Lab 3: Static Array Wrapper

Information

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

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

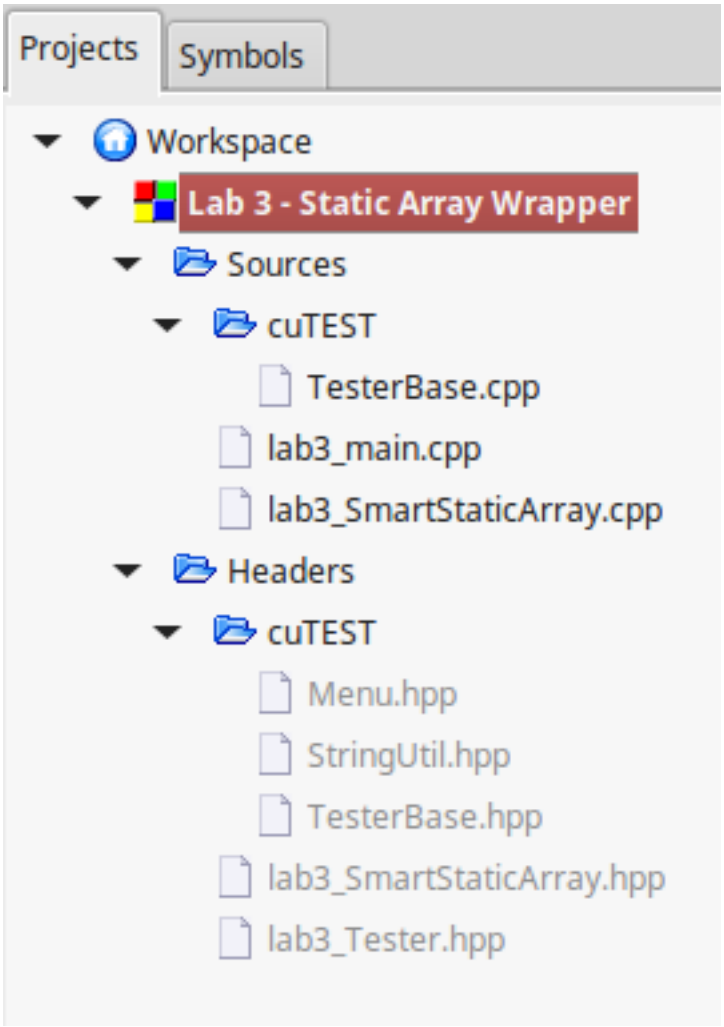
Getting started

Make sure to download the starter code for this project. It contains quite a few files. You will need to create a project and add each of them in.

```
Lab 3 - Static Array Wrapper/
├── lab3_SmartStaticArray.hpp
├── lab3_SmartStaticArray.cpp
├── lab3_main.cpp
├── lab3_tester.hpp
└── cuTEST/
    ├── Menu.hpp
    ├── StringUtil.hpp
    ├── TesterBase.hpp
    └── TesterBase.cpp
```

The items under the **cuTEST** folder is a unit test library that I've written, and is needed in order to use the `lab3_tester.hpp` file. When you run the program you will be able to test one function of your SmartStaticArray at a time.

In Code::Blocks, the project directory will look like this:



Need help setting up the project?
There are helper documents on my Course Common Files repository on GitHub, https://github.com/Rachels-Courses/Course-Common-Files

Go to STUDENT_REFERENCE → HOW_TO, where you will find helper docs on using Code Blocks and Visual Studio.

WHY ALL THESE FILES?! O_O
You will only be implementing the SmartStaticArray class' functions, but I've gone ahead and created the program shell.

All this program does is start up and being the tests, and the tests run

all the functions in the SmartStaticArray class, whose declarations and definitions are included in `lab3_SmartStaticArray.hpp` and `lab3_SmartStaticArray.cpp`.

My cuTEST library also contains some other reusable code, such as `Menu.hpp`, which contains functions to make "pretty" console menus, and `StringUtil.hpp`, which contains my functions to convert between numbers and strings.

What are unit tests?
Unit tests are a type of tests that test one *unit* at a time - usually a single function in a program. The idea behind unit tests is that, for any given function, we know what the **expected outputs** are for some **given inputs**. We can then run the function with those inputs, and compare the **actual output** to the expected output.

You don't need to know how the unit test code in this assignment works, but we will go over writing unit tests later on in class.

When you run the program, you will have a main menu where you can choose which function to test:

```
--------------------------------------------------------------------------------

- cuTEST Main Menu -
--------------------------------------------------------------------------------

1. Push
2. Insert
3. Extend
4. Pop
5. Remove
6. Get
7. Size
8. IsFull
9. IsEmpty
10. operator[]
11. operator=
12. operator==
13. Exit

Enter choice (1 - 13):
```

When you run a test, it will give you an error message if a test fails. This will help you figure out if your function is working correctly, and help protect against logic errors.

```
--------------------------------------------------------------------------------

- TestPop() -
--------------------------------------------------------------------------------
```

```
TEST FAILED: TestPop() A
Tried to pop from an empty array. New list size is negative!

TEST FAILED: TestPop() B
Array's size after pushing MAX_SIZE items is not MAX_SIZE.

TEST FAILED: TestPop() C
Popped all items from an array. IsEmpty is false, but should be true.
```

It will also give you a summary of how many tests passed, so you will be able to see that you're progressing as you go.

Warning!
It is very important to make sure your program compiles and runs at all times! If you keep trying to implement all the functions while the program is not in a building state, you cannot take advantage of the unit tests!

The best way to approach this is to work on **one function at a time**, or even part of one function at a time, and build after every few changes.

Points are taken off for turning in code that doesn't build.

The SmartStaticArray class declaration

The class declaration for the SmartStaticArray has already been written, and looks like this:

```
1  const int MAX_SIZE = 1000;
2
3  class SmartStaticArray
4  {
5      public:
6      SmartStaticArray();
7
8      void Push( const string& newItem );
9      void Insert( int index, const string& newItem );
10     void Extend( const SmartStaticArray& other );
11     void Pop();
12     void Remove( int index );
13     string Get( int index ) const;
14
15     int Size() const;
16     bool IsFull() const;
17     bool IsEmpty() const;
18
19     string operator[]( int index );
20     SmartStaticArray& operator=( const SmartStaticArray&
       other );
21     bool operator==( const SmartStaticArray& other );
22     bool operator!=( const SmartStaticArray& other );
23
24     private:
25     void ShiftRight( int index );
26     void ShiftLeft( int index );
27
28     string m_data[MAX_SIZE];
29     int m_itemCount;
30 };
```

Take note of the private member variables, `m_data` and `m_itemCount`, which will be used within the function definitions.

Implementing the functions

Step through the instructions on how to implement each of these functions.

Note that, for certain tests, certain functions need to be implemented for the tests to work correctly. For example, the `Pop()` function test won't work properly until the `Push()` function has been implemented as well.

SmartStaticArray()

Input parameters: None
Return value: None

In this constructor function, you only need to initialize the private member variable, `m_itemCount`, to 0.

---

int Size()

Input parameters: None
Return value: int, the amount of items stored in the array.

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

---

bool IsFull()

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

You can determine whether the array is full by comparing `m_itemCount` with `MAX_SIZE`. If they are equivalent, then the array is full.

---

bool IsEmpty()

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

You can determine whether the array is full by checking if `m_itemCount` 's value is 0. If it is 0, then the array is empty.

Hint: Shortcut!
While you could write an if/else statement for this and the `IsFull()` functions, you can also simply do...:

```
bool SmartStaticArray::IsEmpty() const
{
    return ( m_itemCount == 0 );
}
```

... which will return true if `( m_itemCount == 0 )`, and false if not.

---

string Get(int index)

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

Error checking: Make sure to check if the index is valid before trying to access the array! The index is invalid if it is less than 0, or greater than or equal to the `m_itemCount`.

If the index is invalid, **throw** an `out_of_range` error with a message: "Cannot get at index - out of range".

Functionality: If the index is valid, then this function will return the value from the private member array, `m_data`, at the position passed in as `index`.

---

void Pop()

Input parameters: None
Return value: None

The Pop() function is used to remove the very last item of the array. However, we are going to do something called a **lazy delete**: we don't *actually* have to change any data; we can simply adjust the `m_itemCount` variable to say that there is one less item.

Therefore, in this function, first check to make sure that `m_itemCount` is greater than 0. If so, then simply decrement `m_itemCount` by one.

Decrementing variables
There are several ways you can decrement a variable:

```
1. num = num - 1;
2. num -= 1;
3. num--;
4. --num;
```

---

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`.

For this function, we are going to shift all the items in the array to the right one space. This is so that we can make space for a new item when the `Insert` function is called.

Create a for loop:

- Start: Create a counter variable `i` and initialize it to `m_itemCount`.
- Loop condition: While `i` is greater than the `index`.
- Update code: Decrement `i` by 1 each time.

Within the loop, set the value of `m_data` at position `i` to the value of `m_data` at position `i-1`.

English → Code
Since we're early on in the class, and you might be a bit rusty at C++, here's what this function is supposed to look like!

```
void SmartStaticArray::ShiftRight( int index )
{
    for ( int i = m_itemCount; i > index; i-- )
    {
        m_data[i] = m_data[i-1];
    }
}
```

---

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`.

When removing an item at a specific index, we will need to close the gap left over; we want all the data in the array to be contiguous. Therefore, we need a ShiftLeft function, that will move all the elements after a certain index back by one. To implement this...

Create a for loop:

- Start: Create a counter variable `i` and initialize it to the `index`.
- Loop condition: While `i` is less than `m_itemCount - 1`.
- Update code: Increment `i` by 1 each time.

Within the loop, set the value of `m_data` at position `i` to the value of `m_data` at position `i+1`.

---

void Push(const string& newItem)

Input parameters: const string& newItem
Return value: None

Error checking: Make sure to check if the array is full before we add any new items, otherwise we will go outside of bounds of the array!

Create an if statement that asks if the array is full (you can use the `IsFull()` function. If the array is full, then **throw** a `length_error` error with a message: "Cannot add new item - array is full!".

Functionality: If the array is not full, then you will add the `newItem` to the array `m_data`, at the position `m_itemCount`. Also make sure to increment `m_itemCount` by one afterwards.

Storing the new value

```
m_data[ m_itemCount ] = newItem;
```

---

void Insert(int index, const string& newItem)

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

Error checking: For this function, there are several things we want to check for before we make any modifications to the array. Make sure to check for...

- If the index is invalid (less than 0, or greater than or equal to `MAX_SIZE`).

  If the index is invalid, then **throw** an `out_of_range` exception with the message, "Cannot insert at index - out of range".

- If the array is full (use `IsFull()`.
  If the array is full, then **throw** a `length_error` exception with the message, "Cannot insert new item - array is full!".

- If the index given is not contiguous in the array.
  If the index is in a bad position, then **throw** an `out_of_range` exception with the message, "Cannot insert at index - must be contiugous!"

Functionality: If everything is OK (no exceptions are thrown), then...

1. Call the `ShiftRight` function, passing in the index.
2. Assign the value `newItem` to the element of `m_data` at the given index.
3. Increment `m_itemCount` by one.

---

void Extend(const SmartStaticArray& other)

Input parameters: const SmartStaticArray& other
Return value: None

This function takes a second SmartStaticArray as its input parameter. The values from the other SmartStaticArray will be appended to the end of the array we're working with from within this function.

Error checking: Check to see if the sum of `m_itemCount` and `other.m_itemCount` is greater than or equal to `MAX_SIZE`. If it is greater, then **throw** a `length_error` with the message, "Cannot append second list - will go out of bounds of array!"

Functionality: If there is no error, then you will create a for loop to iterate through all the items from the `other` SmartStaticArray, and add them to our current SmartStaticArray via the `Push` function.

Hint: Copying over the values

```
for ( int i = 0; i < other.m_itemCount; i++ )
{
    Push( other.Get( i ) );
}
```

---

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`.

Remaining functions

There are several functions that are **overloaded operators**, which are already implemented for you: `operator=`, `operator==`, and `operator!=`.

Running and testing

Make sure your program compiles and runs. Test out each function one at a time via the tester menu.

It is best to make sure that your program runs after every change you make, so that you can use the unit tests to check your work as you're going through.

Grading breakdown

Function	**Point value**
SmartStaticArray constructor	1
Size	1
IsFull	1
IsEmpty	1
Get	2
Pop	2
ShiftRight	3
ShiftLeft	3
Push	3
Insert	5
Extend	5
Remove	3
Total	30

Appendix A: Starter code

lab3_main.cpp

```
#include <iostream>
using namespace std;

#include "lab3_Tester.hpp"

int main()
{
    Tester tester;
    tester.Start();

    return 0;
}
```

lab3_SmartStaticArray.hpp

```
#ifndef _SMART_STATIC_ARRAY_HPP
#define _SMART_STATIC_ARRAY_HPP

#include <iostream>
#include <string>
#include <stdexcept>
using namespace std;

const int MAX_SIZE = 1000;

class SmartStaticArray
{
    public:
    SmartStaticArray();

    void Push( const string& newItem );
    void Insert( int index, const string& newItem );
    void Extend( const SmartStaticArray& other );
    void Pop();
    void Remove( int index );
    string Get( int index ) const;

    int Size() const;
```

```
24      bool IsFull() const;
25      bool IsEmpty() const;
26
27      string operator[]( int index );
28      SmartStaticArray& operator=( const SmartStaticArray&
        other );
29      bool operator==( const SmartStaticArray& other );
30      bool operator!=( const SmartStaticArray& other );
31
32      private:
33      void ShiftRight( int index );
34      void ShiftLeft( int index );
35
36      string m_data[MAX_SIZE];
37      int m_itemCount;
38  };
39
40  #endif
```

lab3_SmartStaticArray.hpp

```
1   #include "lab3_SmartStaticArray.hpp"
2
3   #include "cuTEST/Menu.hpp"
4
5   SmartStaticArray::SmartStaticArray()
6   {
7   }
8
9   int SmartStaticArray::Size() const
10  {
11          return -1; // placeholder
12  }
13
14  bool SmartStaticArray::IsFull() const
15  {
16          return false; // placeholder
17  }
18
19  bool SmartStaticArray::IsEmpty() const
20  {
21          return false; // placeholder
```

```
22 }
23
24 string SmartStaticArray::Get( int index ) const
25 {
26     return ""; // placeholder
27 }
28
29 void SmartStaticArray::Pop()
30 {
31 }
32
33 void SmartStaticArray::ShiftRight( int index )
34 {
35 }
36
37 void SmartStaticArray::ShiftLeft( int index )
38 {
39 }
40
41 void SmartStaticArray::Push( const string& newItem )
42 {
43 }
44
45 void SmartStaticArray::Insert( int index, const string&
     newItem )
46 {
47 }
48
49 void SmartStaticArray::Extend( const SmartStaticArray&
     other )
50 {
51 }
52
53 void SmartStaticArray::Remove( int index )
54 {
55 }
56
57 SmartStaticArray& SmartStaticArray::operator=( const
     SmartStaticArray& other )
58 {
59         for ( int i = 0; i < other.m_itemCount; i++ )
60         {
61         m_data[i] = other.m_data[i];
```

```
62          m_itemCount++;
63          }
64
65          return *this;
66  }
67
68  bool SmartStaticArray::operator==( const
       SmartStaticArray& other )
69  {
70          if ( m_itemCount != other.m_itemCount )
71          {
72          return false;
73          }
74
75          for ( int i = 0; i < m_itemCount; i++ )
76          {
77          if ( m_data[i] != other.m_data[i] )
78          {
79              return false;
80          }
81          }
82
83          return true;
84  }
85
86  bool SmartStaticArray::operator!=( const
       SmartStaticArray& other )
87  {
88          return !( *this == other );
89  }
90
91  string SmartStaticArray::operator[]( int index )
92  {
93      return Get( index );
94  }
```

Additional files

There are other files needed for this lab, though you can download them from the class GitHub page.