Lab 1: STL Containers

Information

Topics: vector, list, stack, queue, map

Turn in: All .cpp files, question document as .txt file.
Every lab *part* should have its own .cpp file.
Give each lab file a unique name, like `lab1_part1.cpp` .

Part 1: STL Vector

STL Vector functions

`void vector::push_back( value_type& value )`
Adds a new element at the end of the vector, after its current last element. The content of val is copied (or moved) to the new element.

`unsigned int vector::size()`
Returns the number of elements in the vector.

`value_type& vector::operator[]( size_type n )`
Returns a reference to the element at position n in the vector container.

http://www.cplusplus.com/reference/vector/vector/

We will start out by working with the **vector** object of the Standard Template Library. You can view the documentation for this object at

`http://www.cplusplus.com/reference/vector/vector/`

Starter code

Begin with the following code:

```
#include <iostream> // required for cout
#include <vector>   // required for vector
#include <string>   // required for strings
using namespace std;

void AddIngredients( vector<string>& ingredients )
{
}

void DisplayIngredients( const vector<string>& ingredients )
{
}

int main()
{
    return 0;
}
```

Notice that the functions `AddIngredients(...)` and `DisplayIngredients(...)` both contain `vector` parameters. Vectors are essentially like *dynamic arrays* , and they can store any data type. In this case - these vectors will store `strings` .

The `AddIngredients(...)` function will be responsible for inserting new strings into the vector, so the vector parameter is being passed in **by-reference** , denoted by the &.

The `DisplayIngredients(...)` function will only display the values stored in the vector, so the parameter is passed as a **const reference** . Why? – Because a vector could be a large object, so it is more efficient to pass it by-reference than by-value. Passing by-value means copying all the data, and that could be *computationally expensive* .

---

main()

Within `main()` , create a `vector` of `strings` named **ingredients**

```
vector<string> ingredients;
```

Next, *call* the `AddIngredients(...)` function, passing in the **ingredients** variable as the argument.

```
AddIngredients( ingredients );
```

Finally, *call* the `DisplayIngredients(...)` function, again passing in the **ingredients** variable as the argument.

```
DisplayIngredients( ingredients );
```

Now the vector variable is being passed around, but these functions currently don't do anything. We will implement these next.

Review: Parameter vs. Argument

When you're writing a function, the variables declared within the ( ) are called **parameters** .

When you're passing data *into* a function call, the items that you're passing in are called **arguments** .

When passing arguments, you can use variables or literals. When passing in a variable, you do not need to specify its data type.

When a parameter is **pass-by-reference** , then you can only pass in a variable. No special symbols are required from the argument side.

---

void AddIngredients(vector<string>& ingredients)

The `vector` class has a function called `push_back` , which allows us to add a new item to the back of the vector's internal array. You can read the documentation for `push_back` at

```
http://www.cplusplus.com/reference/vector/vector/push_back/
```

The `push_back` function has a `void` return type, and its only parameter is a new value. The data type of the parameter depends on what the vector has been declared with. In our case, it is a `string` .

Using the `push_back` function, insert the following items into the vector: (1) lettuce, (2) tomato, (3) mayo, and (4) bread.

```
ingredients.push_back( "lettuce" );
```

---

void DisplayIngredients(const vector<string>& ingredients)

The `vector` class has a function `size()` that will return an `unsigned integer` with the count of items in the vector.

You can also access items at specific elements of the array with the **subscript operator** , [ ]. The vector uses an array behind-the-scenes, so you can access the first element at position 0, like: `ingredients [ 0 ]`

Hint: Iterating over an array

Maybe you're feeling a little rusty. For an array of size n, how do we iterate over all its elements to display them?

```
for ( int i = 0; i < n; i++ )
{
    cout << i << ". " << arr[i] << endl;
}
```

Make sure you review your **core C++ contepts** so that you can spend more time on the design of the data structures in this course, instead of struggling with how to write the basics.

Use a `for loop` to iterate over all items in the vector (index 0 to $size - 1$), and display both the index of each vector element, and its value, using a `cout` statement.

Your program output should look like the following.

Example output

```
0. lettuce
1. tomato
2. mayo
3. bread
```

Review: Arrays

Recall from previous programming courses that when you create an array in C++, the elements of the array are stored **side-by-side** in memory. This is the reason that we can **randomly access** elements of the array, because if we know the address of the first element, then we simply need to step to an offset of *index* × *bytes* to find the element at that index.

`int arr[3] =`

index	address
0	`0x7ffee961a920`
1	`0x7ffee961a924`
2	`0x7ffee961a928`

The size of an integer is 4 bytes, so notice the addresses of the elements are different by 4. `0x...20`, `0x...24`, `0x...28`

Part 2: STL List

STL List functions

`void list::push_back( value_type& value )`
Adds a new element at the end of the list container, after its current last element. The content of val is copied (or moved) to the new element.

`void list::push_front( value_type& value )`
Inserts a new element at the beginning of the list, right before its current first element. The content of val is copied (or moved) to the inserted element.

`void list::sort()`
Sorts the elements in the list, altering their position within the container.

`void list::reverse()`
Reverses the order of the elements in the list container.

http://www.cplusplus.com/reference/list/list/

The **list** is another type of linear data structure from the Standard Template Library. Unlike the `vector`, however, the List is not implemented with an array, but a series of "nodes" that point at the next element in the list. Because of this, you cannot **randomly access elements** of a list like you can with a vector or an array.

`http://www.cplusplus.com/reference/list/list/`

Starter code

Begin with the following code:

```
1  #include <iostream> // required for cout
2  #include <list>     // required for list
3  #include <string>   // required for strings
4  using namespace std;
5
6  void AddCourses( list<string>& courses )
7  {
8  }
9
10 void SortList( list<string>& courses )
11 {
12 }
13
14 void ReverseList( list<string>& courses )
15 {
16 }
17
18 void DisplayCourses( list<string>& courses )
```

```
19 {
20     int counter = 0;
21     // This is how we have to iterate thru a list.
22     for( list<string>::iterator it = courses.begin();
23         it != courses.end(); it++ )
24     {
25         if ( counter != 0 ) { cout << ", "; }
26         cout << counter++ << ". " << (*it);
27     }
28 }
29
30 int main()
31 {
32     return 0;
33 }
```

---

main()

First, in `main()`, create a new list that will store strings, like this:

```
list<string> courses;
```

Then call `AddCourses(...)`, passing in the `courses` variable as the argument. Afterward, call `DisplayCourses(...)`.

Next, call `SortList(...)` and then `DisplayCourses(...)`.

And finally, call `ReverseList(...)` and then `DisplayCourses(...)`.

`DisplayCourses(...)` has already been written for you because using a for loop on a `list` object requires using `iterators`; we can't just iterate from $i = 0$ to $size - 1$ like with a `vector`.

---

void AddCourses(list<string>& courses)

With the `list`, we can add new items to the **front**, using `push_front(...)` or to the **back**, using `push_back(...)`.

Add the following items to the `courses` list, at either the front or the back: *cs 250, cs 200, cs 210, cs 235, cs 134, cs 211*

---

void SortList(list<string>& courses)

The `list` class contains a function called `sort()` that will sort all the elements in the list for you. Call it from this function.

---

void ReverseList(list<string>& courses)

The `list` class contains a function called `reverse()` that will reverse the order of elements in the list. Call it with this function.

Running the program, your output will look something like...

Example output

```
Normal order
0. cs 250, 1. cs 200, 2. cs 235, 3. cs 210, 4. cs 134, 5. cs 211

Sorted order
0. cs 134, 1. cs 200, 2. cs 210, 3. cs 211, 4. cs 235, 5. cs 250

Reverse order
0. cs 250, 1. cs 235, 2. cs 211, 3. cs 210, 4. cs 200, 5. cs 134
```

(Note: The first output will look different based on the order you put the items in.)

Linked Lists

Later in the class, we will be working with **linked lists** .

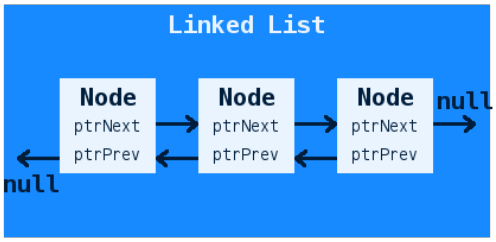
Linked lists use **pointers** to create nodes for a list, by allocating memory for new nodes as needed...

```
Node* newNode = new Node();
newNode->data = 123;

newNode->ptrPrevious = ptrLast; // Link them together
ptrLast->ptrNext = newNode;

ptrLast = newNode // Update the address ptrLast is pointing to
```

The **STL List** is implemented with a **doubly-linked list**



Part 3: STL Queue

STL Queue functions

`value_type& queue::front()`
Returns a reference to the next element in the queue. The next element is the "oldest" element in the queue and the same element that is popped out from the queue when queue::pop is called.

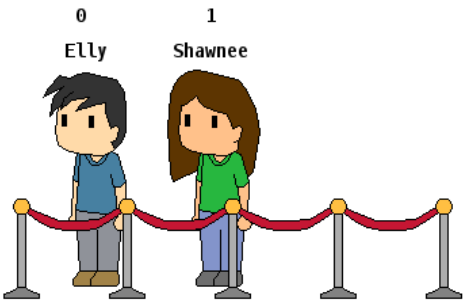
`void queue::push( const value_type& val )`
Inserts a new element at the end of the queue, after its current last element. The content of this new element is initialized to val.

`void queue::pop()`
Removes the next element in the queue, effectively reducing its size by one. The element removed is the "oldest" element in the queue whose value can be retrieved by calling member queue::front.

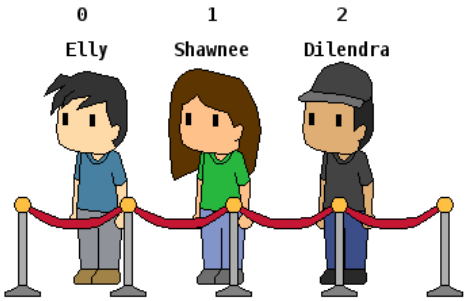
`bool queue::empty()`
Returns whether the queue is empty: i.e. whether its size is zero.

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

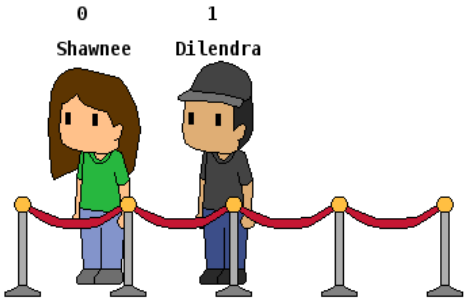
A **queue** is a type of data structure that is linear in nature, but access to the elements are restricted. Queues can be implemented with arrays or with list structures.



Items can be **pushed** into the back of the queue, just as if you were lining up at the store.



The first item that gets to leave the queue is the item at the **front.** Then, everything gets moved forward one spot.



We will be learning more about queues later in the semester, so it is good to get acquainted with them.

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

Starter code

Begin with the following code:

```
#include <iostream> // required for cout
#include <string>   // required for strings
#include <queue>    // required for queues
using namespace std;
```

```
5
6  int main()
7  {
8      float balance = 0.0;
9
10     cout << "Final balance: $" << balance << endl;
11
12     return 0;
13 }
```

---

main()

Within `main()`, create a queue of `floats` called **transactions**.

```
queue<float> transactions;
```

Create a series of values into the queue, both positive and negative values. These will represent deposits and withdraws into an account. The function to add an item to the queue is `push(...)`.

Create a while loop that will keep looping while the queue is not empty. To check if the queue is empty, use the `empty()` function.
While the queue is *not empty* ...

- Access the front item in the queue with `front()`. Display this value with `cout`, and also add this value to the `balance` variable.
- Use the queue's `pop()` function next to remove the front item.

At the end of the program, the final balance will be displayed.

Example output

```
100.42 pushed to account
-5.58 pushed to account
50.78 pushed to account
-20.50 pushed to account

Final balance: $125.12
```

Part 4: STL Stack

STL Stack functions

`value_type& stack::top()`
Returns a reference to the top element in the stack. Since stacks are last-in first-out containers, the top element is the last element inserted into the stack.

`void stack::push( const value_type& val )`
Inserts a new element at the top of the stack, above its current top element. The content of this new element is initialized to a copy of val.

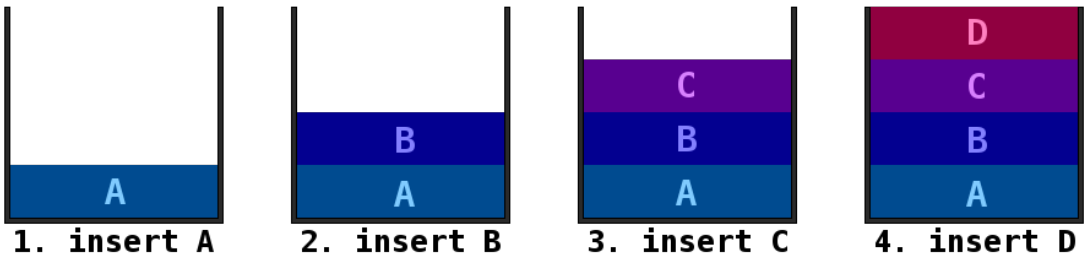
`void stack::pop()`
Removes the element on top of the stack, effectively reducing its size by one. The element removed is the latest element inserted into the stack, whose value can be retrieved by calling member stack::top.

`bool stack::empty()`
Returns whether the stack is empty: i.e. whether its size is zero.

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

A **stack** is similar to a queue in that access to its elements are also restricted, but what you can access is different.

Stacks are usually visualized vertically, like a Pringles tube, where items being added fall to the bottom. Because of this, the first item added ends up being below all later items.



To add a new item to the stack, you use `push(...)`
And to access the top-most item, you use `top()` .
To remove the top-most item, you use the `pop()` function.

View all of the stack's documentation at:

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

Starter code

Begin with the following code:

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

int main()
{
    bool done = false;

    cout << "Enter the next word of the sentence, or UNDO to undo,
    or DONE to stop." << endl;

    while ( !done )
    {
        string word;
        cout << ">> ";
        cin >> word;

        if ( word == "UNDO" )
        {
        }
        else if ( word == "DONE" )
        {
            done = true;
        }
        else
        {
        }
    }

    // Display stack of words
    cout << endl << endl << "Finished sentence: ";

    return 0;
}
```

---

main()

First in `main()`, before the while loop starts, create a stack of strings.

```
stack<float> sentence;
```

As the user enters words, we will **push** them onto the sentence.

Within the `if ( word == "UNDO" )` statement, display a message that you're removing the top-most item from the stack, and then use the `pop()` function to remove the top-most item.

Within the `else` statement, use the `push(...)` function to push the `word` onto the `sentence` stack.

Finally, after the while loop has finished, create a while loop that will *continue running while the* `sentence` *stack is not empty.* Within this loop:

- Display the top-most item with the `top()` function.
- Remove the top-most item with the `pop()` function.

Example output

```
Enter the next word of the sentence, or UNDO to undo, or DONE to
    stop.
>> up
>> you
>> let
>> UNDO
    Removed let
>> give
>> gonna
>> never
>> DONE


Finished sentence: never gonna give you up
```

Example output

```
Enter the next word of the sentence, or UNDO to undo, or DONE to
    stop.
>> a
>> b
>> c
>> UNDO
>> d
>> e
>> f
>> DONE


Finished sentence: f e d b a
```

Part 5: STL Map

STL Map functions

`value_type& map::operator[]( const key_type& k )`

If k matches the key of an element in the container, the function returns a reference to its mapped value. If k does not match the key of any element in the container, the function inserts a new element with that key and returns a reference to its mapped value. Notice that this always increases the container size by one, even if no mapped value is assigned to the element (the element is constructed using its default constructor).

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

The **map** type, sometimes known as a **dictionary** or **hash-table**, is a type of data-structure that relates some *key* to a *value*.

If we were looking at an average array, the keys would be `0, 1, 2, 3, ...` and so on. With a map, the key can be *any data-type*, and in *any order*.

For example, let's say we wanted to store an array of employees, but rather than have employees 0, 1, 2, 3, etc., we want to be able to access employees directly via their employee IDs, which are alphanumeric values.

key	value
hr-32917	Maryam A.
hr-38163	Colton B.
eng-632	Shelley C.
eng-192	Logan D.
qa-291	Elizabeth E.
qa-942	Liangyan F.

Once stored in a map, we can access the values (the employee names) via the key: `cout << employees[ "hr-32917" ] << endl;`

Starter code

Begin with the following code:

```
1 #include <iostream>
2 #include <string>
3 #include <map>
4 using namespace std;
5
6 int main()
```

```
7  {
8      while ( true )
9      {
10         string color;
11         cout << endl << "Enter a color, or QUIT to stop: ";
12         cin >> color;
13
14         if ( color == "QUIT" )
15         {
16             break;
17         }
18     }
19
20     return 0;
21 }
```

---

main()

At the start of `main()`, create a `map` named `colors`, whose key is a `string` and whose value is a `string`.

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

Initialize the map with the following values:

key	value		key	value
red	FF0000		magenta	FF00FF
green	00FF00		yellow	FFFF00
blue	0000FF		cyan	00FFFF

You can create the element simply by writing an assignment statement to set the value at a given key:

```
colors[ "red" ]         = "FF0000";
```

Next, after the user has entered a value for `color`, display the hex value for this color, again by using the map, and the key (as a variable) within the subscript operator [ ].

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

Example output

```
Enter a color, or QUIT to stop: red
Hex: FF0000

Enter a color, or QUIT to stop: green
Hex: 00FF00

Enter a color, or QUIT to stop: blue
Hex: 0000FF
```

Questions and Answers

Answer the following questions in a .txt file and turn them in with the rest of the lab.

1. General vocabulary
   (a) What is a **parameter** and an **argument?**
   (b) In relation to Arrays, what is an **element** and what is an **index?**
2. Vectors
   `http://www.cplusplus.com/reference/vector/vector/`
   (a) What function of the vector class will erase all of its elements?
   (b) What two functions can you use to add items to a vector?
3. Lists
   `http://www.cplusplus.com/reference/list/list/`
   (a) What function gives you the size of the list?
   (b) What function will reverse the items in a list?
4. Queues
   `http://www.cplusplus.com/reference/queue/queue/`
   (a) What is the function to add an item to the queue?
   (b) What is the function to remove the front item from the queue?
   (c) What is the function to access the front item of the queue?
   (d) What is the function to see whether the queue is empty?
5. Stacks
   `http://www.cplusplus.com/reference/stack/stack/`
   (a) What is the function to add an item to the stack?
   (b) What is the function to remove the top item from the stack?
   (c) What is the function to access the top item of the stack?
   (d) How are stacks and queues different?
6. Maps
   `http://www.cplusplus.com/reference/map/map/`
   (a) What is a **key** ?
   (b) What is a **value** ?
   (c) What are some other terms to refer to a map data structure?