CPSC 131 – Data Structures

How do I choose?

Josuttis, The C++ Standard Library 2e, 2012, Pearson Education

Professor T. L. Bettens

Fall 2020

When to Use Which Container

- Contains general statements that might not fit reality
- For example,
  - if you manage only a few elements, you can ignore complexity
  - Short element processing with linear complexity is better than long element processing with logarithmic complexity

(*Josuttis, The C++ Standard Library 2e, 2012, Pearson Education*)

Overview of Container Abilities

	Array	Vector	Deque	List	Forward List	Associative Containers	Unordered Containers
Available since	TR1	C++98	C++98	C++98	C++11	C++98	TR1
Typical internal data structure	Static array	Dynamic array	Array of arrays	Doubly linked list	Singly linked list	Binary tree	Hash table
Element type	Value	Value	Value	Value	Value	Set: value Map: key/value	Set: value Map: key/value
Duplicates allowed	Yes	Yes	Yes	Yes	Yes	Only multiset or multimap	Only multiset or multimap
Iterator category	Random access	Random access	Random access	Bidirectional	Forward	Bidirectional (element/key constant)	Forward (element/key constant)
Growing/shrinking	Never	At one end	At both ends	Everywhere	Everywhere	Everywhere	Everywhere
Random access available	Yes	Yes	Yes	No	No	No	Almost
Search/find elements	Slow	Slow	Slow	Very slow	Very slow	Fast	Very fast
Inserting/removing invalidates iterators	—	On reallocation	Always	Never	Never	Never	On rehashing
Inserting/removing invalidates refs/ptrs	—	On reallocation	Always	Never	Never	Never	Never
Allows memory reservation	—	Yes	No	—	—	—	Yes (buckets)
Frees memory for removed elements	—	Only with `shrink_to_fit()`	Sometimes	Always	Always	Always	Sometimes
Transaction safe (success or no effect)	No	Push/pop at the end	Push/pop at the beginning and the end	All insertions and all erasures	All insertions and all erasures	Single-element insertions and all erasures if comparing doesn't throw	Single-element insertions and all erasures if hashing and comparing don't throw

The Following Rules of Thumb Might Help

- By default, use a **Vector**
  - Simplest internal data structure and provides random access
  - Data access is convenient and flexible
  - Data processing is often fast enough.
- Use a **Deque**
  - If you insert and/or remove elements often at the beginning and the end of a sequence
  - if it is important that the amount of internal memory used by the container shrinks when elements are removed.
- Use a **List**
  - If you insert, remove, and move elements often in the middle of a container
  - Lists provide special member functions to move elements from one container to another in constant time.
  - Note, however, that because a list provides no random access, you might suffer significant performance penalties on access to elements inside the list if you have only the beginning of the list
  - Like all node-based containers, a list doesn't invalidate iterators that refer to elements, as long as those elements are part of the container. Vectors invalidate all their iterators, pointers, and references whenever they exceed their capacity and part of their iterators, pointers, and references on insertions and deletions. Deques invalidate iterators, pointers, and references when they change their size, respectively.
  - If you need a container that handles exceptions so that each operation either succeeds or has no effect
- Use an **Unordered** (**Hash Table**) **Set** or **Multiset**
  - If you often need to search for elements according to a certain criterion
  - However, hash containers have no ordering, so if you need to rely on element order, you should use a set or a multiset that sorts elements according to the search criterion.
- To process key/value pairs, use an unordered (multi)map or, if the element order matters, a (multi)map.
- If you need an associative array, use an unordered map or, if the element order matters, a map.
- If you need a dictionary, use an unordered multimap or, if the element order matters, a multimap.