

Please do not redistribute slides without prior permission.

Introduction to C++ Containers

-- Know Your Data Structures
with Mike Shah

17:15 - 16:15 Fri, November 12, 2023

~60 minutes | Introductory Audience

Social: [@MichaelShah](https://twitter.com/MichaelShah)

Web: mshah.io

Courses: courses.mshah.io

 **YouTube**

www.youtube.com/c/MikeShah

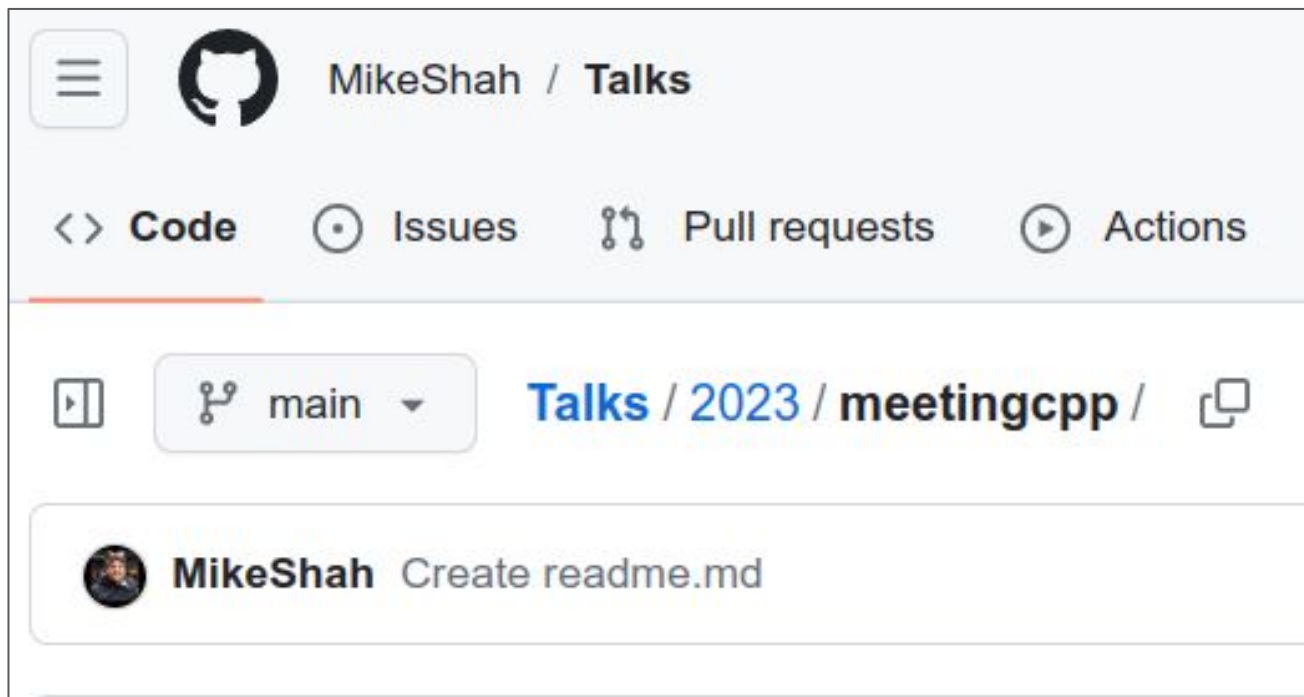
The abstract that you read and enticed you to join me is here!

Abstract

The C++ Standard Library provides a common set of data structures (known as containers) for inserting, updating, and removing data. Since the most recent standardization of C++ 23, additional container and container adaptors have been added. In this talk, I will discuss how C++ organizes these containers (sequence, associative, unordered associative, and adaptors) targeted at a beginner who wants to understand how to navigate the STL. Along this journey trade-offs with each data structure will be discussed. Listeners to this talk will leave with a cheat-sheet of data structures, so they know immediately which data structures to use when starting a project. C++ examples will be shown for how to use each container, the time complexity of the operations, the common implementation of each container. Some other common 'gotchas' regarding thread-safety and iterator invalidation will be displayed in these examples. Finally, time will be spent at the end of this talk highlighting the new C++ 23 flat container containers.

Code for the talk (or Google my name and find talk listed on website)

- Located here: <https://github.com/MikeShah/Talks/tree/main/2023/meetingcpp>



The screenshot shows the GitHub interface for the repository 'MikeShah / Talks'. The navigation bar includes 'Code', 'Issues', 'Pull requests', and 'Actions'. Below the navigation bar, the file path 'Talks / 2023 / meetingcpp' is displayed, with 'main' selected as the branch. A commit message 'Create readme.md' by 'MikeShah' is visible at the bottom of the commit list.

Your Tour Guide for Today

by Mike Shah

- **Associate Teaching Professor** at Northeastern University in Boston, Massachusetts.
 - I **love** teaching: courses in computer systems, computer graphics, geometry, and game engine development.
 - My **research** is divided into computer graphics (geometry) and software engineering (software analysis and visualization tools).
- I do **consulting** and **technical training** on modern C++, DLang, Concurrency, OpenGL, and Vulkan projects
 - Usually graphics or games related -- e.g. Building 3D application plugins
- Outside of work: guitar, running/weights, traveling and cooking are fun to talk about



Web

www.mshah.io












<https://www.youtube.com/c/MikeShah>

Non-Academic Courses

courses.mshah.io

More STL than can fit in a talk....

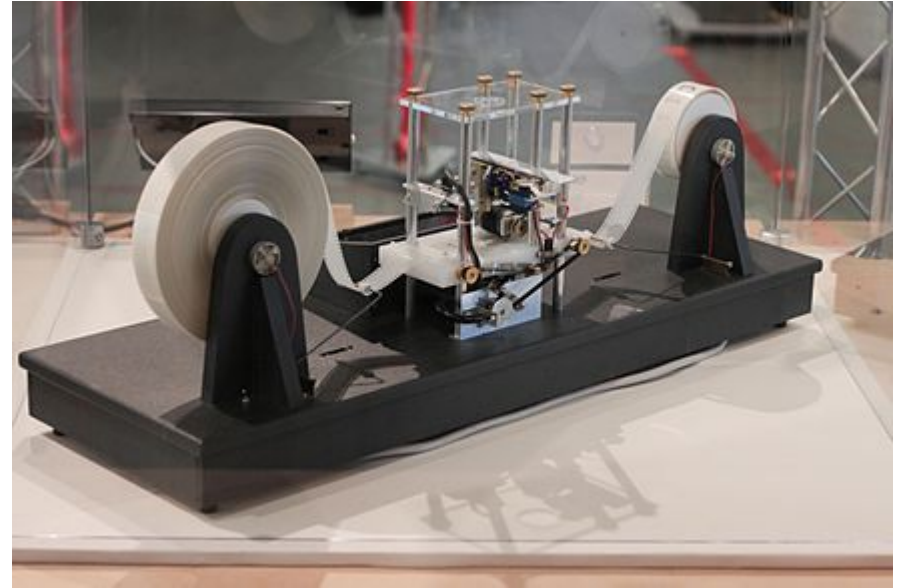
- **NOTE:** I will not cover every container today in depth, so some slides may go fast (but you can pause, and the slides were created to otherwise help you)
- If you don't find what you're looking for:
 -  YouTube "[Mike Shah C++](#)" or "[Mike Shah STL](#)"
 - or
 - www.courses.mshah.io

| | |
|---|---|
|  <p>STL std::array Modern C++ with Mike 23:50</p> | STL std::array (Since C++11) Modern Cpp Series Mike Shah • 1.2K views • 7 months ago |
|  <p>STL std::span Modern C++ with Mike 16:03</p> | STL std::span Modern Cpp Series Mike Shah • 2.3K views • 7 months ago |
|  <p>STL std::vector Modern C++ with Mike 31:01</p> | STL std::vector Modern Cpp Series Mike Shah • 1.2K views • 7 months ago |
|  <p>// To Comment // or Not Comment Modern C++ with Mike 13:02</p> | Commenting the 'why' in your C++ code Modern Cpp Series Mike Shah • 653 views • 7 months ago |
|  <p>STL std::list Modern C++ with Mike 33:48</p> | STL std::list Modern Cpp Series Mike Shah • 1.3K views • 7 months ago |
|  <p>STL std::forward_list Modern C++ with Mike 29:09</p> | STL std::forward_list Modern Cpp Series Mike Shah • 887 views • 7 months ago |
|  <p>STL std::deque Modern C++ with Mike 28:43</p> | STL std::deque Modern Cpp Series Mike Shah • 1K views • 7 months ago |
|  <p>STL std::set Modern C++ with Mike</p> | STL std::set Modern Cpp Series Mike Shah • 1.1K views • 7 months ago |

Data

All we have is data

- At the end of the day--computers are machines that help us transform data as quickly and precisely as possible
 - $\text{Data}_{\text{in}} \rightarrow \text{operation} \rightarrow \text{Data}_{\text{out}}$
- The machine to the right is a 'Turning machine' which interprets one symbol at a time and applies some operation to the data.
 - We have some higher level abstractions in C++ however to help us manage and organize the data.



An example of a Turing Machine
https://en.wikipedia.org/wiki/Turing_machine

Fundamental Data Structures

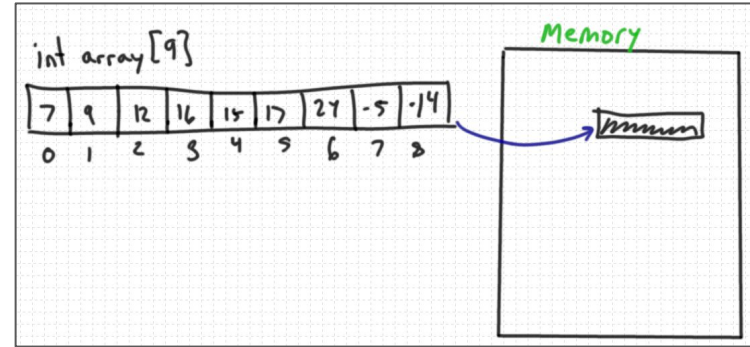
Data Structure Definition

- Similar to our use of a shelf, bookcase, drawer, etc. to organize everyday objects we organizing data in computers with **data structures**

In **computer science**, a **data structure** is a **data** organization, management, and storage format that is usually chosen for **efficient access** to data.^{[1][2][3]} More precisely, a data structure is a collection of data values, the relationships among them, and the functions or operations that can be applied to the data,^[4] i.e., it is an **algebraic structure** about **data**.

Fundamental Data Structure: Built-in Array (1/2)

- In C++ we have built-in arrays which are contiguous chunks of memory.
 - Arrays are a homogeneous data structure where all of the elements are of the same data type.
 - i.e. If you declare 'int array[9]' you have 9 equally sized integers of type 'int'
 - Note: With arrays, we need to be careful not to access something out of bounds (i.e. index 0 to 8 is my range)



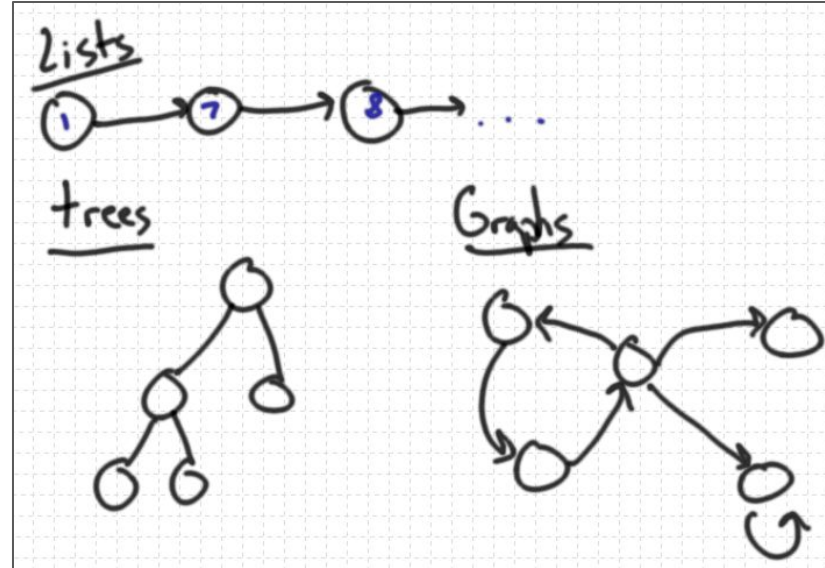
Fundamental Data Structure: Built-in Array (2/2)

- We can allocate an array statically (fixed-size)
 - This array cannot change sizes -- the allocated memory is 'static' or and the size of the array is unchangeable after compiling our code.
- We can allocate an array dynamically at run-time.
 - We acquire a chunk of memory dynamically with 'new' in C++, and we hold the start of that chunk of memory with a pointer of the same type.
 - Through careful management, we can effectively shrink or expand the size of a dynamic array by reallocating a new chunk of memory

```
1 // raw_array.cpp
2 #include <iostream>
3
4 int main(){
5
6     // Fixed-size array at compile-time.
7     // Contiguous chunk of memory for 9 integers
8     // Total memory = 9 * sizeof(int) bytes
9     int fixed_size_raw_array[9];
10
11     // First element of allocation set to '15'
12     fixed_size_raw_array[0] = 15;
13     // Second element (offset of 1) set to '27'
14     fixed_size_raw_array[1] = 21;
15     // ...
16     // Final element (offset of 8) set to 101.
17     fixed_size_raw_array[8] = 101;
18
19
20     // Dynamically allocate array at run-time.
21     // - We do this when we do not know the size
22     // - OR when the allocation is larger than can
23     //   fit in stack memory.
24     int* dynamic_allocation = new int[100000];
25
26     // Similarly, we offset to element with brackets
27     // operator to either read/write data.
28     dynamic_allocation[99999] = 999999;
29
30     // Dynamic allocations must be managed manually
31     delete[] dynamic_allocation;
32
33     return 0;
34 }
```

Fundamental Data Structure: Linked (1/2)

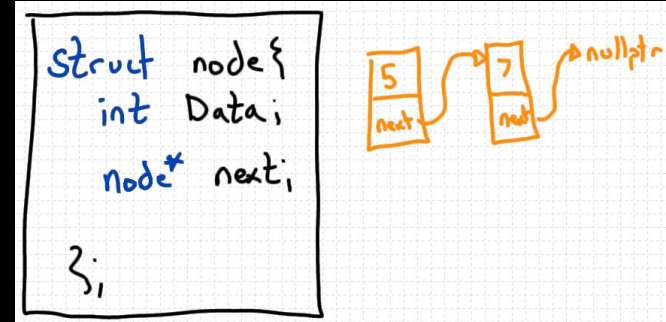
- Another fundamental data structure is a 'linked' data structure
 - Linked data structures are formed of individual pieces of data and associated by 'chaining' them together
 - In C++ **pointers** (which store memory addresses) are the primary mechanism to link data together.



Fundamental Data Structure: Linked (2/2)

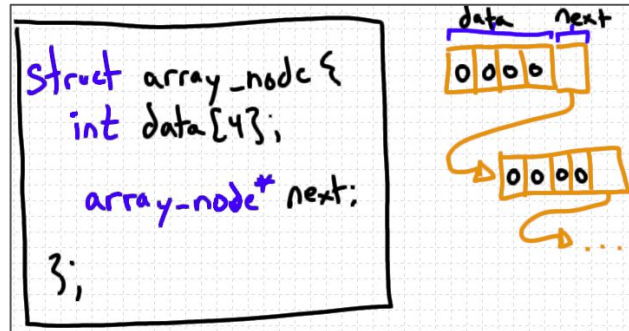
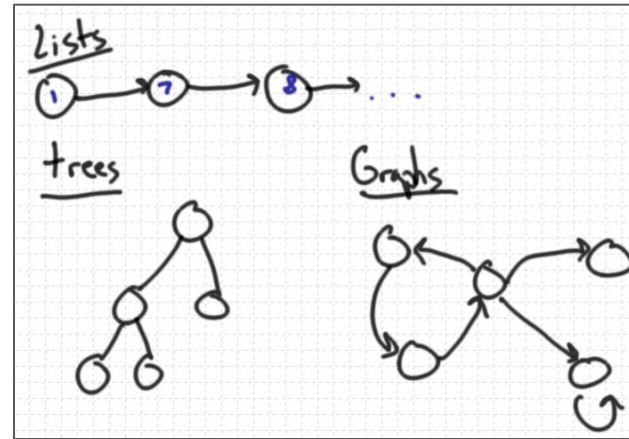
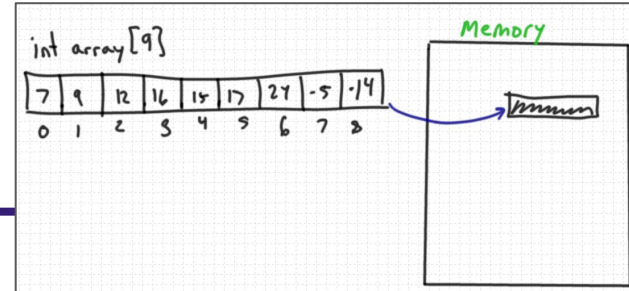
- Usually we wrap together a 'struct' with one member variable as a 'pointer' alongside the data to implement a data linked structure
- Typically these type of data structures are *easy* to expand, because we can add a new link (often called a 'node') to them.
 - However, this again has to be managed.

```
1 // linked.cpp
2 #include <iostream>
3
4 struct node{
5     int data;
6     node* next;
7 };
8
9 int main(){
10
11     node n1;
12     node n2;
13
14     n1.data = 5;
15     n2.data = 7;
16
17     n1.next = &n2;
18     n2.next = nullptr;
19
20     // Print out
21     node* iter = &n1;
22     while(iter!=nullptr){
23         std::cout << iter->data << std::endl;
24         iter=iter->next;
25     }
26
27     return 0;
28 }
```



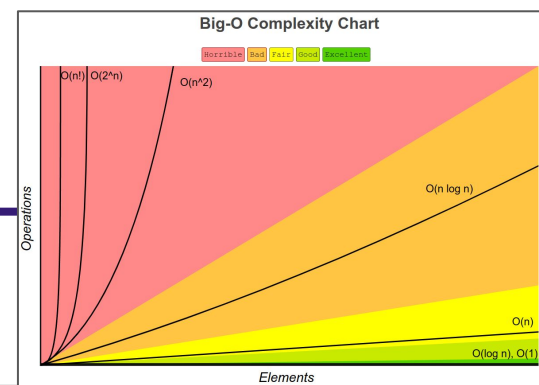
Data Structure Building Blocks

- So our building blocks for data structures are
 - 1.) Built-in Arrays
 - 2.) Linked Nodes (using pointers)
- Note: Also consider the possibilities that we can also create intricate data structures such as linking together arrays from these simple primitives.



Data Structure trade-offs (1/2)

- The abstractions we use to create a data structure create result **trade-offs** in terms of space (storage) and time (run-time):
 - Access time
 - i.e. to retrieve data
 - Search
 - query existence of data
 - Insertion
 - add more data, at the beginning, end, or arbitrary position
 - Deletion
 - Remove data (beginning, end, or arbitrary position)
- There may also be trade-offs regarding:
 - Allocation
 - Fixed or resizable
 - Ease of use/implementation



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

<https://www.bigocheatsheet.com/>

Data Structure trade-offs (2/2)

- The abstractions we use to create result trade-offs of space (storage) and time
 - Access time
 - i.e. to retrieve data
 - Search
 - query existence
 - Insertion
 - add more data, at arbitrary position
 - Deletion
 - Remove data (beginning, end, or arbitrary position)
- There may also be trade-offs regarding:
 - Allocation
 - Fixed or resizable
 - Ease of use/implementation

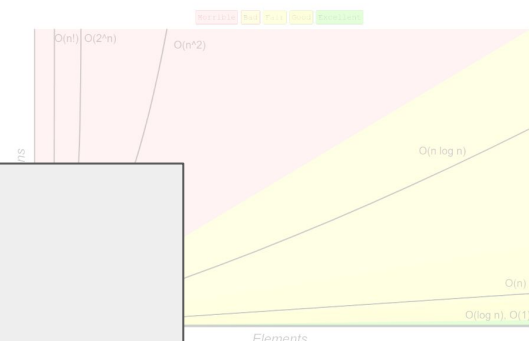
Good News!

The C++ Standard Library comes **powered** with several data structures

I will provide an overview of what is available and how to choose a data structure

| | Search | Insertion | Deletion | Space Complexity Worst |
|--------------------|------------------|------------------|------------------|------------------------|
| Binary Search Tree | $\Theta(\log n)$ | $\Theta(\log n)$ | $\Theta(\log n)$ | $O(n)$ |
| Cartesian Tree | N/A | $\Theta(\log n)$ | $\Theta(\log n)$ | N/A |
| B-Tree | $\Theta(\log n)$ | $\Theta(\log n)$ | $\Theta(\log n)$ | $O(\log n)$ |
| Red-Black Tree | $\Theta(\log n)$ | $\Theta(\log n)$ | $\Theta(\log n)$ | $O(\log n)$ |
| Splay Tree | N/A | $\Theta(\log n)$ | $\Theta(\log n)$ | N/A |
| AVL Tree | $\Theta(\log n)$ | $\Theta(\log n)$ | $\Theta(\log n)$ | $O(\log n)$ |
| KD Tree | $\Theta(\log n)$ | $\Theta(\log n)$ | $\Theta(\log n)$ | $O(n)$ |

<https://www.bigocheatsheet.com/>



| Operations | Search | Insertion | Deletion | Space Complexity Worst |
|----------------|------------------|------------------|------------------|------------------------|
| Array | $\Theta(1)$ | $\Theta(n)$ | $\Theta(n)$ | $O(n)$ |
| Hash Table | $\Theta(1)$ | $\Theta(1)$ | $\Theta(1)$ | $O(n)$ |
| LinkedList | $\Theta(n)$ | $\Theta(1)$ | $\Theta(1)$ | $O(n)$ |
| Stack | $\Theta(1)$ | $\Theta(1)$ | $\Theta(1)$ | $O(n)$ |
| Queue | $\Theta(1)$ | $\Theta(1)$ | $\Theta(1)$ | $O(n)$ |
| Priority Queue | $\Theta(1)$ | $\Theta(n)$ | $\Theta(n)$ | $O(n \log n)$ |
| Heap | $\Theta(1)$ | $\Theta(n)$ | $\Theta(n)$ | $O(n)$ |
| Binary Tree | $\Theta(\log n)$ | $\Theta(\log n)$ | $\Theta(\log n)$ | $O(n)$ |
| B-Tree | $\Theta(\log n)$ | $\Theta(\log n)$ | $\Theta(\log n)$ | $O(\log n)$ |
| Red-Black Tree | $\Theta(\log n)$ | $\Theta(\log n)$ | $\Theta(\log n)$ | $O(\log n)$ |
| AVL Tree | $\Theta(\log n)$ | $\Theta(\log n)$ | $\Theta(\log n)$ | $O(\log n)$ |
| KD Tree | $\Theta(\log n)$ | $\Theta(\log n)$ | $\Theta(\log n)$ | $O(n)$ |



Standard Template Library (STL)

Some History

Standard Template Library (STL) - History

- In 1993 [Alexander Stepanov](#) presented a generics library to the C++ standard committee (in 1994 it was released, and officially adopted in C++98)
 - What this included was many data structures and container structures.
 - Prior to this time there was no 'standard template library' for C++
 - (Folks who have done Java or Python are use to using 'import' to get libraries)
 - Folks rolled their own library of data structures (and many still do this in specific domains)
- The Standard Template Library (STL) provides C++ programmers with a set of standard: algorithms, containers, functions, and iterators.
 - This means regardless of the compiler, we can (for the most part) rely on having a common set of tools to work with and implement C++ in.
 - Most vendors: Clang++, g++, MSVC, etc. have good implementations of the STL available

The C++ Standard Library

- The Standard library offers us many libraries
 - The ‘data structures’ portion we are going to focus on in this talk are listed under the ‘**Containers Library**’
- Note:
 - Things like ‘pair’, ‘tuple’, ‘string’, ‘bitset’, ‘valarray’ are data structures available as well but not discussed today
 - They are special use cases of the generic containers we will talk about today.

C++ reference

C++98, C++03, C++11, C++14, C++17, C++20, C++23 | Compiler support C++11, C++14, C++17, C++20, C++23

| | | |
|---|----------------------------------|--|
| Freestanding implementations | Diagnostics library | Iterators library |
| Language | General utilities library | Ranges library (C++20) |
| Basic concepts | Smart pointers and allocators | Algorithms library |
| Keywords | unique_ptr (C++11) | Numerics library |
| Preprocessor | shared_ptr (C++11) | Common math functions |
| Expressions | Date and time | Mathematical special functions (C++17) |
| Declaration | Function objects – hash (C++11) | Numeric algorithms |
| Initialization | String conversions (C++17) | Pseudo-random number generation |
| Functions | Utility functions | Floating-point environment (C++11) |
| Statements | pair – tuple (C++11) | complex – valarray |
| Classes | optional (C++17) – any (C++17) | Localizations library |
| Overloading | variant (C++17) – format (C++20) | Input/output library |
| Templates | Strings library | Stream-based I/O |
| Exceptions | basic_string | Synchronized output (C++20) |
| Headers | basic_string_view (C++17) | I/O manipulators |
| Named requirements | Null-terminated strings: | Filesystem library (C++17) |
| Feature test macros (C++20) | byte – multibyte – wide | Regular expressions library (C++11) |
| Language support library | Containers library | basic_regex – algorithms |
| Type support – traits (C++11) | array (C++11) – vector – deque | Atomic operations library (C++11) |
| Program utilities | map – unordered_map (C++11) | atomic – atomic_flag |
| Coroutine support (C++20) | set – unordered_set (C++11) | atomic_ref (C++20) |
| Three-way comparison (C++20) | priority_queue – span (C++20) | Thread support library (C++11) |
| numeric_limits – type_info | Other containers: | thread – mutex |
| initializer_list (C++11) | sequence – associative | condition_variable |
| Concepts library (C++20) | unordered associative – adaptors | |
| Technical specifications | | |
| Standard library extensions (library fundamentals TS) | | |
| resource_adaptor – invocation_type | | |
| Standard library extensions v2 (library fundamentals TS v2) | | |
| propagate_const – ostream_joiner – randint | | |
| observer_ptr – detection_idiom | | |
| Standard library extensions v3 (library fundamentals TS v3) | | |
| scope_exit – scope_fail – scope_success – unique_resource | | |
| Concurrency library extensions (concurrency TS) – Transactional Memory (TM TS) | | |
| Concepts (concepts TS) – Ranges (ranges TS) – Reflection (reflection TS) | | |

<https://en.cppreference.com/w/>

C++ reference

C++11, C++14, C++17, C++20, C++23, C++26 | Compiler support C++11, C++14, C++17, C++20, C++23, C++26

Freestanding implementations

ASCII chart

Language

Basic concepts

Keywords

Preprocessor

Expressions

Declarations

Initialization

Functions

Statements

Classes

Overloading

Templates

Exceptions

Standard library (headers)

Named requirements

Feature test macros (C++20)

Language support library

source_location (C++20)

Type support

Program utilities

Coroutine support (C++20)

Three-way comparison (C++20)

numeric_limits – type_info

initializer_list (C++11)

Concepts library (C++20)

Diagnostics library

exception – System error

basic_stacktrace (C++23)

Memory management library

unique_ptr (C++11)

shared_ptr (C++11)

Low level management

Metaprogramming library (C++11)

Type traits – ratio

integer_sequence (C++14)

General utilities library

Function objects – hash (C++11)

Swap – Type operations (C++11)

Integer comparison (C++20)

pair – tuple (C++11)

optional (C++17)

expected (C++23)

variant (C++17) – any (C++17)

String conversions (C++17)

Formatting (C++20)

bitset – Bit manipulation (C++20)

Strings library

basic_string – char_traits

basic_string_view (C++17)

Null-terminated strings:

byte – multibyte – wide

Containers library

array (C++11)

vector – deque

list – forward_list (C++11)

set – multiset

map – multimap

unordered_map (C++11)

unordered_multimap (C++11)

unordered_set (C++11)

unordered_multiset (C++11)

stack – queue – priority_queue

flat_set (C++23)

flat_multiset (C++23)

flat_map (C++23)

flat_multimap (C++23)

span (C++20) – mdspan (C++23)

Iterators library

Ranges library (C++20)

Algorithms library

Execution policies (C++17)

Constrained algorithms (C++20)

Numerics library

Common math functions

Mathematical special functions (C++17)

Mathematical constants (C++20)

Numeric algorithms

Pseudo-random number generation

Floating-point environment (C++11)

complex – valarray

Date and time library

Calendar (C++20) – Time zone (C++20)

Localizations library

locale – Character classification

Input/output library

Print functions (C++23)

Stream-based I/O – I/O manipulators

basic_istream – basic_ostream

Synchronized output (C++20)

Filesystem library (C++17)

path

Regular expressions library (C++11)

basic_regex – algorithms

Concurrency support library (C++11)

thread – jthread (C++20)

atomic – atomic_flag

atomic_ref (C++20)

memory_order – condition_variable

Mutual exclusion – Semaphores (C++20)

future – promise – async

latch (C++20) – barrier (C++20)

C++ reference

C++11, C++14, C++17, C++20, C++23, C++26 | Compiler support C++11, C++14, C++17, C++20, C++23, C++26

Freestanding implementations

ASCII chart

Language

Basic concepts

Keywords

Preprocessor

Expressions

Declarations

Initialization

Functions

Statements

Classes

Overloading

Templates

Exceptions

Standard library (headers)

Named requirements

Feature test macros (C++20)

Language support library

source_location (C++20)

Type support

Program utilities

Coroutine support (C++20)

Three-way comparison (C++20)

numeric_limits – type_info

initializer_list (C++11)

Concepts library (C++20)

Diagnostics library

exception – System error

basic_stacktrace (C++23)

Memory management library

unique_ptr (C++11)

shared_ptr (C++11)

Low level management

Metaprogramming library (C++11)

Type traits – ratio

integer_sequence (C++14)

General utilities library

Function objects – hash (C++11)

Swap – Type operations (C++11)

Integer comparison (C++20)

pair – tuple (C++11)

optional (C++17)

expected (C++23)

variant (C++17) – any (C++17)

String conversions (C++17)

Formatting (C++20)

bitset – Bit manipulation (C++20)

Strings library

basic_string – char_traits

basic_string_view (C++17)

Null-terminated strings:

byte – multibyte – wide

Containers library

array (C++11)

vector – deque

list – forward_list (C++11)

set – multiset

map – multimap

unordered_map (C++11)

unordered_multimap (C++11)

unordered_set (C++11)

unordered_multiset (C++11)

stack – queue – priority_queue

flat_set (C++23)

flat_multiset (C++23)

flat_map (C++23)

flat_multimap (C++23)

span (C++20) – mdspan (C++23)

Iterators library

Ranges library (C++20)

Algorithms library

Execution policies (C++17)

Constrained algorithms (C++20)

Numerics library

Common math functions

Mathematical special functions (C++17)

Mathematical constants (C++20)

Numeric algorithms

Pseudo-random number generation

Floating-point environment (C++11)

complex – valarray

Date and time library

Calendar (C++20) – Time zone (C++20)

Localizations library

locale – Character classification

Input/output library

Print functions (C++23)

Stream-based I/O – I/O manipulators

basic_istream – basic_ostream

Synchronized output (C++20)

Filesystem library (C++17)

path

Regular expressions library (C++11)

basic_regex – algorithms

Concurrency support library (C++11)

thread – jthread (C++20)

atomic – atomic_flag

atomic_ref (C++20)

memory_order – condition_variable

Mutual exclusion – Semaphores (C++20)

future – promise – async

latch (C++20) – barrier (C++20)

C++ reference

C++11, C++14, C++17, C++20, C++23, C++26 | Compiler support C++11, C++14, C++17, C++20, C++23, C++26

Freestanding implementations

ASCII chart

Language

Basic concepts

Keywords

Preprocessor

Expressions

Declarations

Initialization

Functions

Statements

Classes

Overloading

Templates

Exceptions

Standard library

Named requirements

Feature test macros (C++20)

Language support library

source_location (C++20)

Type support

Program utilities

Coroutine support (C++20)

Three-way comparison (C++20)

numeric_limits – type_info

initializer_list (C++11)

Concepts library (C++20)

Diagnostics library

exception – System error

basic_stacktrace (C++23)

Memory management library

unique_ptr (C++11)

shared_ptr (C++11)

Low level management

Metaprogramming library (C++11)

Type traits – ratio

integer_sequence (C++14)

General utilities library

4 main 'categories' of
containers

Containers library

array (C++11)

vector – deque

list – forward_list (C++11)

set – multiset

map – multimap

unordered_map (C++11)

unordered_multimap (C++11)

unordered_set (C++11)

unordered_multiset (C++11)

stack – queue – priority_queue

flat_set (C++23)

flat_multiset (C++23)

flat_map (C++23)

flat_multimap (C++23)

span (C++20) – mdspan (C++23)

Iterators library

Ranges library (C++20)

Algorithms library

Execution policies (C++17)

Constrained algorithms (C++20)

Numerics library

Common math functions

Mathematical special functions (C++17)

Mathematical constants (C++20)

Numeric algorithms

Pseudo-random number generation

Floating-point environment (C++11)

complex – valarray

Date and time library

Calendar (C++20) – Time zone (C++20)

Localizations library

locale – Character classification

Input/output library

Print functions (C++23)

Stream-based I/O – I/O manipulators

basic_istream – basic_ostream

Synchronized output (C++20)

Filesystem library (C++17)

path

Regular expressions library (C++11)

basic_regex – algorithms

Concurrency support library (C++11)

thread – jthread (C++20)

atomic – atomic_flag

atomic_ref (C++20)

memory_order – condition_variable

Mutual exclusion – Semaphores (C++20)

future – promise – async

latch (C++20) – barrier (C++20)

C++ reference

C++11, C++14, C++17, C++20, C++23, C++26 | Compiler support C++11, C++14, C++17, C++20, C++23, C++26

Sequence containers

Sequence containers implement data structures which can be accessed sequentially.

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

Associative containers

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

| | |
|-----------------------|--|
| <code>set</code> | collection of unique keys, sorted by keys (class template) |
| <code>map</code> | collection of key-value pairs, sorted by keys, keys are unique (class template) |
| <code>multiset</code> | collection of keys, sorted by keys (class template) |
| <code>multimap</code> | collection of key-value pairs, sorted by keys (class template) |

17)

Container adaptors

Container adaptors provide a different interface for sequential containers.

| | |
|------------------------------------|---|
| <code>stack</code> | adapts a container to provide stack (LIFO data structure) (class template) |
| <code>queue</code> | adapts a container to provide queue (FIFO data structure) (class template) |
| <code>priority_queue</code> | adapts a container to provide priority queue (class template) |
| <code>flat_set</code> (C++23) | adapts a container to provide a collection of unique keys, sorted by keys (class template) |
| <code>flat_map</code> (C++23) | adapts two containers to provide a collection of key-value pairs, sorted by unique keys (class template) |
| <code>flat_multiset</code> (C++23) | adapts a container to provide a collection of keys, sorted by keys (class template) |
| <code>flat_multimap</code> (C++23) | adapts two containers to provide a collection of key-value pairs, sorted by keys (class template) |

Unordered associative containers (since C++11)

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

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

Sorted strings:

`basic_string` – wide

Containers library

`array` (C++11)
`vector` – `deque`
`list` – `forward_list` (C++11)
`set` – `multiset`
`map` – `multimap`
`unordered_map` (C++11)
`unordered_multimap` (C++11)
`unordered_set` (C++11)
`unordered_multiset` (C++11)
`stack` – `queue` – `priority_queue`
`flat_set` (C++23)
`flat_multiset` (C++23)
`flat_map` (C++23)
`flat_multimap` (C++23)
`span` (C++20) – `mdspan` (C++23)

Language support library

`source_location` (C++20)
Type support
Program utilities
Coroutine support (C++20)
Three-way comparison (C++20)
`numeric_limits` – `type_info`
`initializer_list` (C++11)

Concepts library (C++20)

Diagnostics library

`exception` – System error
`basic_stacktrace` (C++23)

Memory management library

`unique_ptr` (C++11)
`shared_ptr` (C++11)
Low level management

Input/output library

Print functions (C++23)
Stream-based I/O – I/O manipulators
`basic_istream` – `basic_ostream`
Synchronized output (C++20)

Filesystem library (C++17)

`path`

Regular expressions library (C++11)

`basic_regex` – algorithms

Concurrency support library (C++11)

`thread` – `jthread` (C++20)
`atomic` – `atomic_flag`
`atomic_ref` (C++20)
`memory_order` – `condition_variable`
Mutual exclusion – Semaphores (C++20)
`future` – `promise` – `async`
`latch` (C++20) – `barrier` (C++20)

C++ Containers

Containers library

```
array (C++11)
vector – deque
list – forward_list (C++11)
set – multiset
map – multimap
unordered_map (C++11)
unordered_multimap (C++11)
unordered_set (C++11)
unordered_multiset (C++11)
stack – queue – priority_queue
flat_set (C++23)
flat_multiset (C++23)
flat_map (C++23)
flat_multimap (C++23)
span (C++20) – mdspace (C++23)
```

Motivation: Standard Template Library (STL) Containers

- Some thoughts on motivating use of the STL Containers:
 - **First:** These containers are 'generic' -- such that they can be used with any data type.

```
1 // generic
2 #include <vector>
3 #include <array>
4
5 struct UDT{};
6
7 int main(){
8
9     // standard vector container is 'generic'
10    std::vector<int> v1;
11    std::vector<UDT> v2;
12
13    // Class Template Argument Deduction (CTAD)
14    // Can automatically infer template arguments.
15    //
16    // NOTE: I recommend providing arguments regardless
17    std::vector v3 = {1L,2L,3L};
18
19    return 0;
20 }
```

Motivation: Standard Template Library (STL) Containers

- Some thoughts on motivating use of the STL Containers:
 - First:** These containers are 'generic' -- such that they can be used with any data type.

Observe, in many cases I can also simply change the container as well -- as the interfaces are often identical!

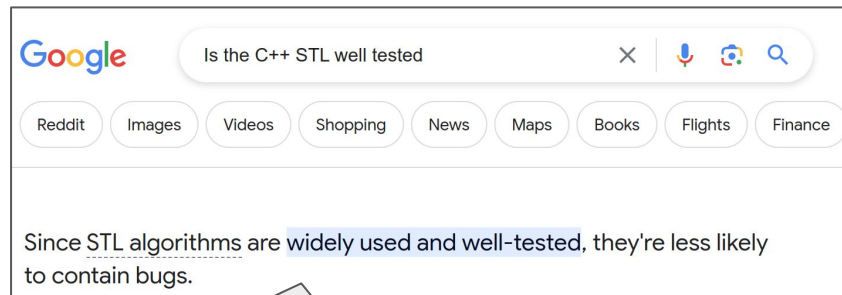
Simply swap container and see if it gives you the desired performance/behavior you need!

```
1 // generic2.cpp
2 #include <vector>
3 #include <list>
4
5 // alias template
6 template<class T>
7 using MyType = std::vector<T>;
8
9 struct UDT{};
10
11 int main(){
12
13     // standard vector container is 'generic'
14     MyType<int> v1;
15     MyType<UDT> v2;
16
17     // Class Template Argument Deduction (CTAD)
18     // Can automatically infer template arguments.
19     //
20     // NOTE: I recommend providing arguments regardless
21     MyType<long> v3 = {1L,2L,3L};
22
23     return 0;
24 }
```

```
1 // generic2.cpp
2 #include <vector>
3 #include <list>
4
5 // alias template
6 template<class T>
7 using MyType = std::list<T>;
8
9 struct UDT{};
10
11 int main(){
12
13     // standard vector container is 'generic'
14     MyType<int> v1;
15     MyType<UDT> v2;
16
17     // Class Template Argument Deduction (CTAD)
18     // Can automatically infer template arguments.
19     //
20     // NOTE: I recommend providing arguments regardless
21     MyType<long> v3 = {1L,2L,3L};
22
23     return 0;
24 }
```

Motivation: Standard Template Library (STL) Containers

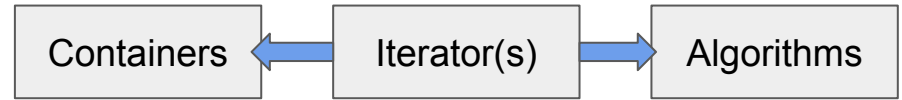
- Some thoughts on motivating use of the STL Containers:
 - **First:** These containers are 'generic' -- such that they can be used with any data type.
 - **Second:** The Containers are well tested, and used by many developers.



Note: I do not have a statistic to prove this-- but keep in mind the STL is the result of decades and millions of C++ programmers writing code, multiple compiler vendors, and top library writers making and testing contributions..

Motivation: Standard Template Library (STL) Containers

- Some thoughts on motivating use of the STL Containers:
 - **First:** These containers are 'generic' -- such that they can be used with any data type.
 - **Second:** The Containers are well tested, and used by many developers.
 - **Third (and final point):** STL Containers work well with rest of the standard library
 - We have 100+ algorithms in the STL available to use with containers



```
1 // iterator.cpp
2 #include <iostream>
3 #include <vector>
4 #include <algorithm>
5
6 int Generate(){
7     static int i=0;
8     return ++i;
9 }
10
11 int main(){
12
13     // Generic collection
14     std::vector<int> v1;
15
16     // Algorithm to generate '5' values from a function
17     std::generate_n(std::back_inserter(v1), 5, Generate);
18
19     // 'Filter' elements by copying any value greater
20     // than '2' to a new collection.
21     std::vector<int> results;
22     std::copy_if(v1.begin(), v1.end(), std::back_inserter(results),
23                [](int x){
24                    return x > 2;
25                });
26
27     // Range-based for loop to output results
28     for(auto elem: results){
29         std::cout << elem << std::endl;
30     }
31
32     return 0;
33 }
```

Sequence containers

Sequence containers implement data structures which can be accessed sequentially.

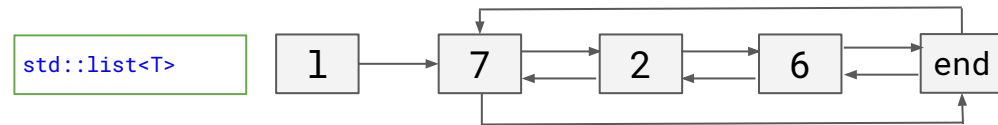
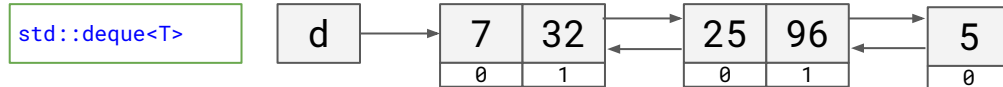
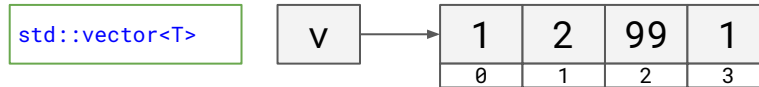
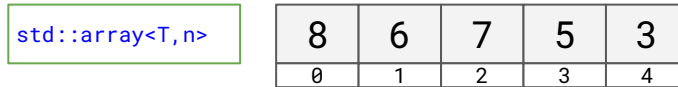
| | |
|-----------------------------|--|
| array (C++11) | static contiguous array (class template) |
| vector | dynamic contiguous array (class template) |
| deque | double-ended queue (class template) |
| forward_list (C++11) | singly-linked list (class template) |
| list | doubly-linked list (class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

Sequence Containers

Sequence Containers

- Containers that can be accessed sequentially
 - Each container has a linear (i.e. line-like) arrangement/shape
 - i.e. Can move from one element to the next
- Observe some sequence containers implemented with 'arrays' and some are 'linked' data structures.

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



```
std::array<T,n>
```

| | | | | |
|---|---|---|---|---|
| 8 | 6 | 7 | 5 | 3 |
| 0 | 1 | 2 | 3 | 4 |

std::array (1/3)

std::array

Defined in header `<array>`

```
template<
    class T,
    std::size_t N (since C++11)
> struct array;
```



Quick Snapshot

Element access

| | |
|---------------------------------|---|
| <code>at</code> (C++11) | access specified element with bounds checking (public member function) |
| <code>operator[]</code> (C++11) | access specified element (public member function) |
| <code>front</code> (C++11) | access the first element (public member function) |
| <code>back</code> (C++11) | access the last element (public member function) |
| <code>data</code> (C++11) | direct access to the underlying array (public member function) |

Capacity

| | |
|-------------------------------|---|
| <code>empty</code> (C++11) | checks whether the container is empty (public member function) |
| <code>size</code> (C++11) | returns the number of elements (public member function) |
| <code>max_size</code> (C++11) | returns the maximum possible number of elements (public member function) |

Operations

| | |
|---------------------------|---|
| <code>fill</code> (C++11) | fill the container with specified value (public member function) |
| <code>swap</code> (C++11) | swaps the contents (public member function) |


```
std::array<T,n>
```

| | | | | |
|---|---|---|---|---|
| 8 | 6 | 7 | 5 | 3 |
| 0 | 1 | 2 | 3 | 4 |

std::array (2/3)

- In C++ 11 we got a new 'array' in the STL
 - The std::array container is functionally exactly the same as a regular raw array.
 - It is a stack-allocated contiguous array
 - Size is fixed at compile-time
- We 'prefer' to use std::array versus raw array (See [to_array](#) for conversion)
 - Can perform bounds checking (e.g. .at() member function)
 - Several useful member functions available
 - (See example on right)
 - Does not decay to a pointer in a function (i.e. it's very clear what data we are passing)

```
10 // @file std_array3.cpp
11 // Bring in a header file on our include path
12 // this happens to be in the standard library
13 // (i.e. default compiler path)
14 #include <iostream>
15 #include <array> // Include the array library
16
17 // Entry point to program 'main' function
18 int main(int argc, char* argv[]){
19
20     // The first parameter is the type
21     // The second parameter is the 'fixed size' of
22     // the array. i.e. how many elements we can store.
23     std::array<int,5> myArray;
24
25     // We can access an array with the index operator,
26     // that is the []'s
27     myArray[0] = 7;
28     // We can use member functions in our array as well
29     // with the .at(position)
30     myArray.at(0) = 9;
31
32     // Some other member functions.
33     // Let's test what they return in GDB!
34     myArray.front();
35     myArray.back();
36     myArray.size();
37     myArray.max_size();
38
39     return 0;
40 }
```

`std::array<T,n>`

| | | | | |
|---|---|---|---|---|
| 8 | 6 | 7 | 5 | 3 |
| 0 | 1 | 2 | 3 | 4 |

std::array (3/3)

Behavior/Performance characteristics

- Allocation:
 - Static and fixed at compile-time
- Access:
 - Random access with an offset into the array
- Search:
 - $O(n)$ if unsorted (i.e. linear search)
 - $O(\log_2 n)$ if sorted (i.e. binary search)
- Notes:
 - `std::array` always knows its length
 - Nicer interface (similar to `std::vector`) versus raw arrays.
 - Prefer curly brace initialization
 - `std::array<int,5> a; // uninitialized memory`
 - `std::array<int,5> a{}; // Default initializes`
 - `std::array<int,5> a{1,2}; // Default initializes remainder`

```
10 // @file std_array3.cpp
11 // Bring in a header file on our include path
12 // this happens to be in the standard library
13 // (i.e. default compiler path)
14 #include <iostream>
15 #include <array> // Include the array library
16
17 // Entry point to program 'main' function
18 int main(int argc, char* argv[]){
19
20     // The first parameter is the type
21     // The second parameter is the 'fixed size' of
22     // the array. i.e. how many elements we can store.
23     std::array<int,5> myArray;
24
25     // We can access an array with the index operator,
26     // that is the []'s
27     myArray[0] = 7;
28     // We can use member functions in our array as well
29     // with the .at(position)
30     myArray.at(0) = 9;
31
32     // Some other member functions.
33     // Let's test what they return in GDB!
34     myArray.front();
35     myArray.back();
36     myArray.size();
37     myArray.max_size();
38
39     return 0;
40 }
```

```
std::vector<T>
```

v

| | | | |
|---|---|----|---|
| 1 | 2 | 99 | 1 |
| 0 | 1 | 2 | 3 |

std::vector (1/14)

std::vector

Defined in header `<vector>`

```
template<
    class T,
    class Allocator = std::allocator<T>
> class vector;
```

Quick Snapshot

Element access

| | |
|-------------------------|---|
| <code>at</code> | access specified element with bounds checking (public member function) |
| <code>operator[]</code> | access specified element (public member function) |
| <code>front</code> | access the first element (public member function) |
| <code>back</code> | access the last element (public member function) |
| <code>data</code> | direct access to the underlying array (public member function) |

Capacity

| | |
|----------------------------------|--|
| <code>empty</code> | checks whether the container is empty (public member function) |
| <code>size</code> | returns the number of elements (public member function) |
| <code>max_size</code> | returns the maximum possible number of elements (public member function) |
| <code>reserve</code> | reserves storage (public member function) |
| <code>capacity</code> | returns the number of elements that can be held in currently allocated storage (public member function) |
| <code>shrink_to_fit</code> (DR*) | reduces memory usage by freeing unused memory (public member function) |

Modifiers

| | |
|-----------------------------------|---|
| <code>clear</code> | clears the contents (public member function) |
| <code>insert</code> | inserts elements (public member function) |
| <code>insert_range</code> (C++23) | inserts a range of elements (public member function) |
| <code>emplace</code> (C++11) | constructs element in-place (public member function) |
| <code>erase</code> | erases elements (public member function) |

`std::vector<T>`

v

| | | | |
|---|---|----|---|
| 1 | 2 | 99 | 1 |
| 0 | 1 | 2 | 3 |

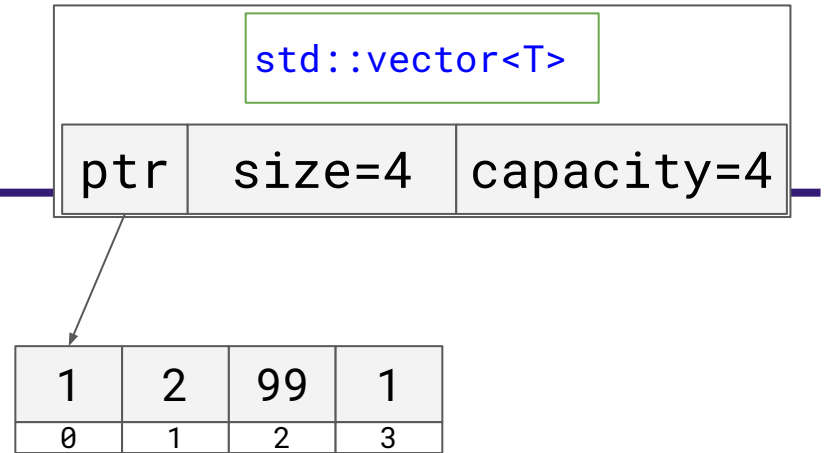
std::vector (2/14)

- A `std::vector` in C++ is not to be confused with a mathematical vector from linear algebra
- A `std::vector` is a resizable array (i.e. a dynamic array)
 - We can push (expand) in as many elements as we want to the vector
 - And we can remove (shrink) existing elements from a vector when no longer needed.

```
10 // @file std_vector.cpp
11 // Bring in a header file on our include path
12 // this happens to be in the standard library
13 // (i.e. default compiler path)
14 #include <iostream>
15 #include <vector>    // Include the vector library
16
17 // Entry point to program 'main' function
18 int main(int argc, char* argv[]){
19
20     // The first parameter is the type
21     std::vector<int> myVector;
22
23     // Add elements to our vector
24     myVector.push_back(1);
25     myVector.push_back(2);
26     myVector.push_back(3);
27     // Access element at a specific position
28     myVector.at(2);
29     // Remove the last element
30     myVector.pop_back();
31     // Print everything in the vector
32     for(int i=0; i < myVector.size(); i++){
33         // Use the [ ] operator to access an element
34         // at the i'th position in the vector
35         std::cout << myVector[i] << std::endl;
36     }
37
38     return 0;
39 }
```

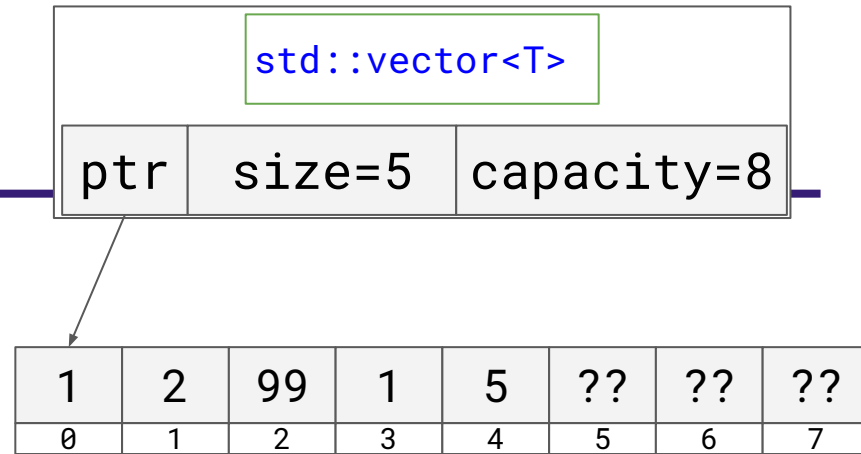
std::vector (3/14)

- It's worth noting that a vector is 'heap allocated'
 - Observe the visualization to the right to best understand that a vector keeps track of the 'size' and 'capacity' of the allocated memory
 - (next slide)

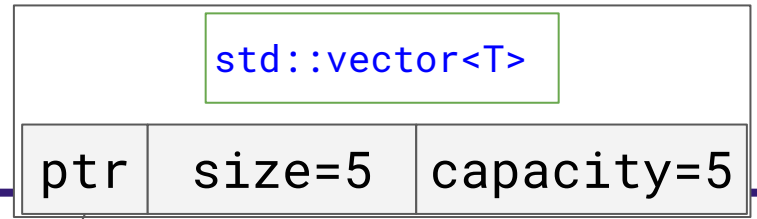


std::vector (4/14)

- It's worth noting that a vector is 'heap allocated'
 - Observe the visualization to the right to best understand that a vector keeps track of the 'size' and 'capacity' of the allocated memory
- When we 'push_back(5)' another element will be added.
 - Sometimes this forces a reallocation.
 - It's typical that a vector's capacity may increase by some factor (e.g. 1.6, 2.0, etc.)



std::vector (5/14)

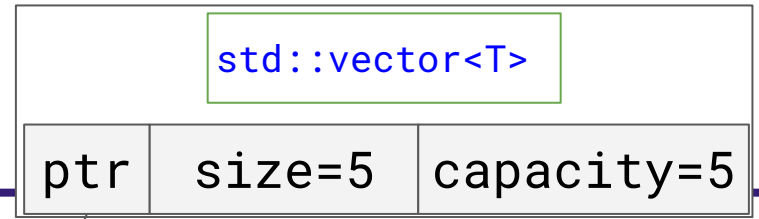


- [shrink_to_fit\(\)](#)
 - A request to try to remove any unused capacity in case our vector starts growing too large (think millions of large objects stored)
 - (Behavior implementation defined)

A diagram showing the data storage of a vector. It is a 2x5 grid. The top row contains the values 1, 2, 99, 1, and 5. The bottom row contains the indices 0, 1, 2, 3, and 4. An arrow points from the `ptr` field in the metadata box above to the first cell of this grid.

| | | | | |
|---|---|----|---|---|
| 1 | 2 | 99 | 1 | 5 |
| 0 | 1 | 2 | 3 | 4 |

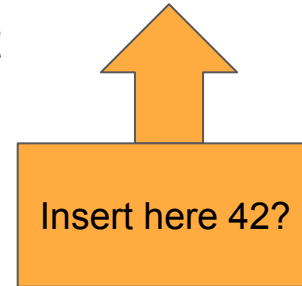
std::vector (6/14)



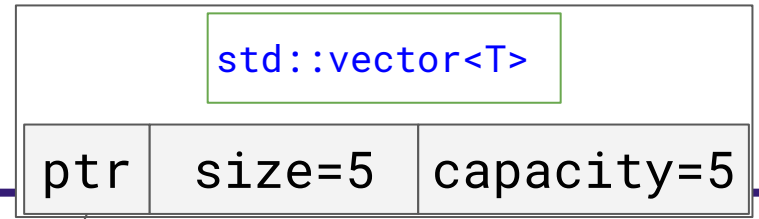
- What if I want to insert in the beginning or middle of a contiguous data structure?
 - `insert()` does allow us to insert at an arbitrary position
- Consider what must happen, and what the performance must be.
 - (next slide for answer)

A diagram of a contiguous data structure represented as a 2x5 grid. The top row contains the values 1, 2, 99, 1, and 5. The bottom row contains the indices 0, 1, 2, 3, and 4. An arrow points from the `ptr` field of the `std::vector` diagram above to the first cell (index 0, value 1).

| | | | | |
|---|---|----|---|---|
| 1 | 2 | 99 | 1 | 5 |
| 0 | 1 | 2 | 3 | 4 |



std::vector (7/14)

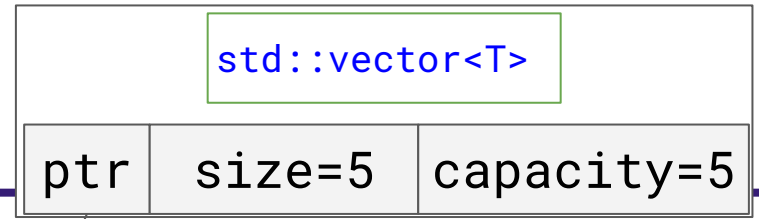


- What if I want to insert in the beginning or middle of a contiguous data structure?
 - `insert()` does allow us to insert at an arbitrary position
- Consider what must happen, and what the performance must be.
 - First need to dynamically allocate new memory large enough (potentially 1.6x or 2.0x the size)

| | | | | |
|---|---|----|---|---|
| 1 | 2 | 99 | 1 | 5 |
| 0 | 1 | 2 | 3 | 4 |

| | | | | | |
|----|----|----|----|----|----|
| ?? | ?? | ?? | ?? | ?? | ?? |
| 0 | 1 | 2 | 3 | 4 | 4 |

std::vector (8/14)

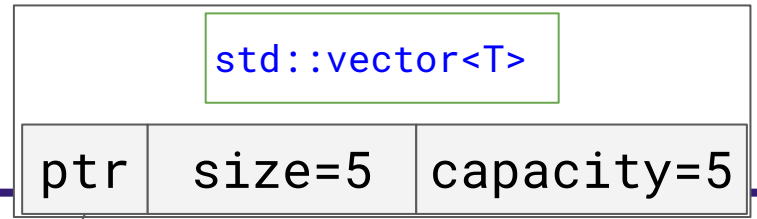


- What if I want to insert in the beginning or middle of a contiguous data structure?
 - `insert()` does allow us to insert at an arbitrary position
- Consider what must happen, and what the performance must be.
 - First need to dynamically allocate new memory large enough (potentially 1.6x or 2.0x the size)
 - Then copy elements (could be expensive if copy is expensive!)

| | | | | |
|---|---|----|---|---|
| 1 | 2 | 99 | 1 | 5 |
| 0 | 1 | 2 | 3 | 4 |

| | | | | | |
|---|----|---|----|---|---|
| 1 | ?? | 2 | 99 | 1 | 5 |
| 0 | 1 | 2 | 3 | 4 | 4 |

std::vector (9/14)

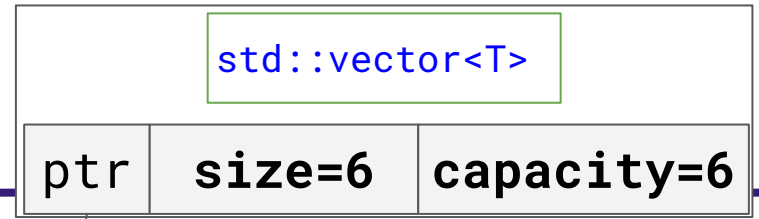


| | | | | |
|---|---|----|---|---|
| 1 | 2 | 99 | 1 | 5 |
| 0 | 1 | 2 | 3 | 4 |

| | | | | | |
|---|----|---|----|---|---|
| 1 | 42 | 2 | 99 | 1 | 5 |
| 0 | 1 | 2 | 3 | 4 | 4 |

- What if I want to insert in the beginning or middle of a contiguous data structure?
 - `insert()` does allow us to insert at an arbitrary position
- Consider what must happen, and what the performance must be.
 - First need to dynamically allocate new memory large enough (potentially 1.6x or 2.0x the size)
 - Then copy elements (could be expensive if copy is expensive!)
 - Insert our element

std::vector (10/14)



- What if I want to insert in the beginning or middle of a contiguous data structure?
 - `insert()` does allow us to insert at an arbitrary position
- Consider what must happen, and what the performance must be.
 - First need to dynamically allocate new memory large enough (potentially 1.6x or 2.0x the size)
 - Then copy elements (could be expensive if copy is expensive!)
 - Insert our element
 - Update our pointer, and reclaim memory

A diagram showing the initial state of a vector. It is a 2x5 grid. The top row contains the values 1, 2, 99, 1, 5. The bottom row contains the indices 0, 1, 2, 3, 4. An arrow points from the `ptr` field in the metadata box above to the top-left cell (index 0).

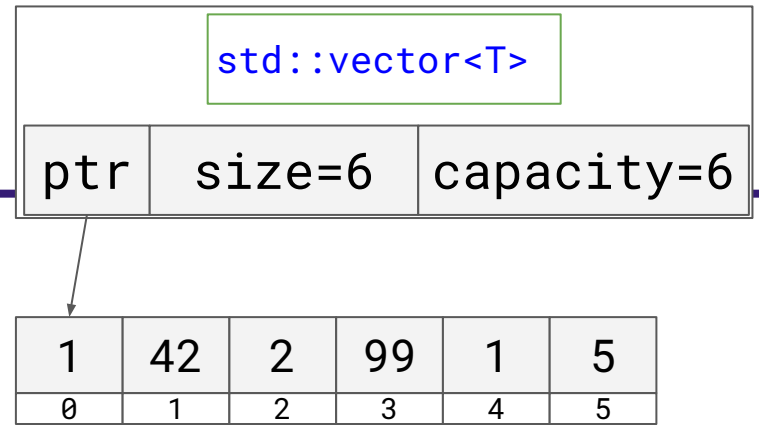
| | | | | |
|---|---|----|---|---|
| 1 | 2 | 99 | 1 | 5 |
| 0 | 1 | 2 | 3 | 4 |

A diagram showing the vector after inserting the value 42 at index 1. It is a 2x6 grid. The top row contains the values 1, 42, 2, 99, 1, 5. The bottom row contains the indices 0, 1, 2, 3, 4, 5. An arrow points from the `ptr` field in the metadata box above to the top-left cell (index 0).

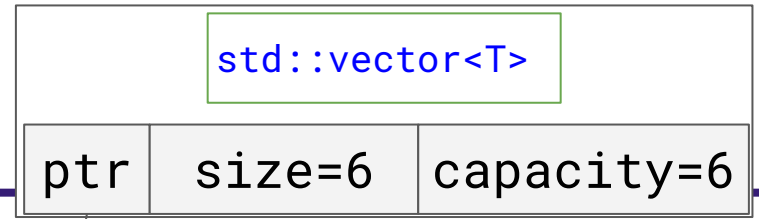
| | | | | | |
|---|----|---|----|---|---|
| 1 | 42 | 2 | 99 | 1 | 5 |
| 0 | 1 | 2 | 3 | 4 | 5 |

std::vector (11/14)

- So std::vector are more flexible than std::array,
 - We have to be careful if we perform insertion (and removal) operations at anywhere other than the end of the std::vector however.
- **Good news however** -- all of the hard work to reallocate is done for us, we just need to use the vector interface
 - It's useful however to see exactly how this stuff works.



std::vector (12/14)



| | | | | | |
|---|----|---|----|---|---|
| 1 | 42 | 2 | 99 | 1 | 5 |
| 0 | 1 | 2 | 3 | 4 | 5 |

- **Aside:**
 - There are some other useful tricks like using `reserve(size_t n)` to set the vector capacity when you initially allocate it.
 - This prevents too many 'reallocations' if you are going to populate the vector with `push_back` -- especially when first initializing the data structure.

`std::vector<T>`

v

| | | | |
|---|---|----|---|
| 1 | 2 | 99 | 1 |
| 0 | 1 | 2 | 3 |

std::vector (13/14)

Behavior/Performance characteristics

- Allocation:
 - Dynamic
- Access:
 - Random access with an offset into the array
- Search:
 - $O(n)$ if unsorted (i.e. linear search)
 - $O(\log_2 n)$ if sorted (i.e. binary search)
- Notes:
 - The default data structure for performance and flexibility
 - (i.e. Probably what you'll use most frequently)

```
10 // @file std_vector.cpp
11 // Bring in a header file on our include path
12 // this happens to be in the standard library
13 // (i.e. default compiler path)
14 #include <iostream>
15 #include <vector> // Include the vector library
16
17 // Entry point to program 'main' function
18 int main(int argc, char* argv[]){
19
20     // The first parameter is the type
21     std::vector<int> myVector;
22
23     // Add elements to our vector
24     myVector.push_back(1);
25     myVector.push_back(2);
26     myVector.push_back(3);
27     // Access element at a specific position
28     myVector.at(2);
29     // Remove the last element
30     myVector.pop_back();
31     // Print everything in the vector
32     for(int i=0; i < myVector.size(); i++){
33         // Use the [ ] operator to access an element
34         // at the i'th position in the vector
35         std::cout << myVector[i] << std::endl;
36     }
37
38     return 0;
39 }
```

`std::vector<T>`

v

1

2

99

1

0

1

2

3

std::vector (14/14)

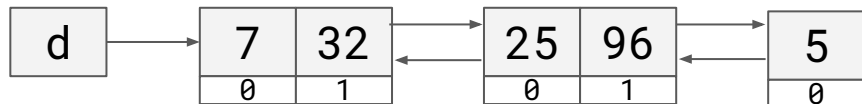
Behavior/Performance characteristics

- More Notes:
 - Optimized for operations at end of the data structure
 - Some mitigation of copying can be done with `.reserve`
 - Use `shrink_to_fit()` to minimize capacity
 - `std::string` effectively a vector that is optimized for primitive types
 - Though `std::vector` of `std::byte` may be useful for a different use case.

```
10 // @file std_vector.cpp
11 // Bring in a header file on our include path
12 // this happens to be in the standard library
13 // (i.e. default compiler path)
14 #include <iostream>
15 #include <vector> // Include the vector library
16
17 // Entry point to program 'main' function
18 int main(int argc, char* argv){
19
20     // The first parameter is the type
21     std::vector<int> myVector;
22
23     // Add elements to our vector
24     myVector.push_back(1);
25     myVector.push_back(2);
26     myVector.push_back(3);
27     // Access element at a specific position
28     myVector.at(2);
29     // Remove the last element
30     myVector.pop_back();
31     // Print everything in the vector
32     for(int i=0; i < myVector.size(); i++){
33         // Use the [ ] operator to access an element
34         // at the i'th position in the vector
35         std::cout << myVector[i] << std::endl;
36     }
37
38     return 0;
39 }
```



```
std::deque<T>
```



std::deque (1/5)

std::deque

Defined in header `<deque>`

```
template<
    class T,
    class Allocator = std::allocator<T>
> class deque;
```

Quick Snapshot

Element access

| | |
|-------------------------|---|
| <code>at</code> | access specified element with bounds checking (public member function) |
| <code>operator[]</code> | access specified element (public member function) |
| <code>front</code> | access the first element (public member function) |
| <code>back</code> | access the last element (public member function) |

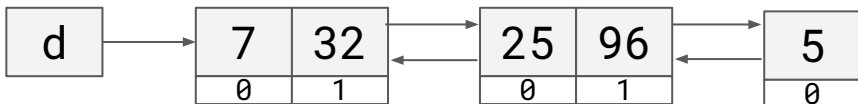
Capacity

| | |
|----------------------------------|---|
| <code>empty</code> | checks whether the container is empty (public member function) |
| <code>size</code> | returns the number of elements (public member function) |
| <code>max_size</code> | returns the maximum possible number of elements (public member function) |
| <code>shrink_to_fit</code> (DR*) | reduces memory usage by freeing unused memory (public member function) |

Modifiers

| | |
|-----------------------------------|---|
| <code>clear</code> | clears the contents (public member function) |
| <code>insert</code> | inserts elements (public member function) |
| <code>insert_range</code> (C++23) | inserts a range of elements (public member function) |
| <code>emplace</code> (C++11) | constructs element in-place (public member function) |
| <code>erase</code> | erases elements (public member function) |
| <code>push_back</code> | adds an element to the end (public member function) |
| <code>emplace_back</code> (C++11) | constructs an element in-place at the end (public member function) |
| <code>append_range</code> (C++23) | adds a range of elements to the end (public member function) |
| <code>pop_back</code> | removes the last element (public member function) |
| <code>push_front</code> | inserts an element to the beginning (public member function) |

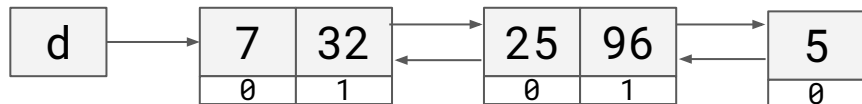
```
std::deque<T>
```



std::deque (2/5)

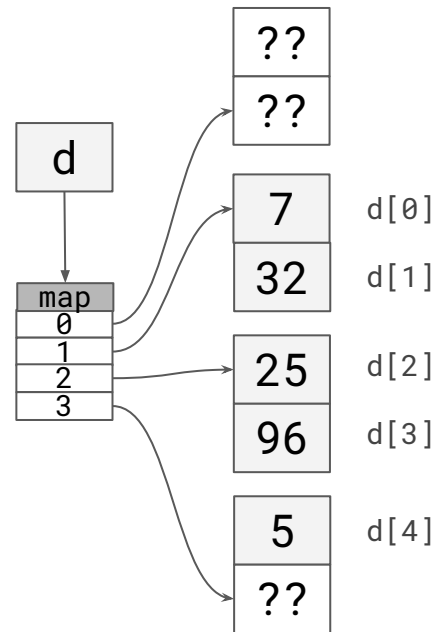
- `std::deque`'s are 'double-ended queues'
 - A careful observation is that they are not typically implemented as a contiguous data structure
 - The top-right view is how we can think of them.
 - Internally however -- there are links between fixed-size arrays.
 - (next slide)

```
std::deque<T>
```

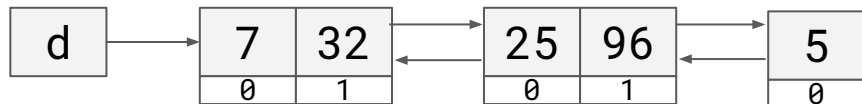


std::deque (3/5)

- std::deque is a 'double-ended queue'
 - A careful observation is that they are not typically implemented as a contiguous data structure
 - The top-right view is how we can think of them.
 - Internally however -- there are links between fixed-size arrays.
 - Some data structure (could be as simple as a std::vector<chunks>) points us to the correct fixed-size element.
 - So what does this mean?
 - (next slide)



```
std::deque<T>
```

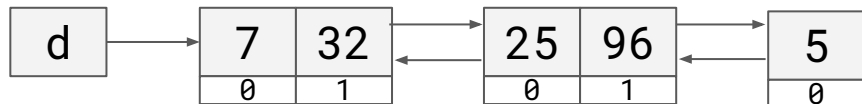


std::deque (4/5)

- std::deque allows for insertion at both the front() and back() of the data structure in constant time!
 - It's easy to allocate another fixed-size array.

```
1 // deque.cpp
2 #include <iostream>
3 #include <deque>
4 #include <algorithm>
5 #include <iterator>
6
7 int main(){
8
9     // Double-ended queue
10    // Usually implemented as links of
11    // fixed-size arrays
12    std::deque<int> d{1,2,3,4};
13
14    // Ability to push on front and back quickly!
15    d.push_back(123);
16    d.push_front(999);
17
18    // Prints out 999,1,2,3,4,123
19    auto print = [](int a){ std::cout << a << ", "; };
20    std::for_each(d.cbegin(),d.cend(), print);
21
22    return 0;
23 }
```

`std::deque<T>`



std::deque (5/5)

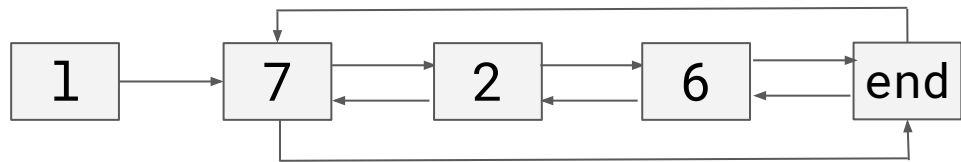
Behavior/Performance characteristics

- Allocation:
 - Dynamically allocated (can resize)
- Access:
 - Constant time and Random access with an offset
- Search:
 - $O(n)$ if unsorted (i.e. linear search)
 - $O(\log_2 n)$ if sorted (i.e. binary search)
- Notes:
 - Ability to `resize()` if needed
 - Slightly extends upon `std::vector` interface with ability to work with first elements.

```
1 // deque.cpp
2 #include <iostream>
3 #include <deque>
4 #include <algorithm>
5 #include <iterator>
6
7 int main(){
8
9     // Double-ended queue
10    // Usually implemented as links of
11    // fixed-size arrays
12    std::deque<int> d{1,2,3,4};
13
14    // Ability to push on front and back quickly!
15    d.push_back(123);
16    d.push_front(999);
17
18    // Prints out 999,1,2,3,4,123
19    auto print = [](int a){ std::cout << a << ", "; };
20    std::for_each(d.cbegin(),d.cend(), print);
21
22    return 0;
23 }
```

std::list (1/4)

```
std::list<T>
```



std::list

Defined in header `<list>`

```
template<
    class T,
    class Allocator = std::allocator<T>
> class list;
```

Quick Snapshot

Element access

| | |
|--------------------|--|
| <code>front</code> | access the first element (public member function) |
| <code>back</code> | access the last element (public member function) |

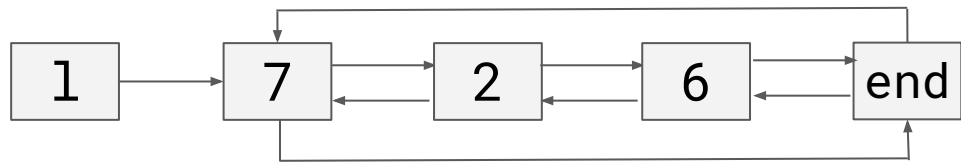
Capacity

| | |
|-----------------------|---|
| <code>empty</code> | checks whether the container is empty (public member function) |
| <code>size</code> | returns the number of elements (public member function) |
| <code>max_size</code> | returns the maximum possible number of elements (public member function) |

Modifiers

| | |
|-----------------------------------|---|
| <code>clear</code> | clears the contents (public member function) |
| <code>insert</code> | inserts elements (public member function) |
| <code>insert_range</code> (C++23) | inserts a range of elements (public member function) |
| <code>emplace</code> (C++11) | constructs element in-place (public member function) |
| <code>erase</code> | erases elements (public member function) |
| <code>push_back</code> | adds an element to the end (public member function) |
| <code>emplace_back</code> (C++11) | constructs an element in-place at the end (public member function) |
| <code>append_range</code> (C++23) | adds a range of elements to the end |

`std::list<T>`

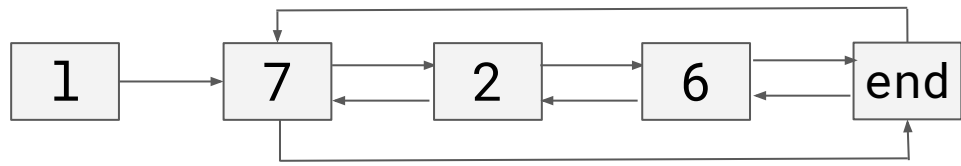


std::list (2/4)

- `std::list` is (usually) an implementation of a doubly-linked list.
 - This means we have fast insertion and deletion
 - Note: We also do not have to worry so much about iterator invalidation!
 - (More on this later)
 - It's also worth noting there exist member functions like '[remove_if](#)' that have slightly different semantics than `std::remove_if`
 - Meaning, we do in fact 'erase' elements.

```
1 // list.cpp
2 #include <iostream>
3 #include <list>
4
5 void printList(const std::list<int>& list){
6     std::cout << "=====\n";
7     for(const auto& e: list){
8         std::cout << e << " ";
9     }
10    std::cout << std::endl;
11 }
12
13 int main(){
14
15     std::list<int> myList;
16     myList.push_back(1);
17     myList.push_back(2);
18     myList.push_back(3);
19
20     // O(1) if front or end, otherwise O(n)
21     myList.insert(begin(myList),5);
22     myList.insert(end(myList),0);
23
24     auto it = cbegin(myList);
25     std::advance(it,myList.size()/2);
26     std::cout << "middle is: " << *it << std::endl;
27
28     myList.sort();
29     myList.reverse();
30
31     // member function 'erases' elements for us
32     myList.remove_if([](int n) {return n<2;});
33
34     printList(myList);
35
36     return 0;
37 }
```

`std::list<T>`



`std::list` (3/4)

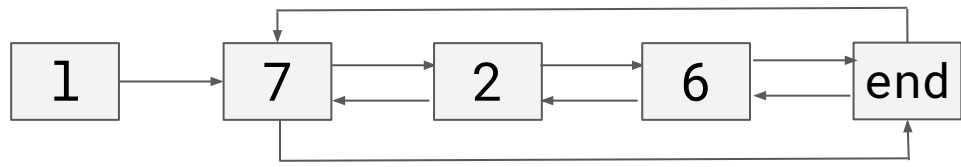
- (Aside) Just another example to show how to splice (move elements) from one list to the other.

```
1 // list2.cpp
2 #include <iostream>
3 #include <list>
4
5 void printList(const std::list<int>& list){
6     std::cout << "=====\n";
7     for(const auto& e: list){
8         std::cout << e << ", ";
9     }
10    std::cout << std::endl;
11 }
12
13 int main(){
14
15     std::list<int> myList;
16     myList.push_back(1);
17     myList.push_back(2);
18     myList.push_back(3);
19
20     // Splice example
21     std::list<int> list3{15,25,35,45};
22     auto list3_iter = list3.begin();
23     std::advance(list3_iter,2);
24
25     myList.splice(end(myList),
26                 list3,
27                 list3_iter,
28                 end(list3));
29
30     printList(myList);
31     printList(list3);
32
33     return 0;
34 }
```

```
=====  
1,2,3,35,45,  
=====  
15,25,
```



```
std::list<T>
```



std::list (4/4)

Behavior/Performance characteristics

- Allocation:
 - Dynamic, just keep adding nodes as needed
- Access:
 - $O(n)$ -- need to traverse list
 - $O(1)$ for first (front) and last (back) element
- Search:
 - Linear -- need to traverse list
- Notes:
 - Take advantage of optimized member functions (rather than generic `<algorithm>`'s) for better performance/behavior

```
1 // list.cpp
2 #include <iostream>
3 #include <list>
4
5 void printList(const std::list<int>& list){
6     std::cout << "=====\n";
7     for(const auto& e: list){
8         std::cout << e << ", ";
9     }
10    std::cout << std::endl;
11 }
12
13 int main(){
14
15     std::list<int> myList;
16     myList.push_back(1);
17     myList.push_back(2);
18     myList.push_back(3);
19
20     // O(1) if front or end, otherwise O(n)
21     myList.insert(begin(myList),5);
22     myList.insert(end(myList),0);
23
24     auto it = cbegin(myList);
25     std::advance(it,myList.size()/2);
26     std::cout << "middle is: " << *it << std::endl;
27
28     myList.sort();
29     myList.reverse();
30
31     // member function 'erases' elements for us
32     myList.remove_if([](int n) {return n<2;});
33
34     printList(myList);
35
36     return 0;
37 }
```

```
std::forward_list<T>
```



std::forward_list (1/3)

std::forward_list

Defined in header `<forward_list>`

```
template<
    class T,
    class Allocator = std::allocator<T>
> class forward_list;
```

Element access

`front` (C++11)

access the first element
(public member function)

Capacity

`empty` (C++11)

checks whether the container is empty
(public member function)

`max_size` (C++11)

returns the maximum possible number of elements
(public member function)

Modifiers

`clear` (C++11)

clears the contents
(public member function)

`insert_after` (C++11)

inserts elements after an element
(public member function)

`emplace_after` (C++11)

constructs elements in-place after an element
(public member function)

`insert_range_after` (C++23)

inserts a range of elements after an element
(public member function)

`erase_after` (C++11)

erases an element after an element
(public member function)

`push_front` (C++11)

inserts an element to the beginning
(public member function)

Operations

`merge` (C++11)

merges two sorted lists
(public member function)

`splICE_after` (C++11)

moves elements from another `forward_list`
(public member function)

`remove`

removes elements satisfying specific criteria

Quick Snapshot

```
std::forward_list<T>
```



std::forward_list (2/3)

- In short, this is singly linked list
 - Fast insertion at front of list with [push_front](#) (constant time)
 - You'll have to come up with your own abstractions otherwise to add flexibility
- Similar to std::list, in some sense, but way less power.
 - Lightweight container added in C++11

```
14 // forward_list.cpp
15 void push_back(std::forward_list<int>& list, int val){
16     auto pos = begin(list);
17     int distance = std::distance(begin(list),
18                                 end(list));
19     std::advance(pos,distance-1);
20     list.insert_after(pos,val);
21 }
22
23 int main(){
24
25     // singly linked list (slist)
26     std::forward_list<int> myList{1,2,3,4};
27
28     myList.push_front(0);
29     // WARNING: You might just want to use a
30     //           std::list, but this is
31     //           practice, showing how to add operations.
32     //           Would probably want to return an iterator
33     //           to speed this up!
34     push_back(myList,5);
35     push_back(myList,6);
36     printList(myList);
37
38     std::forward_list<int> list2{-2,0,3,4,5};
39     // merges two already sorted lists
40     // list2 will become empty after this operation
41     myList.merge(list2);
42
43     printList(myList);
44     printList(list2);
45
46     return 0;
47 }
```

```
std::forward_list<T>
```



std::forward_list (3/3)

Behavior/Performance characteristics

- Allocation:
 - Dynamic (inserting one node at a time at the end)
- Access:
 - $O(n)$, $O(1)$ if you already have iterator handle
- Search:
 - $O(n)$ -- linear search
- Notes:
 - `std::forward_list` does not know its length
 - Useful if you are primarily going to be adding to a data structure and traversing few times
 - Optimized for space storage versus `std::list`

```
14 // forward_list.cpp
15 void push_back(std::forward_list<int>& list, int val){
16     auto pos = begin(list);
17     int distance = std::distance(begin(list),
18                                 end(list));
19     std::advance(pos,distance-1);
20     list.insert_after(pos,val);
21 }
22
23 int main(){
24
25     // singly linked list (slist)
26     std::forward_list<int> myList{1,2,3,4};
27
28     myList.push_front(0);
29     // WARNING: You might just want to use a
30     //           std::list, but this is
31     //           practice, showing how to add operations.
32     //           Would probably want to return an iterator
33     //           to speed this up!
34     push_back(myList,5);
35     push_back(myList,6);
36     printList(myList);
37
38     std::forward_list<int> list2{-2,0,3,4,5};
39     // merges two already sorted lists
40     // list2 will become empty after this operation
41     myList.merge(list2);
42
43     printList(myList);
44     printList(list2);
45
46     return 0;
47 }
```

C++ reference

C++11, C++14, C++17, C++20, C++23, C++26 | Compiler support C++11, C++14, C++17, C++20, C++23, C++26

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 (class template) |
| forward_list (C++11) | singly-linked list (class template) |
| list | doubly-linked list (class template) |

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) |

17)

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) |
| flat_set (C++23) | adapts a container to provide a collection of unique keys, sorted by keys (class template) |
| flat_map (C++23) | adapts two containers to provide a collection of key-value pairs, sorted by unique keys (class template) |
| flat_multiset (C++23) | adapts a container to provide a collection of keys, sorted by keys (class template) |
| flat_multimap (C++23) | adapts two containers to provide a collection of key-value pairs, sorted by keys (class template) |

Unordered associative containers (since C++11)

Unordered associative containers implement unsorted (hashed) data structures that can be quickly searched ($O(1)$ average, $O(n)$ 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) |

Sorted strings:

byte multibyte – wide

Containers library

array (C++11)
 vector – deque
 list – forward_list (C++11)
 set – multiset
 map – multimap
 unordered_map (C++11)
 unordered_multimap (C++11)
 unordered_set (C++11)
 unordered_multiset (C++11)
 stack – queue – priority_queue
 flat_set (C++23)
 flat_multiset (C++23)
 flat_map (C++23)
 flat_multimap (C++23)
 span (C++20) – mdsan (C++23)

Language support library

source_location (C++20)
 Type support
 Program utilities
 Coroutine support (C++20)
 Three-way comparison (C++20)
 numeric_limits – type_info
 initializer_list (C++11)

Concepts library (C++20)

Diagnostics library

exception – System error
 basic_stacktrace (C++23)

Memory management library

unique_ptr (C++11)
 shared_ptr (C++11)
 Low level management

Input/output library

Print functions (C++23)
 Stream-based I/O – I/O manipulators
 basic_istream – basic_ostream
 Synchronized output (C++20)

Filesystem library (C++17)

path

Regular expressions library (C++11)

basic_regex – algorithms

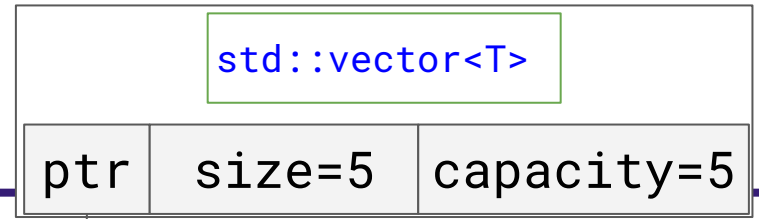
Concurrency support library (C++11)

thread – jthread (C++20)
 atomic – atomic_flag
 atomic_ref (C++20)
 memory_order – condition_variable
 Mutual exclusion – Semaphores (C++20)
 future – promise – async
 latch (C++20) – barrier (C++20)

Couple of 'Gotcha's' with Containers (1/6)

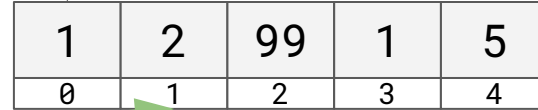
- Containers 'own' the data
 - When we use `push_back` that is making a copy to be placed in the container
 - To avoid copies, can use `emplace` member functions to construct a new object in place
 - (Should be faster, but of course measure to confirm)
- When removing data
 - C++ 20
 - `std::erase` and `std::erase_if` make things easier
 - Otherwise can use the [erase-remove idiom](#)
 - e.g.
 - `v.erase(std::remove(v.begin(), v.end(), 5), v.end());`
 - Careful however if you are holding pointers or otherwise references to other objects, as memory may not actually be freed.
- Generally where possible -- prefer the member functions of containers (especially if you know know that is the member function you will be using)
 - Member function may be more optimized
 - May have a clearer interface

Couple of 'Gotcha's' with Containers (2/6)



- Consider after we insert a new element in a vector, we may have a 'reallocation' to a new block of memory.

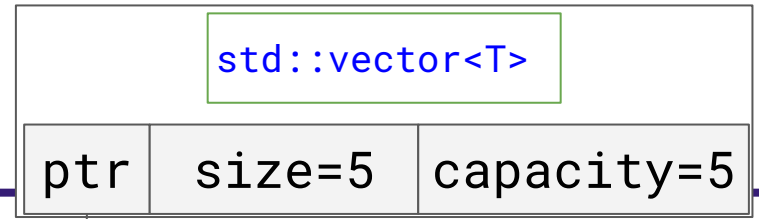
Some iterator



- Any previous iterators are considered 'invalid' as they do not point to the current vectors allocated memory
- e.g.
 - (next slide)

| Iterator invalidation | | | | | | |
|--|--|-------------------------|-------------------|-------------------------------|-------------------|---|
| Read-only methods never <i>invalidate</i> iterators or references. Methods which modify the contents of a container may invalidate iterators and/or references, as summarized in this table. | | | | | | |
| 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 | | Yes | | Insertion changed capacity |
| | deque | No | Yes | Yes, except erased element(s) | No | Before modified element(s) (for insertion only if capacity didn't change) |
| | list | Yes | No | Yes, except erased element(s) | No | At or after modified element(s) |
| | forward_list | Yes | No | Yes, except erased element(s) | No | Modified first or last element |
| Associative containers | set multiset map multimap | Yes | Yes | Yes, except erased element(s) | Yes | Modified middle only |
| Unordered associative containers | unordered_set unordered_multiset unordered_map unordered_multimap | No | Yes | | N/A | Insertion caused rehash |
| | | Yes | Yes | Yes, except erased element(s) | | No rehash |
| | | | | | | |

Couple of 'Gotcha's' with Containers (3/6)



- Consider after we insert a new element in a vector, we may have a 'reallocation' to a new block of memory.

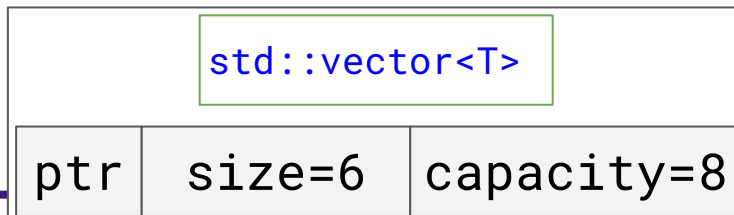
Some iterator

| | | | | |
|---|---|----|---|---|
| 1 | 2 | 99 | 1 | 5 |
| 0 | 1 | 2 | 3 | 4 |

- Any previous iterators are considered 'invalid' as they do not point to the current vectors allocated memory
- e.g.
 - `push_back(7)`
 - Let's assume this forces a new allocation

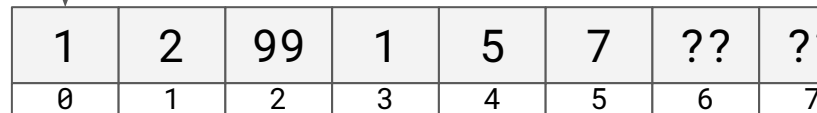
| Iterator invalidation | | | | | | |
|--|--------------------|-------------------------|-------------------------------|-------------------------------|-------------------|---|
| Read-only methods never <i>invalidate</i> iterators or references. Methods which modify the contents of a container may invalidate iterators and/or references, as summarized in this table. | | | | | | |
| 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 | | | | Insertion changed capacity |
| | deque | No | Yes | Yes, except erased element(s) | Yes | Before modified element(s) (for insertion only if capacity didn't change) |
| | list | No | Yes | Yes, except erased element(s) | No | At or after modified element(s) |
| | forward_list | No | Yes | Yes, except erased element(s) | No | Modified first or last element |
| Associative containers | set | Yes | | Yes, except erased element(s) | | Modified middle only |
| | multiset | Yes | | Yes, except erased element(s) | | |
| | map | Yes | | Yes, except erased element(s) | | |
| | multimap | Yes | | Yes, except erased element(s) | | |
| Unordered associative containers | unordered_set | No | | N/A | | Insertion caused rehash |
| | unordered_multiset | Yes | Yes | Yes, except erased element(s) | | No rehash |
| | unordered_map | Yes | Yes | Yes, except erased element(s) | | |
| unordered_multimap | Yes | Yes | Yes, except erased element(s) | | | |

Couple of 'Gotcha's' with Containers (4/6)



- Consider after we insert a new element in a vector, we may have a 'reallocation' to a new block of memory.

Some iterator



- Any previous iterators are considered 'invalid' as they do not point to the current vectors allocated memory
- e.g.
 - push_back(7)
 - Let's assume this forces a new allocation
 - Observe any iterators point to the old data -- thus invalidated.

| Iterator invalidation | | | | | | |
|--|--------------------|-------------------------|-------------------------------|-------------------------------|-------------------------------|---|
| Read-only methods never <i>invalidate</i> iterators or references. Methods which modify the contents of a container may invalidate iterators and/or references, as summarized in this table. | | | | | | |
| 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 | | | | Insertion changed capacity |
| | deque | Yes | Yes | Yes | Yes | Before modified element(s) (for insertion only if capacity didn't change) |
| | list | No | Yes, except erased element(s) | No | No | At or after modified element(s) |
| | forward_list | Yes | No | Yes, except erased element(s) | No | Modified first or last element |
| Associative containers | set | Yes | Yes | Yes, except erased element(s) | Yes, except erased element(s) | Modified middle only |
| | multiset | Yes | Yes | Yes, except erased element(s) | Yes, except erased element(s) | |
| | map | Yes | Yes | Yes, except erased element(s) | Yes, except erased element(s) | |
| Unordered associative containers | unordered_set | No | | | N/A | Insertion caused rehash |
| | unordered_multiset | Yes | Yes | Yes, except erased element(s) | Yes, except erased element(s) | No rehash |
| | unordered_map | Yes | Yes | Yes, except erased element(s) | Yes, except erased element(s) | |
| unordered_multimap | Yes | Yes | Yes, except erased element(s) | Yes, except erased element(s) | | |

Couple of 'Gotcha's' with Containers (5/6)

- When it comes to threading, that's a whole other talk, but consider:
 - You Can:
 - Safely read from a container with multiple threads
 - Safely write to different locations so long as one thread accessing a unique node/index
 - You cannot:
 - Have simultaneous writes to the same location however -- this require some locking mechanism
 - You should think about what granularity makes sense for the problem you are trying to solve and how to appropriately compose the result.

Thread safety

1. All container functions can be called concurrently by different threads on different containers. More generally, the C++ standard library functions do not read objects accessible by other threads unless those objects are directly or indirectly accessible via the function arguments, including the this pointer.
2. All const member functions can be called concurrently by different threads on the same container. In addition, the member functions `begin()`, `end()`, `rbegin()`, `rend()`, `front()`, `back()`, `data()`, `find()`, `lower_bound()`, `upper_bound()`, `equal_range()`, `at()`, and, except in associative containers, `operator[]`, behave as const for the purposes of thread safety (that is, they can also be called concurrently by different threads on the same container). More generally, the C++ standard library functions do not modify objects unless those objects are accessible, directly or indirectly, via the function's non-const arguments, including the this pointer.
3. Different elements in the same container can be modified concurrently by different threads, except for the elements of `std::vector<bool>` (for example, a vector of `std::future` objects can be receiving values from multiple threads). (since C++11)
4. Iterator operations (e.g. incrementing an iterator) read, but do not modify the underlying container, and may be executed concurrently with operations on other iterators on the same container, with the const member functions, or reads from the elements. Container operations that invalidate any iterators modify the container and cannot be executed concurrently with any operations on existing iterators even if those iterators are not invalidated.
5. Elements of the same container can be modified concurrently with those member functions that are not specified to access these elements. More generally, the C++ standard library functions do not read objects indirectly accessible through their arguments (including other elements of a container) except when required by its specification.
6. In any case, container operations (as well as algorithms, or any other C++ standard library functions) may be parallelized internally as long as this does not change the user-visible results (e.g. `std::transform` may be parallelized, but not `std::for_each` which is specified to visit each element of a sequence in order).

<https://en.cppreference.com/w/cpp/container>

Couple of 'Gotcha's' with Containers (6/6)

- A brief summary from `cppreference` about when to use each is below.
- Not sure?
 - Could simply start with `std::vector`, then profile otherwise.
 - Why this is the common advice is `vector`, `array`, or other contiguous data structures provide good cache locality.
 - Consider how often you add/remove, and to what locations (front, back, or middle) to help guide you.

Trade-offs / usage notes

| | |
|--|--|
| <code>std::vector</code> | Fast access but mostly inefficient insertions/deletions |
| <code>std::array</code> | Fast access but fixed number of elements |
| <code>std::list</code> <code>std::forward_list</code> | Efficient insertion/deletion in the middle of the sequence |
| <code>std::deque</code> | Efficient insertion/deletion at the beginning and at the end of the sequence |

C++ reference

C++11, C++14, C++17, C++20, C++23, C++26 | Compiler support C++11, C++14, C++17, C++20, C++23, C++26

Sequence containers

Sequence containers implement data structures which can be accessed sequentially.

| | |
|-----------------------------------|--|
| <code>array</code> (C++11) | static contiguous array (class template) |
| <code>vector</code> | dynamic contiguous array (class template) |
| <code>deque</code> | double-ended (class template) |
| <code>forward_list</code> (C++11) | singly-linked list (class template) |
| <code>list</code> | doubly-linked list (class template) |

Associative containers

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

| | |
|-----------------------|--|
| <code>set</code> | collection of unique keys, sorted by keys (class template) |
| <code>map</code> | collection of key-value pairs, sorted by keys, keys are unique (class template) |
| <code>multiset</code> | collection of keys, sorted by keys (class template) |
| <code>multimap</code> | collection of key-value pairs, sorted by keys (class template) |

17)

Container adaptors

Container adaptors provide a different interface for sequential containers.

| | |
|------------------------------------|---|
| <code>stack</code> | adapts a container to provide stack (LIFO data structure) (class template) |
| <code>queue</code> | adapts a container to provide queue (FIFO data structure) (class template) |
| <code>priority_queue</code> | adapts a container to provide priority queue (class template) |
| <code>flat_set</code> (C++23) | adapts a container to provide a collection of unique keys, sorted by keys (class template) |
| <code>flat_map</code> (C++23) | adapts two containers to provide a collection of key-value pairs, sorted by unique keys (class template) |
| <code>flat_multiset</code> (C++23) | adapts a container to provide a collection of keys, sorted by keys (class template) |
| <code>flat_multimap</code> (C++23) | adapts two containers to provide a collection of key-value pairs, sorted by keys (class template) |

Unordered associative containers (since C++11)

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

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

Sorted strings:

`basic_string` – wide

Language support library

- `source_location` (C++20)
- Type support
- Program utilities
- Coroutine support (C++20)
- Three-way comparison (C++20)
- `numeric_limits` – `type_info`
- `initializer_list` (C++11)

Concepts library (C++20)

Diagnostics library

- `exception` – System error
- `basic_stacktrace` (C++23)

Memory management library

- `unique_ptr` (C++11)
- `shared_ptr` (C++11)
- Low level management

Containers library

- `array` (C++11)
- `vector` – `deque`
- `list` – `forward_list` (C++11)
- `set` – `multiset`
- `map` – `multimap`
- `unordered_map` (C++11)
- `unordered_multimap` (C++11)
- `unordered_set` (C++11)
- `unordered_multiset` (C++11)
- `stack` – `queue` – `priority_queue`
- `flat_set` (C++23)
- `flat_multiset` (C++23)
- `flat_map` (C++23)
- `flat_multimap` (C++23)
- `span` (C++20) – `mdspan` (C++23)

Input/output library

- Print functions (C++23)
- Stream-based I/O – I/O manipulators
- `basic_istream` – `basic_ostream`
- Synchronized output (C++20)

Filesystem library (C++17)

- `path`

Regular expressions library (C++11)

- `basic_regex` – algorithms

Concurrency support library (C++11)

- `thread` – `jthread` (C++20)
- `atomic` – `atomic_flag`
- `atomic_ref` (C++20)
- `memory_order` – `condition_variable`
- Mutual exclusion – Semaphores (C++20)
- `future` – `promise` – `async`
- `latch` (C++20) – `barrier` (C++20)

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) |
| flat_set (C++23) | adapts a container to provide a collection of unique keys, sorted by keys (class template) |
| flat_map (C++23) | adapts two containers to provide a collection of key-value pairs, sorted by unique keys (class template) |
| flat_multiset (C++23) | adapts a container to provide a collection of keys, sorted by keys (class template) |
| flat_multimap (C++23) | adapts two containers to provide a collection of key-value pairs, sorted by keys (class template) |

Container Adaptors

Utilize different interface for containers we have already covered

Container Adaptors

- Container adaptors are not new containers implemented in the STL
 - Rather they modify by either restricting or enhancing the interface to other containers
 - When creating a container adaptor, you get to choose the underlying container (or otherwise accept the default)
- Let's take a look!

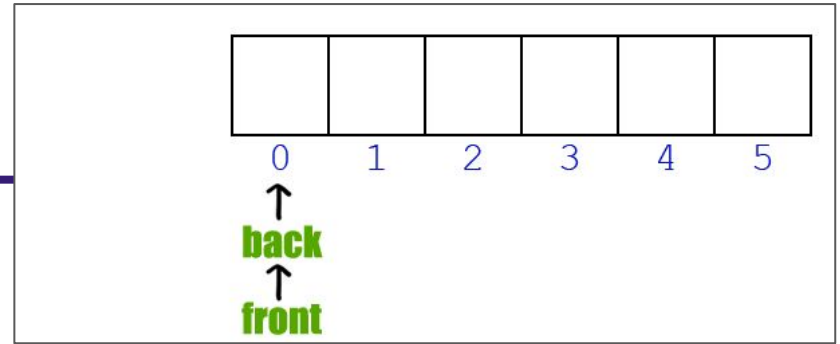
Container adaptors

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) |
| flat_set (C++23) | adapts a container to provide a collection of unique keys, sorted by keys (class template) |
| flat_map (C++23) | adapts two containers to provide a collection of key-value pairs, sorted by unique keys (class template) |
| flat_multiset (C++23) | adapts a container to provide a collection of keys, sorted by keys (class template) |
| flat_multimap (C++23) | adapts two containers to provide a collection of key-value pairs, sorted by keys (class template) |

std::queue (1/3)

- First in First Out (FIFO) data structure
 - Just like a grocery line, first person in, gets served first, last person who lines up gets served last
- With queues, we can only really access the first element, and remove the first element in order to get the next element.



Element access

| | |
|--------------|--|
| front | access the first element (public member function) |
| back | access the last element (public member function) |

Capacity

| | |
|--------------|--|
| empty | checks whether the underlying container is empty (public member function) |
| size | returns the number of elements (public member function) |

Modifiers

| | |
|------------------------|--|
| push | inserts element at the end (public member function) |
| emplace (C++11) | constructs element in-place at the end (public member function) |
| pop | removes the first element (public member function) |
| swap | swaps the contents (public member function) |

std::queue (2/3)

- A `std::queue` uses a `std::deque` by default
 - But a 'list' could also be used -- we need something where we can access first and last element

std::queue

Defined in header `<queue>`

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

Container - The type of the underlying container to use to store the elements. The container must satisfy the requirements of *SequenceContainer*. Additionally, it must provide the following functions with the usual semantics:

- `back()`
- `front()`
- `push_back()`
- `pop_front()`

The standard containers `std::deque` and `std::list` satisfy these requirements.

std::queue (3/3)

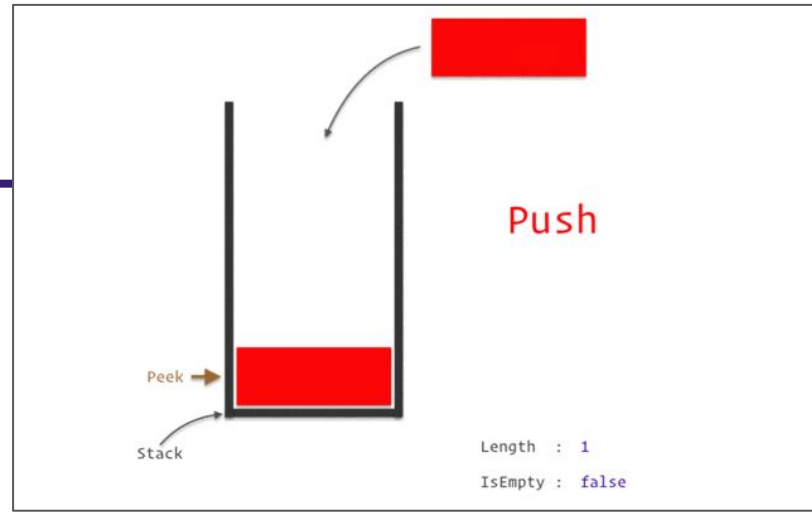
Behavior/Performance characteristics

- Allocation:
 - Dynamic (see underlying data structure)
- Access:
 - $O(1)$ at the front
- Search:
 - N/A; (Could pop off everything in $O(n)$)
- Notes:
 - Restricts underlying storage to give you 'FIFO' behavior.

```
10 // @file std queue.cpp
11 // Bring in a header file on our include path
12 // this happens to be in the standard library
13 // (i.e. default compiler path)
14 #include <iostream>
15 #include <queue> // Include the queue library
16
17 // Entry point to program 'main' function
18 int main(int argc, char* argv){
19
20     // The first parameter is the type
21     std::queue<char> myQueue;
22
23     // Add elements to our queue
24     myQueue.push('a');
25     myQueue.push('b');
26     myQueue.push('c');
27     // peek at the first element
28     std::cout << "front:" << myQueue.front() << std::endl;
29     // remove the first element
30     myQueue.pop();
31     // peek at the first element
32     std::cout << "front:" << myQueue.front() << std::endl;
33     // Report on the size
34     std::cout << "size:" << myQueue.size() << std::endl;
35
36
37
38     return 0;
39 }
```

std::stack (1/3)

- Last in, first out (LIFO) data structure
 - It's like stacking a bunch of dishes, whatever is on top, is the first dish that you take off to wash
- Almost identical to the std::queue interface, but we have the 'top()' member function to read the top of the stack.



Element access

top accesses the top element
(public member function)

Capacity

empty checks whether the underlying container is empty
(public member function)

size returns the number of elements
(public member function)

Modifiers

push inserts element at the top
(public member function)

push_range (C++23) inserts a range of elements at the top
(public member function)

emplace (C++11) constructs element in-place at the top
(public member function)

pop removes the top element
(public member function)

swap (C++11) swaps the contents
(public member function)

std::stack (2/3)

- Uses again a `std::deque` by default
 - Can use a `std::vector` as well -- think about why this works in a performant way (versus a queue where you probably do not want `std::vector`).

std::stack

Defined in header `<stack>`

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

Container - The type of the underlying container to use to store the elements. The container must satisfy the requirements of *SequenceContainer*. Additionally, it must provide the following functions with the usual semantics:

- `back()`
- `push_back()`
- `pop_back()`

The standard containers `std::vector` (including `std::vector<bool>`), `std::deque` and `std::list` satisfy these requirements. By default, if no container class is specified for a particular stack class instantiation, the standard container `std::deque` is used.

std::stack (3/3)

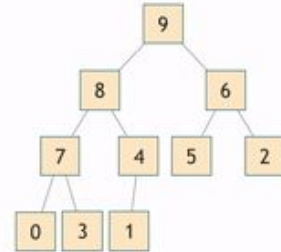
Behavior/Performance characteristics

- Allocation:
 - Dynamic (see underlying data structure)
- Access:
 - $O(1)$ at the front
- Search:
 - N/A; (Could pop off everything in $O(n)$)
- Notes:
 - Restricts underlying storage to give you 'FIFO' behavior.

```
10 // @file std_stack.cpp
11 // Bring in a header file on our include path
12 // this happens to be in the standard library
13 // (i.e. default compiler path)
14 #include <iostream>
15 #include <stack> // Include the stack library
16
17 // Entry point to program 'main' function
18 int main(int argc, char* argv[]){
19
20     // The first parameter is the type
21     std::stack<char> myStack;
22
23     // Add elements to our stack
24     myStack.push('a');
25     myStack.push('b');
26     myStack.push('c');
27     // peek at the top element
28     std::cout << "top:" << myStack.top() << std::endl;
29     // remove the top element
30     myStack.pop();
31     // peek at the top element
32     std::cout << "top:" << myStack.top() << std::endl;
33     // Report on the size
34     std::cout << "size:" << myStack.size() << std::endl;
35
36
37
38     return 0;
39 }
```

std::priority_queue (1/3)

- Similar to std::queue interface, but we have sorting
 - Max element (by default) will always be the top() of priority queue



Element access

top accesses the top element
(public member function)

Capacity

empty checks whether the underlying container is empty
(public member function)

size returns the number of elements
(public member function)

Modifiers

push inserts element and sorts the underlying container
(public member function)

push_range (C++23) inserts a range of elements and sorts the underlying container
(public member function)

emplace (C++11) constructs element in-place and sorts the underlying container
(public member function)

pop removes the top element
(public member function)

swap (C++11) swaps the contents
(public member function)

std::priority_queue (2/3)

std::priority_queue

Defined in header <queue>

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

- Uses again a `std::vector` by default
- May require you to implement a custom comparator in order to maintain heap property in the queue.

Container - The type of the underlying container to use to store the elements. The container must satisfy the requirements of *SequenceContainer*, and its iterators must satisfy the requirements of *LegacyRandomAccessIterator*. Additionally, it must provide the following functions with the usual semantics:

- `front()`
- `push_back()`
- `pop_back()`.

The standard containers `std::vector` (including `std::vector<bool>`) and `std::deque` satisfy these requirements.

Compare - A *Compare* type providing a strict weak ordering.

Note that the *Compare* parameter is defined such that it returns `true` if its first argument comes *before* its second argument in a weak ordering. But because the priority queue outputs largest elements first, the elements that "come before" are actually output last. That is, the front of the queue contains the "last" element according to the weak ordering imposed by *Compare*.

std::priority_queue (3/3)

Behavior/Performance characteristics

- Allocation:
 - Dynamic (see underlying data structure)
- Access:
 - $O(1)$ for top element
- Search:
 - Not really the right problem to solve
 - (Could pop off everything in $O(n)$)
- Notes:
 - Need to write a comparator for non-primitive types so queue can be sorted.

```
1 // priority_queue.cpp
2 #include <iostream>
3 #include <queue> // priority_queue
4 #include <deque>
5
6 int main(){
7
8     std::priority_queue<int> priorityQueue;
9     priorityQueue.push(32);
10    priorityQueue.push(33);
11    priorityQueue.push(31);
12
13    while(!priorityQueue.empty()){
14        int t = priorityQueue.top();
15        std::cout << "ordering: " << t << std::endl;
16        priorityQueue.pop();
17    }
18
19    return 0;
20 }
```

See on YouTube an example of a custom comparator data structure with comparator [C++ STL std::priority_queue \(a container adaptor\) | Modern Cpp Series](#)

std::flat_map (1/2)

- A new adaptor coming in C++23
 - Perhaps your compiler may have this available at the time of watching this recording.
- Note that there are equivalent ‘flat’ data structures for set, multimap, and multiset
- Upcoming in this talk I will show you two versions of a ‘map’
 - One that uses a tree data structure and one that uses a hashmap
 - The flatmap is a ‘third’ option which effectively flattens the tree into a linear sequence (i.e. a sequence data structure we have just discussed).

| Standard library header <code><flat_map></code> (C++23) | |
|---|---|
| This header is part of the containers library. | |
| Includes | |
| <code><compare></code> (C++20) | Three-way comparison operator support |
| <code><initializer_list></code> (C++11) | <code>std::initializer_list</code> class template |
| Classes | |
| <code>flat_map</code> (C++23) | adapts two containers to provide a collection of key-value pairs, sorted by unique keys (class template) |
| <code>flat_multimap</code> (C++23) | adapts two containers to provide a collection of key-value pairs, sorted by keys (class template) |

std::flat_map (2/2)

Behavior/Performance characteristics

- **Allocation:**
 - Dynamic as well, but inserting and deletion are linear time operations
- **Access:**
 - Fast to iterator through
- **Search:**
 - Ordered data structure, so should be $\log_2(n)$ with good cache locality
- **Notes:**
 - Coming soon!

C++ reference

C++11, C++14, C++17, C++20, C++23, C++26 | Compiler support C++11, C++14, C++17, C++20, C++23, C++26

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 (class template) |
| forward_list (C++11) | singly-linked list (class template) |
| list | doubly-linked list (class template) |

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) |

17)

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) |
| flat_set (C++23) | adapts a container to provide a collection of unique keys, sorted by keys (class template) |
| flat_map (C++23) | adapts two containers to provide a collection of key-value pairs, sorted by unique keys (class template) |
| flat_multiset (C++23) | adapts a container to provide a collection of keys, sorted by keys (class template) |
| flat_multimap (C++23) | adapts two containers to provide a collection of key-value pairs, sorted by keys (class template) |

Unordered associative containers (since C++11)

Unordered associative containers implement unsorted (hashed) data structures that can be quickly searched ($O(1)$ average, $O(n)$ 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) |

String containers

string, wstring, multibyte – wide

Language support library

source_location (C++20)
 Type support
 Program utilities
 Coroutine support (C++20)
 Three-way comparison (C++20)
 numeric_limits – type_info
 initializer_list (C++11)

Concepts library (C++20)

Diagnostics library

exception – System error
 basic_stacktrace (C++23)

Memory management library

unique_ptr (C++11)
 shared_ptr (C++11)
 Low level management

Containers library

array (C++11)
 vector – deque
 list – forward_list (C++11)
 set – multiset
 map – multimap
 unordered_map (C++11)
 unordered_multimap (C++11)
 unordered_set (C++11)
 unordered_multiset (C++11)
 stack – queue – priority_queue
 flat_set (C++23)
 flat_multiset (C++23)
 flat_map (C++23)
 flat_multimap (C++23)
 span (C++20) – mdspan (C++23)

Input/output library

Print functions (C++23)
 Stream-based I/O – I/O manipulators
 basic_istream – basic_ostream
 Synchronized output (C++20)

Filesystem library (C++17)

path

Regular expressions library (C++11)

basic_regex – algorithms

Concurrency support library (C++11)

thread – jthread (C++20)
 atomic – atomic_flag
 atomic_ref (C++20)
 memory_order – condition_variable
 Mutual exclusion – Semaphores (C++20)
 future – promise – async
 latch (C++20) – barrier (C++20)

C++ reference

C++11, C++14, C++17, C++20, C++23, C++26 | Compiler support C++11, C++14, C++17, C++20, C++23, C++26

Sequence containers

Sequence containers implement data structures which can be accessed sequentially.

| | |
|-----------------------------------|--|
| <code>array</code> (C++11) | static contiguous array (class template) |
| <code>vector</code> | dynamic contiguous array (class template) |
| <code>deque</code> | double-ended (class template) |
| <code>forward_list</code> (C++11) | singly-linked list (class template) |
| <code>list</code> | doubly-linked list (class template) |

Container adaptors

Container adaptors provide a different interface for sequential containers.

| | |
|------------------------------------|---|
| <code>stack</code> | adapts a container to provide stack (LIFO data structure) (class template) |
| <code>queue</code> | adapts a container to provide queue (FIFO data structure) (class template) |
| <code>priority_queue</code> | adapts a container to provide priority queue (class template) |
| <code>flat_set</code> (C++23) | adapts a container to provide a collection of unique keys (class template) |
| <code>flat_map</code> (C++23) | adapts two containers to provide a collection of key-value pairs, sorted by unique keys (class template) |
| <code>flat_multiset</code> (C++23) | adapts a container to provide a collection of keys, sorted by keys (class template) |
| <code>flat_multimap</code> (C++23) | adapts two containers to provide a collection of key-value pairs, sorted by keys (class template) |

Language support library

- `source_location` (C++20)
- Type support
- Program utilities
- Coroutine support (C++20)
- Three-way comparison (C++20)
- `numeric_limits` – `type_info`
- `initializer_list` (C++11)

Concepts library (C++20)

Diagnostics library

- `exception` – System error
- `basic_stacktrace` (C++23)

Memory management library

- `unique_ptr` (C++11)
- `shared_ptr` (C++11)
- Low level management

Associative containers

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

| | |
|-----------------------|--|
| <code>set</code> | collection of unique keys, sorted by keys (class template) |
| <code>map</code> | collection of key-value pairs, sorted by keys, keys are unique (class template) |
| <code>multiset</code> | collection of keys, sorted by keys (class template) |
| <code>multimap</code> | collection of key-value pairs, sorted by keys (class template) |

Unordered associative containers (since C++11)

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

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

Containers library

- `array` (C++11)
- `vector` – `deque`
- `list` – `forward_list` (C++11)
- `set` – `multiset`
- `map` – `multimap`
- `unordered_map` (C++11)
- `unordered_multimap` (C++11)
- `unordered_set` (C++11)
- `unordered_multiset` (C++11)
- `stack` – `queue` – `priority_queue`
- `flat_set` (C++23)
- `flat_multiset` (C++23)
- `flat_map` (C++23)
- `flat_multimap` (C++23)
- `span` (C++20) – `mdspan` (C++23)

Input/output library

- Print functions (C++23)
- Stream-based I/O – I/O manipulators
- `basic_istream` – `basic_ostream`
- Synchronized output (C++20)

Filesystem library (C++17)

- `path`

Regular expressions library (C++11)

- `basic_regex` – algorithms

Concurrency support library (C++11)

- `thread` – `jthread` (C++20)
- `atomic` – `atomic_flag`
- `atomic_ref` (C++20)
- `memory_order` – `condition_variable`
- Mutual exclusion – Semaphores (C++20)
- `future` – `promise` – `async`
- `latch` (C++20) – `barrier` (C++20)

Associative and Unordered Associative Containers (1/5)

- I'm going to talk about these two container type side-by-side
- As can be observed from the name, one container is 'unordered'
 - The implication of this means that we can choose one container over the other based on the ordering.

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 <small>(class template)</small> |
| map | collection of key-value pairs, sorted by keys, keys are unique <small>(class template)</small> |
| multiset | collection of keys, sorted by keys <small>(class template)</small> |
| multimap | collection of key-value pairs, sorted by keys <small>(class template)</small> |

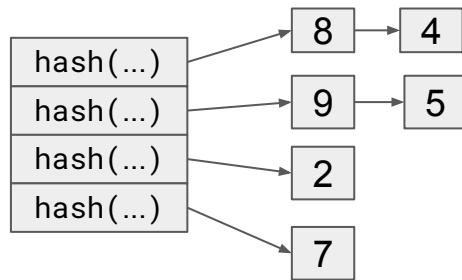
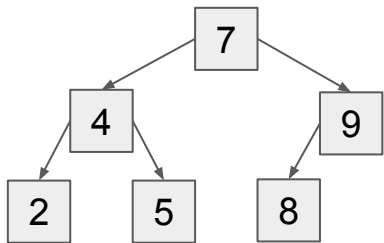
Unordered associative containers (since C++11)

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

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

Associative and Unordered Associative Containers (2/5)

- Associative containers typically have a self-balancing binary tree (rb-tree) to represent them.
- Unordered associative containers have a hash table
 - Observe a `std::set` to the left and observe an `std::unordered_set` on the right



Associative containers

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

| | |
|-----------------------|--|
| <code>set</code> | collection of unique keys, sorted by keys (class template) |
| <code>map</code> | collection of key-value pairs, sorted by keys, keys are unique (class template) |
| <code>multiset</code> | collection of keys, sorted by keys (class template) |
| <code>multimap</code> | collection of key-value pairs, sorted by keys (class template) |

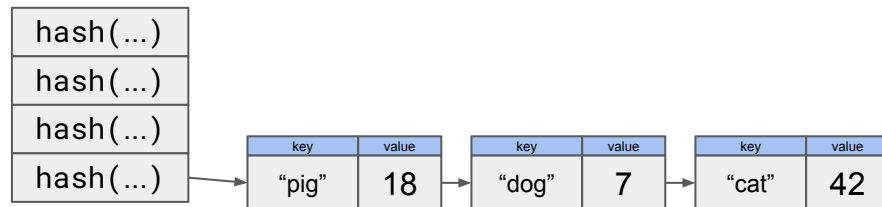
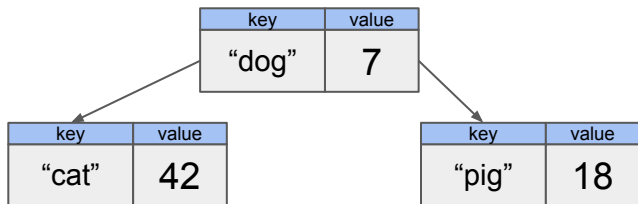
Unordered associative containers (since C++11)

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

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

Associative and Unordered Associative Containers (3/5)

- Observe a `std::map` to the left and observe an `std::unordered_map` on the right
 - This time having a key and value pair. The key is what is sorted this time, and the value is the information alongside in a node



Associative containers

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

| | |
|-----------------------|--|
| <code>set</code> | collection of unique keys, sorted by keys (class template) |
| <code>map</code> | collection of key-value pairs, sorted by keys, keys are unique (class template) |
| <code>multiset</code> | collection of keys, sorted by keys (class template) |
| <code>multimap</code> | collection of key-value pairs, sorted by keys (class template) |

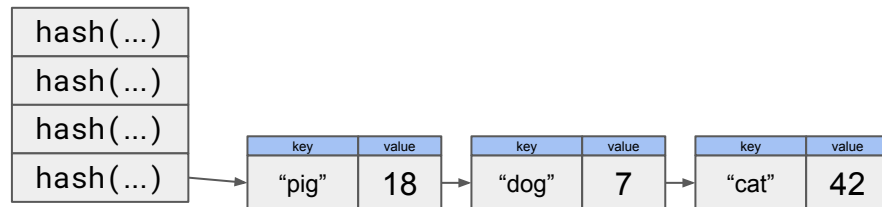
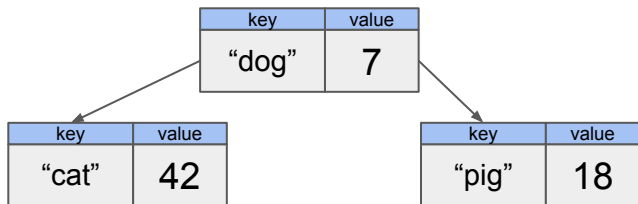
Unordered associative containers (since C++11)

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

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

Associative and Unordered Associative Containers (4/5)

- Observe on the right in the `std::unordered_map` -- we do not want too many nodes to 'hash' the key to the same 'bucket'
 - If this happens, we no longer get average $O(1)$ performance on unordered containers
 - For custom data types, this will be something we have to think about.



Associative containers

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

| | |
|-----------------------|---|
| <code>set</code> | collection of unique keys, sorted by keys <small>(class template)</small> |
| <code>map</code> | collection of key-value pairs, sorted by keys, keys are unique <small>(class template)</small> |
| <code>multiset</code> | collection of keys, sorted by keys <small>(class template)</small> |
| <code>multimap</code> | collection of key-value pairs, sorted by keys <small>(class template)</small> |

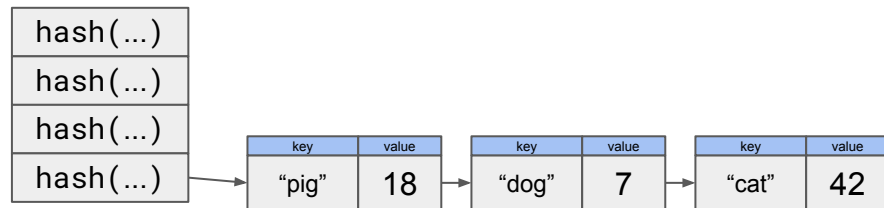
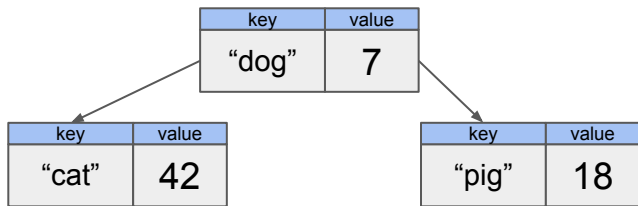
Unordered associative containers (since C++11)

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

| | |
|---|---|
| <code>unordered_set</code> (C++11) | collection of unique keys, hashed by keys <small>(class template)</small> |
| <code>unordered_map</code> (C++11) | collection of key-value pairs, hashed by keys, keys are unique <small>(class template)</small> |
| <code>unordered_multiset</code> (C++11) | collection of keys, hashed by keys <small>(class template)</small> |
| <code>unordered_multimap</code> (C++11) | collection of key-value pairs, hashed by keys <small>(class template)</small> |

Associative and Unordered Associative Containers (5/5)

- As a final note -- `std::multiset`, `std::multimap`, `std::unordered_multiset`, and `std::unordered_multimap` allow for duplicate keys



Associative containers

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

| | |
|-----------------------|---|
| <code>set</code> | collection of unique keys, sorted by keys <small>(class template)</small> |
| <code>map</code> | collection of key-value pairs, sorted by keys, keys are unique <small>(class template)</small> |
| <code>multiset</code> | collection of keys, sorted by keys <small>(class template)</small> |
| <code>multimap</code> | collection of key-value pairs, sorted by keys <small>(class template)</small> |

Unordered associative containers (since C++11)

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

| | |
|---|---|
| <code>unordered_set</code> (C++11) | collection of unique keys, hashed by keys <small>(class template)</small> |
| <code>unordered_map</code> (C++11) | collection of key-value pairs, hashed by keys, keys are unique <small>(class template)</small> |
| <code>unordered_multiset</code> (C++11) | collection of keys, hashed by keys <small>(class template)</small> |
| <code>unordered_multimap</code> (C++11) | collection of key-value pairs, hashed by keys <small>(class template)</small> |

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) |

Associative Containers

(All maintain 'sorted' data)

std::set (1/2)

- All keys are unique and in sorted order
 - Implementation likely a self-balancing tree like a red-black tree



<https://en.cppreference.com/w/cpp/container/set>

std::set

Defined in header <set>

```
template<
    class Key,
    class Compare = std::less<Key>,
    class Allocator = std::allocator<Key>
> class set;
```

| Capacity | |
|-----------------------------|---|
| empty | checks whether the container is empty (public member function) |
| size | returns the number of elements (public member function) |
| max_size | returns the maximum possible number of elements (public member function) |
| Modifiers | |
| clear | clears the contents (public member function) |
| insert | inserts elements or nodes (since C++17) (public member function) |
| insert_range (C++23) | inserts a range of elements (public member function) |
| emplace (C++11) | constructs element in-place (public member function) |
| emplace_hint (C++11) | constructs elements in-place using a hint (public member function) |
| erase | erases elements (public member function) |
| swap | swaps the contents (public member function) |
| extract (C++17) | extracts nodes from the container (public member function) |
| merge (C++17) | splices nodes from another container (public member function) |
| Lookup | |
| count | returns the number of elements matching specific key (public member function) |
| find | finds element with specific key (public member function) |
| contains (C++20) | checks if the container contains element with specific key (public member function) |
| equal_range | returns range of elements matching a specific key (public member function) |
| lower_bound | returns an iterator to the first element <i>not less</i> than the given key (public member function) |
| upper_bound | returns an iterator to the first element <i>greater</i> than the given key (public member function) |

std::set (2/2)

Behavior/Performance characteristics

- Allocation:
 - Dynamic, can expand -- one unique key however
- Access:
 - $O(\log_2(n))$
- Search:
 - $O(\log_2(n))$
- Notes:
 - sorted container

```
10 // @file std_set.cpp
11 // Bring in a header file on our include path
12 // this happens to be in the standard library
13 // (i.e. default compiler path)
14 #include <iostream>
15 #include <set> // Include the set library
16 #include <string> // String data type
17
18 // Entry point to program 'main' function
19 int main(int argc, char* argv[]){
20
21     // A single key
22     std::set<std::string > mySet;
23
24     // Add elements to our set
25     mySet.insert("cat");
26     mySet.insert("cat");
27     mySet.insert("cat");
28     mySet.insert("cat");
29     mySet.insert("cat");
30
31     // Lookup a value by the key
32     std::cout << "Unique items: " << mySet.size() << std::endl;
33
34     return 0;
35 }
```

std::map (1/2)

- An associative data structure consisting of a “key” and “value” pair
 - The “key” is what we are sorting on



<https://en.cppreference.com/w/cpp/container/set>

std::map

Defined in header <map>

```
template<
    class Key,
    class T,
    class Compare = std::less<Key>,
    class Allocator = std::allocator<std::pair<const Key, T>>
> class map;
```

| Element access | |
|---------------------------------|--|
| at | access specified element with bounds checking (public member function) |
| operator[] | access or insert specified element (public member function) |
| Iterators | |
| begin | returns an iterator to the beginning (public member function) |
| cbegin (C++11) | returns an iterator to the beginning (public member function) |
| end | returns an iterator to the end (public member function) |
| cbegin (C++11) | returns a reverse iterator to the beginning (public member function) |
| rend | returns a reverse iterator to the end (public member function) |
| crend (C++11) | returns a reverse iterator to the end (public member function) |
| Capacity | |
| empty | checks whether the container is empty (public member function) |
| size | returns the number of elements (public member function) |
| max_size | returns the maximum possible number of elements (public member function) |
| Modifiers | |
| clear | clears the contents (public member function) |
| insert | inserts elements or nodes (since C++17) (public member function) |
| insert_range (C++23) | inserts a range of elements (public member function) |
| insert_or_assign (C++17) | inserts an element or assigns to the current element if the key already exists (public member function) |
| emplace (C++11) | constructs element in-place (public member function) |
| emplace_hint (C++11) | constructs elements in-place using a hint (public member function) |
| try_emplace (C++17) | inserts in-place if the key does not exist, does nothing if the key exists (public member function) |
| erase | erases elements (public member function) |
| swap | swaps the contents (public member function) |
| extract (C++17) | extracts nodes from the container (public member function) |
| merge (C++17) | splices nodes from another container (public member function) |
| Lookup | |
| count | returns the number of elements matching specific key (public member function) |
| find | finds element with specific key (public member function) |
| contains (C++20) | checks if the container contains element with specific key (public member function) |
| equal_range | returns range of elements matching a specific key (public member function) |

std::map (2/2)

Behavior/Performance characteristics

- Allocation:
 - Dynamic, can expand -- one unique key however
- Access:
 - $O(\log_2(n))$
- Search:
 - $O(\log_2(n))$
- Notes:
 - sorted container by the 'key'
 - Consists of a key/value pair (std::pair)

```
10 // @file std_map.cpp
11 // Bring in a header file on our include path
12 // this happens to be in the standard library
13 // (i.e. default compiler path)
14 #include <iostream>
15 #include <map> // Include the map library
16 #include <string> // String data type
17
18 // Entry point to program 'main' function
19 int main(int argc, char* argv[]){
20
21     // Key and value pair
22     // - A string will be the key
23     // - A string will then be the value
24     std::map<std::string, std::string > dictionary;
25
26     // Create a key/value pair
27     std::pair<std::string, std::string> cat("cat", "a funny animal");
28
29     // Add elements to our map
30     dictionary.insert(cat);
31
32     // Lookup a value by the key
33     std::cout << dictionary["cat"] << std::endl;
34
35     return 0;
36 }
```

std::unordered_set (1/2)

- All keys are unique and unordered
 - Stored in a hash table
 - Available since C++ 11



std::unordered_set

Defined in header `<unordered_set>`

```
template<  
    class Key,  
    class Hash = std::hash<Key>,  
    class KeyEqual = std::equal_to<Key>,  
    class Allocator = std::allocator<Key>  
> class unordered_set;
```

Capacity

| | |
|-------------------------|---|
| empty (C++11) | checks whether the container is empty (public member function) |
| size (C++11) | returns the number of elements (public member function) |
| max_size (C++11) | returns the maximum possible number of elements (public member function) |

Modifiers

| | |
|-----------------------------|---|
| clear (C++11) | clears the contents (public member function) |
| insert (C++11) | inserts elements or nodes (since C++17) (public member function) |
| insert_range (C++23) | inserts a range of elements (public member function) |
| emplace (C++11) | constructs element in-place (public member function) |
| emplace_hint (C++11) | constructs elements in-place using a hint (public member function) |
| erase (C++11) | erases elements (public member function) |
| swap (C++11) | swaps the contents (public member function) |
| extract (C++17) | extracts nodes from the container (public member function) |
| merge (C++17) | splices nodes from another container (public member function) |

Lookup

| | |
|----------------------------|--|
| count (C++11) | returns the number of elements matching specific key (public member function) |
| find (C++11) | finds element with specific key (public member function) |
| contains (C++20) | checks if the container contains element with specific key (public member function) |
| equal_range (C++11) | returns range of elements matching a specific key (public member function) |

Bucket interface

| | |
|---|--|
| begin (size_type) cbegin (size_type) (C++11) | returns an iterator to the beginning of the specified bucket (public member function) |
| end (size_type) cend (size_type) (C++11) | returns an iterator to the end of the specified bucket (public member function) |
| bucket_count (C++11) | returns the number of buckets (public member function) |
| max_bucket_count (C++11) | returns the maximum number of buckets (public member function) |
| bucket_size (C++11) | returns the number of elements in specific bucket (public member function) |
| bucket (C++11) | returns the bucket for specific key (public member function) |

Hash policy

| | |
|----------------------------|---|
| load_factor (C++11) | returns average number of elements per bucket (public member function) |
|----------------------------|---|

std::unordered_set (2/2)

Behavior/Performance characteristics

- Allocation:
 - Dynamic, can expand -- one unique key however
- Access:
 - $O(1)$ on average
- Search:
 - $O(1)$ on average
- Notes:
 - Not sorted
 - Frequent resizing or a bad hash function harms performance

```
13 int gen(){
14     static int i=0;
15     return ++i;
16 }
17
18
19 int main(){
20
21     std::unordered_set<int> s{1,2,3,4,5,6};
22     std::unordered_set<int> s2;
23
24     std::generate_n(std::inserter(s2,s2.begin()),10,gen);
25
26     s.merge(s2);
27     if(s.erase(11)==1){
28         std::cout << "we removed 11\n";
29     }
30
31     if(!s.contains(12)){
32         s.insert(12);
33     }
34
35     std::cout << s.bucket_count() << std::endl;
36     std::cout << s.load_factor() << std::endl;
37
38     printUnorderedSet(s);
39
40     /*
41     // Printing out bucket sizes
42     for(int i=0; i< s.bucket_count(); i++){
43         std::cout << s.bucket_size(i) << std::endl;
44     }
45     */
46
47     return 0;
48 }
```

std::unordered_map (1/2)

- All keys are unique and unordered
 - Stored in a hash table
 - An associative data structure consisting of a “key” and “value” pair
 - The “key” is what we are sorting on



<https://en.cppreference.com/w/cpp/container/set>

std::unordered_map

Defined in header `<unordered_map>`

```
template<
    class Key,
    class T,
    class Hash = std::hash<Key>,
    class KeyEqual = std::equal_to<Key>,
    class Allocator = std::allocator<std::pair<const Key, T>>
> class unordered_map;
```

Capacity

| | |
|---------------------------------------|--|
| <code>empty</code> (C++11) | checks whether the container is empty (public member function) |
| <code>size</code> (C++11) | returns the number of elements (public member function) |
| <code>max_size</code> (C++11) | returns the maximum possible number of elements (public member function) |
| Modifiers | |
| <code>clear</code> (C++11) | clears the contents (public member function) |
| <code>insert</code> (C++11) | inserts elements or nodes (since C++17) (public member function) |
| <code>insert_range</code> (C++23) | inserts a range of elements (public member function) |
| <code>insert_or_assign</code> (C++17) | inserts an element or assigns to the current element if the key already exists (public member function) |
| <code>emplace</code> (C++11) | constructs element in-place (public member function) |
| <code>emplace_hint</code> (C++11) | constructs elements in-place using a hint (public member function) |
| <code>try_emplace</code> (C++17) | inserts in-place if the key does not exist, does nothing if the key exists (public member function) |
| <code>erase</code> (C++11) | erases elements (public member function) |
| <code>swap</code> (C++11) | swaps the contents (public member function) |
| <code>extract</code> (C++17) | extracts nodes from the container (public member function) |
| <code>merge</code> (C++17) | splices nodes from another container (public member function) |

Lookup

| | |
|----------------------------------|--|
| <code>at</code> (C++11) | access specified element with bounds checking (public member function) |
| <code>operator []</code> (C++11) | access or insert specified element (public member function) |
| <code>count</code> (C++11) | returns the number of elements matching specific key (public member function) |
| <code>find</code> (C++11) | finds element with specific key (public member function) |
| <code>contains</code> (C++20) | checks if the container contains element with specific key (public member function) |
| <code>equal_range</code> (C++11) | returns range of elements matching a specific key (public member function) |

std::unordered_map (2/2)

Behavior/Performance characteristics

- Allocation:
 - Dynamic, can expand -- one unique key however
- Access:
 - O(1) on average
- Search:
 - O(1) on average
- Notes:
 - Not sorted
 - Frequent resizing or a bad hash function harms performance

```
10 // @file std_unordered_map.cpp
11 // Bring in a header file on our include path
12 // this happens to be in the standard library
13 // (i.e. default compiler path)
14 #include <iostream>
15 #include <unordered_map> // Include the unordered_map library
16 #include <string> // String data type
17
18 // Entry point to program 'main' function
19 int main(int argc, char* argv[]){
20
21     // Key and value pair
22     // - A string will be the key
23     // - A string will then be the value
24     std::unordered_map<std::string, std::string > dictionary;
25
26     // Create a key/value pair
27     std::pair<std::string, std::string> cat("cat", "a funny animal");
28
29     // Add elements to our unordered_map
30     dictionary.insert(cat);
31
32     // Lookup a value by the key
33     std::cout << dictionary["cat"] << std::endl;
34
35     return 0;
36 }
```

Common ‘Gotcha’s’ with associative containers

- Important to note that ‘at’ is a read operation
- operator [] will create the key if it does not exist, otherwise update the key
 - It may be worth updating the operations
- Generally, unordered variants are drop in replacements
 - i.e. try to replace a `std::map` with `std::unordered_map` in your code if you do not need sorting

Associative containers

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

| | |
|-----------------------|---|
| <code>set</code> | collection of unique keys, sorted by keys <small>(class template)</small> |
| <code>map</code> | collection of key-value pairs, sorted by keys, keys are unique <small>(class template)</small> |
| <code>multiset</code> | collection of keys, sorted by keys <small>(class template)</small> |
| <code>multimap</code> | collection of key-value pairs, sorted by keys <small>(class template)</small> |

Unordered associative containers (since C++11)

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

| | |
|---|---|
| <code>unordered_set</code> (C++11) | collection of unique keys, hashed by keys <small>(class template)</small> |
| <code>unordered_map</code> (C++11) | collection of key-value pairs, hashed by keys, keys are unique <small>(class template)</small> |
| <code>unordered_multiset</code> (C++11) | collection of keys, hashed by keys <small>(class template)</small> |
| <code>unordered_multimap</code> (C++11) | collection of key-value pairs, hashed by keys <small>(class template)</small> |

Wrapping Up

Summary

- We have had a tour of the containers in the C++ Standard Library
 - My goal is that you now understand there are a variety of data structures available for you to get started and tackle your programming challenges!
 - Choosing the right container can often make a large impact on performance and ease of solving a problem.

Thank you Meeting C++ 2023!

Introduction to C++ Containers

-- Know Your Data Structures

with Mike Shah

17:15 - 16:15 Fri, November 12, 2023

~60 minutes | Introductory Audience

Social: [@MichaelShah](https://twitter.com/MichaelShah)

Web: mshah.io

Courses: courses.mshah.io

 **YouTube**

www.youtube.com/c/MikeShah

Thank you!