or memory that has already been freed.

Example 1:

```
1 // not initialized to an address;
2 // pointing to garbage.
3 char* uninit;
4
5 // trying to de-reference the pointer.
6 cout << *uninit;
```

Example 2:

```
1 int* listOfNumbers = new int[5]; // allocating memory
2 // ...(Stuff happens)...
3 delete [] listOfNumbers; // freeing the memory
4
5 listOfNumbers[3] = 300; // trying to access
```

Memory Leaks - occurs when memory is allocated but never freed.

Example:

```
int main()
{
    int classSize;
    cin >> classSize;

    // Allocating memory
    string* students = new string[ classSize ];

    // No delete here!

    return 0;
}
```

Missing Allocation - occurs when you try to free memory that has already been freed.

Example:

```
int main()
{
    int * myArray = new int[50];
    // (...)
    delete [] myArray;

    // later...

    delete [] myArray; // trying to free it again

    return 0;
}
```

Counts will be pointed off for memory errors in ALL programs in this class, so make sure to be responsible with your memory management! :)

Implementing the functions

Always build!
The unit tests provided for you will help make sure your program functions correctly and doesn't have logic errors, but to be able to *use* those unit tests, your program has to actually build and run!

Don't try to just implement all the functions without testing in-between, make sure you get one function working at a time!

Also note that some tests have **prerequisites**, which are listed on the test output. These are functions that the test relies on to function properly.

New functions

SmartDynamicArray()

Input parameters: None
Return value: None

In this constructor, you sohuld initialize `m_itemCount` and `m_arraySize` to 0, and you should initialize the `m_data` array to point to `nullptr`.

~SmartDynamicArray()

Input parameters: None
Return value: None

In the destructor, call the `DeallocateMemory` function.

---

void AllocateMemory()

This is an overloaded function, so there are two versions.

Input parameters: None
Return value: None

Call the other version of `AllocateMemory`, passing in a default size value of 10.

---

void AllocateMemory(int newSize)

This is an overloaded function, so there are two versions.

Input parameters: newSize, an int
Return value: None

Use the `m_data` pointer to create a dynamic array, if the pointer is not already pointing to some memory address.

- If `m_data` is pointing to `nullptr`...
  - Assign `m_arraySize` to the value passed in as `newSize`
  - Initialize `m_itemCount` to 0
  - Allocate space for a new array of size *m_arraySize* via the `m_data` pointer, and using the **new** command.
- Otherwise...
  - Throw a **logic_error** with a message that memory cannot be allocated because `m_data` is already pointing somewhere.

---

void DeallocateMemory()

Input parameters: None
Return value: None

If `m_data` is *not* pointing to `nullptr`, then free the memory with the **delete** command, and reset the `m_data` pointer to point to `nullptr`.

bool IsInvalidIndex(int index)

Input parameters: some index, an integer
Return value: None
Specifier: This function won't throw an exception. Mark it as `noexcept`.

Any index that is 0 or above is potentially a valid index, so this function only checks to see if the index passed in is less than 0. If the index is < 0, then return true - it is invalid. Otherwise, return false.

---

bool IsNonContiguousIndex(int index)

Input parameters: some index, an integer
Return value: None
Specifier: This function won't throw an exception. Mark it as `noexcept`.

An index is non-contiguous if adding an item at that index would cause the array to have a gap in it. For example:

0	1	2	3
A	B		C

In this case, if we allow "C" to be inserted at position 3, then we will have a gap in our elements. This Dynamic Array wrapper should enforce the design rule that all items are next to each other.

So, if the index is greater than the current `m_itemCount`, then it should return true - it is non-contiguous. Otherwise, return false.

---

void Resize()

This is an overloaded function, so there are two versions.

Input parameters: None
Return value: None

Call the other version of `Resize`, passing in a default value of the current `m_arraySize` plus 10.

void Resize(int newSize)

This is an overloaded function, so there are two versions.

Input parameters: newSize, an integer
Return value: None

Follow these steps to resize the dynamic array.

1. First, check to see if `m_data` is pointing to `nullptr`. If so:
   (a) Call (AllocateMemory) with the `newSize` that was passed in.
   (b) Call `return` - we don't need to continue executing this function.
2. Create a new dynamic array. Declare a local string-pointer variable and use the **new** command to create a new string array, whose size is `newSize`.
3. Make for loop that iterates from 0 to `m_arraySize` to copy items over from `m_data` to your new array that you declared in step 2.
4. Free the old memory by calling **delete** [ ] on the `m_data` pointer.
5. Update the `m_data` pointer, and have it point to the same address as your new array from step 2.
6. Update `m_arraySize`, set it to the value of `newSize`.

Confused?
I illustrate these steps in my Dynamic Arrays lecture from CS 200:
http://edu.moosader.com/course/cs200/viewbyassignment.php

Updated functions

void Push(const string& newItem)

Input parameters: const string& newItem
Return value: None

Error checking: In this version of the array wrapper, we won't need to throw any exceptions from within Push. We still need to check for a couple of errors, and resolve them *before* adding a new item...

1. If `m_data` is currently pointing to `nullptr`, then call `AllocateMemory()` before continuing.
2. If the `IsFull()` function returns true, then call `Resize()` before continuing.
   Specifier: This function won't throw an exception. Mark it as `noexcept`.

In both these casees, once resolved we can continue adding our new item as usual, so make sure your actual code to add the item to the array *is not in an else statement*! We will add the new item no matter what - it's just that we needed some prep ahead of time.

Store the new value: As you did with the SmartStaticArray, add the new item at position `m_itemCount` in the array, and make sure to increment `m_itemCount` by 1.

---

bool IsFull()

Input parameters: None
Return value: bool, true if the array is full, or false if it is not.

This function will now return true if `m_itemCount` and `m_arraySize` are equal values, or false if not. We no longer work with a MAX_SIZE variable.

void Insert(int index, const string& newItem)

Input parameters: int index, const string& newItem
Return value: None

This function will work *mostly* the same as the original, but with one exception: If `IsFull()` returns true, then you call `Resize()` before continuing. This also means that you will remove the error check for `index >= MAX_SIZE` since there is no longer a MAX_SIZE.

---

void Extend(const SmartStaticArray& other)

Input parameters: const SmartStaticArray& other
Return value: None

This will work very similarly to the SmartStaticArray version, EXCEPT that you won't be throwing any exceptions now! If the size of the current array + new array is more than the current `m_arraySize`, simply call `Resize(...)` before the extend functionality happens.

In other words...

```
if ( m_itemCount + other.m_itemCount >= m_arraySize )
{
    Resize( m_itemCount + other.m_itemCount );
}
```

Same functionality as with SmartStaticArray

---

void Pop()

Input parameters: None
Return value: None
Specifier: This function won't throw an exception. Mark it as `noexcept`.

Simply do a "lazy-delete", and just decrement `m_itemCount` by 1, if it is above 0.

---

bool IsEmpty()

Input parameters: None
Return value: bool, true if the array is empty, or false if it is not.
Specifier: This function won't throw an exception. Mark it as `noexcept`.

Just return true if the `m_itemCount` is set to 0. Otherwise, return false.

---

void ShiftRight(int index)

Input parameters: int index, the location to begin pushing items forward
Return value: None
Specifier: This function won't throw an exception. Mark it as `noexcept`.

Every element at the given index and after it should be shifted right by one space.

void ShiftLeft(int index)

Input parameters: int index, the location to begin pulling items backwards
Return value: None
Specifier: This function won't throw an exception. Mark it as `noexcept`.

Every element at the given index and after it should be shifted left by one space.

---

int Size()

Input parameters: None
Return value: int, the amount of items stored in the array.
Specifier: This function won't throw an exception. Mark it as `noexcept`.

This function will only return the current value of the private member variable, `m_itemCount`.

---

string Get(int index)

Input parameters: int index
Return value: string, the value from the array

If the index is invalid (less than 0, or greater than or equal to the `m_itemCount`), then throw an **out_of_range** exception.

Otherwise, return the element from `m_data` at the `index` passed in.

void Remove(int index)

Input parameters: None
Return value: None

Error checking: Check to see if the index is invalid (less than 0 or greater than or equal to `m_itemCount`). If it is invalid, then **throw** an `out_of_range` exception with the message, "Cannot insert at index - out of range".

Functionality: If there is no exception thrown, then call the `ShiftLeft` function, passing in the index. Again, we are *lazy deleting* the data by simply overwriting it with this function call.

Afterwards, make sure to decrement `m_itemCount`.

Grading breakdown

Function	**Point value**
SmartDynamicArray constructor	3
SmartDynamicArray destructor	3
AllocateMemory functions	4
DeallocateMemory	3
IsInvalidIndex	1
IsNonContiguousIndex	1
Resize functions	8
Push	2
IsFull	1
Insert	2
Extend	2
Total	30