Data Structures

Ahsan Ijaz

Review

- A container class allows you to store an arbitrary number of things
- A sequence container is a container whose elements can be accessed sequentially.
- Sequence containers include vectors, stacks, queues, lists, and priority queues (among others).
- The performance characteristics of various sequence containers, and why you might choose one over another.

Review

- A container class allows you to store an arbitrary number of things
- A sequence container is a container whose elements can be accessed sequentially.
- Sequence containers include vectors, stacks, queues, lists, and priority queues (among others).
- The performance characteristics of various sequence containers, and why you might choose one over another.

Review

- A container class allows you to store an arbitrary number of things
- A sequence container is a container whose elements can be accessed sequentially.
- Sequence containers include vectors, stacks, queues, lists, and priority queues (among others).
- The performance characteristics of various sequence containers, and why you might choose one over another.

Review

- A container class allows you to store an arbitrary number of things
- A sequence container is a container whose elements can be accessed sequentially.
- Sequence containers include vectors, stacks, queues, lists, and priority queues (among others).
- The performance characteristics of various sequence containers, and why you might choose one over another.

Stack

Let's look at the code

STLStack

Vector

Quick Demo: Vector
STLVector

STL <vector> Push Front

Why is there no push_front method?

- Pushing an element to the front of the vector requires shifting all other elements in the vector down by one, which can be very slow.
- To demonstrate this, let's say we had this nice little vector:

Figure: Vector

STL <vector> Push Front

Now, let's say that push_front existed, and that you wanted to insert an 8 at the beginning of this vector.

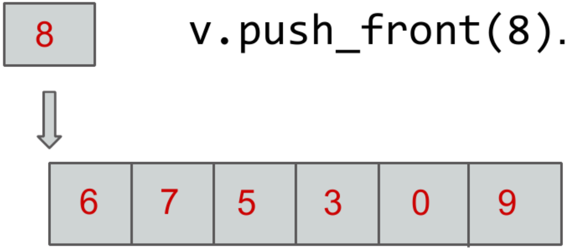

Figure: Vector

Push Front

First, we may have to expand the capacity of the vector

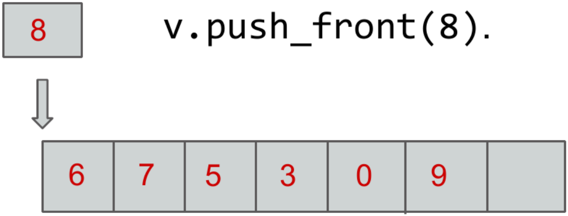

Figure: Vector

Push Front

Then, we'll need to shift every single element down one position

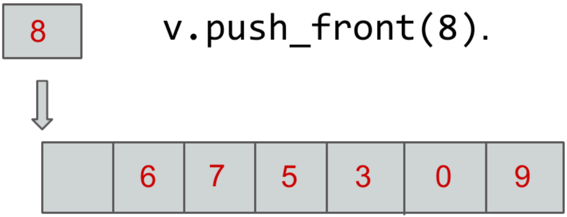

Figure: Vector

Push Front

Finally, we can actually insert the element we wanted to insert.

`v.push_front(8).`

Figure: Vector

STL <deque>

- A deque is a double ended queue.
- Unlike a vector, it's possible (and fast) to push_front.
- The implementation of a deque isn't as straightforward as a vector though

Deque

Let's look at the code
STLDeque

STL<deque>: Implementation

There's no single specification for representing a deque, but it might be laid out something like this

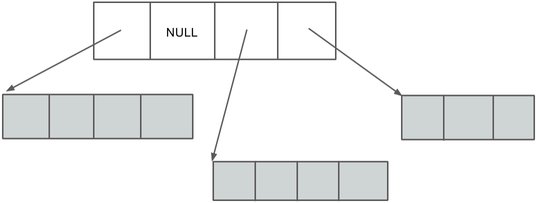

Figure: Vector

STL<deque>: Implementation

You could support efficient insertion by keeping some reserved space in front of the vector representing the first elements of the deque

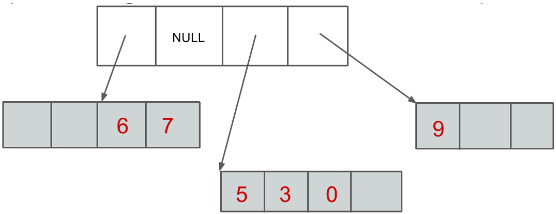

Figure: Vector

STL<deque>: Implementation

You could support efficient insertion by keeping some reserved space in front of the vector representing the first elements of the deque

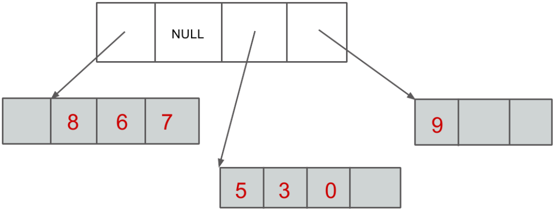

Figure: Vector

STL <deque>: Performance

- We can now use the push_front function, and it will run much faster than if we had used a vector.
- However, if all you're doing is iterating, resizing, and push_backing, then using a vector will be faster.
- Let's see how this looks in real world performance numbers.

Associative Containers

- Unsurprisingly, associative containers are containers (objects you can store data in)
- Associative containers use the idea of a **key**, which is used to lookup a value.
- Maps and Sets are among associative containers.

STL<set>

- The set data structure can be thought of as a checklist of items. We can add elements to a set, or remove elements from one. Then, we can ask the set if it contains a particular item or not.
- We can add duplicates, but only one copy will be stored. That is because sets are only concerned about whether an item appears in the data structure or not.

STL<set>

Let's take a quick peek at the code though, so we can see what STL set code looks like

Iterator: Motivation

- How do you iterate through all the elements of a set?
- How do you iterate through all the elements of a map?

Because maps and sets aren't sequence containers, we can't just go from 0 to vector. or pop elements off of a stack until it's empty.

Iterators: example

As we first see them, iterators will allow us to iterate through all the elements of an unsequenced collection of elements (like a set or a map)

- Let's first try and get a conceptual model of what an iterator is.
- Say that we have a set of integers. Say the set was named 's'.

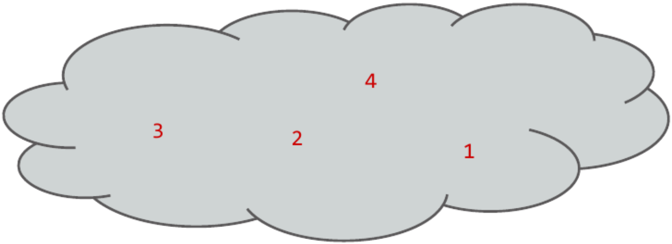

Figure: Set Data

- Let's first try and get a conceptual model of what an iterator is.
- Iterators allow us to view an unordered collection in a linear order

Figure: Linear Picture

- Let's first try and get a conceptual model of what an iterator is.
- We can construct an iterator 'i' to point to the first element in the set

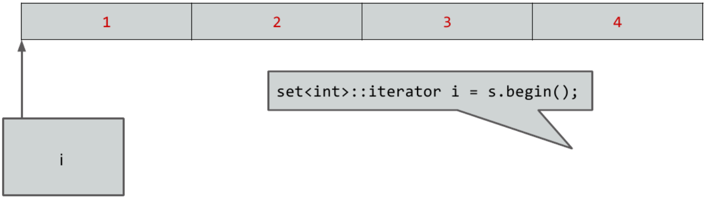

Figure: Start a Iterator

- Let's first try and get a conceptual model of what an iterator is.
- We can dereference our iterator to read the value the iterator is currently on

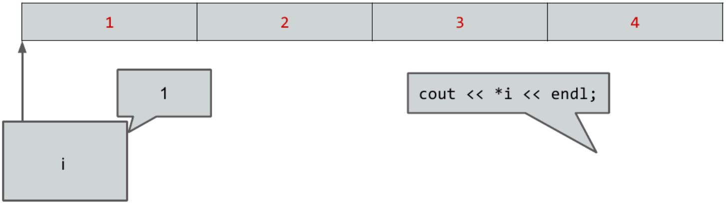

Figure: Dereference Iterator

- Let's first try and get a conceptual model of what an iterator is.
- We can **advance** our iterator

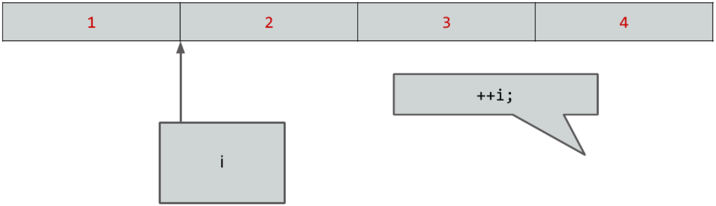

Figure: Advancing Iterator

- Let's first try and get a conceptual model of what an iterator is.
- We can **dereference** our iterator **again** and read a different value

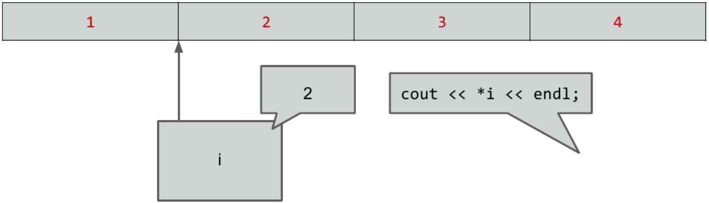

Figure: Reading next value

Iterators

Eventually, we reach the end of a container. You can check if an iterator has iterated through every element in the container by comparing it to the .end() element.

```
if (i == s.end())
cout << "We're done!" << endl;
```

Iterator

Remember the four most fundamental iterator operations:

-
- Create an iterator
- Dereference an iterator and read the value it's currently looking at
- Advance an iterator
- Compare an iterator against another iterator (especially one from the .end()) method

Other uses of Iterator

STL containers often use iterators to specify individual elements inside a container.

```
vector<int> v;
for (int i = 0; i < 10; i++) {
v.push_back(i);
}
v.erase(v.begin() + 5, v.end());
// v now contains 0, 1, 2, 3, 4
```

Other uses of Iterators

- Iterator's don't always have to iterate through all of a container.
- For example, they could iterate through a range of elements.

Other uses of Iterators

For example, here's the code to iterate through all the integers in a set:

```
set<int>::iterator i = s.begin();
set<int>::iterator end = s.end();
while (i != end) {
cout << *i << endl;
++i;
}
```

Other uses of Iterators

For example, here's the code to iterate through all the integers greater than 7 and less than 23 in a set:

```
set<int>::iterator i = s.lower_bound(7);
set<int>::iterator end = s.upper_bound(23);
while (i != end) {
cout << *i << endl;
++i;
}
```