CS106L: **STL Containers and Adaptors**

Lecture Notes by Sathya Edamadaka, Frankie Cerkvenik

September 30, 2021

Abstract

We'll open by motivating why we need to care about containers in the first place. We'll then discuss containers provided to us by the STL (Standard Template Library). We'll compare and contrast sequence and associative containers. Lastly, we'll go through in-depth performance evaluations of each function, and we'll leave you with a solid understanding of when to use which data structure.

1 Why care about C++ data structures?

- We need data structures (ds) to store data in programs. `structs`, as we've discussed in the past two lectures, are one way to store different variables. So what do we do when running into extremely common problems, like storing a list of variables or trying to create a queue. As a (bad) example, for the former, should we create a struct with 1000 elements and hope no one uses more than 1000 elements?

- It looks like we need a way to store data in several ways, each needing to accommodate for an *unknown* amount of data. We could have 3 elements in a list, or 100,000! Our data structures need to be able to account for that.

- Here are some classic examples of data structures in two languages you may be familiar with! A few common data structures in Python are shown below:

```
lst = []
dictionary = {}
hash_table = set()
string = "adc"
```

  and here are a few in Stanford's C++ Library!

```
Vector<int> lst;
Map<int, int> dictionary;
Set<int> hash_table;
std::string str = "adc"; //
```

  If you're wondering, a std::string can be considered as a data structure, as it stores any number of elements (characters), also in any order!

- If we were to design a queue (think of a grocery store checkout line), which 4 methods (functions) does this queue *need* to have? This is a great chance to practice data structure and class design (i.e. given a certain real-life situation, how would you represent it in code). The answer is that it would need:

  1. a method to enqueue elements to the back of the queue,

2. a method to dequeue elements from the front of the queue,
3. a method to check the size of the queue, and
4. a method to check which element is at the front of the queue.

- Like we hinted at earlier, we've just designed an extremely generalizable data structure. In other words, although we can apply it to representing our specific grocery store shopping line, we can also reuse it for many other situations. In fact, it seems like many other programmers would have similar situations in which they'd need a queue, e.g.:
  - for representing a car wash
  - for representing a burst of API requests
- What if there was a standard library where hundreds of similarly generalized, extremely well-tested data structures (called containers) were available to all C++ programmers?

2 Introducing the Standard Template Library (and Containers)

- Introducing C++'s Standard Template Library, or STL! There are four main components to the STL:
  1. Containers
  2. Iterators
  3. Functions
  4. Algorithms
- We'll discuss each part of the STL in a different lecture. Today, we'll talk about containers. We'll discuss iterators next lecture, take a break to discuss how all of these are implemented through template classes and classes, and return to talk about STL functions and algorithms.
- Let's dive into the types of "containers" (any data structure that stores a collection of other elements) given to us in the STL, how to use each one, and when you'd utilize them.
- Sequence containers are all variants of lists. They all store elements in some order, but vary in how programmers can access the elements.
- As a gentle introduction to sequence containers, let's compare Stanford's C++ and Standard C++ vectors (on the next page)! Vectors are the most fundamental sequence container.

When using STL containers, you usually need to add an import statement at the beginning of your file. Then, make sure to either type std::NameOfContainerHere, or a "using std::NameOfContainerHere;" at the beginning of the file. Below is a quick example!

```
#import <vector>

int main () {
    std::vector<int> vec;
    ...
}
```

What you want to do	Stanford `Vector<int>`	`std::vector<int>`
Create a new, empty vector	`Vector<int> vec;`	`std::vector<int> vec;`
Create a vector with `n` copies of 0	`Vector<int> vec(n);`	`std::vector<int> vec(n);`
Create a vector with `n` copies of a value `k`	`Vector<int> vec(n, k);`	`std::vector<int> vec(n, k);`
Add a value `k` to the end of a vector	`vec.add(k);`	`vec.push_back(k);`
Remove all elements of a vector	`vec.clear();`	`vec.clear();`
Get the element at index `i`	`int k = vec[i];`	`int k = vec[i];` (does **not** bounds check)
Check size of vector	`vec.size();`	`vec.size();`
Loop through vector by index `i`	`for (int i = 0; i < vec.size(); ++i) ...`	`for (std::size_t i = 0; i < vec.size(); ++i) ...`
Replace the element at index `i`	`vec[i] = k;`	`vec[i] = k;` (does **not** bounds check)

What you want to do	Stanford `Vector<int>`	`std::vector<int>`
Insert k at some index i	`vec.insert(i, k);`	`vec.insert(vec.begin() + i, k);`
Remove the element at index i	`vec.remove(i);`	`vec.erase(vec.begin() + i);`
Get the sublist in range [i, j)	`v.subList(i, j);`	`std::vector<int> sum (vec.begin() + i, vec.begin() + j);`
Create a vector that is two vectors appended to each other	`Vector<int> v = v1 + v2;`	`// pretty complicated ngl`
Add j to the front of a vector	`vec.insert(0, i);`	`vec.insert(vec.begin(), k);`

- Here are a few more sequence containers.

Name of Container	Description	Implementation
`std::vector`	A vector is the most common container you'll use. It's comparable to a list in Python and Arraylist in Java. It can hold any amount of elements. You can insert elements at any index, change any element, and access any element by index. However, adding to the front/middle of a vector is extremely slow (that's why there's only a `std::vector::push_back` method, and no `std::vector::push_front`; in general, C++ doesn't expose a method for something that's inherently inefficient). All elements must be of the same type.	Internally, a vector consists of a fixed-size array. It automatically will resize the array (by creating a new one and moving the elements) so that you never run out of space. As a result, behind the scenes, the array has a "size" (the number of filled elements in the vector) and a "capacity" (the length of the array). When the capacity equals the size, then the array is resized to give the programmer more space.
`std::deque`	A deque is a variant of a vector. It's extremely similar, except supports fast insertion anywhere into the container. All elements must be of the same type.	Deques don't have a single, standard implementation. On common implementation, though, is to think of them as an array of arrays (check out the lecture slides 31-46 for an animation to make this clearer).
`std::list`	A list is very different from `list` in python. It is another type of sequence container. You can add and remove elements anywhere in the list, but unlike a deque and vector, you can't access any element by index. You must iterate from front to back, and access them in sequential order. All elements must be of the same type.	Lists are implemented as doubly-linked lists. The details are not extremely important to know (though you can google "doubly linked list C++" to learn more), but here's some intuition for why lists don't offer "random access", or the ability to access any element at will: doubly linked lists are stored in memory as just the first element in the list (which has a sort of "link" to the next element, and so on). As a result, in order to get to the nth element in a list, we need to manually traverse from the first element in the list to the nth, which is very inefficient for large n. As a result, C++ does not expose a random access method in lists.
`std::tuple`	A tuple is just like a python tuple. Unlike the previous containers, a tuple may have elements of different types. However, after its creation, it may not be edited nor changed (tuples are "immutable", as opposed to being "mutable" like the other containers). However, it's very rarely used (if you're using a tuple, you might as well create your own struct), so we won't discuss it further.	Tuples are implemented with "variadic templates". Stick around for two more lectures to see what these are!

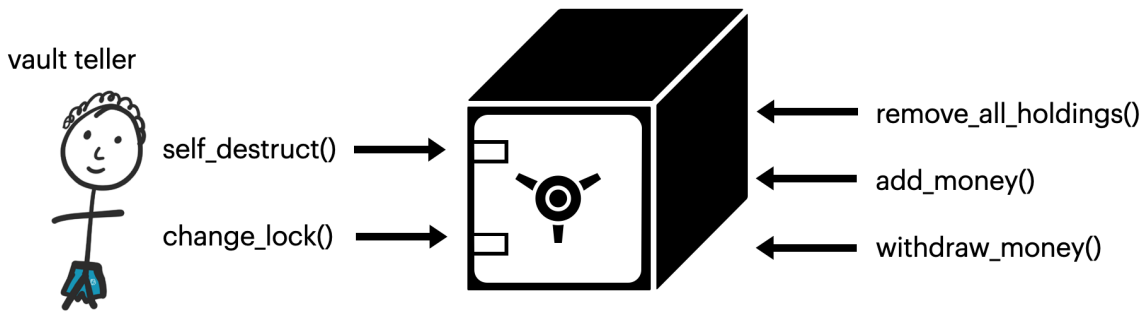
- That was a lot! Let us leave you with a table that shows you when to use each sequence container:

What you want to do	`std::vector`	`std::deque`	`std::list`
Insert/remove in the front	`Slow`	`Fast`	`Fast`
Insert/remove in the back	`Super Fast`	`Very Fast`	`Fast`
Indexed Access	`Super Fast`	`Fast`	`Impossible`
Insert/remove in the middle	`Slow`	`Fast`	`Very Fast`
Memory usage	`Low`	`High`	`High`
Combining (splicing/joining)	`Slow`	`Very Slow`	`Fast`
Stability* (iterators/concurrency)	`Bad`	`Very Bad`	`Good`

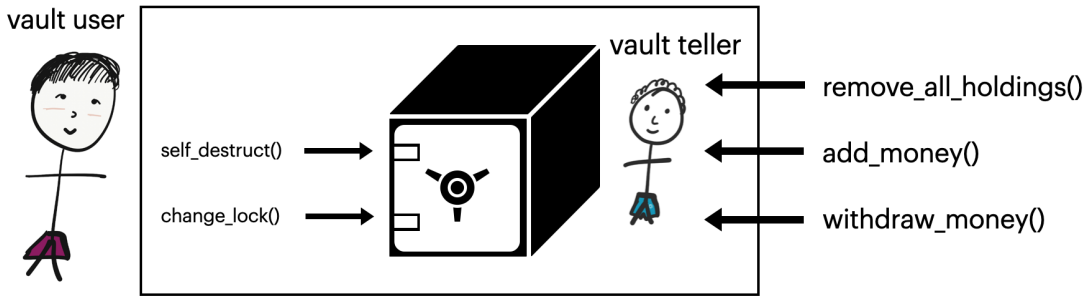
These two are the most common!

Don't worry if you don't know what stability means! It's a fairly advanced concept that you don't need to understand in order to grasp the core of this slide.

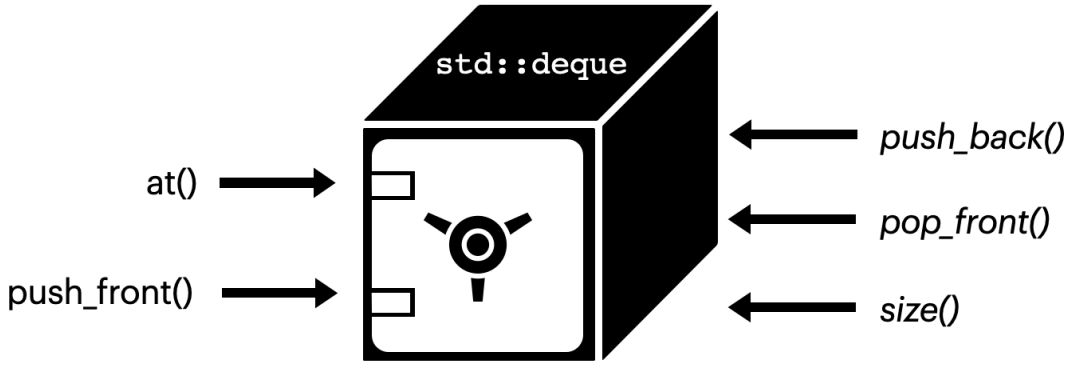
- Basically, use `std::vector` for almost everything, `std::deque` if you need fast insertion to the front AND back of your container, and `std::list` very rarely, and only if you don't need random access to each element of your container.
- Next, let's talk about "Adaptors". These include stacks, queues, and priority queues (which you'll use for the second assignment). Adaptors, or wrappers, are conceptually shells around other programming entities that only expose a few of the underlying entities' methods. For example, consider a teller to a vault. The vault teller has access to many methods, as shown below:



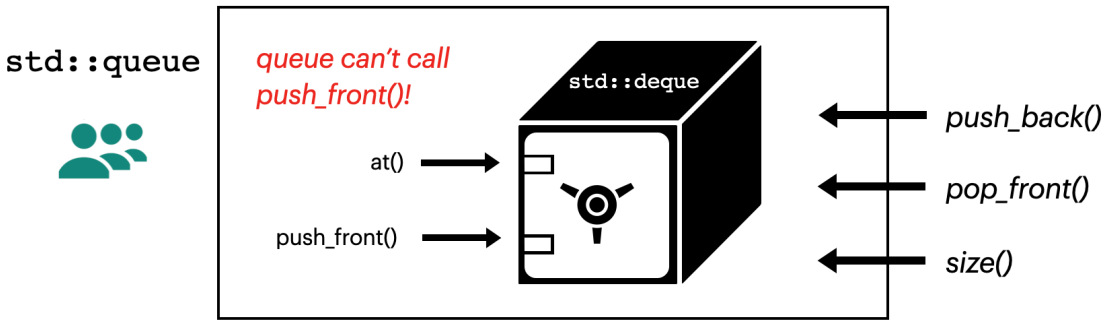
Now, let's think about how we'd allow public clients of this vault to interact with it. We wouldn't want any clients to be able to change the vault lock, or self destruct the vault! As a result, we can add a wrapper around the vault teller that only exposes a few methods that we think public vault clients should have access to, like `add_money()` and `withdraw_money()`.



- Similarly, adaptors (or wrappers) in C++ are abstractions of sequence containers. For instance, Let's consider a deque:



Instead of needing to create an entirely separate container for a queue, we can just expose certain methods of a deque analogous to enqueue(), dequeue(), front(), and size()!



- In fact, if we take a gander at the official C++ documentation for a stack and queue, we can see that this is exactly the case:

std::queue

Defined in header <queue>

```
template<
    class T,
    class Container = std::deque<T>
> class queue;
```

The `std::queue` class is a container adapter that gives the programmer the functionality of a queue - specifically, a FIFO (first-in, first-out) data structure.

The class template acts as a wrapper to the underlying container - only a specific set of functions is provided. The queue pushes the elements on the back of the underlying container and pops them from the front.

std::stack

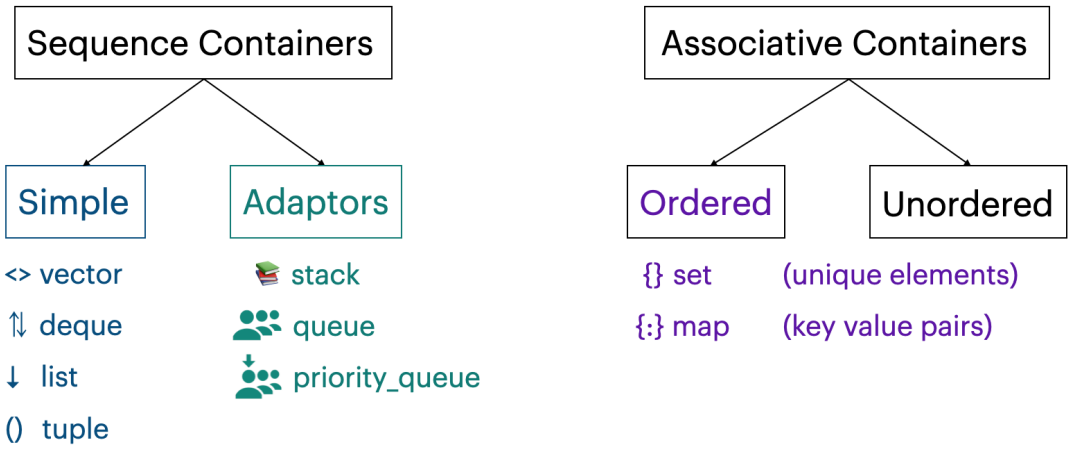
Defined in header <stack>

```
template<
    class T,
    class Container = std::deque<T>
> class stack;
```

The `std::stack` class is a container adapter that gives the programmer the functionality of a stack - specifically, a LIFO (last-in, first-out) data structure.

The class template acts as a wrapper to the underlying container - only a specific set of functions is provided. The stack pushes and pops the element from the back of the underlying container, known as the top of the stack.

- As a concrete example, we can use several different underlying containers to represent a queue! One example would be a deque, and another would be a list. However, should we use a vector? No! This is because removing from the front of a vector would not be efficient (dequeuing an element is removing it from the front of the underlying container, and enqueuing an element is adding it to the end).
- The next major type of container is an associative container. If this sounds foreign, don't worry! You've also definitely seen these before:



- Sets (hash are vectors that can only store one of each element (no duplicates). Maps are associative containers that map a key to a value (dictionaries in pythons), also called hash maps. Here's the difference between Stanford Set and Map and Standard C++ set and map:

What you want to do	Stanford `Set<int>`	`std::set<int>`
Create an empty set	`Set<int> s;`	`std::set<int> s;`
Add a value `k` to the set	`s.add(k);`	`s.insert(k);`
Remove value `k` from the set	`s.remove(k);`	`s.erase(k);`
Check if a value `k` is in the set	`if (s.contains(k)) ...`	`if (s.count(k)) ...`
Check if vector is empty	`if (vec.isEmpty()) ...`	`if (vec.empty()) ...`

What you want to do	Stanford `Map<int, char>`	`std::map<int, char>`
Create an empty map	`Map<int, char> m;`	`std::map<int, char> m;`
Add key k with value v into the map	`m.put(k, v);` `m[k] = v;`	`m.insert({k, v});` `m[k] = v;`
Remove key k from the map	`m.remove(k);`	`m.erase(k);`
Check if key k is in the map	`if (m.containsKey(k)) ...`	`if (m.count(k)) ...`
Check if the map is empty	`if (m.isEmpty()) ...`	`if (m.empty()) ...`
Retrieve or overwrite value associated with key k ***(error** if key isn't in map)*	`Impossible (but does auto-insert)`	`char c = m.at(k);` `m.at(k) = v;`
Retrieve or overwrite value associated with key k ***(auto-insert** if key isn't in map)*	`char c = m[k];` `m[k] = v;`	`char c = m[k];` `m[k] = v;`

- Here's how to iterate through maps and sets:

```
std::set<...> s;
std::map<..., ...> m;

for (const auto& element : s) {
    // do stuff with element
}

for (const auto& [key, value] : m) {
    // do stuff with key and value
}
```

- By default, the type (for sets) or key's type (for maps) must have a comparison operator (<) defined. For example, ints can be compared with <, but `std::ifstream`s can't.
- `std::unordered_set` and `std::unordered_map` are faster than their ordered counterparts. They also don't require the type (for sets) or key's type (for maps) to have a comparison operator (<) defined, but they do allow you to pass in a "hash" function that converts your type to a number. Therefore, this allows you to have a set or map of any object. In addition, adding, removing, and accessing elements is much faster than their ordered counterparts.
- That's a lot! We've just covered a ton of information about containers. Let's recap:

- Sequence Containers
  - `std::vector` - use for almost everything
  - `std::deque` - use when you need fast insertion to front AND back
- Container Adaptors
  - `std::stack` and `std::queue`
  - if using simple data types/you're familiar with hash functions, use `std::unordered_map` and `std::unordered_set`