Lab : The Standard Template Library

Information

In-class labs are meant to introduce you to a new topic and provide some practice with that new topic.

Topics: Vectors, Lists, Stack, Queue, Map

Solo work: Labs should be worked on by each individual student, though asking others for help is permitted. Do not copy code from other sources, and do not give your code to other students. **Students who commit or aid in plagiarism will receive a 0% on the assignment and be reported.**

Building and running: If you are using Visual Studio, make sure to run with debugging. **Do not 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();
```

Turn in: Once you're ready to turn in your code, prepare the files by doing the following: **(1)** Make a copy of your project folder and name it `LASTNAME-FIRSTNAME-LABNAME`. **(2)** Make sure that all source files (.cpp, .hpp, and/or .h files) and the `Makefile` files are all present. **(3)** Remove all Visual Studio files - I only want the source files and Makefiles. **(4)** Zip your project folder as `LASTNAME-FIRSTNAME-LABNAME.zip`

Never turn in Visual Studio files!

Starter files: Download from GitHub.

Grading: Grading is based on completion, if the program functions as intended, and absense of errors. **Programs that don't build will receive 0%.** Besides build errors, runtime errors, logic errors, memory leaks, and ugly code will reduce your score.

Contents

1.1 About

Data structures have been implemented countless times for various languages and various platforms. In the real world, the chances that you'll need to implement a data structure from scratch is somewhat minimal (depending on what you're working on), but it is important to understand *how* different data structures work so that you can make the best design decisions.

Before we get into the specifics of *how* to make a data structure, it is good to get an introductory understanding of why there are different types of structures, what they do, and how they're used.

1.2 Setting up the project

Follow these instructions to keep your source code separate from your project files. This is so that when you submit your source code, it is more managable.

1. From the GitHub Labs page, click on the **starter files** folder.
2. Download the appropriate **zip** file for this lab.
3. **Extract** the zip file contents to your desktop (or other harddrive location). [1] [2]
4. Open your IDE [3] of choice.
5. Create a project/solution in a **different directory** from your source files.
6. When adding files to your project, navigate to the lab folder and add the source files.

[1]If you work off a flash drive, the compile process will be very slow.
[2]If you *don't* extract the zip file, the compile process won't work.
[3]Integrated Development Environment

1.3 Lab specifications

Each data structure's purpose is to store data and make it accessible to the user. Because of this, common operations include **add** an item, **remove** an item, and **access** an item. Depending on the data structure, where an item can be accessed differs.

1.3.1 Part 1: STL Vector

> "Vectors are sequence containers representing arrays that can change in size.
>
> Just like arrays, vectors use contiguous storage locations for their elements, which means that their elements can also be accessed using offsets on regular pointers to its elements, and just as efficiently as in arrays. But unlike arrays, their size can change dynamically, with their storage being handled automatically by the container.
>
> Internally, vectors use a dynamically allocated array to store their elements."
>
> From http://www.cplusplus.com/reference/vector/vector/

The STL Vector is essentially a class that wraps [4] a dynamic array.

With the vector, we can add new items with the `push_back` function, and remove the last item in the vector's array with the `pop_back` function.

Because a vector is implemented with an array, we can randomly access an element of the vector's array with the subscript operator [5]

And because the dynamic array is wrapped inside a class (the vector class), it gives us a function to get the current amount of items in the array: the `size` function. If we had implemented a dynamic array on our own, we would have to create a separate integer variable to keep track of the amount of items in the array. [6]

[4] To "wrap" something is to store that item in a class, adding a public interface via the class' functions, and possibly adding helper variables.

[5] The subscript operator is [ ].

[6] Or use the `sizeof` the array, divided by the size of the element data type.

Use the documentation to implement the following functionality.

Documentation links

STL Vector:	cplusplus.com/reference/vector/vector
push_back:	cplusplus.com/reference/vector/vector/push_back
pop_back:	cplusplus.com/reference/vector/vector/pop_back
size:	cplusplus.com/reference/vector/vector/size
operator[]:	cplusplus.com/reference/vector/vector/operator[]
back:	cplusplus.com/reference/vector/vector/back

Create a program loop that will continue running until the user decides to quit. The main menu will have four options: 1. Add a new course, 2. Remove the last course, 3. Display the course list, and 4. Quit.

The program will need a `vector<string>` item to store an array of courses. Make sure that this is declared prior to the program loop starting; if it is declared within the loop, then you'll be working with an empty vector each iteration.

If the user wants to add a new item, ask them for the course name, then use the `push_back` function to add it to the vector.

If the user wants to remove the last course, then use the `pop_back` function to remove the last item from the vector.

If the user wants to display the entire course list, use the `size` function and the subscript operator to display each item. The easiest way to do this would be with a for loop, going from $i = 0$ to the vector's $size - 1$. [7]

See the **Example Output** section for what the program should look like.

[7] Your compiler might complain about using an integer variable when comparing to the vector's size, so you could use an `unsigned int` instead.

1.3.2 Part 2: STL List

> "Lists are sequence containers that allow constant time insert and erase operations anywhere within the sequence, and iteration in both directions.
>
> List containers are implemented as doubly-linked lists; Doubly linked lists can store each of the elements they contain in different and unrelated storage locations. The ordering is kept internally by the association to each element of a link to the element preceding it and a link to the element following it. [...]
>
> The main drawback of lists [...] compared to these other sequence containers is that they lack direct access to the elements by their position; For example, to access the sixth element in a list, one has to iterate from a known position (like the beginning or the end) to that position, which takes linear time in the distance between these. They also consume some extra memory to keep the linking information associated to each element (which may be an important factor for large lists of small-sized elements)."
>
> From http://www.cplusplus.com/reference/list/list/

We will eventually learn about what a *linked list* is. For now, know that a STL List object is *not* implemented with an array. The elements of an array are stored contiguously[8] in memory, for a linked list the elements are stored in arbitrary[9] places. Because of this, we cannot randomly access elements of a list like we can in a vector. More specifically, this means we cannot access items with the [ ] operator.

The easiest elements of a list to access are the **head** (aka front or first item) and **tail** (aka back or last item). To step through each element of a list, we need to use something called **iterators**, which you may not have covered in previous classes. Iterators are covered in Interlude 7, but only the basics will be explained for now.

Additionally, the STL List also has some useful functions that will sort and reverse the elements of the list.

[8]side-by-side-by-side
[9]virtually random

Use the documentation to implement the following functionality.

Documentation links

STL List:	cplusplus.com/reference/list/list
push_back:	cplusplus.com/reference/list/list/push_back
pop_back:	cplusplus.com/reference/list/list/pop_back
back:	cplusplus.com/reference/list/list/back
push_front:	cplusplus.com/reference/list/list/push_front
pop_front:	cplusplus.com/reference/list/list/pop_front
front:	cplusplus.com/reference/list/list/front
size:	cplusplus.com/reference/list/list/size
begin iterator:	cplusplus.com/reference/list/list/begin
sort:	cplusplus.com/reference/list/list/sort
reverse:	cplusplus.com/reference/list/list/reverse

Create a program loop that will continue running until the user decides to continue. The main menu has the options: (1) Add new states to front of list, (2) Add new state to back of list, (3) Pop state from front of list, (4) Pop state from end of list, and (5) Continue.

The program will need a `list<string>` item to store a list of states.

If the user wants to add a state to the front of the list, use the `push_front` function.

If the user wants to add a state to the back of the list, use the `push_back` function.

If the user wants to remove the state at the front of the list, use the `pop_front` function.

If the user wants to remove the state at the back of the list, use the `pop_back` function.

Once the user selects **continue**, it will display the original list, the reversed list, the sorted list, and the reverse-sorted list.

You can use the following function to display the list using an iterator:

```
void DisplayList( list<string>& states )
{
    for (   list<string>::iterator it = states.begin();
            it != states.end();
            it++ )
    {
        cout << *it << "\t";
    }
    cout << endl;
}
```

After the user has continued, display the original list of states.

Then, reverse the states list and display that.

Then, sort the states list and display that.

Then, reverse the sorted states list and display that.

View the **Example Output** section to see how the program should run.

1.3.3 Part 3: STL Queue

"queues are a type of container adaptor, specifically designed to operate in a FIFO context (first-in first-out), where elements are inserted into one end of the container and extracted from the other. [...]

Elements are pushed into the "back" of the specific container and popped from its "front"."

From http://www.cplusplus.com/reference/queue/queue/

Documentation links

STL Queue:	cplusplus.com/reference/queue/queue
push:	cplusplus.com/reference/queue/queue/push
pop:	cplusplus.com/reference/queue/queue/pop
front:	cplusplus.com/reference/queue/queue/front
size:	cplusplus.com/reference/queue/queue/size
empty:	cplusplus.com/reference/queue/queue/empty

For the queue program, do the following:

Create a `queue<float>` to store a list of pending transactions.

Create a loop, and continue asking the user for the next transaction amount, until they decide to **continue**. Transactions can be positive (adding to account) or negative (withdrawing from account).

Each time the user adds a transaction amount, push this new amount to the transactions queue.

Once the user chooses to continue, create a `float` variable to store the user's balance. Initialize it to `0`.

Loop through the transactions queue while it is not empty. Within the loop...
  Display the front-most transaction amount to the user.
  Add the front-most amount to the balance variable.
  Pop the front of the queue.

Once the loop has completed, display the final balance.

1.3.4 Part 4: STL Stack

"Stacks are a type of container adaptor, specifically designed to operate in a LIFO context (last-in first-out), where elements are inserted and extracted only from one end of the container. [...]

Elements are pushed/popped from the "back" of the specific container, which is known as the top of the stack."

From http://www.cplusplus.com/reference/stack/stack/

Documentation links

STL Stack:	cplusplus.com/reference/stack/stack
push:	cplusplus.com/reference/stack/stack/push
pop:	cplusplus.com/reference/stack/stack/pop
top:	cplusplus.com/reference/stack/stack/top
size:	cplusplus.com/reference/stack/stack/size
empty:	cplusplus.com/reference/stack/stack/empty

For the stack program, create a `stack<string>` item to store the letters of a word.

Create a loop. Let the user enter the next letter(s) of the word, OR type "UNDO" to undo their last letter, or "DONE" once they are finished.

When adding a new letter to the word, use the `push` function.

When removing the last-added letter from the word, use the `pop` function.

Once the user is done entering letters, create a loop that will continue while the stack is not empty. Within this loop...
  Display the top-most letter in the word stack.
  Pop the top-most letter in the word stack.

See the **Example Output** section for a preview of the program.

1.3.5 Part 5: STL Map

"Maps are associative containers that store elements formed by a combination of a key value and a mapped value, following a specific order.

In a map, the key values are generally used to sort and uniquely identify the elements, while the mapped values store the content associated to this key. The types of key and mapped value may differ"

From http://www.cplusplus.com/reference/map/map/

A map is sometimes known as a **dictionary** or a **hash table**. We will learn more about dictionaries later in Chapter 18.

For a dictionary, a **key** and a **value** are associated together. The key can be any data-type, and the value can be any data-type. They may look similar to arrays at first, except instead of having indices from 0 to $size - 1$, the "index" can be any data-type.

Documentation links

STL Map: cplusplus.com/reference/map/map

At the beginning of your map program, create a map that has a `char` as the key, and a `string` as the value. The declaration will look like this:

```
map<char, string> colors;
```

Assign the following values to the colors map:

key	value		key	value
r	FF0000		c	00FFFF
g	00FF00		m	FF00FF
b	0000FF		y	FFFF00

Assigning these values will look like this:

```
colors[ 'r' ] = "FF0000";
```

Next, create a loop. Have the user enter a `char` value.

If their choice is `'q'`, then exit the program.

Otherwise, use the user's input as a key, and try to display the color associated with it.

```
cout << "Hex:  " << colors[ color ] << endl;
```

(If no color is associated with that key, then it will be blank.)

See the **Example Output** section for the output.

1.4 Example output

1.4.1 Part 1: STL Vector

```
----------------------------
Course list size: 0

1. Add a new course                2. Remove last course
3. Display the course list         4. Quit

>> 1

NEW COURSE
Enter new course name: CS 200

----------------------------
Course list size: 1

1. Add a new course                2. Remove last course
3. Display the course list         4. Quit

>> 1

NEW COURSE
Enter new course name: CS 235

----------------------------
Course list size: 2

1. Add a new course                2. Remove last course
3. Display the course list         4. Quit

>> 3

VIEW COURSE LIST
courses[0] = CS 200
courses[1] = CS 235

----------------------------
Course list size: 2

1. Add a new course                2. Remove last course
3. Display the course list         4. Quit
```

```
>> 2

REMOVE COURSE
CS 235 removed

----------------------------
Course list size: 1

1. Add a new course             2. Remove last course
3. Display the course list      4. Quit

>> 1

NEW COURSE
Enter new course name: CS 250

----------------------------
Course list size: 2

1. Add a new course             2. Remove last course
3. Display the course list      4. Quit

>> 1

NEW COURSE
Enter new course name: CS 211

----------------------------
Course list size: 3

1. Add a new course             2. Remove last course
3. Display the course list      4. Quit

>> 3

VIEW COURSE LIST
courses[0] = CS 200
courses[1] = CS 250
courses[2] = CS 211

----------------------------
Course list size: 3
```

```

1. Add a new course              2. Remove last course
3. Display the course list       4. Quit

>> 4

Goodbye
```

1.4.2 Part 2: STL List

```
----------------------------
State list size: 0

1. Add new state to front    2. Add new state to back
3. Pop front state           4. Pop back state
5. Continue

>> 1

ADD STATE TO FRONT
Enter new state name: Missouri

----------------------------
State list size: 1

1. Add new state to front    2. Add new state to back
3. Pop front state           4. Pop back state
5. Continue

>> 1

ADD STATE TO FRONT
Enter new state name: Nebraska

----------------------------
State list size: 2

1. Add new state to front    2. Add new state to back
3. Pop front state           4. Pop back state
5. Continue

>> 2

ADD STATE TO BACK
Enter new state name: Kansas

----------------------------
State list size: 3

1. Add new state to front    2. Add new state to back
3. Pop front state           4. Pop back state
```

```
5. Continue

>> 2

ADD STATE TO BACK
Enter new state name: Alaska

----------------------------
State list size: 4

1. Add new state to front    2. Add new state to back
3. Pop front state          4. Pop back state
5. Continue

>> 3

REMOVE STATE FROM FRONT
Nebraska removed

----------------------------
State list size: 3

1. Add new state to front    2. Add new state to back
3. Pop front state           4. Pop back state
5. Continue

>> 4

REMOVE STATE FROM BACK
Alaska removed

----------------------------
State list size: 2

1. Add new state to front    2. Add new state to back
3. Pop front state           4. Pop back state
5. Continue

>> 1

ADD STATE TO FRONT
Enter new state name: Iowa
```

```
----------------------------
State list size: 3

1. Add new state to front    2. Add new state to back
3. Pop front state           4. Pop back state
5. Continue

>> 2

ADD STATE TO BACK
Enter new state name: California

----------------------------
State list size: 4

1. Add new state to front    2. Add new state to back
3. Pop front state           4. Pop back state
5. Continue

>> 5

ORIGINAL LIST:
Iowa    Missouri    Kansas  California

REVERSED LIST:
California  Kansas  Missouri    Iowa

SORTED LIST:
California  Iowa    Kansas  Missouri

REVERSE-SORTED LIST:
Missouri    Kansas  Iowa    California

Goodbye
```

1.4.3 Part 3: STL Queue

```
----------------------------
Transactions queued: 0

1. Enqueue transaction   2. Continue

>> 1

Enter amount (positive or negative) for next transaction
   : 9.99

----------------------------
Transactions queued: 1

1. Enqueue transaction   2. Continue

>> 1

Enter amount (positive or negative) for next transaction
   : -5.20

----------------------------
Transactions queued: 2

1. Enqueue transaction   2. Continue

>> 1

Enter amount (positive or negative) for next transaction
   : 15.38

----------------------------
Transactions queued: 3

1. Enqueue transaction   2. Continue

>> 1

Enter amount (positive or negative) for next transaction
   : -2.33
```

```
----------------------------
Transactions queued: 4

1. Enqueue transaction   2. Continue

>> 2

9.99 pushed to account
-5.20 pushed to account
15.38 pushed to account
-2.33 pushed to account

Final balance: $17.84

Goodbye
```

1.4.4 Part 4: STL Stack

```
Enter the next letter of the word, or UNDO to undo, or
   DONE to stop.
>> e
>> r
>> r
>> UNDO
      Removed r
>> e
>> h
>> t
>> -
>> h
>> UNDO
      Removed h
>> o
>> l
>> l
>> e
>> h
>> DONE


Finished word: hello-there
```

1.4.5 Part 5: STL Map

```
Enter a color letter, or 'q' to stop: r
Hex: FF0000

Enter a color letter, or 'q' to stop: g
Hex: 00FF00

Enter a color letter, or 'q' to stop: b
Hex: 0000FF

Enter a color letter, or 'q' to stop: c
Hex: 00FFFF

Enter a color letter, or 'q' to stop: m
Hex: FF00FF

Enter a color letter, or 'q' to stop: y
Hex: FFFF00

Enter a color letter, or 'q' to stop: t
Hex:

Enter a color letter, or 'q' to stop: q
Goodbye
```