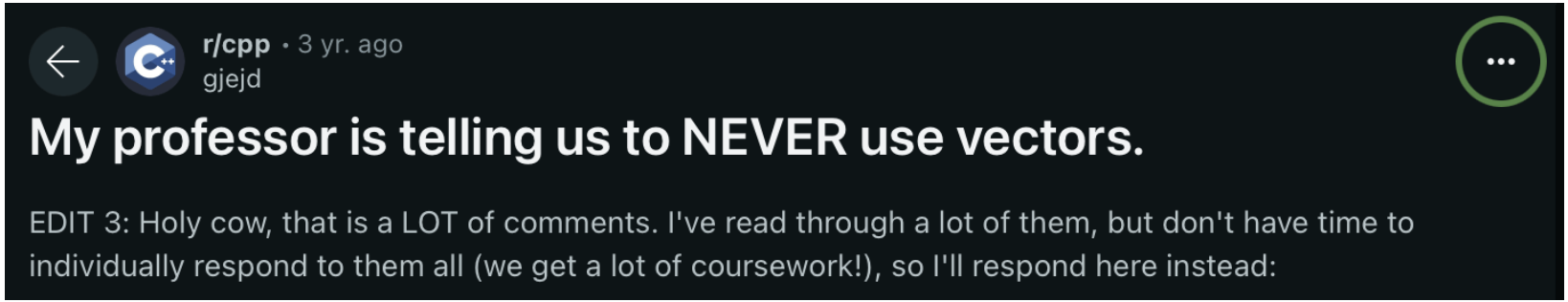
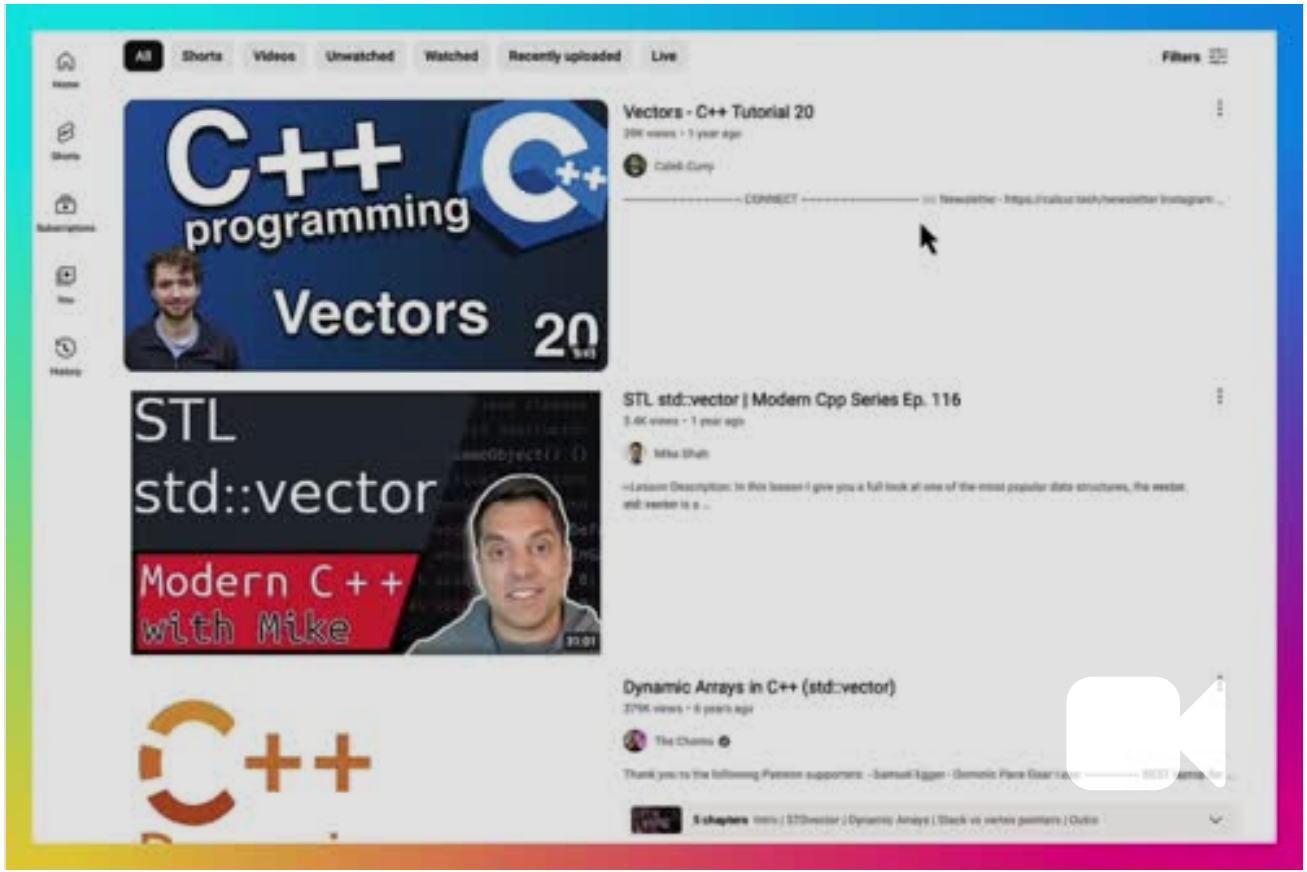
Back To Basics
Almost Always Vector

KEVIN CARPENTER

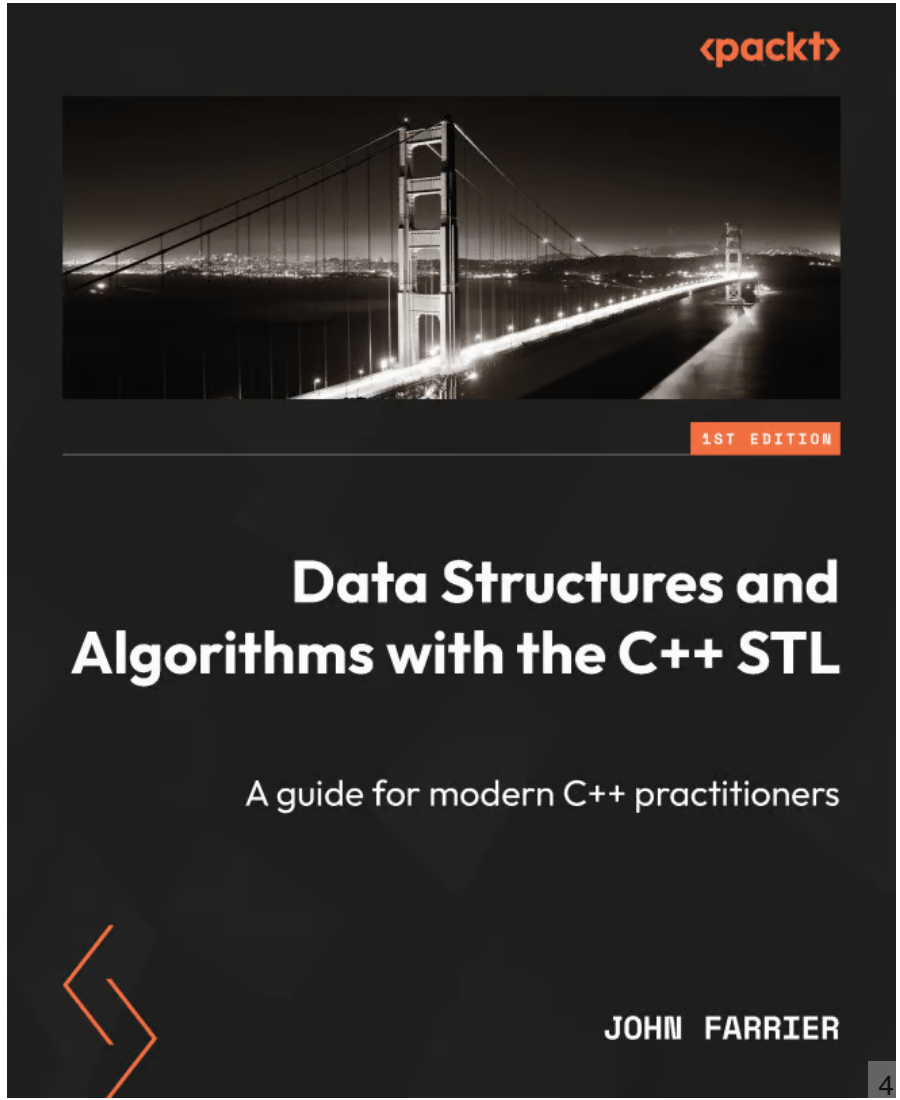



```cpp
#include <array>
#include <iostream>
#include <vector>

int a[] = {0, 1, 2, 3, 4};

std::vector<int> c = {0, 1, 2, 3, 4};

auto main() -> int {
  std::cout << "C style array: " << sizeof(a) / sizeof(a[0]) << std::endl;
  std::cout << "Vector size: " << c.size() << std::endl;

  return 0;
}
```

https://www.reddit.com/r/cpp/comments/qm6i25/my_professor_is_telling_us_to_never_use_vectors/



more std::vector?



How do you read a technical book front to back?

As a technical editor...

The Old Way...

```c
#include <stdio.h>

int main() {
  int cppconEarlyYears[6] = {2014, 2015, 2016, 2017, 2018, 2019};

  for (int i = 0; i < 6; i++) {
    printf("%d ", cppconEarlyYears[i]);
  }

  return 0;
}
```

```
[Running]  gcc 1_the_old_way.c -o 1_the_old_way
2014 2015 2016 2017 2018 2019
[Done] exited with code=0 in 0.926 seconds
```

Deleting value's

```c
#include <stdio.h>

int main() {
  int cppconEarlyYears[6] = {2014, 2015, 2016, 2017, 2018, 2019};

  cppconEarlyYears[5] = 0;  // can't delete - fixed length

  for (int i = 0; i < 6; i++) {
    printf("%d ", cppconEarlyYears[i]);
  }

  return 0;
}
```

```
[Running] gcc 2_swapping_values.c -o 2_swapping_values
2014 2015 2016 2017 2018 0
[Done] exited with code=0 in 0.234 seconds
```

The Limitations...

```c
#include <stdio.h>

int main() {
  int cppconEarlyYears[6] = {2014, 2015, 2016, 2017, 2018, 2019};
  cppconEarlyYears[5] = 0; // <- can't delete
  // cppconEarlyYears[6] = 2020; // <- can't add

  int cppconEarlyRevised[6];
  // cppconEarlyRevised = cppconEarlyYears; // <- can't copy or assign

  for (int i = 0; i < 5; i++) {
    cppconEarlyRevised[i] = cppconEarlyYears[i];
  }

  for (int i = 0; i < 7; i++) { // <- yes - it prints
    printf("%d ", cppconEarlyRevised[i]);
  }

  return 0;
}
```

```
[Running] gcc 3_the_limitations.c -o 3_the_limitations
2014 2015 2016 2017 2018 1 2014
[Done] exited with code=0 in 0.227 seconds
```

Generic Dynamic Arrays in C++

Uwe Steinmueller

Alexander von Zitzewitz

SNI AP 44

Siemens Nixdorf Informationssysteme AG 1992

1. Why are dynamic arrays needed in C++?

Arrays are a widely used low-level abstraction for storing elements of a common type. C++ provides arrays as a built-in type. So why should one want to implement a generic array class? There are basically the following reasons for doing this:

- C++ does not provide dynamic arrays. The size of an array must be known at compile time and cannot be changed at runtime. This implies a practice of using fixed sized limits. In many cases these "maximum" limits are no natural limit at all and lead to programs which are very hard to be modified or extended. On the other hand using dynamic arrays in C is error prone and boring.
- C++ does not support copying and assignment of arrays.
- Generic dynamic arrays are an important building block for the implementation of many kinds of container classes, nearly every serious application will need it.

Thank you WG21/N0083

The Plan...

- Basics
- Memory Management
- Iterators
- Algorithms
- Container Comparisons
- Why AAV

The Basics
of DynArray

Original Definition

```
1  template <class T> class DynArray {
2  public:
3      // Constructors
4      DynArray();                  // Array of size 0
5      DynArray (unsigned size);    // Array of given size
6
7      // Destructor
8      ~DynArray();     // destructs elements and frees dynamic store
9
10     // Copy and assign
11     DynArray(const DynArray& source);
12     const DynArray& operator=(const DynArray& source);
13
14     // Element access
15     T& operator[] (unsigned index);
16     const T& operator[] (unsigned index) const;
17
18     // Low level aaccess to array elements
19     T* base() ;   // get address of first element
20     const T* base() const:
21
22     // Size
23     unsigned size() const;       // returns size of array
24     unsigned size(unsigned sz); // change size to sz
25
26     // Low level helper functions
27     void swap (DynArray& other); // swap contents
28 };
```

Alexander von Zitzewitz (He/Him) • 8:09 AM

At that time Uwe and me were building a C++ class library called "Generic". That was even before the STL became available. It has collection classes, strings with reference counting etc. The G_DynArray<T> class was the backbone of many of the collection classes. Generic had happy users well into the early 2000's but then it died since the STL was the way to go. (Edited)

From DynArray to vector?

2. Proposal for a generic dynamic array class

Since the proposed class should be used as a low-level building block it only provides basic functionality.

The name "DynArray" for this class is selected as "array" is a built-in type of C++ and the alternative "vector" may be expected to be a mathematical class (and would have different member functions).

and

std::vector

was born!

What happened in 32 years?

32 member functions!

More ways to access elements.

Iterators

Allocators

Modifiers

Oh My!

Member functions

(constructor)	constructs the `vector` (public member function)
(destructor)	destructs the `vector` (public member function)
`operator=`	assigns values to the container (public member function)
`assign`	assigns values to the container (public member function)
`assign_range` (C++23)	assigns a range of values to the container (public member function)
`get_allocator`	returns the associated allocator (public member function)
Element access	
`at`	access specified element with bounds checking (public member function)
`operator[]`	access specified element (public member function)
`front`	access the first element (public member function)
`back`	access the last element (public member function)
`data`	direct access to the underlying contiguous storage (public member function)
Iterators	
`begin` `cbegin` (C++11)	returns an iterator to the beginning (public member function)
`end` `cend` (C++11)	returns an iterator to the end (public member function)
`rbegin` `crbegin` (C++11)	returns a reverse iterator to the beginning (public member function)
`rend` `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)
`reserve`	reserves storage (public member function)
`capacity`	returns the number of elements that can be held in currently allocated storage (public member function)
`shrink_to_fit` (DR*)	reduces memory usage by freeing unused memory (public member function)
Modifiers	
`clear`	clears the contents (public member function)
`insert`	inserts elements (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)
`erase`	erases elements (public member function)
`push_back`	adds an element to the end (public member function)
`emplace_back` (C++11)	constructs an element in-place at the end (public member function)
`append_range` (C++23)	adds a range of elements to the end (public member function)
`pop_back`	removes the last element (public member function)
`resize`	changes the number of elements stored (public member function)
`swap`	swaps the contents (public member function)

The Basics

Creating a std::vector

```cpp
#include <iostream>
#include <vector>

int main() {
  std::vector<u_short> wa_years; // empty vector of years

  wa_years.push_back(2014);      // adding to the Washington years.
  wa_years.push_back(2015);
  wa_years.push_back(2016);
  wa_years.push_back(2017);
  wa_years.push_back(2018);

  std::vector<u_short> co_years = {2019, 2020, 2021,
                                   2022, 2023, 2024}; // initializer list

  std::vector<u_short> cppcon_total_years(11);  // construct with a size

  std::vector<u_short> cppcon_space_odessy(11, 2001);  // with size and a default

  for (auto yr : cppcon_total_years) {
    std::cout << yr << ' ';
  }
  std::cout << std::endl;

  return 0;
}
```

```
[Running] && g++ -std=c++20 1_create.cpp -o 1_create

0 0 0 0 0 0 0 0 0 0 0

[Done] exited with code=0 in 0.883 seconds
```

Access data in a vector

```cpp
#include <iostream>
#include <vector>

int main() {
  std::vector<u_short> co_years = {2019, 2020, 2021,
                                   2022, 2023, 2024}; // initializer list

  std::cout << "Accessing the first elements...\n";
  std::cout << co_years[0] << std::endl;
  std::cout << co_years.front() << std::endl;

  std::cout << "Accessing the last elements...\n";
  std::cout << co_years.back() << std::endl;
  std::cout << co_years.at(5) << std::endl;

  std::cout << "Showing them all...\n";
  for (auto yr : co_years) {
    std::cout << yr << ' ';
  }
  std::cout << std::endl;

  return 0;
}
```

```
[Running] g++ -std=c++20 2_accessing.cpp -o 2_accessing
Accessing the first elements...
2019
2019

Accessing the last elements...
2024
2024

Showing them all...
2019 2020 2021 2022 2023 2024

[Done] exited with code=0 in 0.57 seconds
```

If [0] then why at(0)

```cpp
#include <iostream>
#include <vector>

int main() {
  std::vector<u_short> co_years = {2019, 2020, 2021,
                                   2022, 2023, 2024}; // initializer list

  std::cout << "When it works...\n";
  std::cout << co_years[0] << "\n";
  std::cout << co_years.at(5) << "\n";

  return 0;
}
```

```
[Running] g++ -std=c++20 3_why_at.cpp -o 3_why_at
When it works...
2019
2024

[Done] exited with code=0 in 0.589 seconds
```

then at() can catch

```cpp
#include <iostream>
#include <vector>

int main() {
  std::vector<u_short> co_years = {2019, 2020, 2021,
                                   2022, 2023, 2024}; // initializer list

  std::cout << "\nWhen it doesn't...\n";
  std::cout << co_years[7] << "\n";

  std::cout << "at() can catch it...\n";
  std::cout << co_years.at(7) << "\n";

  return 0;
}
```

```
[Running] g++ -std=c++20 4_at_can_catch.cpp -o 4_at_can_catch
When it doesn't...
0
at() can catch it...
libc++abi: terminating due to uncaught exception of type std::out_of_range: vector
/bin/sh: line 1: 25723 Abort trap: 6

[Done] exited with code=134 in 0.583 seconds
```

.data() when you want to c

```cpp
#include <iostream>
#include <vector>

int main() {
  std::vector<u_short> co_years = {2019, 2020, 2021, 2022, 2023, 2024};
  u_short *yrPtr = co_years.data();

  std::cout << "Years using data() pointer: ";
  for (size_t i = 0; i < co_years.size(); ++i) {
    std::cout << *(yrPtr + i) << " ";
  }
  yrPtr[1] = 9999;

  std::cout << "\nModified vector: ";
  for (int num : co_years) {
    std::cout << num << " ";
  }

  return 0;
}
```

```
[Running] g++ -std=c++20 5_data_when_c.cpp -o 5_data_when_c
Years using data() pointer:2019 2020 2021 2022 2023 2024
Modified vector: 2019 9999 2021 2022 2023 2024
[Done] exited with code=0 in 0.566 seconds
```

Memory Management

stack vs heap

```cpp
#include <iostream>
#include <vector>

int main() {
  using year = u_short;
  year cppconEarlyYears[5] = {2014, 2015, 2016, 2017, 2018};
  std::vector<year> cppconAuroraYears = {2019, 2021, 2022, 2023, 2024};

  std::cout << "Bellevue: ";
  for (int i = 0; i < 5; i++) {
    std::cout << cppconEarlyYears[i] << " ";
  }
  std::cout << "\nAurora: ";
  for (auto dy : cppconAuroraYears) {
    std::cout << dy << " ";
  }

  return 0;
}
```

```
[Running] g++ -std=c++20 0_stack_vs_heap.cpp -o 0_stack_vs_heap
Bellevue: 2014 2015 2016 2017 2018
Aurora: 2019 2021 2022 2023 2024
[Done] exited with code=0 in 0.576 seconds
```

what's the stack limit?

```cpp
#include <iostream>
// ulimit -s on mac/linux works too!
void useStack(int depth) {
  char buffer[1024]; // Allocate 1 KB on the stack
  std::cout << "Depth: " << depth << " Address: " << &buffer << std::endl;
  useStack(depth + 1);
}

int main() {
  useStack(1);
  return 0;
}
```

```
[Running] g++ -std=c++20 2_stack_limit.cpp -o 2_stack_limit
Depth: 2 Address: 0x16f55e978
Depth: 3 Address: 0x16f55e548
...
Depth: 7802 Address: 0x16ed652f8
Depth: 7803 Address: 0x16ed64ec8
Depth: 7804 Address: 0x16ed64a98
/bin/sh: line 1: 52763 Segmentation fault: 11  ./almost-always-vector/src/3_memory/2_stack_limi

[Done] exited with code=139 in 0.626 seconds
```

the trade offs

Stack

Fast - pointer adjustment.

Automatic - easy clean up.

Predictable - easy to debug.

Locality - cache performance.

Safety - see automatic.

Use when data is small, doesn't need to persist beyond the function and you want speed.

Heap

Flexible - dynamic at runtime.

Large - bigger is better right?

Lifetime - !f(x) dependent.

Sharing - between two threads.

Use when data is large, needs to persist, be flexible.

Watch your toes.

size vs capacity

```cpp
#include <iostream>
#include <vector>

int main() {
  std::vector<int> co = {2019, 2020, 2021, 2022, 2023, 2024};
  std::cout << "Size/Capacity: " << co.size() << "/" << co.capacity() << "\n";

  std::vector<int> wa;
  std::cout << "Size/Capacity: " << wa.size() << "/" << wa.capacity() << "\n";
  wa.push_back(2014);
  std::cout << "Size/Capacity: " << wa.size() << "/" << wa.capacity() << "\n";
  wa.push_back(2015);
  std::cout << "Size/Capacity: " << wa.size() << "/" << wa.capacity() << "\n";
  wa.push_back(2016);
  std::cout << "Size/Capacity: " << wa.size() << "/" << wa.capacity() << "\n";

  return 0;
}
```

```
[Running] g++ -std=c++20 2_size_vs_capacity.cpp -o 2_size_vs_capacity
Size/Capacity: 6/6
Size/Capacity: 0/0
Size/Capacity: 1/1
Size/Capacity: 2/2
Size/Capacity: 3/4
[Done] exited with code=0 in 0.669 seconds
```

old c style reallocation

```cpp
#include <iostream>
int main() {
  int *cArray = new int[5];
  for (int i = 0; i < 5; ++i) {
    cArray[i] = i + 1;
  }
  int *temp = new int[10];
  for (int i = 0; i < 5; ++i) {
    temp[i] = cArray[i];
  }
  delete[] cArray; // Important: free the old memory
  cArray = temp;
  for (int i = 5; i < 10; ++i) {
    cArray[i] = i + 1;
  }
  for (int i = 0; i < 10; ++i) {
    std::cout << cArray[i] << " ";
  }
  std::cout << "\n";
  delete[] cArray;
  return 0;
}
```

```
[Running] g++ -std=c++20 3_c_style_array.cpp -o 3_c_style_array
1 2 3 4 5 6 7 8 9 10
[Done] exited with code=0 in 0.523 seconds
```

Managing allocation

```cpp
#include <iostream>
#include <vector>

void addYears(std::vector<int> &vec, int start, int years) {
  for (auto i = start; i < start + years; ++i) {
    vec.push_back(i);
  }
}

int main() {
  std::vector<int> co;
  addYears(co, 2019, 4);
  std::cout << "Size/Capacity: " << co.size() << "/" << co.capacity() << "\n";
  addYears(co, co.back(), 6);
  std::cout << "Size/Capacity: " << co.size() << "/" << co.capacity() << "\n";
  addYears(co, co.back(), 10);
  std::cout << "Size/Capacity: " << co.size() << "/" << co.capacity() << "\n";
  return 0;
}
```

```
[Running] g++ -std=c++20 4_managing_allocation.cpp -o 4_managing_allocation
Size/Capacity: 4/4
Size/Capacity: 10/16
Size/Capacity: 20/32
[Done] exited with code=0 in 0.587 seconds
```

reserve to prevent allocations

```cpp
#include <iostream>
#include <vector>

void addYears(std::vector<int> &vec, int start, int years) {
  for (auto i = start; i < start + years; ++i) {
    vec.push_back(i);
  }
}

int main() {
  std::vector<int> co;
  addYears(co, 2019, 4);
  std::cout << "Size/Capacity: " << co.size() << "/" << co.capacity() << "\n";
  co.reserve(40);
  addYears(co, co.back(), 6);
  std::cout << "Size/Capacity: " << co.size() << "/" << co.capacity() << "\n";
  addYears(co, co.back(), 10);
  std::cout << "Size/Capacity: " << co.size() << "/" << co.capacity() << "\n";
  return 0;
}
```

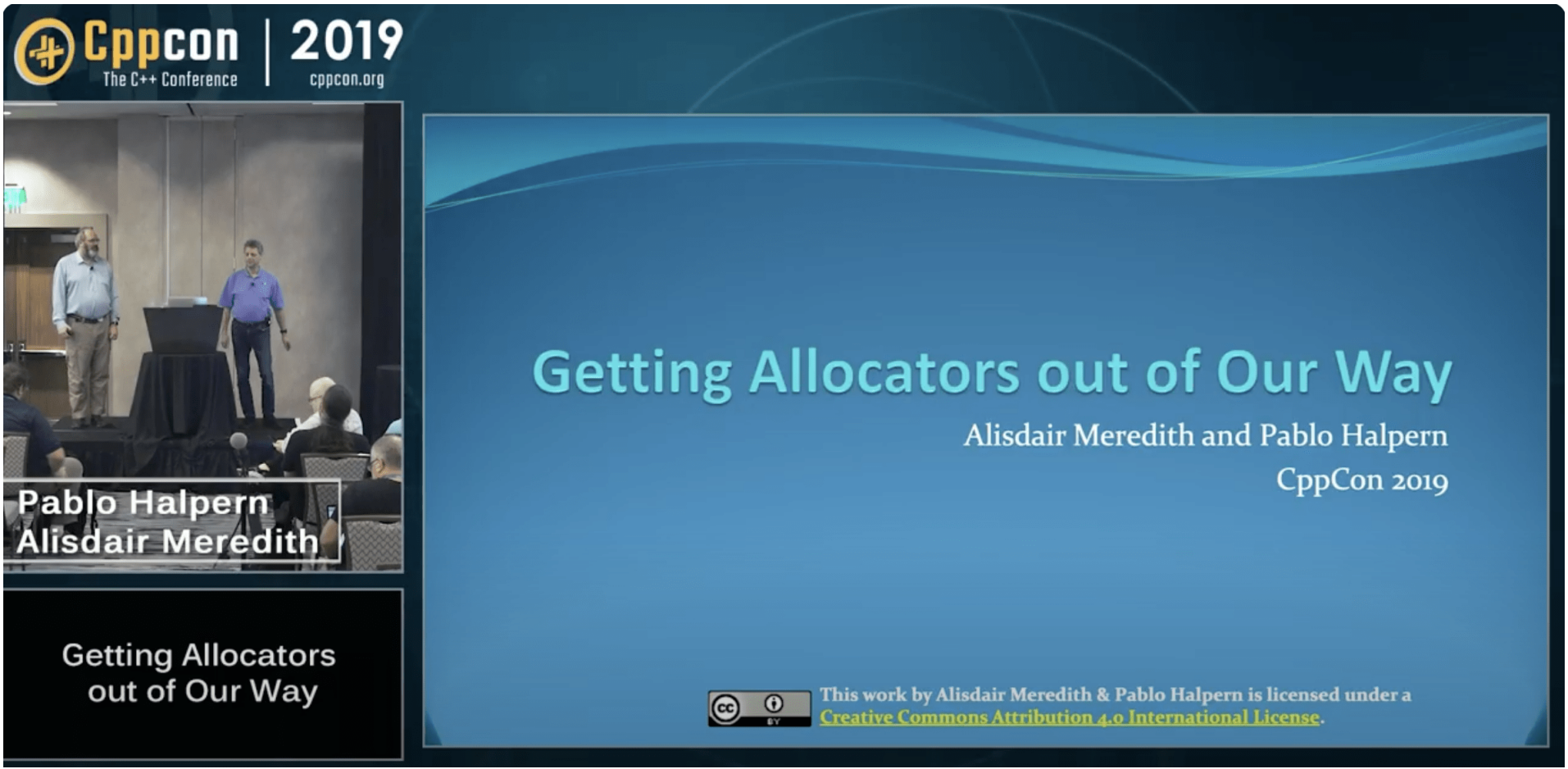
```
[Running] g++ -std=c++20 5_reserve_allocations.cpp -o 5_reserve_allocations
Size/Capacity: 4/4
Size/Capacity: 10/40
Size/Capacity: 20/40
[Done] exited with code=0 in 0.642 seconds
```

shrink_to_fit

```cpp
#include <iostream>
#include <vector>

void addYears(std::vector<int> &vec, int start, int years) { // same implementation...}

int main() {
  std::vector<int> co;
  addYears(co, 2019, 4);
  addYears(co, co.back(), 6);
  addYears(co, co.back(), 12);
  std::cout << "Size/Capacity: " << co.size() << "/" << co.capacity() << "\n";
  co.shrink_to_fit(); // non-binding
  std::cout << "Size/Capacity: " << co.size() << "/" << co.capacity() << "\n";

  for (auto y : co) {
    std::cout << y << " ";
  }

  return 0;
}
```

```
[Running] g++ -std=c++20 6_shrink_to_fit.cpp -o 6_shrink_to_fit
Size/Capacity: 22/32
Size/Capacity: 22/22
2019 2020 2021 2022 2022 2023 2024 2025 2026 2027 2027 2028 2029 2030 2031 2032 2033 2034 2035 2(
[Done] exited with code=0 in 0.562 seconds
```

std::vector on the stack?



https://www.youtube.com/watch?v=RLezJuqNcEQ&t=2676s

Iterators

what is an iterator

```cpp
#include <iostream>
#include <vector>

int main() {
  std::vector<int> co = {2019, 2020, 2021, 2022, 2023, 2024};

  auto it = co.begin();
  std::cout << "Value pointed to by iterator: " << *it << "\n"; // Output: 10
  std::cout << "Address of element in vector: " << &(*it)
            << "\n"; // Address of 10 in the vector
  std::cout << "Address of the iterator itself: " << &it
            << "\n"; // Address of the iterator object

  std::cout << "Iterator value at 3: " << *(it + 3) << "\n";

  return 0;
}
```

```
[Running] g++ -std=c++20 0_hidden_iterator.cpp -o 0_hidden_iterator
Value pointed to by iterator: 2019
Address of element in vector: 0x139605f60
Address of the iterator itself: 0x16d7fb168
Iterator value at 3: 2022

[Done] exited with code=0 in 0.569 seconds
```

pointers be like...

what is an iterator

Though a pointer can be thought of as an iterator, as it allows us to iterate over elements in a block of memory.

Iterators in the stl abstract away and give guard rails to the pointer. Allowing the benifits and helping minimize mistakes.

Its a pointer - in bubble wrap

different iterators

```cpp
#include <iostream>
#include <vector>

int main() {
  std::vector<int> co = {2019, 2020, 2021, 2022, 2023, 2024};

  std::cout << "Forward iteration using begin() and end():"
            << "\n";
  for (auto it = co.begin(); it != co.end(); ++it) {
    std::cout << *it << " "; // Dereference the iterator to get the value
  }

  std::cout << "\nReverse iteration using rbegin() and rend():"
            << "\n";
  for (auto rit = co.rbegin(); rit != co.rend(); ++rit) {
    std::cout << *rit << " "; // Dereference the reverse iterator
  }

  return 0;
}
```

```
[Running] g++ -std=c++20 1_different_iterators.cpp -o 1_different_iterators
Forward iteration using begin() and end():
2019 2020 2021 2022 2023 2024
Reverse iteration using rbegin() and rend():
2024 2023 2022 2021 2020 2019
[Done] exited with code=0 in 0.583 seconds
```

favorite not iterator

```cpp
#include <iostream>
#include <vector>

int main() {
  std::vector<int> co = {2019, 2020, 2021, 2022, 2023, 2024};

  std::cout << "Is range based for loop an iterator?s\n";
  for (auto yr : co) {
    std::cout << yr << " ";
  }

  std::cout << "\nIterator works as a pointer and not a copy...\n";
  for (auto it = co.begin(); it != co.end(); ++it) {
    std::cout << *it << " "; // Dereference the iterator to get the value
  }

  return 0;
}
```

```
[Running] g++ -std=c++20 2_favorite_not_iterator.cpp -o 2_favorite_not_iterator
Is range based for loop an iterator?
2019 2020 2021 2022 2023 2024
Iterator works as a pointer and not a copy...
2019 2020 2021 2022 2023 2024
[Done] exited with code=0 in 0.645 seconds
```

different types of iterators

Iterator category	Operations and storage requirement						
	write	read	increment		decrement	random access	contiguous storage
			without multiple passes	with multiple passes			
LegacyIterator			Required				
LegacyOutputIterator	Required		Required				
LegacyInputIterator (mutable if supports write operation)		Required	Required				
LegacyForwardIterator (also satisfies *LegacyInputIterator*)		Required	Required	Required			
LegacyBidirectionalIterator (also satisfies *LegacyForwardIterator*)		Required	Required	Required	Required		
LegacyRandomAccessIterator (also satisfies *LegacyBidirectionalIterator*)		Required	Required	Required	Required	Required	
LegacyContiguousIterator[1] (also satisfies *LegacyRandomAccessIterator*)		Required	Required	Required	Required	Required	Required

invalidating iterators

```cpp
#include <iostream>
#include <vector>

int main() {
  std::vector<int> cppconYears = {2019, 2020, 2021, 2022, 2023, 2024};
  auto it = cppconYears.begin() + 1;

  std::cout << "Lost year: " << *it << std::endl;
  cppconYears.push_back(2025);
  std::cout << "After push_back: " << *it << std::endl;

  for (auto yr : cppconYears) {
    std::cout << yr << ' ';
  }

  return 0;
}
```

```
[Running] g++ -std=c++20 3_invalidating.cpp -o 3_invalidating
Lost year: 2020
After push_back, iterator now points to: 0
2019 2020 2021 2022 2023 2024 2025
[Done] exited with code=0 in 0.586 seconds
```

Category	Container	After **insertion**, are...		After **erasure**, are...		Conditionally
		iterators valid?	**references** valid?	**iterators** valid?	**references** valid?	
Sequence containers	`array`	N/A		N/A		
	`vector`	No		N/A		Insertion changed capacity
		Yes		Yes		Before modified element(s) (for insertion only if capacity didn't change)
		No		No		At or after modified element(s)
	`deque`	No	Yes	Yes, except erased element(s)		Modified first or last element
			No	No		Modified middle only
	`list`	Yes		Yes, except erased element(s)		
	`forward_list`	Yes		Yes, except erased element(s)		

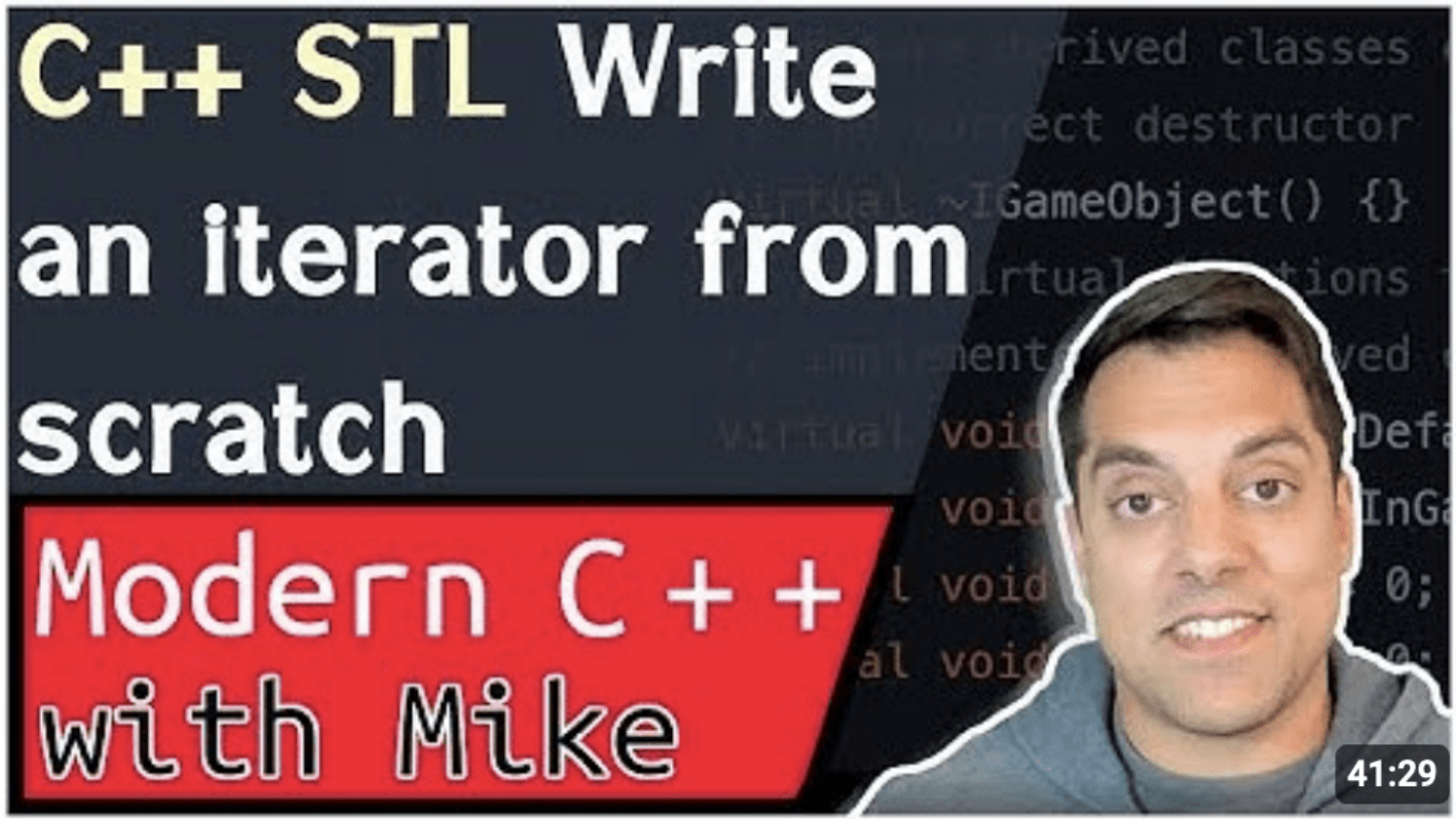
Associative containers	`set`
	`multiset`
	`map`
	`multimap`
		Yes		Yes, except erased element(s)		

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

		Yes		Yes, except erased element(s)		No rehash

https://en.cppreference.com/w/cpp/container#Iterator_invalidation

excellent deep dive on iterators



https://youtu.be/Fv8oj8EdssY?si=AIfpp3gOqXNXQkfI

Algorithms

Unary Predicates + Iterators

predicate as a function

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

// Unary predicate function that checks if a number is even
bool isEven(int n) { return n % 2 == 0; }

int main() {
  std::vector<int> co = {2019, 2020, 2021, 2022, 2023, 2024};

  // Using std::find_if with a unary predicate to find the first even number
  auto it = std::find_if(co.begin(), co.end(), isEven);

  if (it != co.end()) {
    std::cout << "First even number: " << *it << '\n';
  } else {
    std::cout << "No even number found." << '\n';
  }

  return 0;
}
```

```
[Running] g++ -std=c++20 0_pred_as_function.cpp -o 0_pred_as_function
First even number: 2020
[Done] exited with code=0 in 0.565 seconds
```

adjusting our iterator

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

// Unary predicate function that checks if a number is even
bool isEven(int n) { return n % 2 == 0; }

int main() {
  std::vector<int> co = {2019, 2020, 2021, 2022, 2023, 2024};

  // Using std::find_if with a unary predicate to find the first even number
  auto it = std::find_if(co.begin() + 2, co.end(), isEven);

  if (it != co.end()) {
    std::cout << "First even number: " << *it << '\n';
  } else {
    std::cout << "No even number found." << '\n';
  }

  return 0;
}
```

```
[Running] g++ -std=c++20 0_pred_as_function.cpp -o 0_pred_as_function
First even number: 2022
[Done] exited with code=0 in 0.565 seconds
```

as a lambda

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

// Unary predicate function that checks if a number is even
bool isEven(int n) { return n % 2 == 0; }

int main() {
  std::vector<int> co = {2019, 2020, 2021, 2022, 2023, 2024};

  // Using std::find_if with a unary predicate to find the first even number
  auto it = std::find_if(co.begin(), co.end(), [](int n) { return n % 2 == 0; });

  if (it != co.end()) {
    std::cout << "First even number: " << *it << '\n';
  } else {
    std::cout << "No even number found." << '\n';
  }

  return 0;
}
```

```
[Running] g++ -std=c++20 2_pred_as_lambda.cpp -o 2_pred_as_lambda
First even number: 2020
[Done] exited with code=0 in 0.566 seconds
```

Thank you Michael Maniscalco

fun with algorithms

```cpp
#include "nlohmann/json.hpp"
#include <iostream>
#include <string>

using json = nlohmann::json;

struct CreditCardInfo {
  std::string type;
  std::string name;
  std::string number;
  std::string cvv;
  std::string expiry;
  long amount;

  void print() {
    std::cout << "Type: " << type << "\n";
    std::cout << "Name: " << name << "\n";
    std::cout << "Number: " << number << "\n";
    std::cout << "CVV: " << cvv << "\n";
    std::cout << "Expiry: " << expiry << "\n";
    std::cout << "Amount: " << amount / 10 << "\n\n";
  }
};
```

Given the following json example data:

```
{
   "type": "visa",
   "name": "Prof. Madyson Towne",
   "number": "4251080104364294",
   "cvv": "372",
   "expiry": "11/25",
  "amount": 4593
},
```

Please create for me a test data file with 2000 elements, randomizing the type between visa, mc, and amex. Also generate random names, card numbers, cvv, expiry dates and amounts.

Generated Card Data

#	type	name	number	cvv	expiry	ar
1	mc	Carlos Wall	2293938772721091	374	03/33	4
2	amex	Thomas Norris	343222645097342	5883	12/31	7
3	amex	Mia Patel	376179634990491	2197	05/30	5
4	visa	Debbie Parker	4377117124962472	789	05/33	4
5	visa	Derrick Sanchez	4592035961242288	778	08/32	1
6	mc	Kathleen Leon	5394879336599576	283	08/34	8

I have generated the test data file with 2000 elements as requested. You can download it using the link below:

Download card_data_2000.json

repeat the last request but randomize the amount from 1 to 999999 for a total of 10000 elements please

Generated Large Card Data

#	type	name	number	cvv	expiry
1	visa	Veronica Dunn	4507852921415182	494	10/32
2	mc	Matthew Williams	2257192028873488	236	08/33
3	visa	Tammy Tyler	4224236903363457	605	04/26
4	visa	Luke Morgan	4882604419015841	767	05/34
5	visa	Patricia Holt	4118438764472844	497	07/26
6	mc	Yolanda Castro	2294696121980565	986	11/33

The new test data file with 10,000 elements has been generated. You can download it from the link below:

Download large_card_data_10000.json

please repeat this task for a 100000 elements

loading json card data

```
1  nlohmann::json load_card_data(std::string file_path) {
2    std::ifstream file(file_path);
3
4    if (!file.is_open()) {
5      std::cerr << "Could not open the file!" << std::endl;
6      return 1;
7    }
8
9    // Parse the file content as JSON
10   nlohmann::json json_data;
11   file >> json_data;
12
13   return json_data;
14 }
```

std::accumulate 10,000 items

```cpp
int main() {
  auto json_data = load_card_data("../large_card_data_10000.json");
  std::vector<CreditCardInfo> cards = json_data.get<std::vector<CreditCardInfo>>();
  std::vector<std::string> card_types = {"visa", "mc", "amex"};

  std::cout << "item count: " << cards.size() << "\n";
  for (auto ct : card_types) {
    // Use std::accumulate to sum amounts of the specific card type
    long totalAmount = std::accumulate(cards.begin(), cards.end(), 0L,
                          [&ct](long sum, const CreditCardInfo &card) {
                            return sum + (card.type == ct ? card.amount : 0);
                          });

    std::cout << "Total amount for " << ct << ": " << totalAmount / 10 << "\n";
  }

  return 0;
}
```

```
[Running] g++ -std=c++20 3_accumulate_cc.cpp -o 3_accumulate_cc
item count: 10000
Total amount for visa: 166488890
Total amount for mc: 163266022
Total amount for amex: 170784874
[Done] exited with code=0 in 1.429 seconds
```

std::accumulate 100,000

```cpp
int main() {
  auto json_data = load_card_data("../large_card_data_100000.json");
  std::vector<CreditCardInfo> cards = json_data.get<std::vector<CreditCardInfo>>();
  std::vector<std::string> card_types = {"visa", "mc", "amex"};

  std::cout << "item count: " << cards.size() << "\n";
  for (auto ct : card_types) {
    // Use std::accumulate to sum amounts of the specific card type
    long totalAmount = std::accumulate(cards.begin(), cards.end(), 0L,
                          [&ct](long sum, const CreditCardInfo &card) {
                            return sum + (card.type == ct ? card.amount : 0);
                          });

    std::cout << "Total amount for " << ct << ": " << totalAmount / 10 << "\n";
  }

  return 0;
}
```

```
[Running] g++ -std=c++20 3_accumulate_cc.cpp -o 3_accumulate_cc
item count: 100000
Total amount for visa: 1681038353
Total amount for mc: 1660805502
Total amount for amex: 1663977829
[Done] exited with code=0 in 2.749 seconds
```

How does it compare?

vs. std::list

- A doubly linked list, no random access (O(n) to traverse).
- Efficient for insertions/deletions anywhere (O(1) with an iterator) but has higher memory overhead and slower iteration due to poor cache locality.

Choose std::list when you need frequent insertions/deletions at arbitrary positions.

Otherwise... just start with std::vector

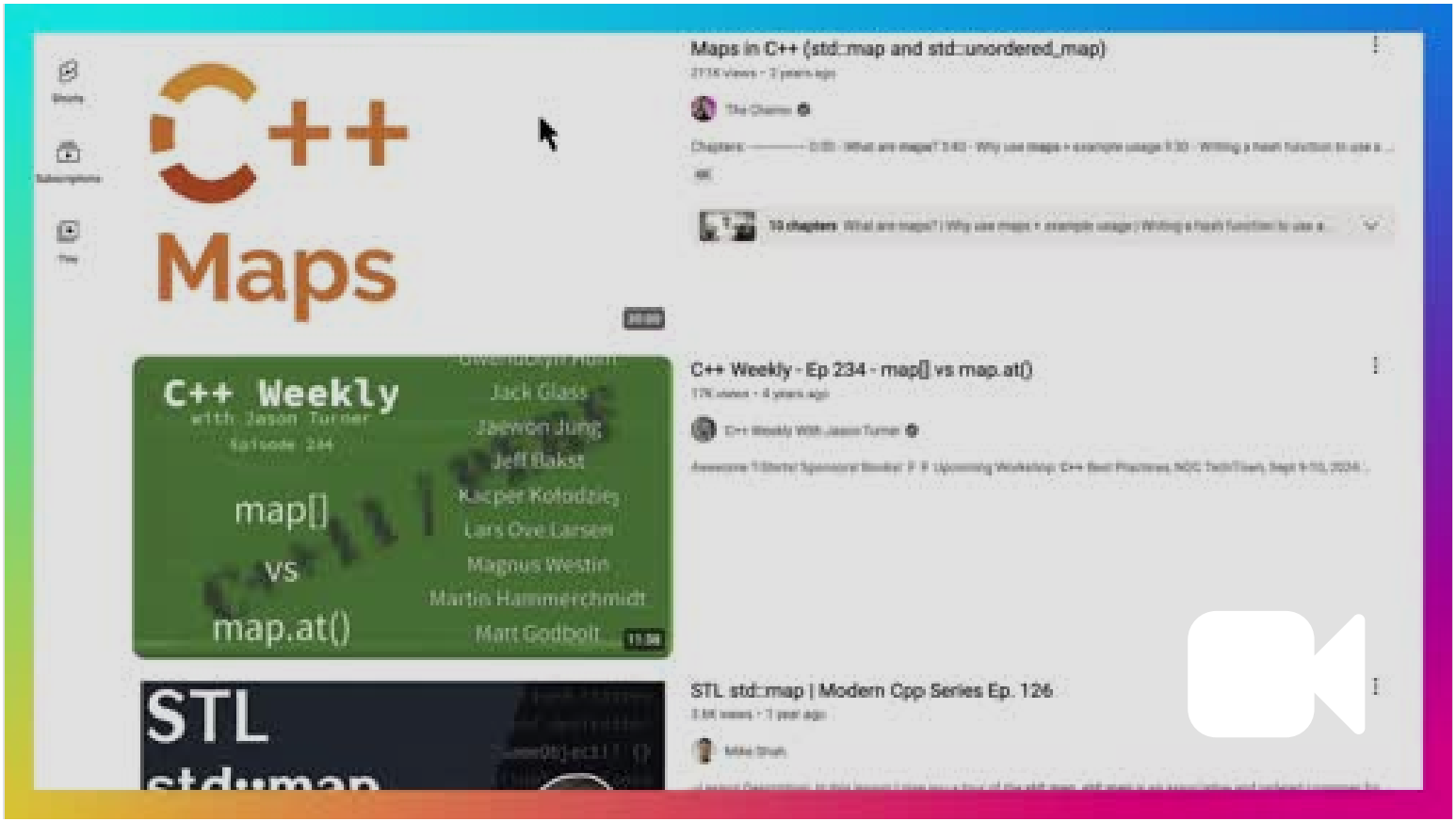
vs std::deque

- Non-contiguous memory, but still supports random access (slightly slower).
- Fast insertions/removals at **both ends** (O(1)).
- Less cache efficient.
- No need for reallocation on resizing.

Choose std::deque when you need efficient insertions/removals at both the front and back.

Otherwise... just start with std::vector

vs std::map



apples and oranges...

Why we
Almost Always Vector

Efficiency

Cache-Friendly, Fast Access: std::vector stores elements in **contiguous memory**, which is highly efficient because modern CPUs are optimized for sequential data access. Improving **cache locality**, meaning accessing elements in sequence is faster than containers like std::list, which have non-contiguous memory.

Amortized Constant-Time Growth: When adding elements, std::vector resizes by allocating more memory in chunks (usually doubling its capacity). Though resizing has a cost, this occurs infrequently, making most insertions run in **amortized constant time**. Ensuring adding elements is generally efficient.

Practicality

Automatic Memory Management: One of the biggest practical benefits of std::vector is that it handles memory allocation and deallocation automatically. You don’t need to worry about tracking memory manually or knowing the collection size in advance. It grows as you need it.

Random Access: You can access elements in constant time via indexing (i.e., vec[i]). This practicality is a big advantage compared to containers like std::list, where you’d need linear time to access an element at a specific position.

Versatility

Wide Standard Library Support: std::vector is fully integrated with the C++ Standard Library, meaning it works seamlessly with algorithms like std::sort, std::find, and other utilities. You can pass std::vector directly to algorithms, apply transformations, and leverage the iterator interface.

Dynamic Sizing and Flexible Insertion: While arrays (std::array) require you to know the size at compile time, std::vector can dynamically grow or shrink, making it much more versatile. It supports useful operations like push_back(), emplace_back(), and insert() for flexible management of elements.

Thanks

AAV

FTW

https://github.com/kevinbcarpenter/almost-always-vector