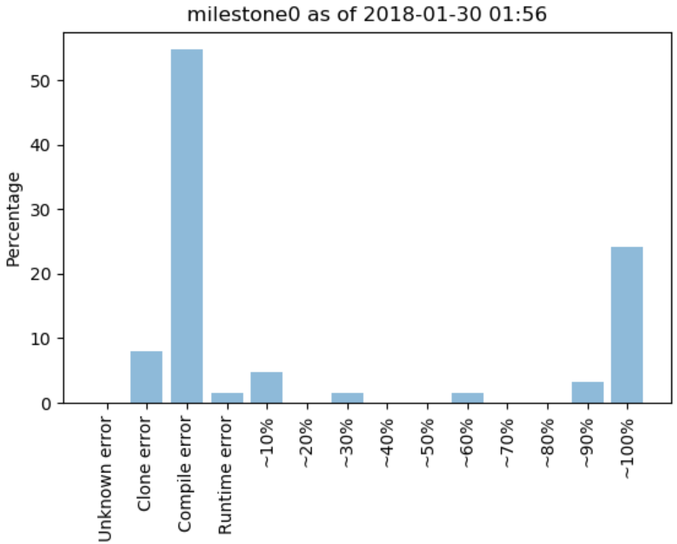
ECE 3574: C++ Standard Library

Changwoo Min

Where we are?



- TA Help Session: Tu 1/30 6:30-7:45pm
- SWEL office hour
- My office hour: 2-4 PM Friday

Rough project deadlines

Milestone 0: 2/5

Milestone 1: 2/20

Milestone 2: 3/13 (spring break)

Milestone 3: 4/3

Milestone 4: week of 4/23

Review test cases

```
TEST_CASE( "test empty stream", "[lexer]" ) {
  {
    std::string input = "    \t    \r      ";
    std::istringstream iss(input);

    TokenList tl = tokenize(iss);

    REQUIRE(tl.size() == 0);
  }
}
```

Review test cases

```
TEST_CASE( "test equal token", "[lexer]" ) {

  std::string input = R"(
  .data
  NAME1 = 1
  NAME2 = 2
  NAME3 = -3)";

  std::istringstream iss(input);
  TokenList tl = tokenize(iss);

  REQUIRE(tl.size() == 14);
}
```

- C++11 raw strings literals
  - Used to avoid escaping of any character: `R"(...)"`
  - Link 1, Link 2, Link 3

Meeting 5: C++ Standard Library

- The goal of today's meeting is to review the standard library
  - Containers and Iterators
  - Algorithms
  - Exercise 5

> *The best code is that already written and tested*

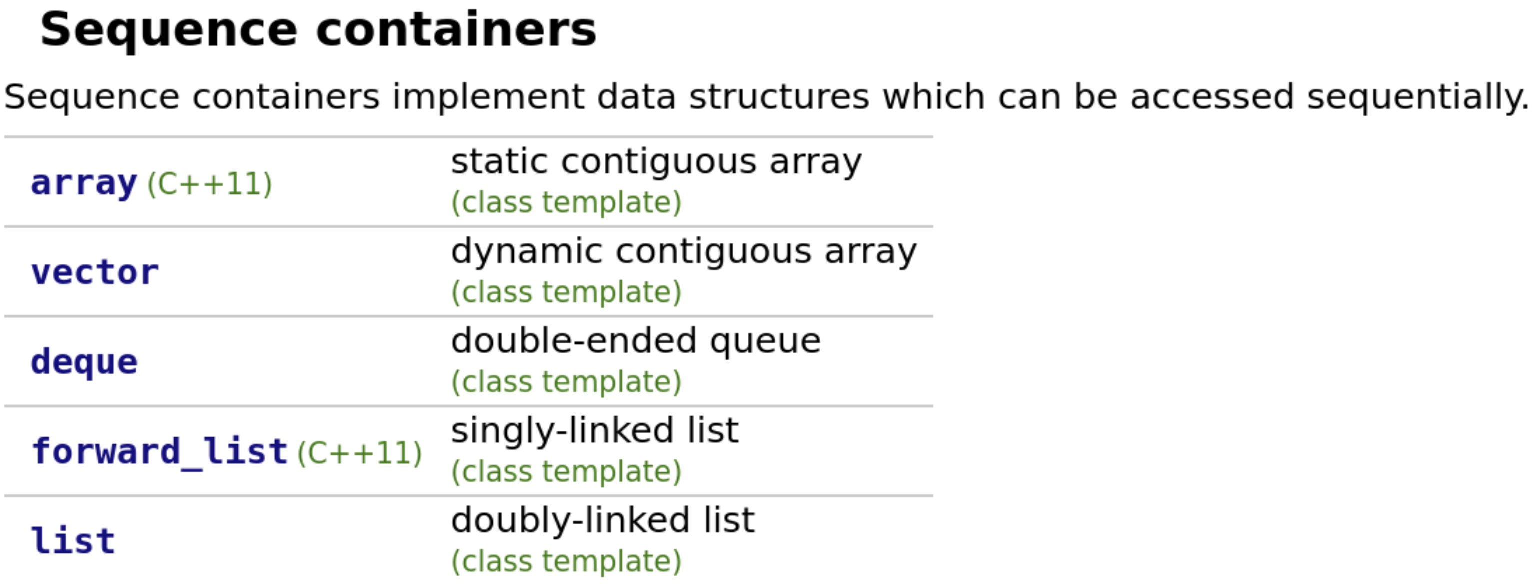
The C++ standard library is well-constructed and tested

- Prefer to use containers and algorithms from the standard library rather than hand-coded data structures and algorithms.
- In 2574 you saw how to implement data structures and common algorithms for sorting and searching. However, the C++ standard library provides implementations of these that are efficient and well tested, so you should prefer to use them over hand-coded approaches whenever feasible.

Three terms you need to know

- **Container**
  - list, hash table, tree, …
- **Iterator**
  - Index to traverse a container
- **Iterator invalidation**
  - What happen to your iterator if you delete an entry while your are traversing a container.

Sequence containers



Sequence containers

Sequence containers implement data structures which can be accessed sequentially.

array (C++11)	static contiguous array (class template)
vector	dynamic contiguous array (class template)
deque	double-ended queue (class template)
forward_list (C++11)	singly-linked list (class template)
list	doubly-linked list (class template)

- Reference

`std::array` is a wrapper around raw arrays

- supports standard access members (at, [], front, back)
- has a size() member
- supports fill and swap
- can be empty
- very low overhead

```
std::array<int,10> a;
a.fill(1);
assert(a[3] == 1);
assert(a.size() == 10);
```

`std::vector` is a dynamically sized array-based container

- the most useful linear data structure
- see members size, capacity, and reserve
- grows exponentially
- supports insert - much more efficient than you might think
- watch out for iterator invalidation

```
std::vector<int> v;
std::cout << v.capacity() << std::endl;
for(int i = 0; i < 100; ++i){
    v.push_back(i);
    std::cout << v.capacity() << std::endl;
}
```

`std::deque` is a dynamically sized double ended queue

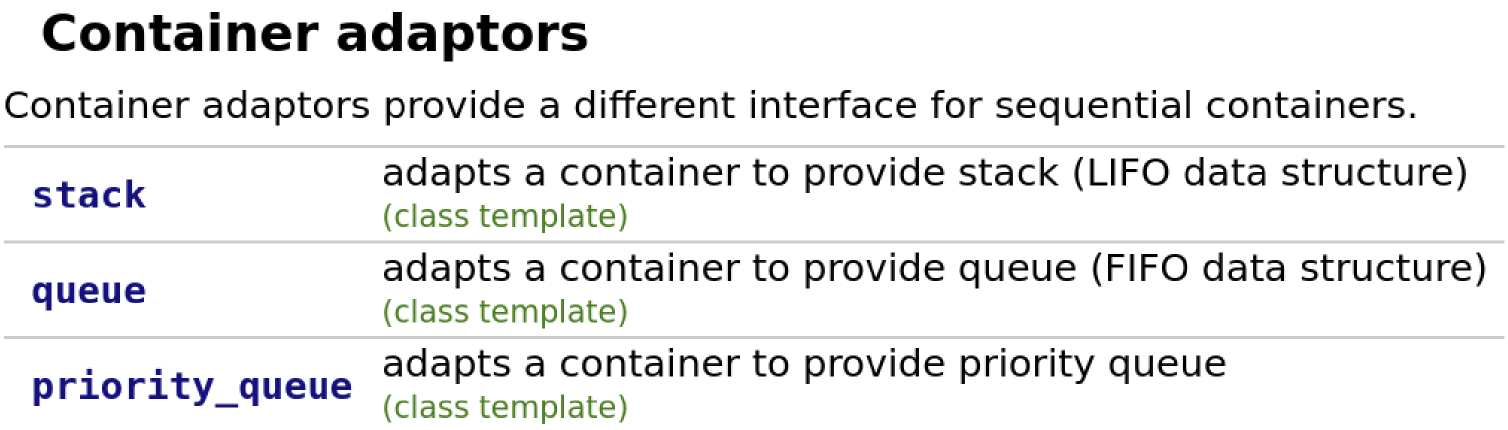
- not contiguous in memory
- access either end: push_front or push_back
- generally better performance than std::list

```
std::deque<int> d;
for(int i = 0; i < 100; ++i){
    d.push_back(i);
    d.push_front(i);
}
return 0;
```

`std::list` and `std::forward_list`

- doubly and singly linked-lists respectively
- constant time insertion anywhere
- no random access
- `std::list` supports bidirectional iteration
- space efficient, no extra space as in `std::vector`
- often less efficient than `std::vector` because of cache misses

Container adaptors



Container adaptors

Container adaptors provide a different interface for sequential containers.

`stack`	adapts a container to provide stack (LIFO data structure) (class template)
`queue`	adapts a container to provide queue (FIFO data structure) (class template)
`priority_queue`	adapts a container to provide priority queue (class template)

- Reference

Adaptors provide wrappers around other containers

- stack (wraps a deque)
- queue (wraps a deque)
- priority_queue (a heap using vector for storage)

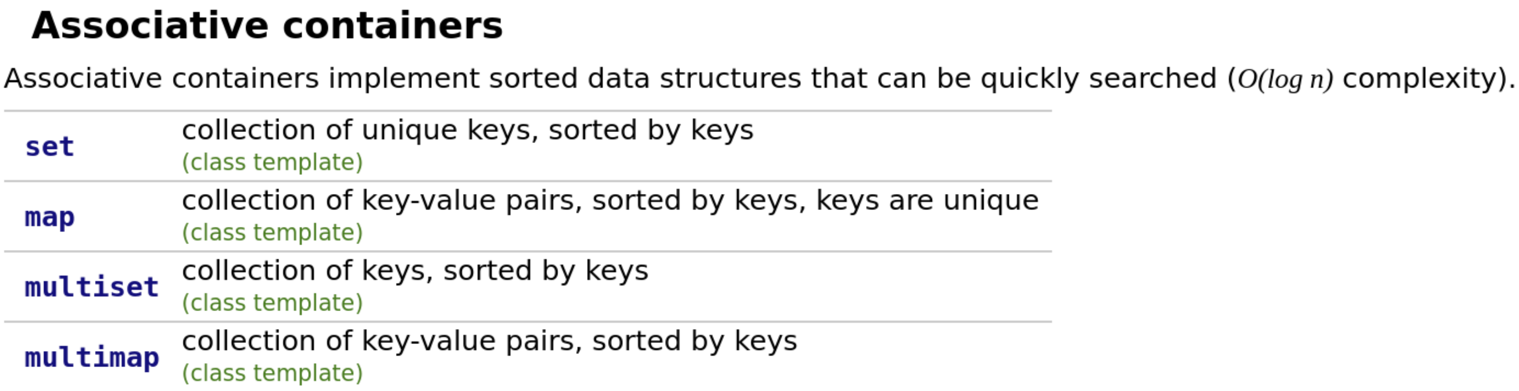
Adaptors provide wrappers around other containers

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

template<
    class T,
    class Container = std::deque<T>    // @_@
> class queue;

template<
    class T,
    class Container = std::vector<T>,  // @_@
    class Compare = std::less<typename Container::value_type>
> class priority_queue;
```

Associative containers



Associative containers

Associative containers implement sorted data structures that can be quickly searched (*O(log n)* complexity).

`set`	collection of unique keys, sorted by keys (class template)
`map`	collection of key-value pairs, sorted by keys, keys are unique (class template)
`multiset`	collection of keys, sorted by keys (class template)
`multimap`	collection of key-value pairs, sorted by keys (class template)

- Reference

`std::map` is a dictionary (key,value)

- `std::map` requires unique keys and value
- implemented as red-black tree (balanced binary tree)
- index `operator[]` is very handy

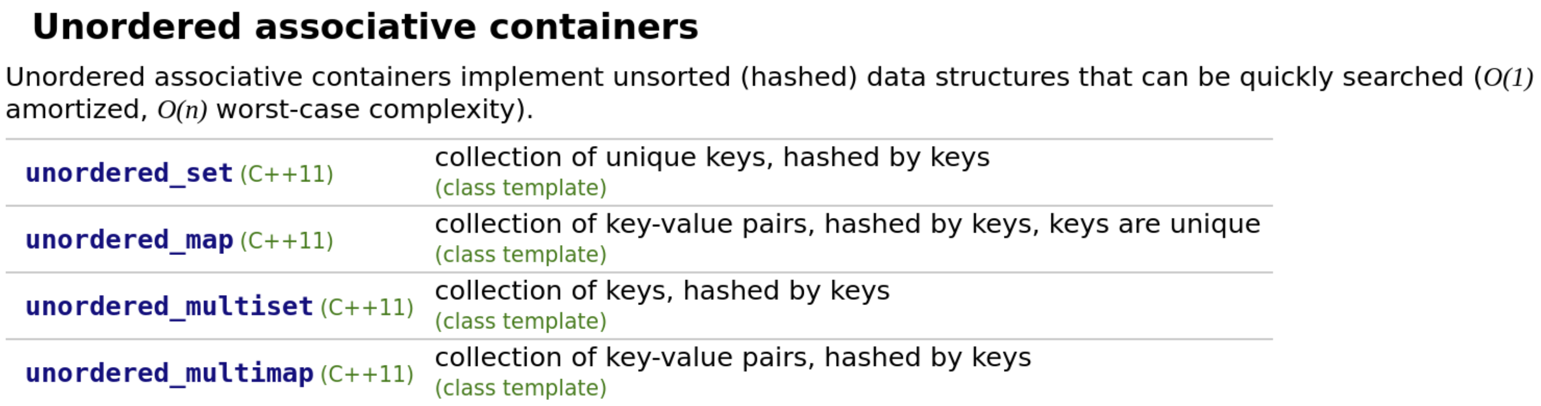
```
std::map<std::string, int> occurances;
occurances["hello"] += 1;
occurances["hello"] += 1;
occurances["goodbye"] += 1;

for(auto it = occurances.begin();
    it != occurances.end();
    ++it)
{
    std::cout << "You said " << it->first << " "
    << it->second << " times." << std::endl;
}
```

`std::multimap`

- `std::multimap` is a dictionary like `std::map`, while permitting multiple entries with the same key.
- *See also* `std::set` *and* `std::multiset` *(no value, just a key)*

Unordered associative containers



Unordered associative containers

Unordered associative containers implement unsorted (hashed) data structures that can be quickly searched (*O(1)* amortized, *O(n)* worst-case complexity).

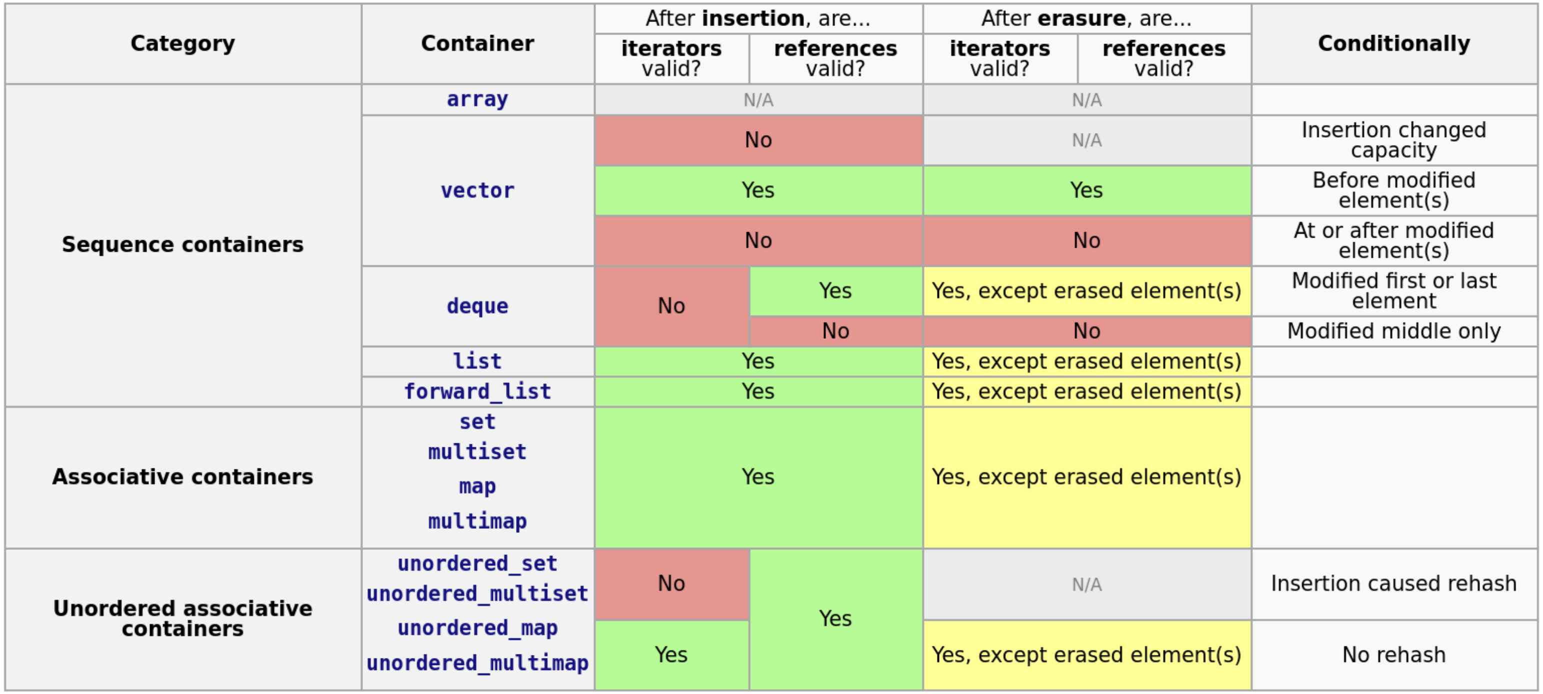
unordered_set (C++11)	collection of unique keys, hashed by keys (class template)
unordered_map (C++11)	collection of key-value pairs, hashed by keys, keys are unique (class template)
unordered_multiset (C++11)	collection of keys, hashed by keys (class template)
unordered_multimap (C++11)	collection of key-value pairs, hashed by keys (class template)

- Reference

Hash tables are in the C++ stdlib now!

- `unordered_set` / `unordered_map`
- `unordered_multiset` / `unordered_multimap`
- constant (amortized) time find, insert, remove

Iterator invalidation



Category	Container	After **insertion**, are...		After **erasure**, are...		Conditionally
		iterators valid?	**references** valid?	**iterators** valid?	**references** valid?	
Sequence containers	`array`	N/A		N/A		
	`vector`	No		N/A		Insertion changed capacity
		Yes		Yes		Before modified element(s)
		No		No		At or after modified element(s)
	`deque`	No	Yes	Yes, except erased element(s)		Modified first or last element
			No	No		Modified middle only
	`list`	Yes		Yes, except erased element(s)		
	`forward_list`	Yes		Yes, except erased element(s)		

Associative containers	`set`
	`multiset`
	`map`
	`multimap`
		Yes		Yes, except erased element(s)		

Unordered associative containers	`unordered_set`
	`unordered_multiset`
	`unordered_map`
	`unordered_multimap`
		No	Yes	N/A		Insertion caused rehash

		Yes		Yes, except erased element(s)		No rehash

- Reference

Iterator invalidation example

```
#include <iostream>
#include <vector>
#include <iterator>
#include <algorithm>

int main()
{
    std::vector<int> vecArr;
    for(int i = 1; i <- 10; i++)
        vecArr.push_back(i);

    for(auto it = vecArr.begin(); it != vecArr.end(); it++)
        std::cout<<(*it)<<"  ";
    std::cout<<std::endl;

    // Erase and element with value 5.
    auto it = std::find(vecArr.begin(), vecArr.end(), 5);
    if(it != vecArr.end())
        vecArr.erase(it); // Now iterator 'it' is invalidated

    for(; it != vecArr.end(); it++)   // Unpredicted Behavior
        std::cout<<(*it)<<"  ";       // Unpredicted Behavior
    return 0;
}
```

Iterator invalidation example

- How to solve this?

```
auto it = std::find(vecArr.begin(), vecArr.end(), 5);
if(it != vecArr.end())
   it = vecArr.erase(it);
```

- Reference

Algorithms library

- Non-modifying sequence operations
- Modifying sequence operations
- Partitioning operations
- Binary search
- Set operations
- Heap operations
- min/max
- numeric (see random number generators too)

Exercise 5

See Website

Useful C++ features

- `auto` specifier (since C++11)
  - For variables, specifies that the type of the variable that is being declared will be automatically deduced from its initializer.
- `std::ifstream`: Link 1, Link 2
- `std::getline()`: Link 1, Link 2, Link 3
- `std::basic_streambuf`: Link
- `std::map`: Link 1, Link 2
- `std::unordered_map`: Link 1, Link 2
- `std::multimap:insert()`: Link 1, Link 2

Next Actions and Reminders

- Read The Pragmatic Programmer Sections 7, 8, 26