STL Containers, Part I

Introduction

Arguably the most powerful library in the C++ language, the Standard Template Library (STL) is a programmer's dream. It offers quick, efficient ways to store, access, and manipulate data and is designed for maximum extensibility. However, as you might expect out of such a powerful library, the STL is relatively complicated and takes a lot of practice to adjust to. Over the next two weeks, we'll cover a sizable chunk of the STL, starting with container classes and concluding with algorithms.

STL Containers vs CS106 ADTs

You're probably just getting used to the CS106 ADTs, so switching over to the STL container classes might be a bit jarring. Fortunately, for almost all of the CS106 ADTs there is a corresponding STL container class (the notable exception is `Grid`). Most of the time, the STL versions of the CS106 ADTs have the same name, except in lowercase: for example, the CS106 `Vector` is analogous to the STL `vector`. Furthermore, anything you can do with the CS106 ADTs you can also do with the STL container classes, although the syntax might be more cryptic. The main advantages of the STL might not be immediately apparent, but next weeks when we cover STL algorithms you'll see exactly how much the STL's capability dwarfs that of the CS106 ADT library.

To use the STL containers, you'll need to use a different set of headers than those you'll use for the CS106 ADTs. The header files for the STL use angle brackets (< and >) and don't end in .h. For example, the header for the STL `vector` is `<vector>`, instead of the CS106 `"vector.h"`. And don't forget `using namespace std`!

A Word on Safety

The biggest difference between the CS106 library and the STL container classes is safety. With the CS106 libraries, if you try to access the tenth element of a nine-element `Vector`, you'll get an error message alerting you to the fact and your program will terminate. With the STL, doing this results in *undefined behavior*. This means that anything could happen – your program might continue with garbage data, it might crash, or it might even cause yellow ducks to show up on the screen and start dancing. What's important to note, though, is that this means that the STL can easily introduce nasty bugs into your code. Most of the time, if you read one spot over the end in an STL `vector`, nothing serious will happen, but when something bad does come up you'll get crashes that can be tricky to track down.

On the plus side, however, this lack of error checking means the STL container classes are much, *much* faster than anything you'll get from the CS106 libraries. In fact, the STL is so fast that it's reasonable to assume that if there's an STL way to accomplish a task and a non-STL way to accomplish the task, the STL way is going to be faster.*

---

* Actually, this is true of almost all the C++ Standard Library functions and classes and is a good lesson to take to heart. Except for practice and experience, avoid "reinventing the wheel" in your code.

What does this all mean? Basically, you're going to have to stop relying on user-friendly error messages as a means for debugging your code. If you have bounds-checking issues with your STL containers, or try to pop a nonexistent element off of a `stack`, you're probably going to be directed to a hideous-looking implementation file where the error occurred. It's the price you pay for efficiency.

A Word on Compiler Errors

Another issue that deserves mention are the compiler errors you're going to get when writing anything with the STL. Normally, if you leave a semicolon off a line, or try to assign a `Vector<string>` to a `Map<string>`, you'll get error messages that are somewhat comprehensible. Not so with the STL. In fact, STL errors are so difficult to read that it's going to take you a while before you'll have any idea what you're looking at.

When you get STL compiler errors, make sure to take the time to slowly read them and to compare the different types that they mention. If you have a type mismatch, it won't be immediately obvious, but it will be there. Similarly, if you try to call a member function that doesn't exist in a container class, be prepared to spend a few seconds reading what type the compiler reports before you even get to the member function that caused the error.

`stack`

Perhaps the simplest STL container is the `stack`. The STL `stack` is similar to the CS106 `Stack` in most ways – it's a last-in, first-out (LIFO) container that supports `push` and `pop`; you can test to see if it's empty and, if not, how tall it is; and you can peek at the top element. However, there are a few subtle differences between the CS106 `Stack` and the STL `stack`, as you'll see in a minute.

The following table summarizes the member functions you can perform on a `stack`. Again, we have not covered the `const` keyword yet, so for now feel free to ignore it.

CS106 `Stack`	STL `stack`	Differences
`int size()`	`size_type size() const`	Although the STL stack's return type is `size_type` instead of `int`, these functions behave identically.
`bool isEmpty()`	`bool empty() const`	Don't get confused by the name: `empty` does *not* empty out a stack. The STL uses the function name `empty` instead of the CS106 `isEmpty`, but they do exactly the same thing.
`T& peek()`	`T& top()` `const T& top() const`	These functions do exactly the same thing, except that the STL `top` function does not perform any bounds-checking.
`void push(T toAdd)`	`void push(const T& toAdd)`	Basically identical.
`T pop();`	`void pop();`	See the next section
`void clear();`	-	There is no STL `stack` equivalent of `clear`.

As you can see, most of the functions from the CS106 `Stack` work the same way in the STL `stack`, albeit with a few syntactic differences. The big difference, however, is the `pop` function. With the CS106 `Stack`, you can write code like this:

```
int topElem = myCS106Stack.pop();
```

This is *not* the case with the STL `stack`. The STL function `pop` removes the top element from the stack but *does not* return a value. Thus, an STL version of the above code might look something like this:

```
int topElem = mySTLStack.top();
mySTLStack.pop();
```

While this may make your code slightly less readable, it will improve your runtime speed because the `top` function doesn't create a copy of the object on top of the `stack`. It's much faster to return a reference to an object than a copy of it, and if your `stack` stores non-primitive types the performance boost over the CS106 ADTs can be astounding.

`queue`

Another simple STL container class is the `queue`, which behaves much the same way as the CS106 `Queue`. Here's another summary table:

CS106 `Queue`	STL `queue`	Differences
`int size()`	`size_type size() const`	Although the STL `queue`'s return type is `size_type` instead of `int`, these functions behave in the same way.
`bool isEmpty()`	`bool empty() const`	See the `stack` chart for words of warning.
`T& peek()`	`T& front()` `const T& front() const`	These functions do exactly the same thing, except that the STL `front` function does not perform any bounds-checking.
`void enqueue(T toAdd)`	`void push(const T& toAdd)`	Although it's irksome that the `queue` uses "push" and "pop" instead of "enqueue" and "dequeue," these functions are the same.
`T dequeue();`	`void pop();`	As with the `stack`, `pop` does not return the value it removes.
`void clear()`	-	There is no `queue` "clear" function.
-	`T& back();` `const T& back() const`	Returns a reference to the back of the `queue`, which is the last element that was `push`ed on.

`vector`

Fortunately (or unfortunately, depending on how you look at it) for those of you who thought that the STL was exactly the same as the CS106 libraries with different function names, the only two "simple" STL containers are `stack` and `queue`.* It's now time to deal with the STL `vector`, one of the STL's most ubiquitous containers. In this next section we'll talk about some basic `vector` functionality, and at the end I'll provide a lengthy table listing most of the important member functions.

Fixed-sized `vector`s

The simplest way to use a `vector` is to treat it as though it's a managed, fixed-size array. For now, we'll ignore the `vector`'s ability to dynamically resize itself and instead focus on the functions you can use to access the `vector`.

---

* There is a third "simple" container, the `priority_queue`, which we will cover in several weeks.

When you create a vector, you can set its default size as a parameter to the constructor like this:

```
vector<int> myIntVector;      // vector of size 0
vector<int> myIntVector(10);  // vector of size 10, elements set to zero.
vector<int> myIntVector(50, 137);   // size 50, all elements are 137.
vector<string> myStrings(4, "blank");  // size 4, elements set to "blank"
```

Note that this is different from the CS106 ADT `Vector`. With the CS106 `Vector`, if you write code like this:

```
Vector<int> myIntVector(100);
```

you are *not* creating a `Vector` with 100 elements. Instead you're providing a size hint about the expected size of the `Vector`. Thus, if you're porting code from CS106 to the STL equivalents, make sure to watch out for this discrepancy – it won't work the way you want it to!

Now, of course, a `vector` can't do much anything if you can't access any of its data. Fortunately, the STL `vector` is much like the CS106 `Vector` in that you can access its data in two ways, as shown below:

```
vector<int> myInts(10);
myInts[0] = 10;     // Set the first element to 10.
myInts.at(0) = 10;  // Set the first element to 10.
int ninthElement = myInts[9];
int ninthElement = myInts.at(9);
```

There is a subtle difference between the two syntaxes, but for now it's safe to ignore it. Just remember that in both cases, if you go off the ends of the array, you're likely to get your program to crash, so make sure to bounds-check it!

As with the `stack` and `queue`, the `vector` uses `size` and `empty` to return the size and determine if the `vector` is empty, respectively. For example:

```
vector<int> myVector(10);
for(vector<int>::size_type h = 0; h < myVector.size(); h++)
      myVector[h] = h;
```

In this above code, note that I used the type `vector<int>::size_type` as an iterating parameter instead of the more traditional `int`. As with `string`s, all STL container classes define a `size_type` type because they cannot contain negative numbers of entries. In nearly every case it's completely safe to ignore the distinction and use `int`s for tracking these variables, but it's good to know the distinction since the compiler might complain about "implicit type conversions with possible losses of data."

If you want to clear out all of the entries of a `vector`, use the `clear` member function, which behaves the same way as the CS106 ADT `clear` member functions:

```
vector<int> myVector(1000);
cout << myVector.size() << endl;    // Prints 1000
myVector.clear();
cout << myVector.size() << endl;    // Prints 0
```

Variable-sized `vectors`

These above functions are indeed useful, but they treat the vector as though it was simply a fixed-sized array. However, the real power of the `vector` shows up when you resize it. The simplest way to resize a `vector` is with the `resize` member function. `resize` has the same syntax as the `vector` constructor; that is, you can specify only a target size, or both a target size and a default value for any inserted elements. `resize` can be used both to expand and shrink a `vector`. If the `vector` expands, any new elements are appended to the end. If it shrinks, they're deleted off the end.

(For the examples below, assume we have a `PrintVector` function that takes in a `vector<int>` and prints it to the screen.*)

```
vector<int> myVector;   // Defaults to empty vector
PrintVector(myVector);  // Output: [nothing]
myVector.resize(10, 0); // Grow the vector, setting new elements to 0
PrintVector(myVector);  // Output: 0 0 0 0 0 0 0 0 0 0
myVector.resize(5);     // Shrink the vector
PrintVector(myVector);  // Output: 0 0 0 0 0
myVector.resize(7, 1);  // Grow the vector, setting new elements to 1
PrintVector(myVector);  // Output: 0 0 0 0 0 1 1
myVector.resize(1, 7);  // The second parameter is effectively ignored.
PrintVector(myVector);  // Output: 0
```

Of course, sometimes you only want to add one element to the `vector`. If that new element is at the end of the `vector`, you can use the `push_back` method to "push" an element onto the back of the `vector`. This is equivalent to the CS106 `add` function. For example:

```
vector<int> myVector;
for(int i = 0; i < 10; i++)
      myVector.push_back(i);
PrintVector(myVector);  // Output: 0 1 2 3 4 5 6 7 8 9
```

Similarly, to take an element off the back of a `vector`, you can use `pop_back`, with behavior almost identical to that of the `stack`'s `pop` member function.

```
vector<int> myVector(10, 0);
while(!myVector.empty())
      myVector.pop_back();
```

The `vector` also has a member function `back` that returns a reference to the last element in the `vector`.

`push_back` and `pop_back` are nice, but what if you want to insert an element into an arbitrary point in the `vector`? It turns out that it is possible to do this, but the syntax is a bit weird. Next week, when we cover iterators, the syntax will make more sense, but for now, please just trust me that this works.

To insert an element into a `vector` at an arbitrary point, use this syntax:

```
myVector.insert(myVector.begin() + n, element);
```

---

* For now you can just use a simple for loop to do this. Next week, we'll see a much cooler way that involves iterators and algorithms.

Here, `n` represents the index at which you want to insert the element. So, for example, to insert the number 10 at the very beginning of an integer `vector`, you'd write

```
myVector.insert(myVector.begin(), 10);
```

Just as you can use `resize` to grow a `vector` and fill in the newly added elements, you can use `insert` to insert multiple copies of a single element by using this syntax:

```
myVector.insert(myVector.begin() + n, numCopies, element);
```

For example, to insert five copies of the number 137 at the start of a `vector`, you could use the syntax `myVector.insert(myVector.begin(), 5, 137)`.

To remove a single element from a random point in a `vector`, use the `erase` method as follows:

```
myVector.erase(myVector.begin() + n);
```

where `n` represents the index of the element to erase. If you want to remove elements in the range [start, stop), you can alternatively use this syntax:

```
myVector.erase(myVector.begin() + start, myVector.begin() + stop);
```

The following table summarizes most of the important member functions of the `vector`:

Again, we haven't covered `const` yet, so it's safe to ignore it for now. We also haven't covered iterators yet, so don't feel scared when you see iterator functions.

Constructor: `vector<T> ()`	`vector<int> myVector;` Constructs an empty `vector`.
Constructor: `vector<T> (size_type size)`	`vector<int> myVector(10);` Constructs a `vector` of the specified size where all elements use their default values (for integral types, this is zero).
Constructor: `vector<T> (size_type size, const T &default)`	`vector<string> myVector(5, "blank");` Constructs a `vector` of the specified size where each element is equal to the specified default value.
`size_type size() const;`	`for(int i = 0; i < myVector.size(); i++) { ... }` Returns the number of elements in the `vector`.
`bool empty() const;`	`while(!myVector.empty()) { ... }` Returns whether the `vector` is empty.
`void clear();`	`myVector.clear();` Erases all the elements in the `vector` and sets the size to zero.

`vector` functions, contd.

`T& operator [] (size_type position);`
`const T& operator [] (size_type position) const;`
 	
`T& at(size_type position);`
`const T& at(size_type position) const;`
	`myVector[0] = 100;`
	`int x = myVector[0];`
	`myVector.at(0) = 100;`
	`int x = myVector.at(0);`
	
	Returns a reference to the element at the specified position. The bracket notation [] does not do any bounds checking and has undefined behavior past the end of the data. The `at` member function will throw an exception if you try to access data beyond the end. We will cover exception handling later in the quarter.

`void resize(size_type newSize);`
`void resize(size_type newSize, T fill);`
	`myVector.resize(10);`
	`myVector.resize(10, "default");`
	
	Resizes the `vector` so that it's guaranteed to be the specified size. In the second version, the `vector` elements are initialized to the value specified by the second parameter. Elements are added to and removed from the end of the `vector`, so you can't use `resize` to add elements to or remove elements from the start of the `vector`.

`void push_back();`	`myVector.push_back(100);` Appends an element to the `vector`.

`T& back();`
`const T& back() const;`
	`myVector.back() = 5;`
	`int lastElem = myVector.back();`
	
	Returns a reference to the last element in the `vector`.

`T& front();`
`const T& front() const;`
	`myVector.front() = 0;`
	`int firstElem = myVector.front();`
	
	Returns a reference to the first element in the `vector`.

`void pop_back();`	`myVector.pop_back();` Removes the last element from the `vector`, decreasing its size by 1.
`iterator begin();` `const_iterator begin() const;`	`vector<int>::iterator itr = myVector.begin();` Returns an iterator that points to the first element in the `vector`.
`iterator end();` `const_iterator end() const;`	`while(itr != myVector.end());` Returns an iterator that points the element *after* the last. That is, the iterator returned by `end` does not point to an element in the `vector`.

`iterator insert( iterator position, const T& value);`
`void insert( iterator start, size_type numCopies, const T& value);`
	`myVector.insert(myVector.begin() + 4, "Hello");`
	`myVector.insert(myVector.begin(), 2, "Yo!");`
	
	The first version inserts the specified value into the `vector`, and the second inserts `numCopies` copies of the value into the `vector`. Both calls invalidate all outstanding iterators for the `vector`.

`vector` functions, contd.

`iterator erase(iterator position);`
`iterator erase(iterator start,`
`iterator end);`
	`myVector.erase(myVector.begin());`
	`myVector.erase(startItr, endItr);`
	
	The first version erases the element at the position pointed to by `position`. The second version erases all elements in the range [`startItr`, `endItr`). Note that this does **not** erase the element pointed to by `endItr`.

As you can see, the `vector` is very powerful and has a whole bunch of useful functions. Of course, there are more that I didn't document, so be sure to consult a reference for some of the other things you can do with a `vector`.

deque

The final container we'll cover today is the `deque`. `deque` is pronounced "deck," as in a deck of cards, and not "dequeue," the queue's remove element function. Originally, `deque` stood for "**d**ouble-**e**nded **que**ue," but now is most commonly implemented like a deck of cards, hence the pronunciation.

What's interesting about a `deque` is that anything you can do with a `vector` you can also do with a `deque`. Any of the above listed functions also work on `deque`s. However, `deque`s also support two more functions, `push_front` and `pop_front`, which work like the `vector`'s `push_back` and `pop_back` except that they insert and remove elements from the front of the `deque`. In case this doesn't seem familiar, it's some of the behavior exhibited by the `stack`'s `push` and `pop`. In fact, this is not a coincidence – the `stack` is actually implemented as a wrapper class that manipulates a `deque`. Because the `stack` "adapts" the `deque` from one data type to another, the `stack` is technically not a container, but a "container adapter." The same is true of the `queue`.

If `deque` has more functionality than `vector`, why don't people use it instead of `vector`? The main reason is speed. `deque`s and `vector`s are implemented in two different ways – `deque`s maintain a list of different "pages" that store information, while `vector`s manage single arrays. This means that accessing single elements is slightly faster in a `vector`, but insertions and deletions to and from a `deque` are significantly faster than in a `vector`. Also, the `deque`'s destructor takes a considerable amount of time to clean up the memory it allocates, while the `vector` can almost instantly dispose of its single internal array.

If you ever find yourself about to use a `vector`, check to see what you're doing with it. If you need to optimize for fast access, keep using a `vector`. If you're going to be inserting or deleting elements frequently, consider using a `deque` instead. Also, if you plan on making and deleting many container objects, stay away from the `deque` because the cleanup time is much greater.

More To Explore

There's so much to explore with the STL that we could spend all of our lectures teaching the STL without ever running out of material. If you're interested in some more advanced topics relating to this material and the STL in general, consider reading on these topics:

1. **`bitset`**: An STL container that might be worth reading on is the `bitset`, a container that manages a set of bits. It allows you to store binary flags in a very small space and without too much hassle. Effectively, it's a class that does bitwise manipulations for you. Those of you with a C background should especially consider looking into this one.
2. **`vector::capacity` and `vector::reserve`**: Because the `vector` manages an internal array that grows in response to requests for more space, sometimes insertions on the `vector` can take a very long time as the buffer has to be resized. Using the `reserve` member function, however, you can tell the `vector` in advance how much space you plan to use. This lets you reduce the overhead time for insertions and resize operations. The `capacity` member function reports how much space the `vector` is currently capable of managing without a resize, and you can use this function to determine whether to reserve space.
3. **`valarray`**: The `valarray` class is similar to a `vector` in that it's a managed array that can hold elements of any type. However, unlike `vector`, `valarray` is designed for numerical computations. `valarray`s do not support insertions or deletions, but have intrinsic and quick support for mathematical operators. For example, you can use the syntax `myValArray *= 2` to multiply all of the entries in a `valarray` by two. If you're interested in numeric or computational programming, consider looking into the `valarray`.
4. There's an excellent article online comparing the performances of the `vector` and `deque` containers. If you're interested, you can see it at http://www.codeproject.com/vcpp/stl/vector_vs_deque.asp.

Practice Questions

Everything we've covered this lecture primarily concerns the basic syntax of STL containers and thus doesn't lend itself to "practice problems" in the traditional sense. However, I highly recommend that you try some of the following exercises:

1. Take your code from the ADT Client assignment from CS106B and CS106X and rewrite it using the STL. This should help you get a feel for the STL without having to worry about tackling new algorithmic challenges.
2. Deliberately write code with the STL that contains syntax errors and look at the error messages you get. See if you have any idea what they say, and, if not, practice until you can get a feel for what's going on. If you want a real mess, try making mistakes with `vector<string>`.
3. Using the time trial code from lecture (available at the CS106L website), compare the performances of the `vector`'s bracket operator syntax (e.g. `myVector[0]`) to the `vector`'s `at` function (e.g. `myVector.at(0)`). Given that the `at` function is bounds-checked, how serious are the runtime penalties of bounds-checking?