Project 1: Unit Tests and Linear Structures Homework projects should be worked on by each individual student. Brainstorming and sketching out problems on paper or on a whiteboard together are permitted, but do not copy code from someone else or allow your code to be copied. Students who commit or aid in plagiarism will receive a 0% on the assignment and be reported.

Information

Topics: Design, testing, UML, unit tests

Turn in: Turn in all source files - .cpp, .hpp, and/or .h files. **Do not turn in Visual Studio files**.

Starter files: Download from GitHub.

Building and running: If you are using Visual Studio, make sure to run with debugging. (Don't run without debugging!) Using the debugger will help you find errors.
To prevent a program exit, use this before `return 0;`

```
cin.ignore();
cin.get();
```

Tips:

- Always make sure it builds. Only add a few lines of code at a time and build after each small change to ensure your program still builds.
- One feature at a time. Only implement one feature (or one function) at a time. Make sure it builds, runs, and works as intended before moving on to the next feature.
- Search for build errors. Chances are someone else has had the same build error before. Copy the message into a search engine and look for information on *why* it occurs, and *how* to resolve it.
- Use debug tools, such as breakpoints, stack trace, and variable watch.
- Don't implement everything in one go. Don't just try typing out all the code in one go without building, running, and testing. It will be much harder to debug if you've tried to program everything all at once.

Contents

About

Linear data structures may be implemented in different ways behind-the-scenes, but the general functionality should still result in the same outcome. For this project, you will design and implement unit tests for a linear structure, and these tests will be tested against the Bag data type, a smart Dynamic Array, and eventually the Linked List object.

Test files

In the starter code, there are the following files:

- `tester_program.cpp` - contains `main()`
- `Tester.hpp` - the class and function declarations
- `Tester.cpp` - the tester function definitions
- `List.hpp` - A List class, with stub functions

The List class will have placeholders in the methods so that the program will be buildable. You will remove the placeholders as the functions are being implemented.

Before implementing the functions, your first task is to implement the tests. As you implement the tests, they will start of failing (for the most part), which is good. Then, when you implement the actual functionality, you can validate that it works as intended by whether it passes the tests.

Your tests may not be correct at first since this might be a new concept, so in some cases it is okay to adjust your tests. You might also have to add additional test cases to more fully cover the functionality.

Once these tests are implemented and working with a static array-based list, then the tests should also theoretically work with other linear structures.

Three turn ins

You will turn in this project in three parts: First, the Test Outline document (edit in LibreOffice or MS Word and turn in). Second, the test implementations in the code. Third, the full List implementation.

Part 1 - Planning Tests

Make sure you download the **Test Outline** document. This is a document you can open up in MS Word or LibreOffice. Work on planning your test sets during class. You can also brainstorm with other students on this part.

Your tests should, at minimum, cover all reasonable outputs. For example, for something that returns a `boolean`, we should have tests that expect a true and a false as outputs. For a function that returns an `integer`, such as `Size()`, you won't test all possible outputs, but at least for 0, 1, and several other values.

bool IsEmpty() const

Test #	State setup	Inputs	Expected output
1.	Create a List, do nothing	(none)	IsEmpty() returns true
2.	Create a List, insert 1 item	(none)	IsEmpty() returns false

Figure 1.1: An example of tests for the IsEmpty() function, in the Test Outline document.

We will work on this part during class, and for Part 1 of the project you can brainstorm and collaborate with other students. However, keep in mind that your code in Part 2 and Part 3 should be your own.

Part 2 - Writing Tests

For this part of the project you should be working just on the tests, not the List functions. You will upload this project in two parts: Just the tests, and then the tests with the List code.

Reference the **Function Specifications** section for descriptions of the functions

Writing unit tests

Each unit test is meant to test one function each, though you might have to rely on multiple functions to fully validate a function. For example, to test the `Size()` function, you need to be able to add new items to the list, so `Push(...)` will have to be implemented for the `Size()` test to work properly.

Each test function may need multiple tests to properly coverage all reasonable use cases.

For each test, you will need to decide on **inputs** to pass in, and the **expected output**. To get the **actual output**, you will call the function being tested, and store its return value in a variable.

You can use an if statement to check whether the actual output matches the expected output. If they match, the individual test can *pass*. If they don't match, the test *fails*.

It's up to you for what to output to the screen, though generally it helps if you output the inputs and outputs, and whether the test passes or fails.

Tests to implement

The tests to implement are:

Test_Init()	Test_ShiftRight()	Test_ShiftLeft()
Test_Size()	Test_IsEmpty()	Test_IsFull()
Test_PushFront()	Test_PushBack()	Test_PopFront()
Test_PopBack()	Test_Clear()	Test_Get()
Test_GetFront()	Test_GetBack()	Test_GetCountOf()
Test_Contains()	Test_Remove()	Test_Insert()

Example test:

```
1  void Tester::Test_Size()
2  {
3      DrawLine();
4      cout << "TEST: Test_Size" << endl;
5
6      {   // Test begin
7          cout << "\n \n Test 1";
8          List<int> testList;
9          int expectedSize = 0;
10         int actualSize = testList.Size();
11
12         cout << "\n Expected size: " << expectedSize;
13         cout << "\n Actual size:   " << actualSize;
14
15         if ( actualSize == expectedSize )
16             cout << "\n Pass";
17         else
18             cout << "\n Fail";
19     }   // Test end
20
21     {   // Test begin
22         cout << endl << "Test 2" << endl;
23         List<int> testList;
24
25         testList.PushBack( 1 );
26         testList.PushBack( 3 );
27
28         int expectedSize = 2;
29         int actualSize = testList.Size();
30
31         cout << "\n Expected size: " << expectedSize;
32         cout << "\n Actual size:   " << actualSize;
33
34         if ( actualSize == expectedSize )
35             cout << "\n Pass";
36         else
37             cout << "\n Fail";
38     }   // Test end
39 }
```

Example output:

An example of a couple of tests failing, since the List functions haven't been implemented yet.

```
--------------------------------------------------------------------------------
TEST: Test_Size

Test 1
Expected size: 0
Actual size:   -1585510992
Fail

Test 2
Expected size: 1
Actual size:   -1585510992
Fail
```

Part 3 - Implementing the List

Once you've implemented the tests, you will be implementing the List functions based on the functionality specified. As you get each function working, your tests should be passing.

Note that certain tests may rely on the implementation of other functions, such as `Size()` needing a Push function implemented in order to test for more cases than just an empty list.

Function Specifications

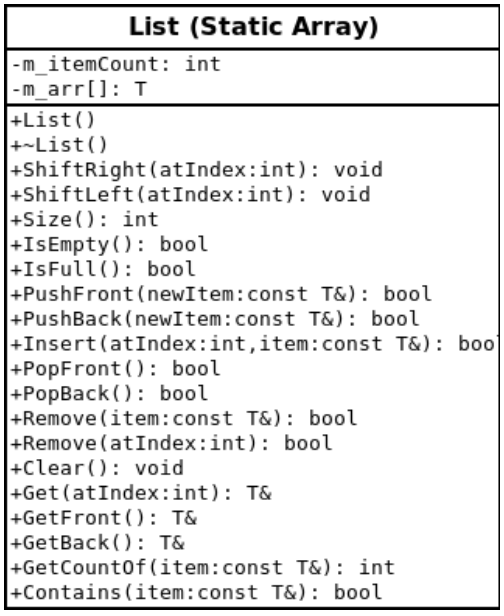

Figure 1.2: A List class UML diagram, implemented with a static array

IsEmpty

Function header: `bool IsEmpty() const`

Inputs: None

Outputs: `true` if there is nothing in the list (i.e., the size is 0), or `false` if not.

IsFull

Function header: `bool IsFull() const`

Inputs: None

Outputs: `true` if the list is full (i.e., the `m_itemCount` is equal to `ARRAY_SIZE`), or `false` if not.

Size

Function header: `int Size() const`

Inputs: None

Outputs: Returns the amount of items stored in the list. Will be ≥ 0.

GetCountOf

Function header: `int GetCountOf( const T& item ) const`

Inputs: `const T& item`, where the list contains items of type `T` (it is a template). The item to be searched for and counted is `item`.

Outputs: Returns the amount of instances of `item` being found in the list. Will be ≥ 0.

Contains

Function header: `bool Contains( const T& item ) const`

Inputs: `const T& item`, where `item` is the item to search for.

Outputs: Returns `true` if `item` is found in the list, and `false` otherwise.

List constructor

Function header: `List()`

Initializes the List class. The list should start off empty.

Inputs: None

List destructor (no tests)

Nothing required for the static array version of the List.

PushFront

Function header: `bool PushFront( const T& newItem )`

Inputs: `const T& item`, the new item to add to the list at the beginning.

Outputs: Returns `true` if the operation is a success, and `false` otherwise.

PushBack

Function header: `bool PushBack( const T& newItem )`

Inputs: `const T& item`, the new item to add to the list at the end.

Outputs: Returns `true` if the operation is a success, and `false` otherwise.

Insert

Function header: `bool Insert( int atIndex, const T& item )`

Inserts a new item at the given index. Existing items shouldn't be replaced. If there is an item at that position, shift everything right and then insert the new item to that index.

Only allow **contiguous** elements - don't allow someone to insert a new item past `m_itemCount`. Therefore, valid indices you can insert at are $0 \leq$ atIndex $\leq$ `m_itemCount`.

Inputs:

- `int atIndex` - the index where to insert the new item.
- `const T& item` - the new item to insert.

Outputs: Returns `true` if the operation is a success, and `false` otherwise.

Get

Function header: `T* Get( int atIndex )`

Inputs: `int atIndex`, the index of the item to return.

Outputs: Returns the address of the item at the index given, or `nullptr` if it isn't found.

GetFront

Function header: `T* GetFront()`

Inputs: None

Outputs: Returns the address of the item at the front of the list, or `nullptr` if the list is empty.

GetBack

Function header: `T* GetBack()`

Inputs: None

Outputs: Returns the address of the item at the end of the list, or `nullptr` if the list is empty.

PopFront

Function header: `bool PopFront()`

Inputs: None

Outputs: Returns `true` if removing the item is successful, and `false` otherwise.

PopBack

Function header: `bool PopBack()`

Inputs: None

Outputs: Returns `true` if removing the item is successful, and `false` otherwise.

Remove

Function header: `bool Remove( const T& item )` Function header: `bool Remove( int atIndex )`

Inputs:

- `const T& item` - locates all instances of this item in the list and removes it.
- `int atIndex` - removes the item at this index.

Outputs: Returns `true` if removing the item is successful, and `false` otherwise.

Clear

Function header: `void Clear()`

Clears out the list. To test this function, you should ensure that the size of the list returns to 0 after this function is called.

Inputs: None

Outputs: None

ShiftRight

Function header: `bool ShiftRight( int atIndex )`

Moves everything at the index and to the right of the index further to the right by one position.

Inputs: `int atIndex` - The index at which to begin moving items forward.

Outputs: Returns `true` if the shift is successful, or `false` if not.

ShiftLeft

Function header: `bool ShiftLeft( int atIndex )`

Moves everything to the right of the index further to the left by one position. This will overwrite the item at the `atIndex`.

Inputs: None

Outputs: Returns `true` if the shift is successful, or `false` if not.

Grading breakdown

Breakdown	
Score	
Item	**Task weight**
Part 1: Test Outline (10%)	
- Adequate test coverage planned out	10.00%
Part 2: Unit Tests (40%)	
- Test_Init()	2.00%
- Test_ShiftRight()	3.00%
- Test_ShiftLeft()	3.00%
- Test_Size()	2.00%
- Test_IsEmpty()	2.00%
- Test_IsFull()	2.00%
- Test_PushFront()	3.00%
- Test_PushBack()	3.00%
- Test_Insert()	3.00%
- Test_PopFront()	3.00%
- Test_PopBack()	3.00%
- Test_Remove()	3.00%
- Test_Clear()	2.00%
- Test_Get()	2.00%
- Test_GetFront()	2.00%
- Test_GetBack()	2.00%
Part 3: Function Implementations (50%)	
- List()	1.00%
- ShiftRight()	5.00%
- ShiftLeft()	5.00%
- Size()	2.00%
- IsEmpty()	2.00%
- IsFull()	2.00%
- PushFront()	3.00%
- PushBack()	3.00%
- Insert()	3.00%
- PopFront()	3.00%
- PopBack()	3.00%
- Remove()	3.00%
- Clear()	2.00%
- Get()	3.00%
- GetFront()	3.00%
- GetBack()	3.00%
- GetCountOf()	2.00%
- Contains()	2.00%
Score totals	100.00%

Penalties	
Item	**Max penalty**
Syntax errors (doesn't build)	-50.00%
Logic errors	-10.00%
Run-time errors	-10.00%
Memory errors (leaks, bad memory access)	-10.00%
Ugly code (bad indentation, no whitespacing)	-5.00%
Ugly UI (no whitespacing, no prompts, hard to use)	-5.00%
Not citing code from other sources	-100.00%
Not all tests run (Tests crash)	-10.00%
Penalty totals	