16.11 — std::vector and stack behavior

Consider the case where you are writing a program where the user will enter a list of values (such as a bunch of test scores). In this case, the number of values that they will be entering is not known at compile time, and could vary every time they run the program. You will be storing these values in a `std::vector` for display and/or processing.

Based on what we've discussed so far, there are a few ways in which you might approach this:

First, you could ask the user how many entries they have, create a vector with that length, and then ask the user to enter that number of values.

This isn't a bad approach, but it requires the user to know exactly how many entries they have ahead of time, and to not have made a mistake in counting. Manually counting more than ten or twenty items can be tedious -- and why ask the user to count the number of values entered when we should be doing that for them?

Alternatively, we could just assume the user won't want to enter more than some number of values (e.g. 30), and create (or resize) a vector with that many elements. Then we can ask the user to enter data until they are done (or until they reach 30 entered values). Because the length of a vector is meant to be the number of used elements, we can then resize the vector to the number of values the user actually entered.

The downside of this approach is that the user is limited to entering 30 values, and we have no idea if that is too many or too few. If the user wants to enter more values, well, too bad.

We could address that problem by adding some logic to resize the vector larger whenever the user reaches the maximum number of values. But this means we're now having to mix array size management with our program logic, and that is going to increase the complexity of our program significantly (which will inevitably lead to bugs).

The real problem here is that we're trying to guess how many elements the user may enter, so we can manage the size of the vector appropriately. For situations where the number of elements to be entered is truly not known ahead of time, there is a better approach.

But before we can get there, we need to sidebar briefly.

What is a stack?

Analogy time! Consider a stack of plates in a cafeteria. For some unknown reason, these plates are extra heavy and can only be lifted one at a time. Because the plates are stacked and heavy, you can only modify the stack of plates in one of two ways:

1. Put a new plate on top of the stack (hiding the plate underneath, if it exists)
2. Remove the top plate from the stack (exposing the plate underneath, if it exists)

Adding or removing a plate from the middle or bottom of the stack is not allowed, as that would require lifting more than one plate at a time.

The order in which items are added to and removed from a stack can be described as **last-in, first-out (LIFO)**. The last plate added onto the stack will be the first plate that is removed.

Stacks in programming

In programming, a **stack** is a container data type where the insertion and removal of elements occurs in a LIFO manner. This is commonly implemented via two operations named **push** and **pop**:

Operation Name	Behavior	Required?	Notes
Push	Put new element on top of stack	Yes	
Pop	Remove the top element from the stack	Yes	May return the removed element or void

Many stack implementations optionally support other useful operations as well:

Operation Name	Behavior	Required?	Notes
Top or Peek	Get the top element on the stack	Optional	Does not remove item
Empty	Determine if stack has no elements	Optional	
Size	Count of how many elements are on the stack	Optional	

Stacks are common in programming. In lesson 3.9 -- Using an integrated debugger: The call stack, we discussed the call stack, which keeps track of which functions have been called. The call stack is… a stack! (I know, that reveal was a letdown). When a function is called, an entry with information about that function is added to the top of the call stack. When the function returns, the entry containing information about that function is removed from the top

of the call stack. In this way, the top of the call stack always represents the currently executing function, and each subsequent entry represents the function that was previously executing.

For example, here's a short sequence showing how pushing and popping on a stack works:

```
       (Stack: empty)
Push 1 (Stack: 1)
Push 2 (Stack: 1 2)
Push 3 (Stack: 1 2 3)
Pop    (Stack: 1 2)
Push 4 (Stack: 1 2 4)
Pop    (Stack: 1 2)
Pop    (Stack: 1)
Pop    (Stack: empty)
```

Stacks in C++

In some languages, a stack is implemented as its own discrete container type (separate from other containers). However, this can be quite limiting. Consider the case where we want to print all the values in a stack without modifying the stack. A pure stack interface does not provide a direct method to do this.

In C++, stack-like operations were instead added (as member functions) to the existing standard library container classes that support efficient insertion and removal of elements at one end (`std::vector`, `std::deque` and `std::list`). This allows any of these containers to be used as stacks in addition to their native capabilities.

As an aside…

The stack of plates analogy is a good one, but we can make a better analogy that will help us understand how a stack can be implemented using an array. Instead of a stack of plates that can vary in the number of plates it is currently holding, consider a column of mailboxes, all stacked on top of each other. Each mailbox can only hold one item, and all mailboxes start out empty. Each mailbox is nailed to the mailbox below it, and the top of the column is covered in poisonous spikes, so no new mailboxes can be inserted anywhere.

If we can't change the number of mailboxes, how do we get a stack-like behavior?

First, we use a marker (like a post-it note) to keep track of where the top of the stack is -- this will always be the lowest empty mailbox. At the start, the stack is empty, so the marker goes on the bottom mailbox.

When we push an item onto our mailbox stack, we put it in the mailbox that is marked (which is the lowest empty mailbox), and move the marker up one mailbox. When we pop an item off the stack, we move the marker down one mailbox (so it's pointed at the top non-empty

mailbox) and remove the item from that mailbox so that it is now empty.

Items below the marker are considered “on the stack”. Items at or above the marker are not on the stack.

Now, call the marker `length`, and the number of mailboxes `capacity`…

In the rest of this lesson, we’ll examine how the stack interface of `std::vector` works, and then we’ll conclude by showing how it helps us solve the challenge introduced at the top of the lesson.

Stack behavior with `std::vector`

Stack behavior in `std::vector` is implemented via the following member functions:

Function Name	Stack Operation	Behavior	Notes
push_back()	Push	Put new element on top of stack	Adds the element to end of vector
pop_back()	Pop	Remove the top element from the stack	Returns void, removes element at end of vector
back()	Top or Peek	Get the top element on the stack	Does not remove item
emplace_back()	Push	Alternate form of push_back() that can be more efficient (see below)	Adds element to end of vector

Let’s take a look at an example that uses some of these functions:

```
#include <iostream>
#include <vector>

void printStack(const std::vector<int>& stack)
{
	if (stack.empty()) // if stack.size == 0
		std::cout << "Empty";

	for (auto element : stack)
		std::cout << element << ' ';

	// \t is a tab character, to help align the text
	std::cout << "\tCapacity: " << stack.capacity() << "  Length " <<
stack.size() << "\n";
}

int main()
{
	std::vector<int> stack{}; // empty stack

	printStack(stack);

	stack.push_back(1); // push_back() pushes an element on the stack
	printStack(stack);

	stack.push_back(2);
	printStack(stack);

	stack.push_back(3);
	printStack(stack);

	std::cout << "Top: " << stack.back() << '\n'; // back() returns the last
element

	stack.pop_back(); // pop_back() pops an element off the stack
	printStack(stack);

	stack.pop_back();
	printStack(stack);

	stack.pop_back();
	printStack(stack);

	return 0;
}
```

On GCC or Clang, this prints:

```
Empty   Capacity: 0  Length: 0
1       Capacity: 1  Length: 1
1 2     Capacity: 2  Length: 2
1 2 3   Capacity: 4  Length: 3
Top:3
1 2     Capacity: 4  Length: 2
1       Capacity: 4  Length: 1
Empty   Capacity: 4  Length: 0
```

Remember, length is the number of elements in the vector, which in this case, is the number of elements on our stack.

Unlike the subscript operator `operator[]` or the `at()` member function, `push_back()` (and `emplace_back()`) will increment the length of the vector, and will cause a reallocation to occur if the capacity is not sufficient to insert the value.

In the example above, the vector gets reallocated 3 times (from a capacity of 0 to 1, 1 to 2, and 2 to 4).

Key insight

`push_back()` and `emplace_back()` will increment the length of a `std::vector`, and will cause a reallocation to occur if the capacity is not sufficient to insert the value.

Extra capacity from pushing

In the above output, note that when the last of the three reallocations occurred, the capacity jumped from 2 to 4 (even though we were only pushing one element). When pushing triggers a reallocation, `std::vector` will typically allocate some extra capacity to allow additional elements to be added without triggering another reallocation the next time an element is added.

How much extra capacity is allocated is left up to the compiler's implementation of `std::vector`, and different compilers typically do different things:

- GCC and Clang doubles the current capacity. When the last resize is triggered, the capacity is doubled from 2 to 4.
- Visual Studio 2022 multiplies the current capacity by 1.5. When the last resize is triggered, the capacity is changed from 2 to 3.

As a result, the prior program may have a slightly different output depending on what compiler you are using.

Resizing a vector doesn't work with stack behavior

Reallocating a vector is computationally expensive (proportional to the length of the vector), so we want to avoid reallocations when reasonable to do so. In the example above, we could avoid the vector being reallocated 3 times if we manually resized the vector to capacity 3 at the start of the program.

Let's look at what happens if we change line 18 in the above example to the following:

```
std::vector<int> stack(3); // parenthesis init to set vector's capacity to 3
```

Now when we run the program again, we get the following output:

```
0 0 0   Capacity: 3  Length: 3
0 0 0 1         Capacity: 4  Length: 4
0 0 0 1 2       Capacity: 6  Length: 5
0 0 0 1 2 3     Capacity: 6  Length: 6
Top: 3
0 0 0 1 2       Capacity: 6  Length: 5
0 0 0 1         Capacity: 6  Length: 4
0 0 0   Capacity: 6  Length: 3
```

That's not right -- somehow we have a bunch of `0` values at the start of our stack! The issue here is that parenthesis initialization (to set the initial size of the vector) and the `resize()` function set both the capacity AND the length. Our vector is starting with a capacity of 3 (which is what we wanted), but the length is also being set to 3. So our vector is starting with 3 elements having value 0. The elements we push on later are pushed on top of these initial elements.

The `resize()` member function changing the length of the vector is fine when we're intending to use subscripts to access elements (since our indices need to be less than the length to be valid), but it causes problems when we're using the vector as a stack.

What we really want is some way to change the capacity (to avoid future reallocations) without changing the length (which has the side effect of adding new elements to our stack).

The `reserve()` member function changes the capacity (but not the length)

The `reserve()` member function can be used to reallocate a `std::vector` without changing the current length.

Here's the same example as before, but with an added call to `reserve()` to set the capacity:

```
#include <iostream>
#include <vector>

void printStack(const std::vector<int>& stack)
{
        if (stack.empty()) // if stack.size == 0
                std::cout << "Empty";

        for (auto element : stack)
                std::cout << element << ' ';

        // \t is a tab character, to help align the text
        std::cout << "\tCapacity: " << stack.capacity() << "  Length " <<
stack.size() << "\n";
}

int main()
{
        std::vector<int> stack{};

        printStack(stack);

        stack.reserve(6); // reserve space for 6 elements (but do not change length)
        printStack(stack);

        stack.push_back(1);
        printStack(stack);

        stack.push_back(2);
        printStack(stack);

        stack.push_back(3);
        printStack(stack);

        std::cout << "Top: " << stack.back() << '\n';

        stack.pop_back();
        printStack(stack);

        stack.pop_back();
        printStack(stack);

        stack.pop_back();
        printStack(stack);

        return 0;
}
```

On the author’s machine, this prints:

```
Empty   Capacity: 0  Length: 0
Empty   Capacity: 6  Length: 0
1       Capacity: 6  Length: 1
1 2     Capacity: 6  Length: 2
1 2 3   Capacity: 6  Length: 3
Top: 3
1 2     Capacity: 6  Length: 2
1       Capacity: 6  Length: 1
Empty   Capacity: 6  Length: 0
```

You can see that calling `reserve(6)` changed the capacity to 6, but did not affect the length. No more reallocations occur because the std::vector is large enough to accommodate all of the elements we are pushing.

Key insight

The `resize()` member function changes the length of the vector, and the capacity (if necessary).
The `reserve()` member function changes just the capacity (if necessary)

Tip

To increase the number of elements in a `std::vector`:
Use `resize()` when accessing a vector via indexing. This changes the length of the vector so your indices will be valid.
Use `reserve()` when accessing a vector using stack operations. This adds capacity without changing the length of the vector.

`push_back()` vs `emplace_back()`

Both `push_back()` and `emplace_back()` push an element onto the stack. If the object to be pushed already exists, `push_back()` and `emplace_back()` are equivalent, and `push_back()` should be preferred.

However, in cases where we are creating a temporary object (of the same type as the vector's element) for the purpose of pushing it onto the vector, `emplace_back()` can be more efficient:

```
#include <iostream>
#include <string>
#include <string_view>
#include <vector>

class Foo
{
private:
    std::string m_a{};
    int m_b{};

public:
    Foo(std::string_view a, int b)
        : m_a { a }, m_b { b }
        {}

    explicit Foo(int b)
        : m_a {}, m_b { b }
        {};
};

int main()
{
        std::vector<Foo> stack{};

        // When we already have an object, push_back and emplace_back are similar in
efficiency
        Foo f{ "a", 2 };
        stack.push_back(f);    // prefer this one
        stack.emplace_back(f);

        // When we need to create a temporary object to push, emplace_back is more
efficient
        stack.push_back({ "a", 2 }); // creates a temporary object, and then copies
it into the vector
        stack.emplace_back("a", 2);  // forwards the arguments so the object can be
created directly in the vector (no copy made)

        // push_back won't use explicit constructors, emplace_back will
        stack.push_back({ 2 }); // compile error: Foo(int) is explicit
        stack.emplace_back(2);  // ok

        return 0;
}
```

In the above example, we have a vector of `Foo` objects. With `push_back({ "a", 2 })`, we're creating and initializing a temporary `Foo` object, which then gets copied into the vector. For expensive to copy types (like `std::string`) this copy can result in a performance hit.

With `emplace_back()`, we don't need to create a temporary object to pass. Instead, we pass the arguments that would be used to create the temporary object, and `emplace_back()` forwards them (using a feature called perfect forwarding) to the vector, where they are used to create and initialize the object inside the vector. This avoids a copy that would have otherwise been made.

Of note, `push_back()` won't use explicit constructors, whereas `emplace_back()` will. This makes `emplace_back` more dangerous, as it is easier to accidentally invoke an explicit constructor to perform some conversion that doesn't make sense.

Prior to C++20, `emplace_back()` doesn't work with aggregate initialization.

Best practice

Prefer `emplace_back()` when creating a new temporary object to add to the container, or when you need to access an explicit constructor.

Prefer `push_back()` otherwise.

This article has more explanation for this best practice.

Addressing our challenge using stack operations

It should now be obvious how we should address the challenge introduced at the top of the lesson. If we don't know in advance how many elements will be added to our `std::vector`, using the stack functions to insert those elements is the way to go.

Here's an example:

```
#include <iostream>
#include <limits>
#include <vector>

int main()
{
	std::vector<int> scoreList{};

	while (true)
	{
		std::cout << "Enter a score (or -1 to finish): ";
		int x{};
		std::cin >> x;

		if (!std::cin) // handle bad input
		{
			std::cin.clear();
			std::cin.ignore(std::numeric_limits<std::streamsize>::max(),
'\n');
			continue;
		}

		// If we're done, break out of loop
		if (x == -1)
			break;

		// The user entered a valid element, so let's push it on the vector
		scoreList.push_back(x);
	}

	std::cout << "Your list of scores: \n";

	for (const auto& score : scoreList)
		std::cout << score << ' ';

	return 0;
}
```

This program lets the user enter test scores, adding each score to a vector. After the user has finished adding scores, all the values in the vector are printed.

Note how in this program, we don’t have to do any counting, indexing, or deal with array lengths at all! We can just focus on the logic of what we want the program to do, and let the vector handle all of the storage issues!

Quiz time

Question #1

Write a program that pushes and pops values, and outputs the following sequence:

```
       (Stack: empty)
Push 1 (Stack: 1)
Push 2 (Stack: 1 2)
Push 3 (Stack: 1 2 3)
Pop    (Stack: 1 2)
Push 4 (Stack: 1 2 4)
Pop    (Stack: 1 2)
Pop    (Stack: 1)
Pop    (Stack: empty)
```

Show Solution