Data Structures Overview

STL

History

Containers

Iterators

Smart pointers

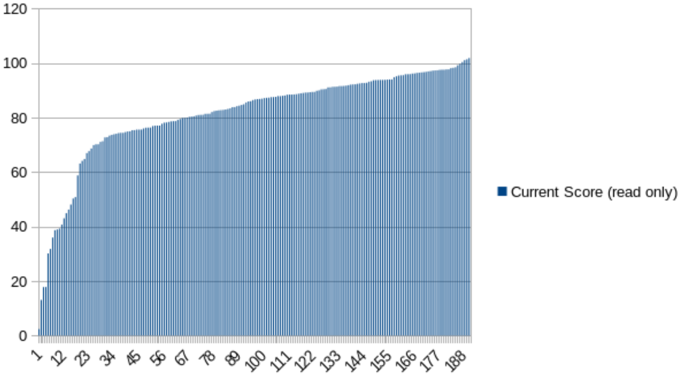
Review, perspectives, lessons learned

Comp Sci 1575 Data Structures

Missouri S&T | Computer Science

Almost done!

A quite high mean this semester



Significantly more than half the class has a B or higher. This distribution is indicative of high grades being predictably accessible with appropriately directed work. Based on this, besides the possible CET bonus, definitely no global curve or drop needed. Difficulty hype is not all it is made out to be (this is a pretty typcial distribution).

Outline

1 Data Structures Overview

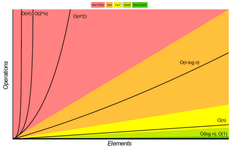
2 STL
History
Containers
Iterators

3 Smart pointers

Data structures

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

Color key:



Sorting algorithms

```
                    COLLECTION
                   /          \
UNORDERED_COLLECTION     ORDERED_COLLECTION
        |                     |          |
       BAG                  LINEAR     NON_LINEAR
        |                     |          |         |
       SET              LINKED_LIST    TREE   DIRECTED_GRAPH
        |                     |          |         |
    HASH_TABLE              DEQUE     BIN_TREE  UNDIRECTED_GRAPH
                           /    \      /    \
                       STACK  QUEUE  BST   HEAP
```

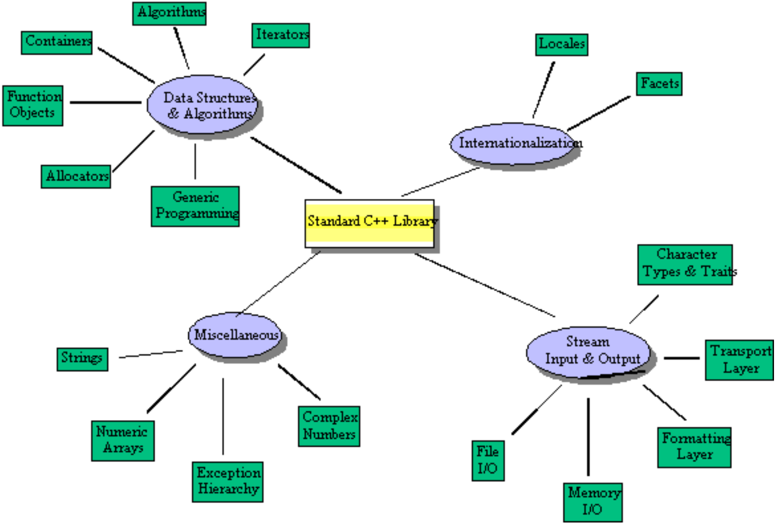
Sorting algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)

Outline

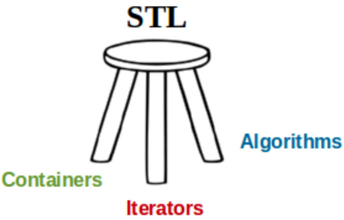
Outline

STL is now just part of standard namespace



- Old diagram of the **S**tandard **T**emplate **L**ibrary (STL)

Main pillars of the STL



- **Containers** manage storage space for elements and provide member functions to access them. Implemented as templated classes, with flexibility of types as elements.
- **Algorithms** act on containers, and perform operations like initialization, sorting, searching, and transforming of the contents of the aforementioned containers.
- **Iterators** step through elements of collections of objects in containers or subsets of containers. Pointer-like iterators can traverse many container classes in modularly.

Outline

1 Data Structures Overview

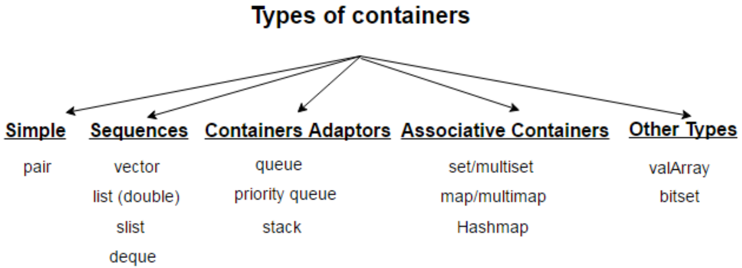
2 STL
- History
- Containers
- Iterators

3 Smart pointers

Containers

- Containers library is a generic collection of class templates and algorithms that allow programmers to easily implement common data structures like queues, lists, and stacks.
- Classes of containers, each of which is designed to support a different set of operations:
  1. sequence containers
  2. associative containers
  3. un-ordered associative containers
  4. container adaptors (modify above)
- Containers manage storage space that is allocated for their elements and provides member functions to access them, either directly or through iterators (objects with properties similar to pointers).
- The set of containers have at least several member functions in common with each other, and share functionalities.

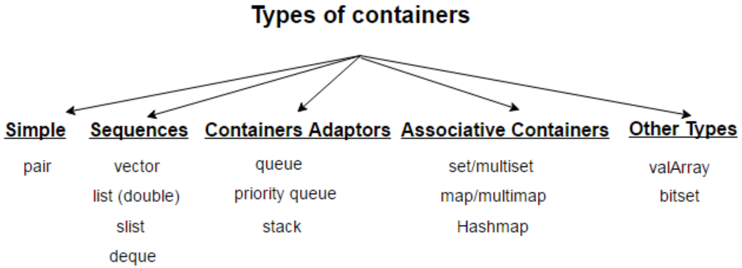
Containers



For a comprehensive list, see:

- `http://en.cppreference.com/w/cpp/container`
- `http://www.cplusplus.com/reference/stl/`
- `https://en.wikipedia.org/wiki/Standard_Template_Library`

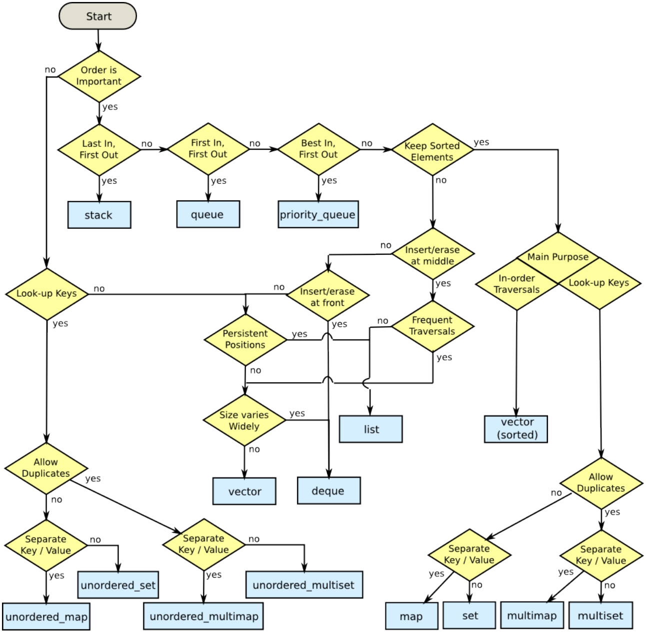
Simple utilities: pair, tuple



Not in the "Containers" but "Utilities"

- `http://en.cppreference.com/w/cpp/utility/pair`
- `http://en.cppreference.com/w/cpp/utility/tuple`

How to choose your container?



Outline

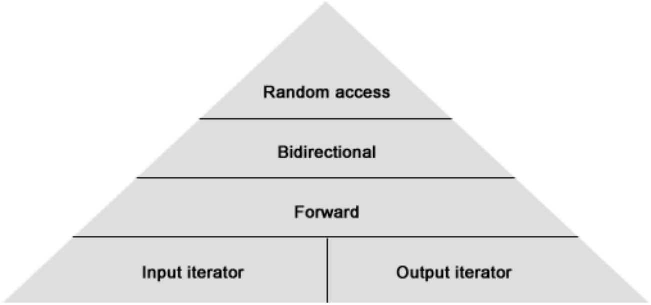
Iterators

An iterator can be imagined as a pointer to a given element in the container, with overloaded operators to provide a subset of well-defined functions normally provided by pointers:

- $Operator*$ Dereferencing the iterator returns the element that the iterator is currently pointing at.
- $Operator++$ Moves the iterator to the next element in the container. Most iterators also provide $Operator--$ to move to the previous element.
- $Operator ==$ and $Operator! =$ Basic comparison operators to determine if two iterators point to the same element. To compare the values that two iterators are pointing at, dereference the iterators first, and then use a comparison operator.
- $Operator =$ Assign the iterator to a new position (typically the start or end of the container's elements). To assign the value of the element the iterator is point at, dereference the iterator first, then use the assign operator.

Iterators

Iterator categories



Iterator categories	Provider
Input iterator	istream
Output iterator	ostream
Forward iterator	
Bidirectional iterator	list, set, multiset, map, multimap
Random access iterator	vector, deque, array

- Where does forward_list go?
- `http://en.cppreference.com/w/cpp/iterator`
- `http://www.cplusplus.com/reference/iterator/`

Iterators

category				properties	valid expressions
all categories				*copy-constructible*, *copy-assignable* and *destructible*	`X b(a);` `b = a;`
				Can be incremented	`++a` `a++`
Random Access	Bidirectional	Forward	Input	Supports equality/inequality comparisons	`a == b` `a != b`
				Can be dereferenced as an *rvalue*	`*a` `a->m`
			Output	Can be dereferenced as an *lvalue* (only for *mutable iterator types*)	`*a = t` `*a++ = t`
				default-constructible	`X a;` `X()`
				Multi-pass: neither dereferencing nor incrementing affects dereferenceability	`{ b=a; *a++; *b; }`

				Can be decremented	`--a`
					`a--`
					`*a--`

				Supports arithmetic operators + and -	`a + n`
					`n + a`
					`a - n`
					`a - b`

				Supports inequality comparisons (<, >, <= and >=) between iterators	`a < b`
					`a > b`
					`a <= b`
					`a >= b`

				Supports compound assignment operations += and -=	`a += n` `a -= n`
				Supports offset dereference operator ([])	`a[n]`

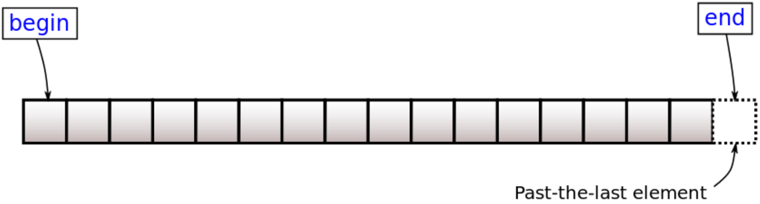
Each category of iterator is defined by the operations that can be performed on it. Any type that supports the necessary operations can be used as an iterator, e.g., pointers support all of the operations required by RandomAccessIterator, so pointers can be used anywhere a RandomAccessIterator is expected.

Iterators

Each container includes at least 4 member functions for the operator= to set the values of named LHS iterators.

- begin() returns an iterator to the first element.
- end() returns an iterator one past the last element.
- cbegin() returns a const (read-only) iterator to the first element.
- cend() returns a const (read-only) iterator one past the last element.

end() doesn't point to the last element in the list. This makes looping easy: iterating over the elements can continue until the iterator reaches end(), and then you know you're done.



iterator example


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

int main()
{
    std::vector<int> ints {1, 2, 4, 8, 16};
    std::vector<std::string> fruits {"orange", "apple", "raspberry"};
    std::vector<char> empty;

    // Sums all integers in the vector ints (if any), printing only the result.
    int sum = 0;

    for (auto it = ints.cbegin(); it != ints.cend(); it++)
        sum += *it;

    std::cout << "Sum of ints: " << sum << "\n";

    // Prints the first fruit in the vector fruits, without checking
    std::cout << "First fruit: " << *fruits.begin() << "\n";

    // checks
    cout << empty.empty();
    if (empty.begin() == empty.end())
        std::cout << "vector 'empty' is indeed empty.\n";

    // Alternative syntax
    auto it1 = ints.begin();
    auto it2 = std::begin(ints);
}
```

Container have different iterator invalidation rules

Each container has different rules for when an iterator will be invalidated after operations on the container:
http://en.cppreference.com/w/cpp/container

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

Outline

1 Data Structures Overview

2 STL
History
Containers
Iterators

3 Smart pointers

Smart pointers

Want to avoid dynamic memory management, but still use it?

- https://en.cppreference.com/book/intro/smart_pointers
- https://en.cppreference.com/w/cpp/memory
- https://www.learncpp.com/cpp-tutorial/15-1-intro-to-smart-pointers-move-semantics/

Final project code file overview