Welcome to Week 3! Link to Attendance Form ↓

What type of initialization can all types use?

- Structured binding
  - `auto [first, second] = p;`
- Member-wise
  - `student.name = “Jacob”`
- Uniform initialization
  - `Student jacob { “Jacob”, ”NM”, 21 }`

What type of initialization can all types use?

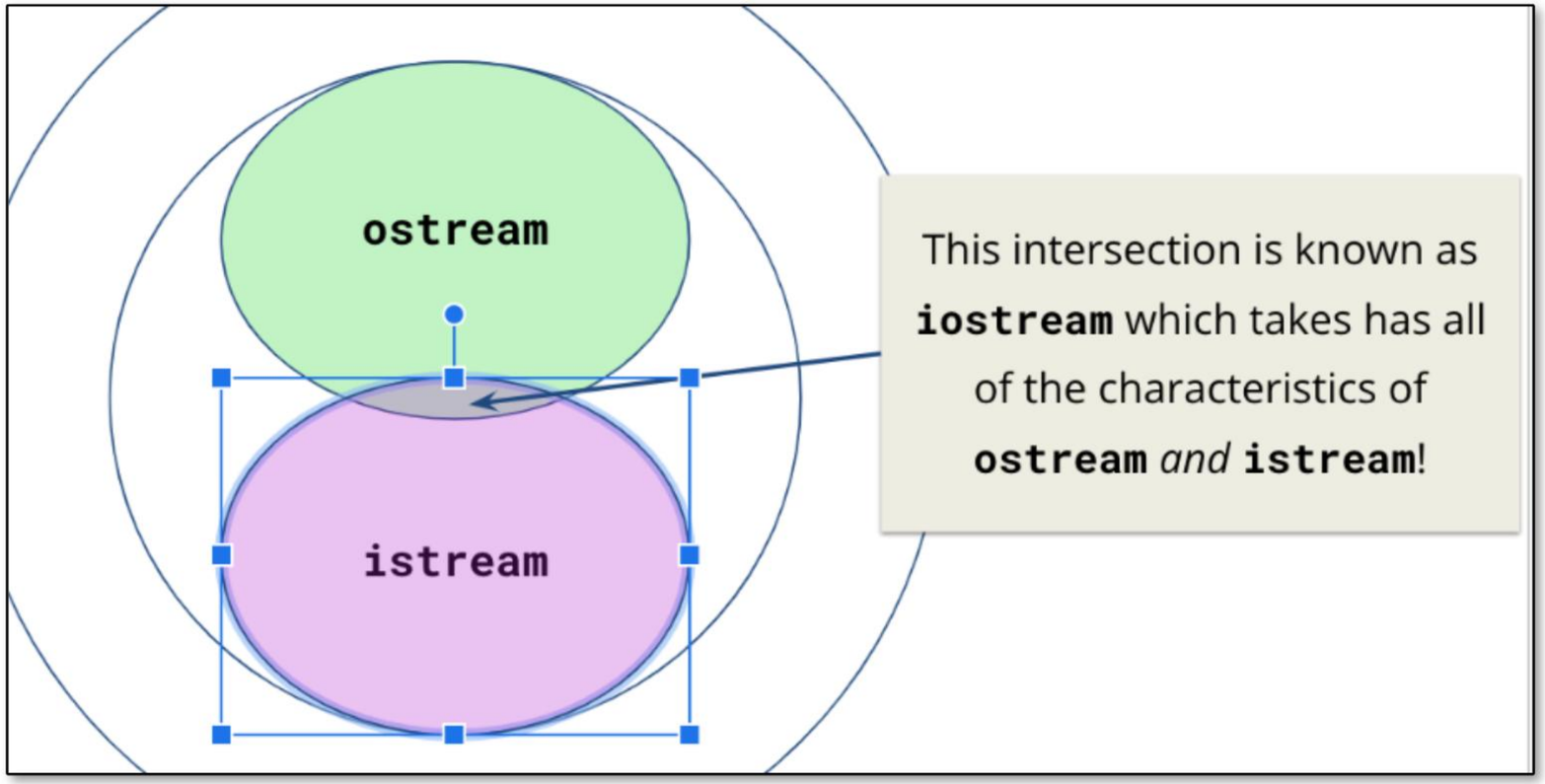
- Structured binding
  - `auto [first, second] = p;`
- Member-wise
  - `student.name = “Jacob”`
- ✅ **Uniform initialization**
  - **`Student jacob { “Jacob”, ”NM”, 21 }`**

A `stringstream` is an…

- Input stream
- Output stream
- Both!
- Neither!

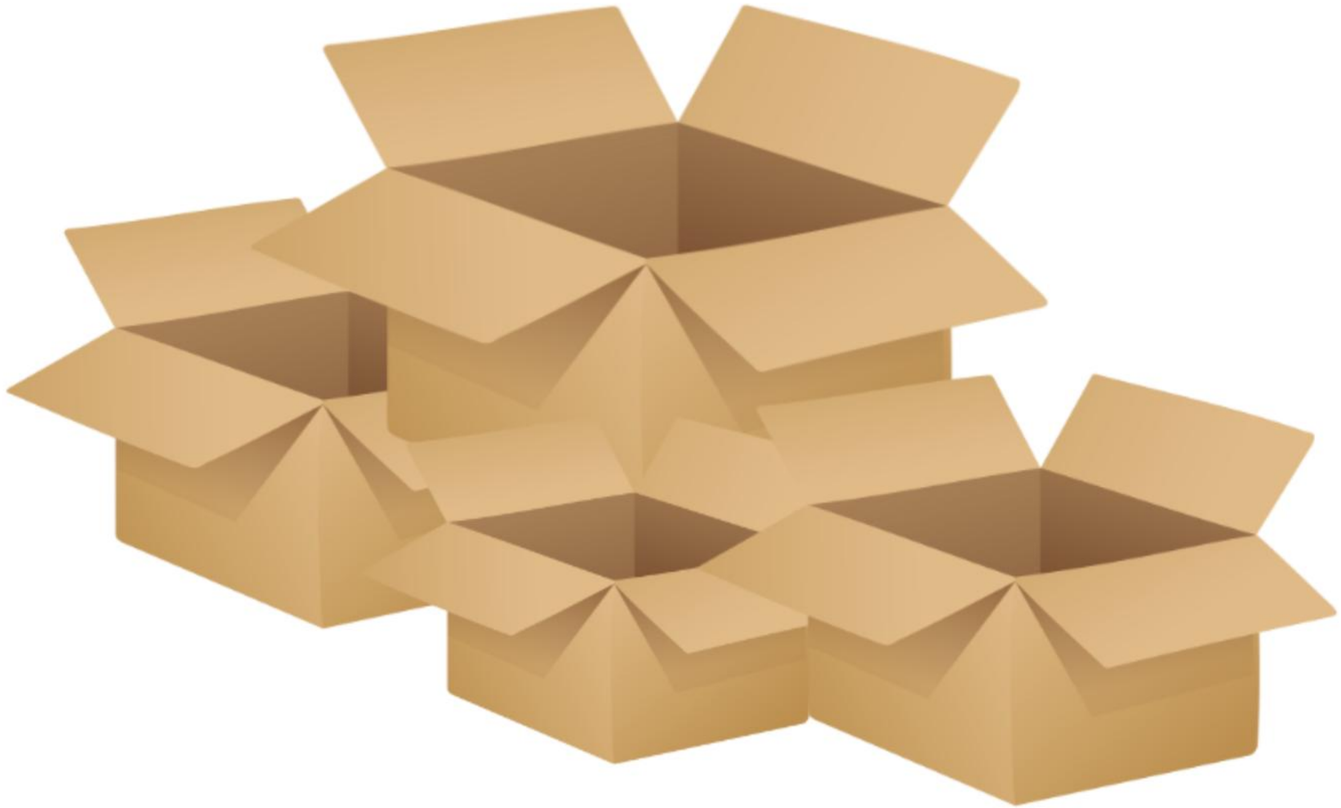
A `stringstream` is an…

- Input stream
- Output stream
- ✅ **Both!**
- Neither!




makita
305mm(12") Slide Compound Miter Saw
LIVEX
B4003PB
BOX 1 of 3

A Disorganized Garage




DIY with style
ETC.
HUSKY

An Organized Garage

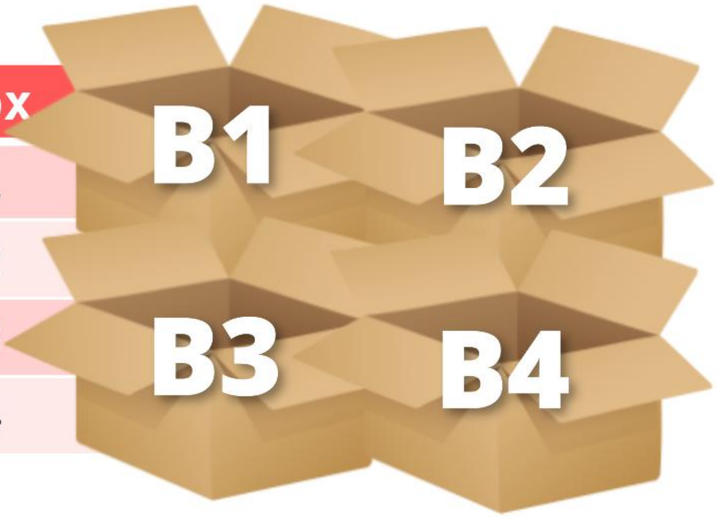
Item	Box
Tools	B1
DVDs	B2
Books	B3
Snacks	B4

Disorganized

Disorganized

Organized

Item	Box
Tools	B1
DVDs	B2
Books	B3
Snacks	B4



- Space efficient
- Slow to lookup item

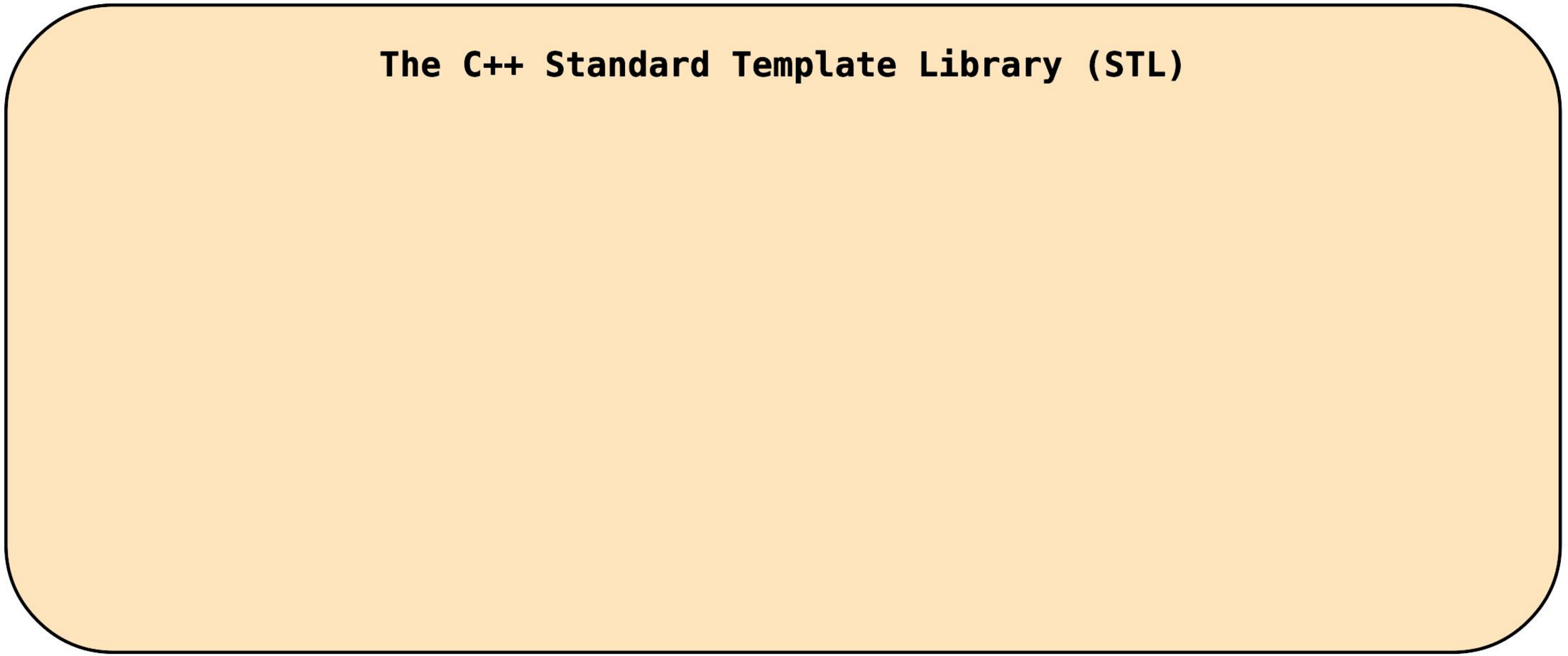
Lecture 5: Containers

Stanford CS106L, Winter 2025

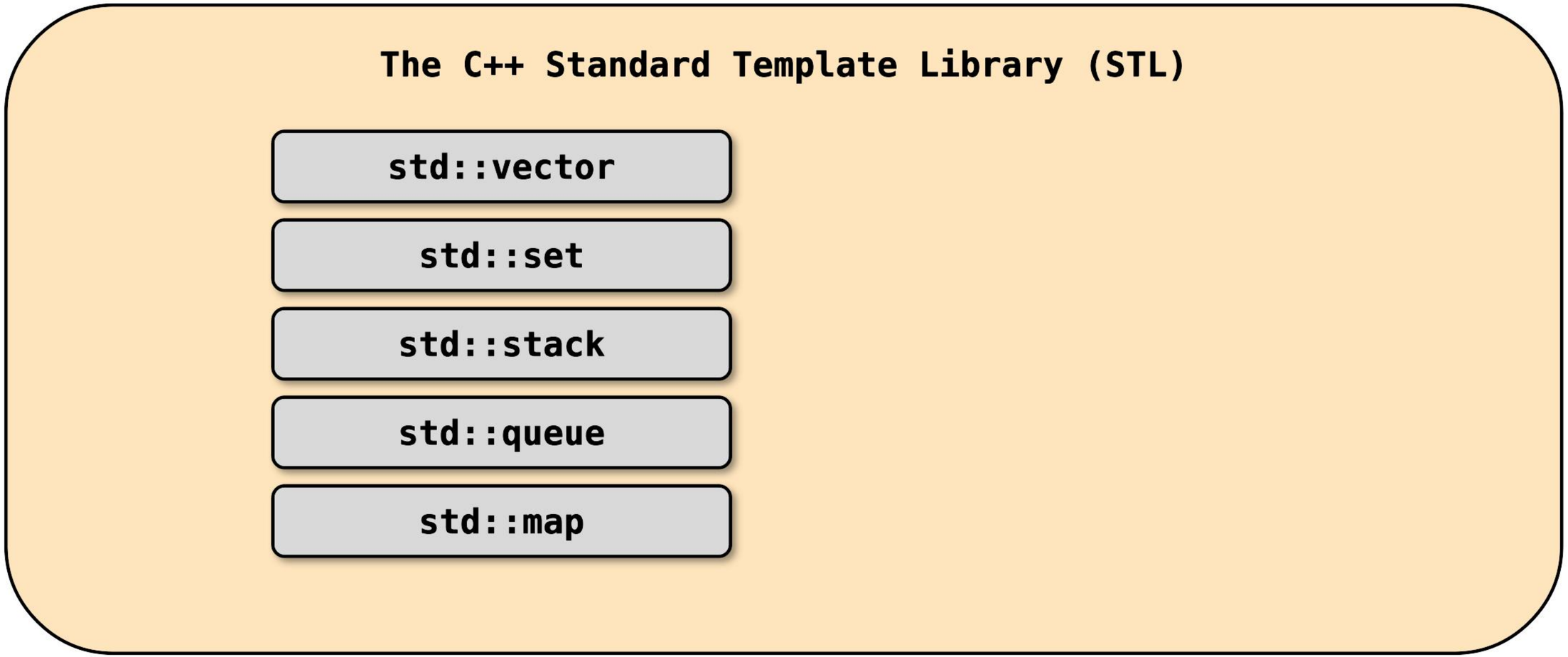
Today we're going beyond the Stanford C++ libraries!

Today we're going beyond the Stanford C++ libraries!

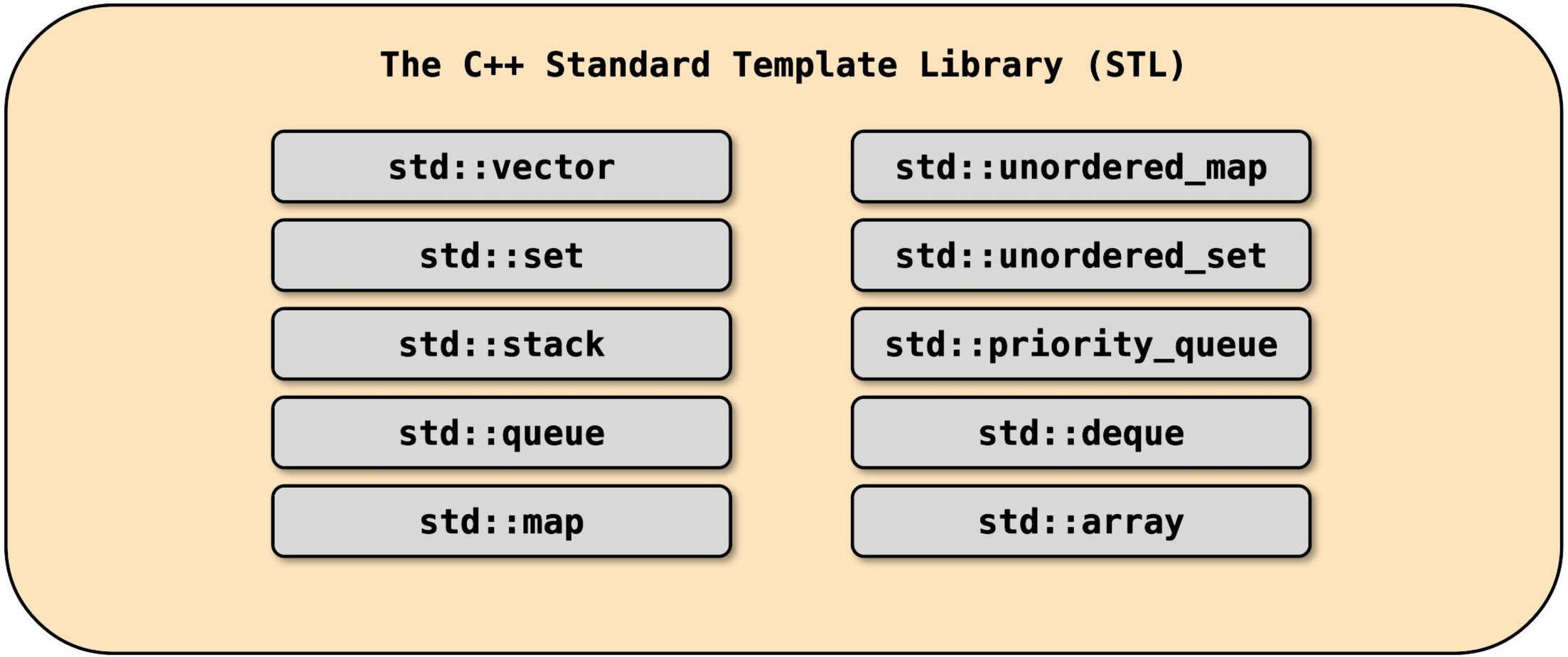
The many containers of C++



The many containers of C++

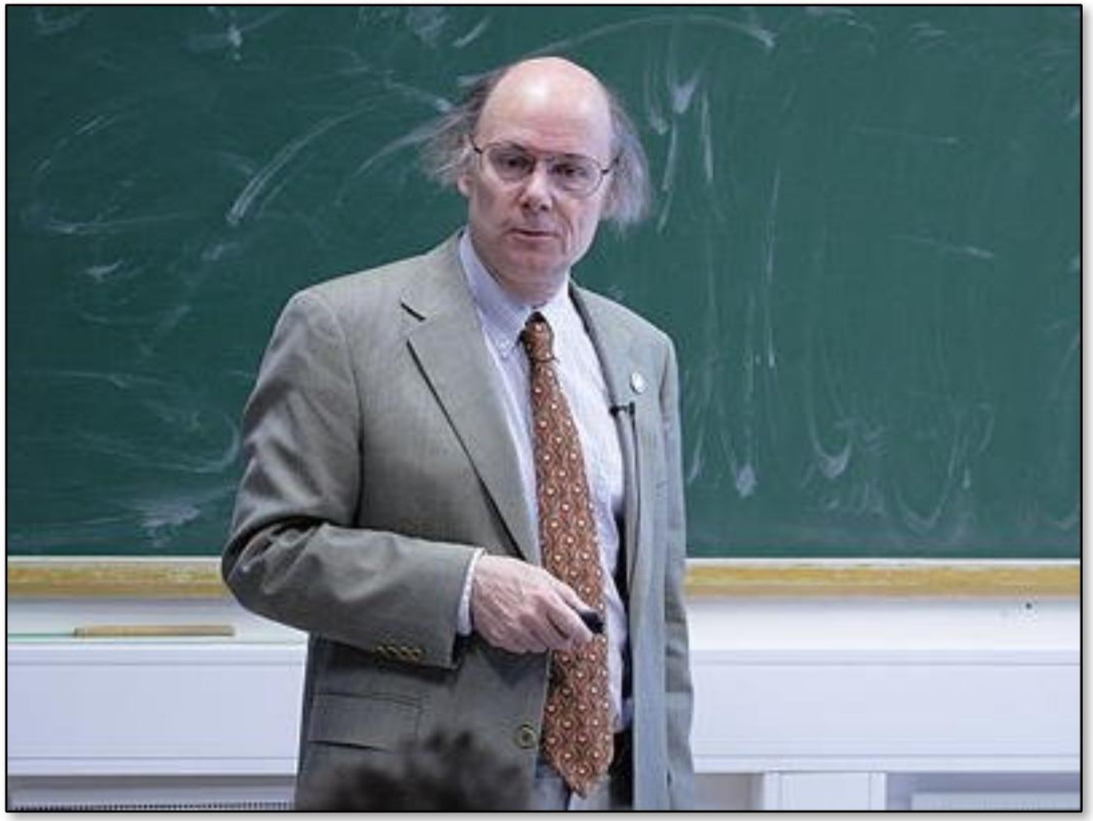


The many containers of C++



Which container do I use?


The space-time tradeoff



“Space is time”
— Bjarne Stroustrup

[source]

Today’s Agenda

Today’s Agenda

- What the heck is the STL? What are templates?
  - ”The Standard Template Library”
- Sequence Containers
  - A linear sequence of elements
- Associative Containers
  - A set of elements organized by unique keys

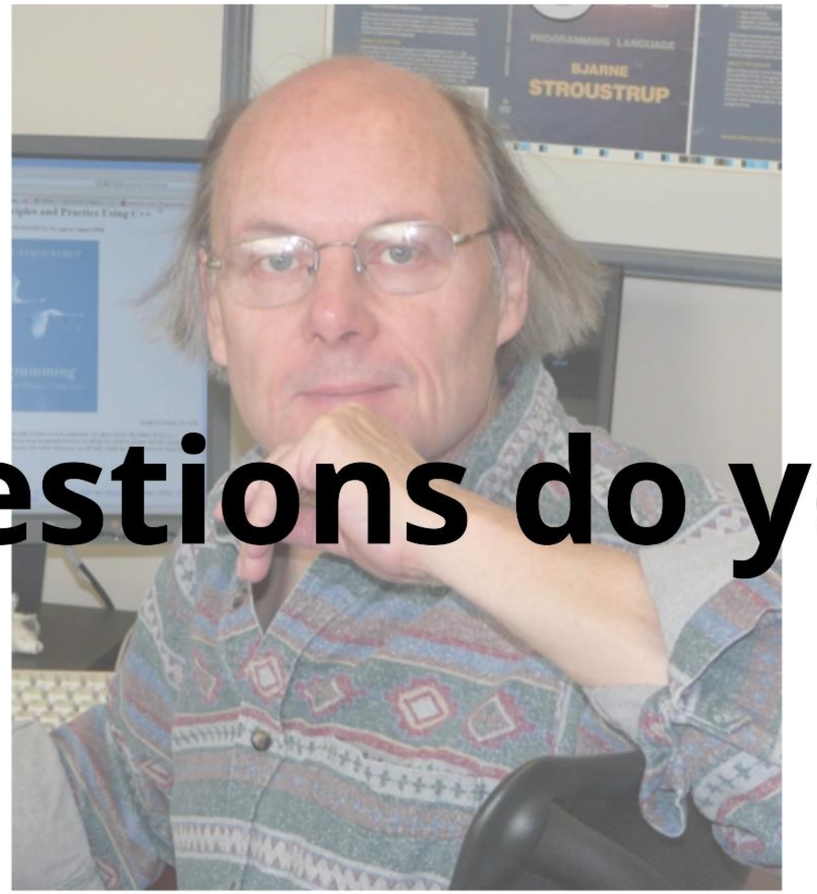
Disclaimer: We’re covering a lot of material!

(try not to get lost in the details!)

But also: we can’t cover everything!

(please ask us questions or reach out on Ed!)

What questions do you have?



bjarne_about_to_raise_hand

What is the STL?

STL: Standard Template Library

What are templates?

What are templates?

What are templates?

What are templates?

What are templates?

What are templates?

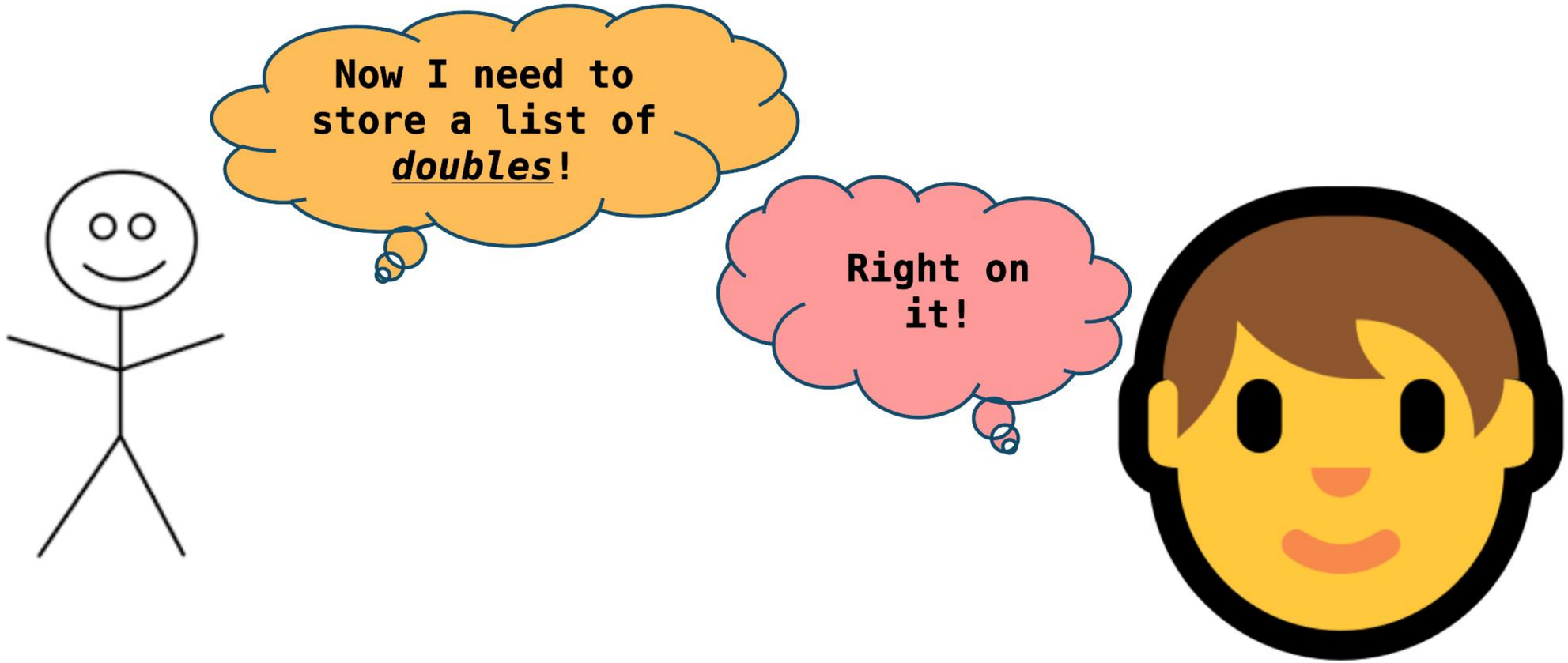
What are templates?


```
class IntVector {
   // Code to store
   // a list of
   // integers…
};
```

What are templates?

What are templates?

What are templates?



What are templates?

What are templates?

What are templates?

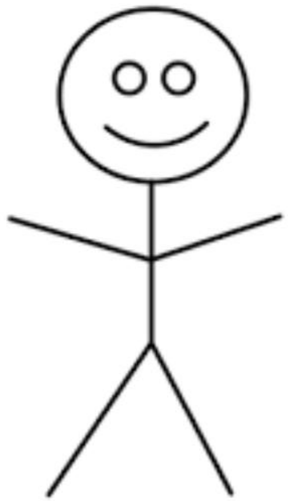

```
class DoubleVector {
   // Code to store
   // a list of
   // doubles…
};
```

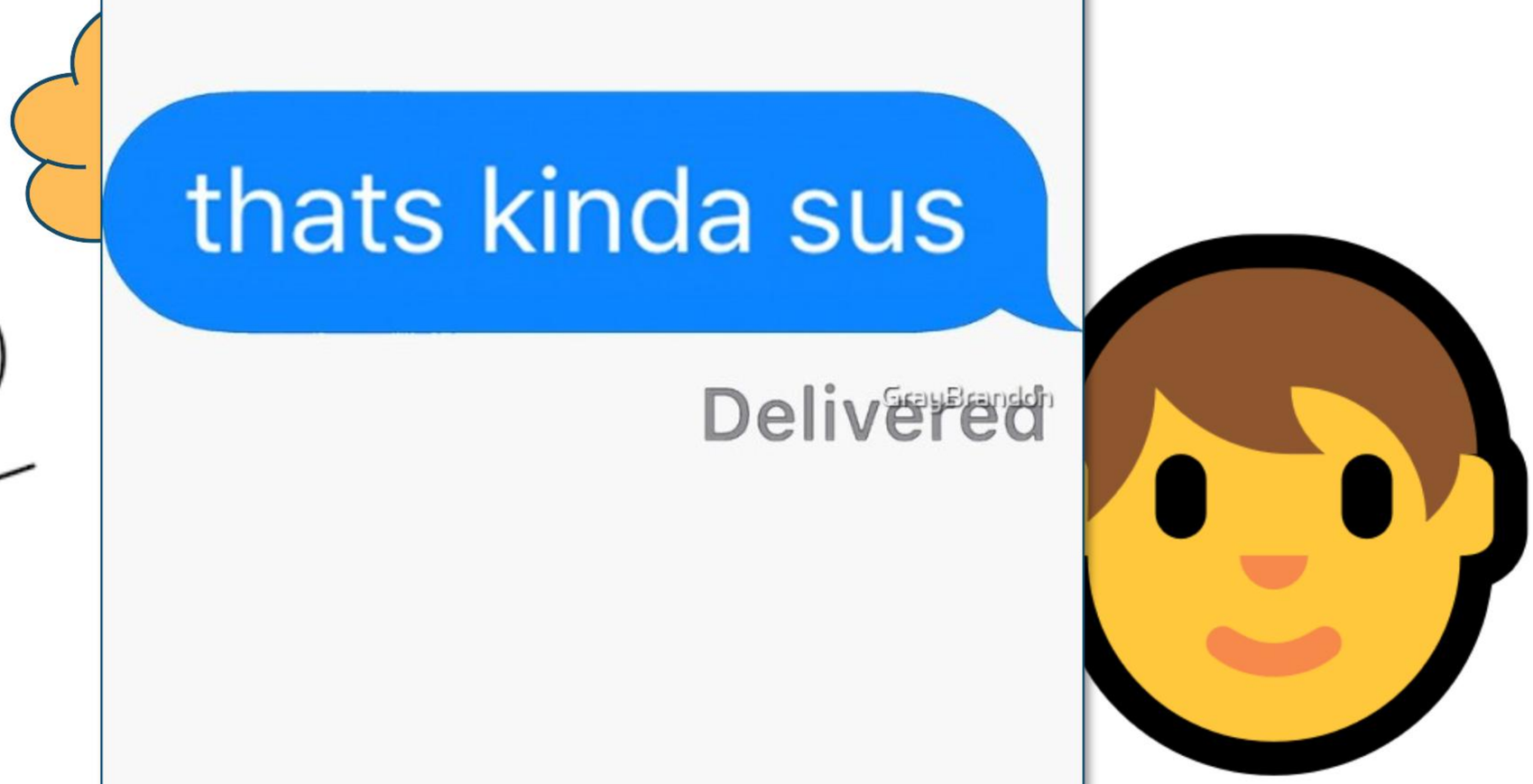
What are templates?

What are templates?

What are templates?

What are tem





What if we could keep the logic, but change the type?

What are templates?

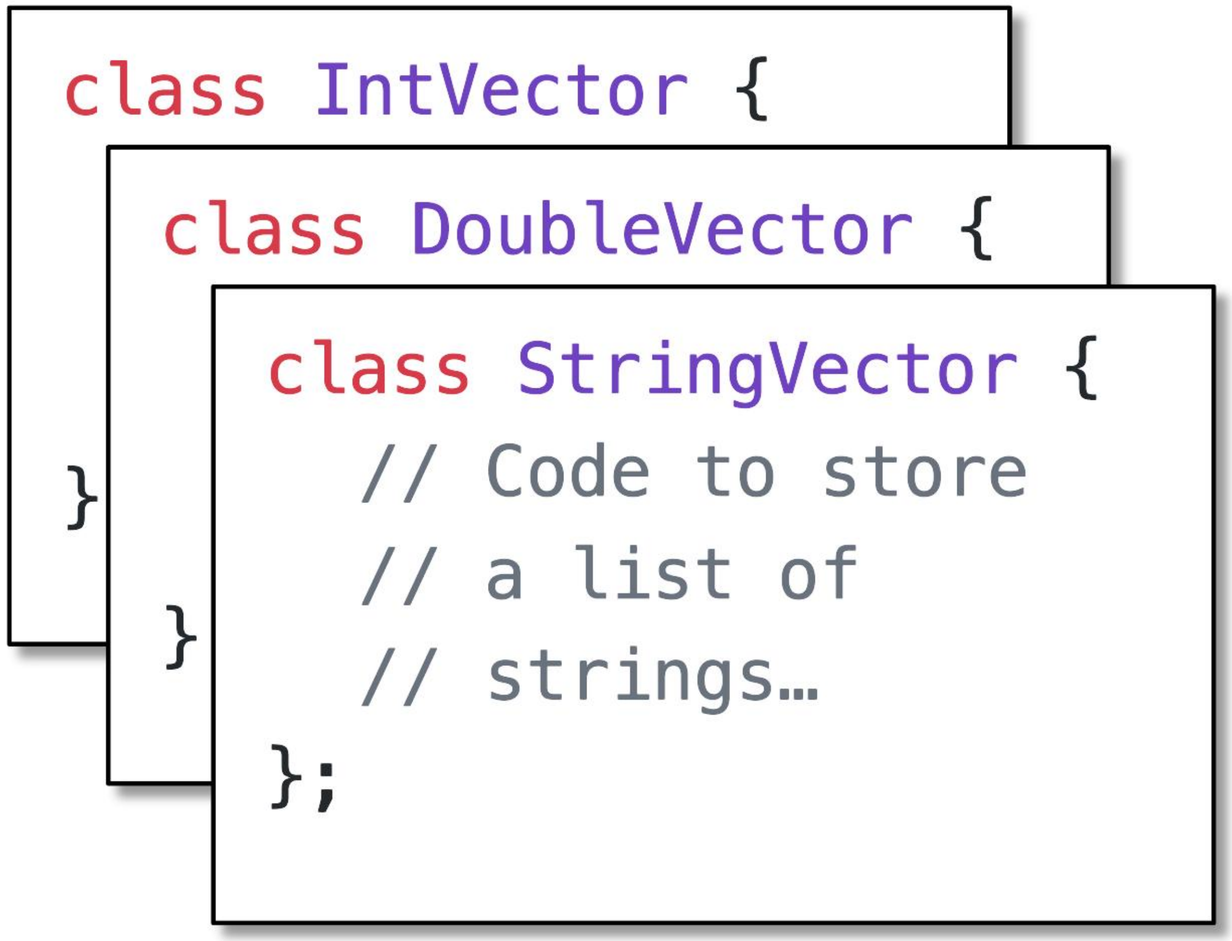
```
class IntVector {
   // Code to store
   // a list of
   // integers…
};
```

What are templates?

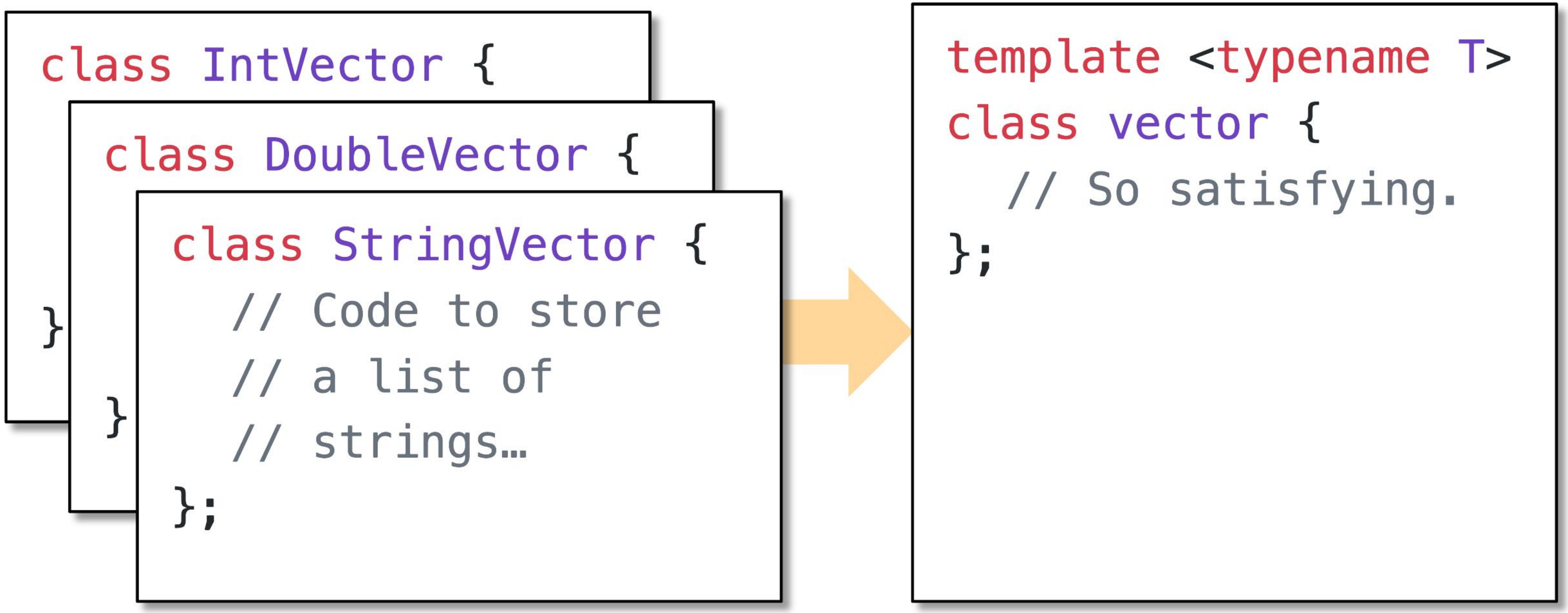
```
class IntVector {

}
```

```
class DoubleVector {
   // Code to store
   // a list of
   // doubles…
};
```

What are templates?



What are templates?

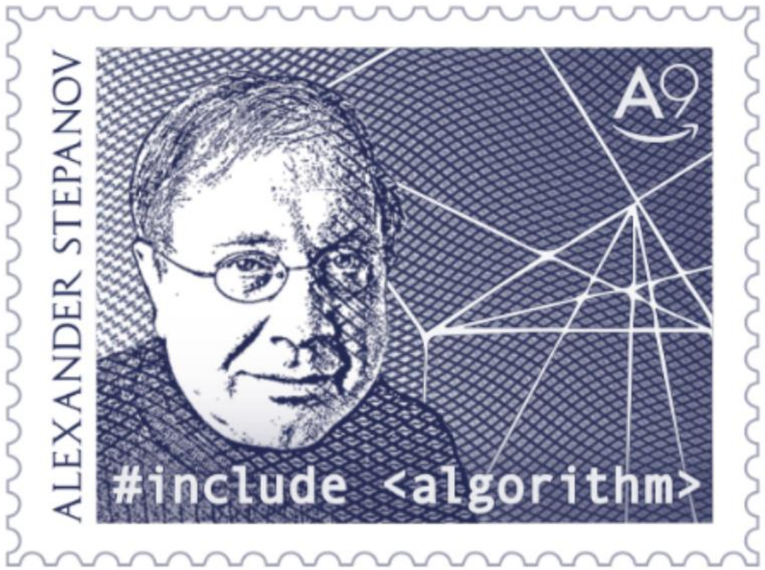


All STL containers are templates!

The Standard Template Library (STL)

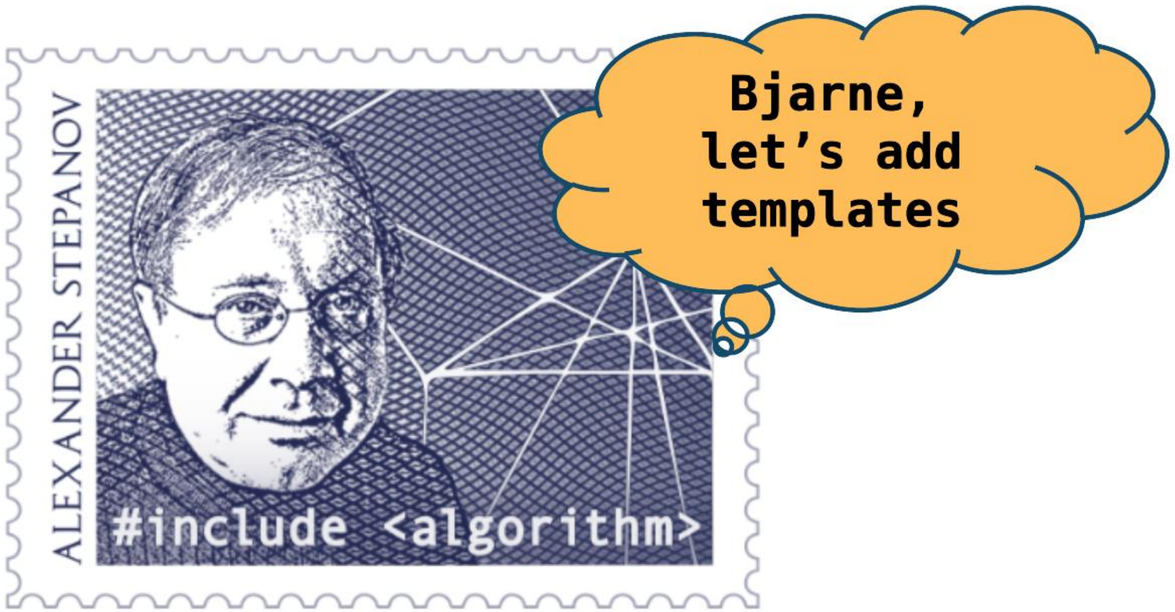
The Standard Template Library (STL)

- Created by Alexander Stepanov



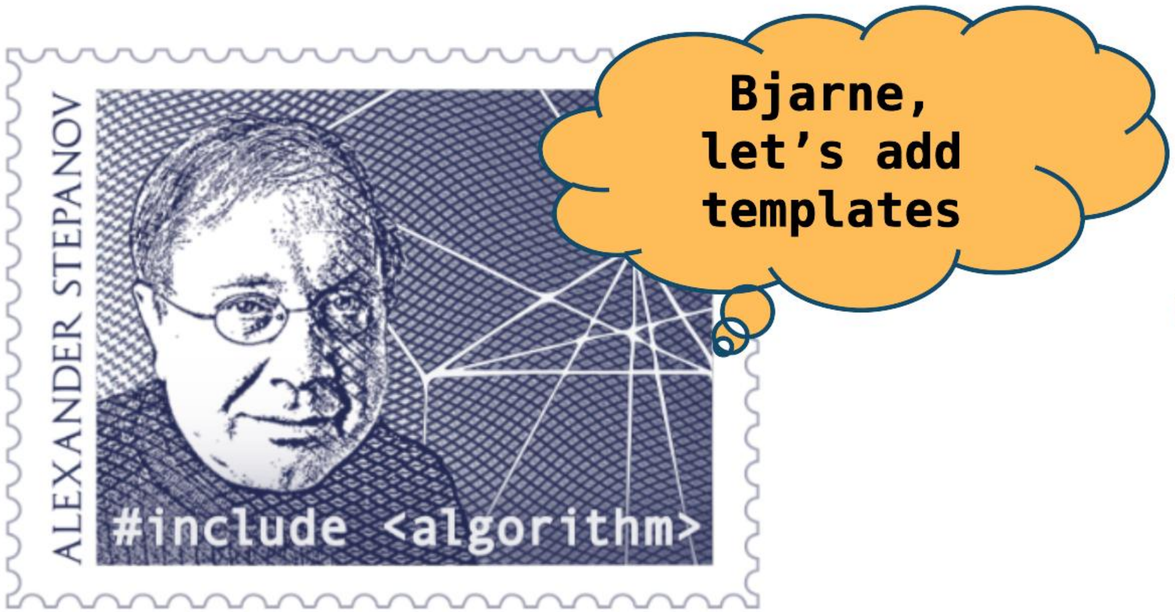
The Standard Template Library (STL)

- Created by Alexander Stepanov
- Added templates to C++ and built a well-known library



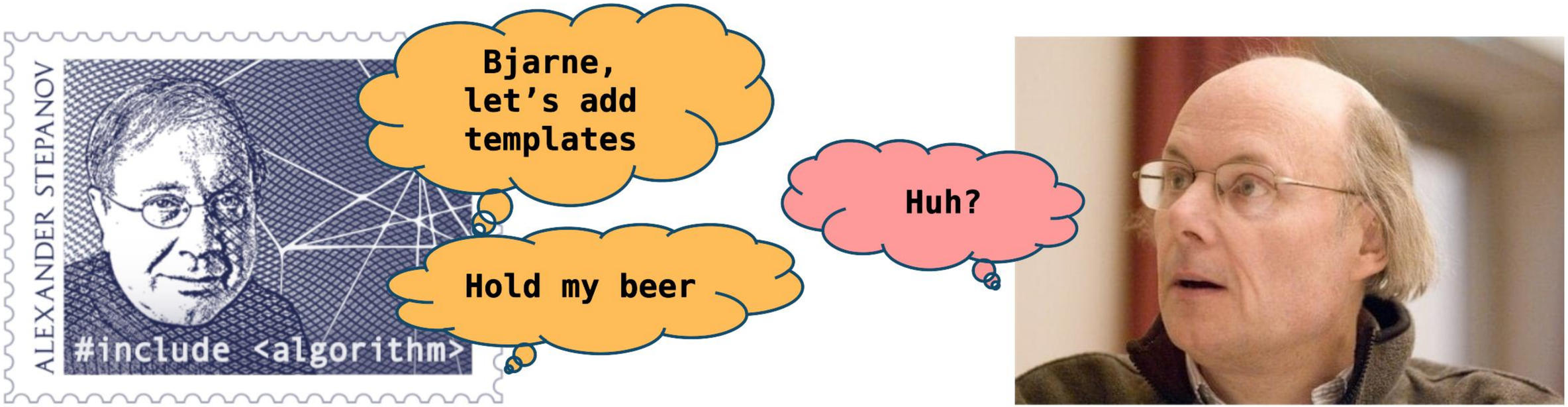
The Standard Template Library (STL)

- Created by Alexander Stepanov
- Added templates to C++ and built a well-known library

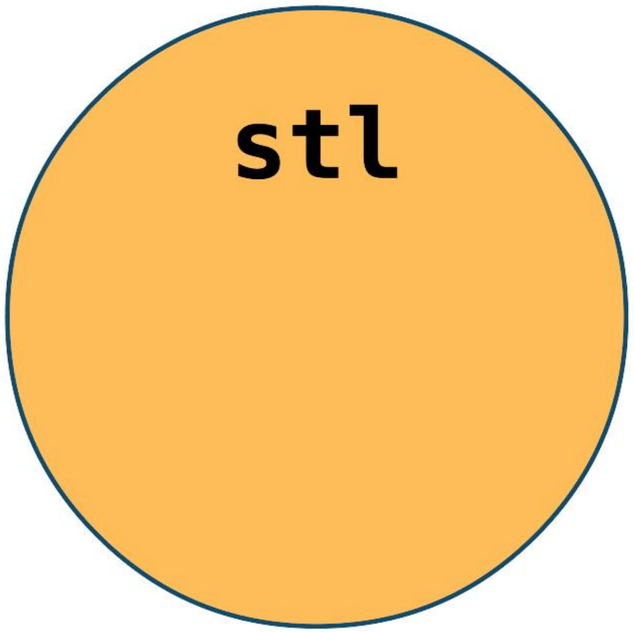


The Standard Template Library (STL)

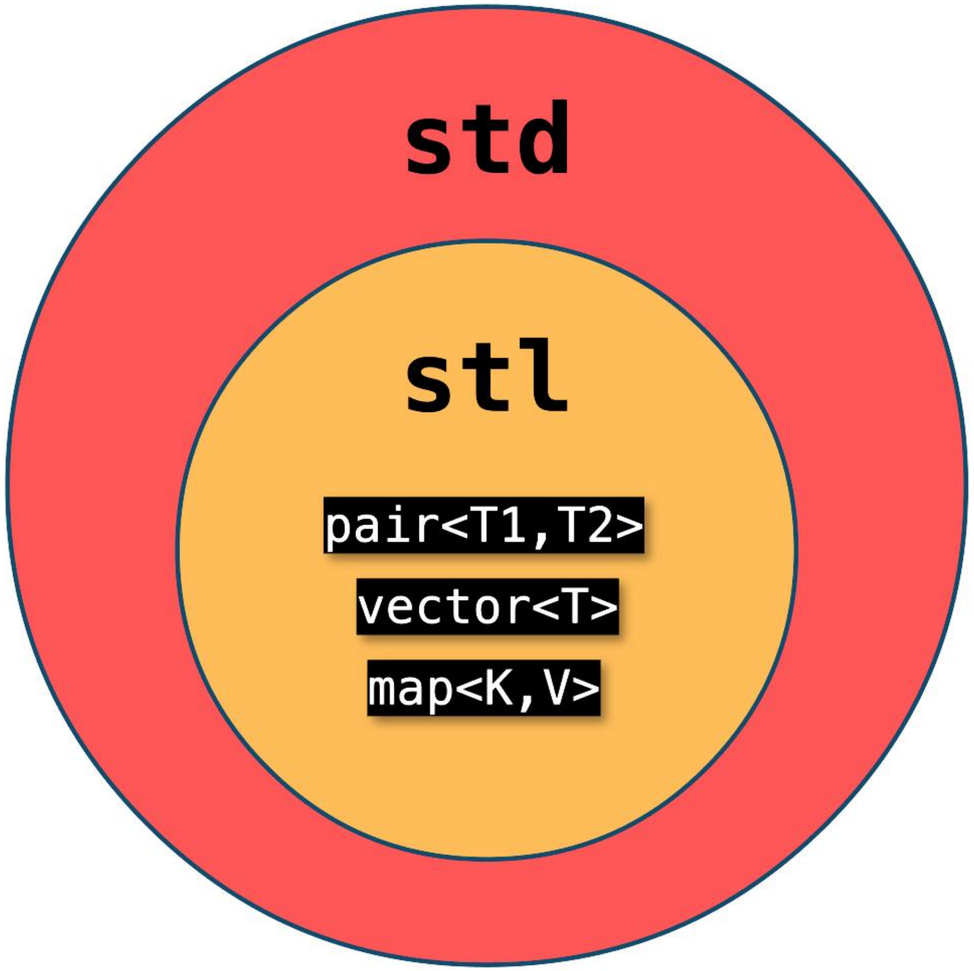
- Created by Alexander Stepanov
- Added templates to C++ and built a well-known library



The Standard Template Library (STL)



The Standard Template Library (STL)



The Standard Template Library (STL)

Containers

How do we store groups of things?

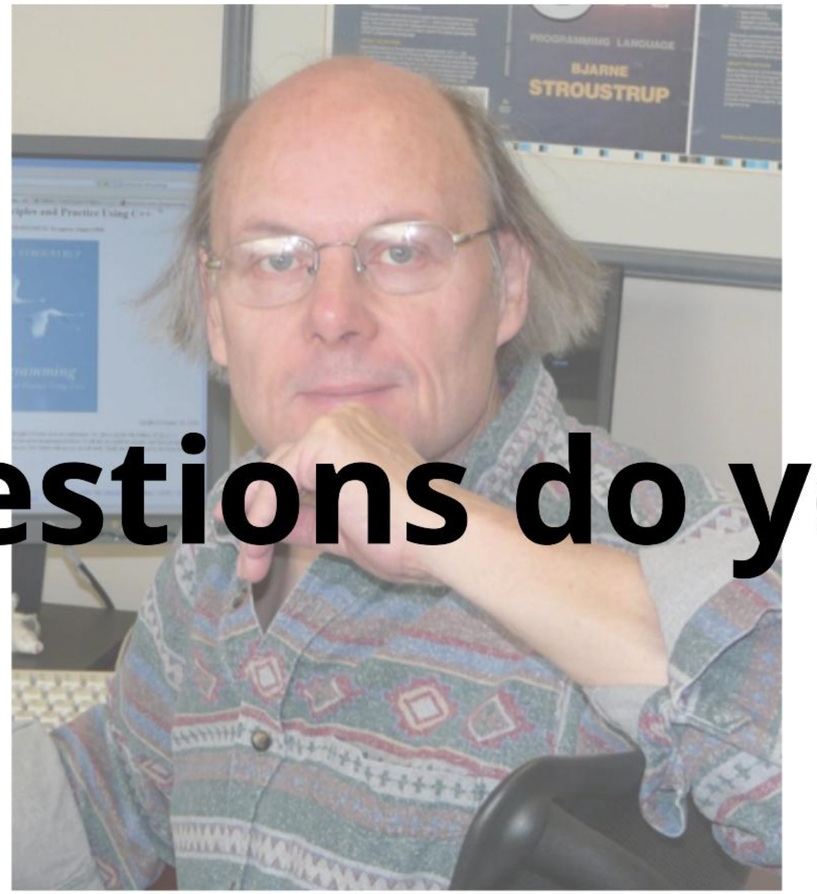
The Standard Template Library (STL)

Containers
How do we store groups of things?

Iterators
How do we traverse containers?

Functors
How can we represent functions as objects?

What questions do you have?



bjarne_about_to_raise_hand

Sequence Containers

Sequence containers store a linear sequence of elements

std::vector

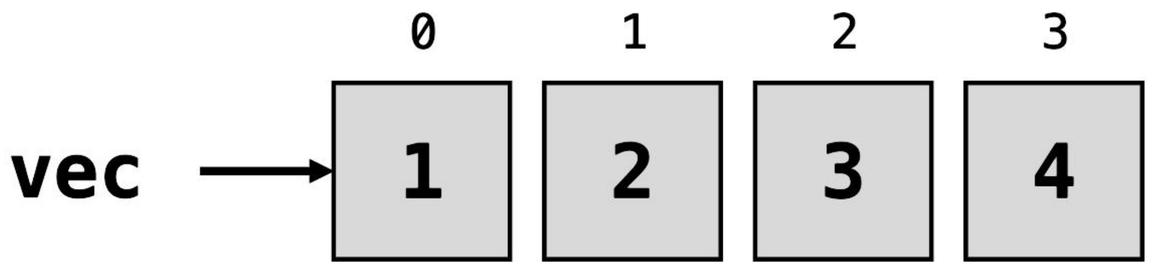
```
#include <vector>
```

std::vector stores a list of elements

```
std::vector<int> vec { 1, 2, 3, 4 };
vec.push_back(5);
vec.push_back(6);
vec[1] = 20;

for (size_t i = 0; i < vec.size(); i++) {
   std::cout << vec[i] << " ";
}
```

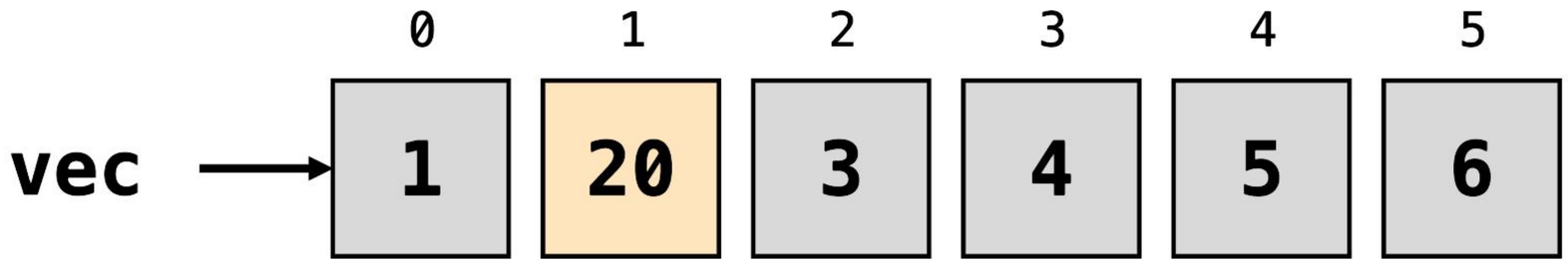
std::vector stores a list of elements

```
std::vector<int> vec { 1, 2, 3, 4 };
vec.push_back(5);
vec.push_back(6);
vec[1] = 20;

for (size_t i = 0; i < vec.size(); i++) {
   std::cout << vec[i] << " ";
}
```



std::vector stores a list of elements

```
std::vector<int> vec { 1, 2, 3, 4 };
vec.push_back(5);
vec.push_back(6);
vec[1] = 20;

for (size_t i = 0; i < vec.size(); i++) {
   std::cout << vec[i] << " ";
}
```



Output:

1 20 3 4 5 6

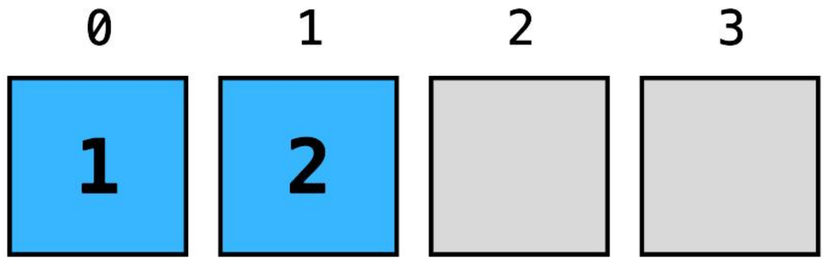
Stanford vs. STL vector

What you want to do?	**Stanford `Vector<int>`**	**`std::vector<int>`**
Create an empty vector	`Vector<int> v;`	`std::vector<int> v;`
Create a vector with **`n`** copies of **`0`**	`Vector<int> v(n);`	`std::vector<int> v(n);`
Create a vector with **`n`** copies of value **`k`**	`Vector<int> v(n, k);`	`std::vector<int> v(n, k);`

Stanford vs. STL vector

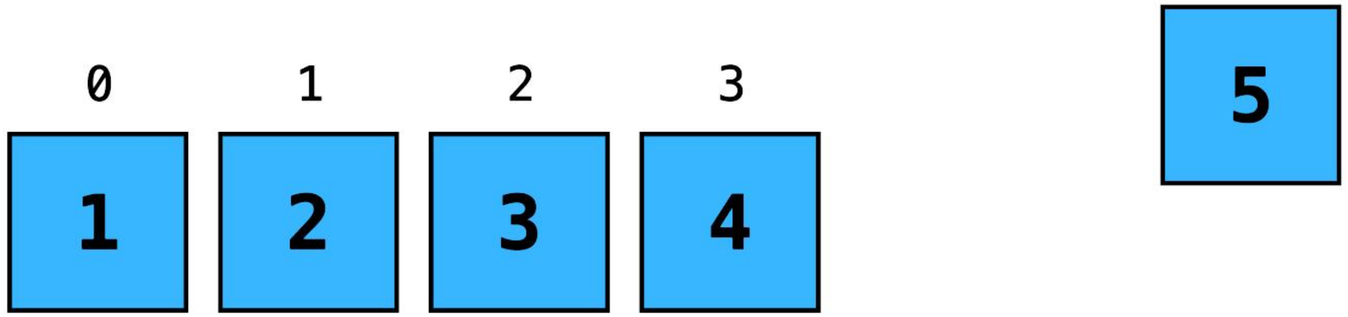
What you want to do?	Stanford `Vector<int>`	`std::vector<int>`
Create an empty vector	`Vector<int> v;`	`std::vector<int> v;`
Create a vector with **n** copies of **0**	`Vector<int> v(n);`	`std::vector<int> v(n);`
Create a vector with **n** copies of value **k**	`Vector<int> v(n, k);`	`std::vector<int> v(n, k);`
Add **k** to the end of the vector	`v.add(k);`	`v.push_back(k);`
Clear vector	`v.clear();`	`v.clear();`
Check if **v** is empty	`if (v.isEmpty())`	`if (v.empty())`
Get the element at index **i**	`int v = v.get(i);` `int k = v[i];`	`int k = v.at(i);` `int k = v[i];`
Replace the element at index **i**	`v.get(i) = k;` `v[i] = k;`	`v.at(i) = k;` `v[i] = k;`

How is vector implemented?



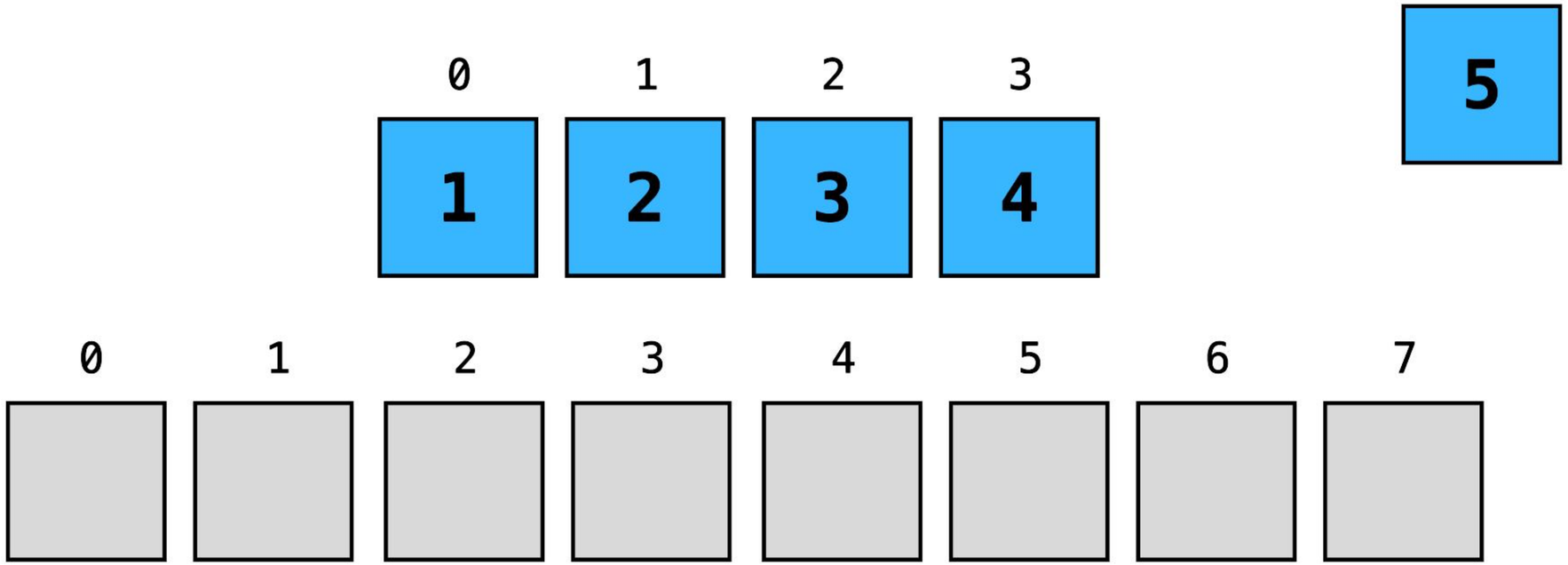
`size = 2, capacity = 4`

How is vector implemented?



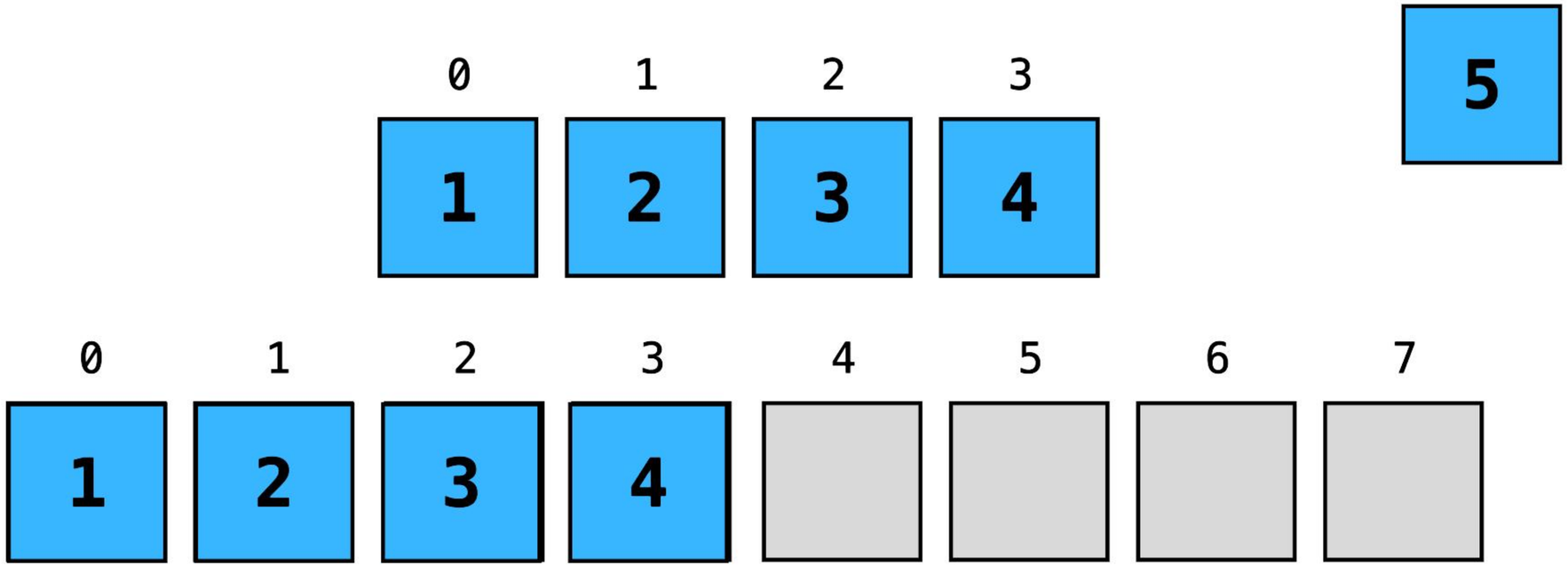
`size = 4, capacity = 4`

How is vector implemented?



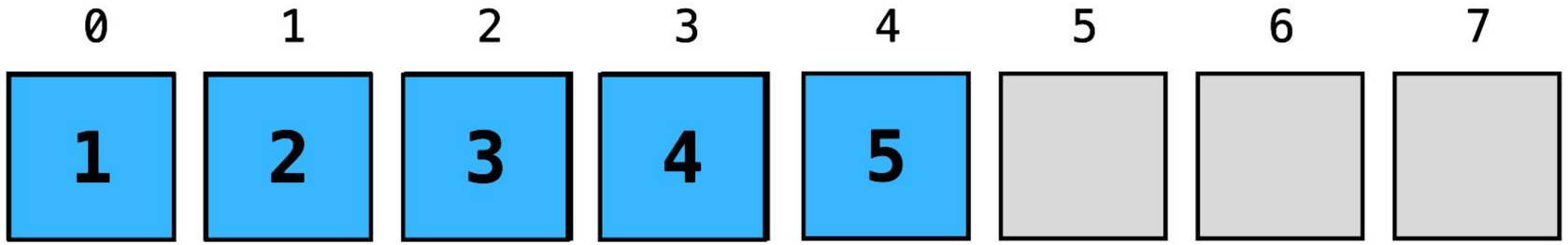
`size = 4, capacity = 4`

How is vector implemented?



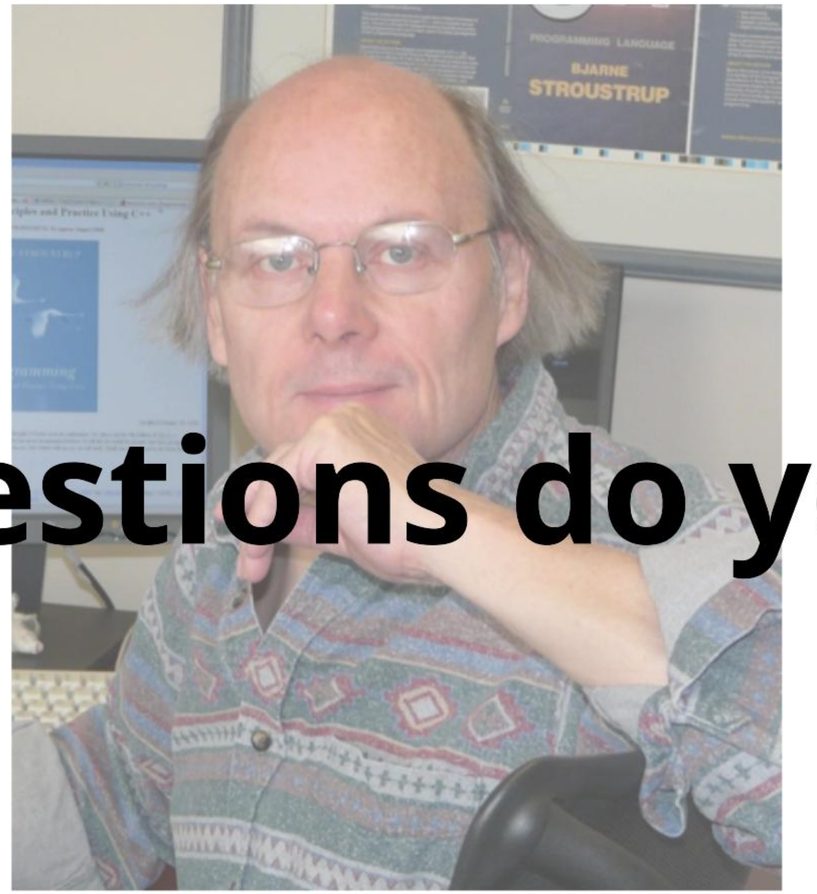
`size = 4, capacity = 4`

How is vector implemented?



`size = 5, capacity = 8`

What questions do you have?



bjarne_about_to_raise_hand

Tip: Use range-based for when possible

```
for (size_t i = 0; i < vec.size(); i++) {
   std::cout << vec[i] << " ";
}
```

Tip: Use range-based for when possible

```
for (size_t i = 0; i < vec.size(); i++) {
   std::cout << vec[i] << " ";
}
```

```
for (auto elem : vec) {
   std::cout << elem << " ";
}
```

Tip: Use `const auto&` when possible

```
std::vector<MassiveType> vec { ... };
for (auto elem : vec) ...
```

Tip: Use `const auto&` when possible

```
std::vector<MassiveType> vec { ... };
for (auto elem : vec) ...
```

```
for (const auto& elem : v)
```

- Applies for all iterable containers, not just `std::vector`
- Saves making a potentially expensive copy of each element

Stanford vs. STL vector

What you want to do?	Stanford `Vector<int>`	`std::vector<int>`
Create an empty vector	`Vector<int> v;`	`std::vector<int> v;`
Create a vector with `n` copies of `0`	`Vector<int> v(n);`	`std::vector<int> v(n);`
Create a vector with `n` copies of value `k`	`Vector<int> v(n, k);`	`std::vector<int> v(n, k);`
Add `k` to the end of the vector	`v.add(k);`	`v.push_back(k);`
Clear vector	`v.clear();`	`v.clear();`
Check if `v` is empty	`if (v.isEmpty())`	`if (v.empty())`
Get the element at index `i`	`int v = v.get(i);` `int k = v[i];`	`int k = v.at(i);` `int k = v[i];`
Replace the element at index `i`	`v.get(i) = k;` `v[i] = k;`	`v.at(i) = k;` `v[i] = k;`

`operator[]` does not perform bounds checking

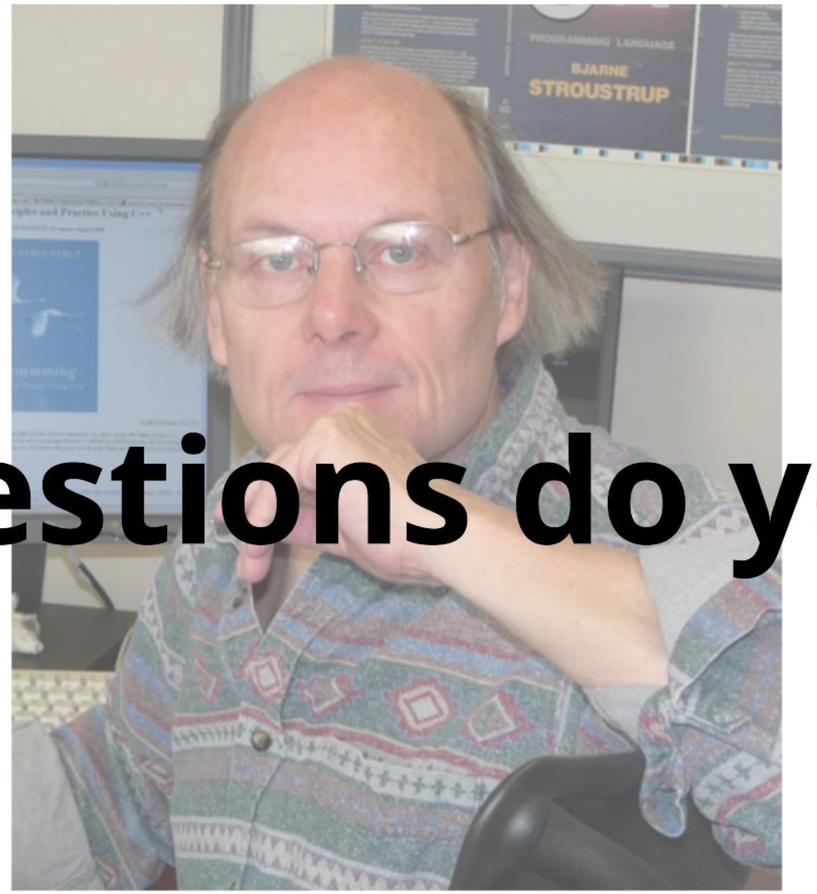
Zero-overhead principle

The *zero-overhead principle* is a C++ design principle that states:

1. You don't pay for what you don't use.
2. What you do use is just as efficient as what you could reasonably write by hand.

[cppreference]

What questions do you have?



bjarne_about_to_raise_hand

`std::vector` is not the best for all cases…

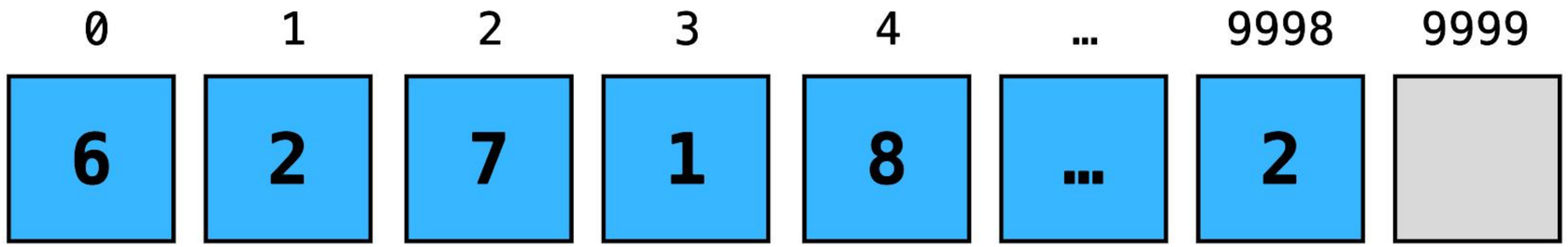
`std::vector` is not the best for all cases…

- Suppose we need to observe the last 10,000 prices of a stock
- What might be concerning about the code below?

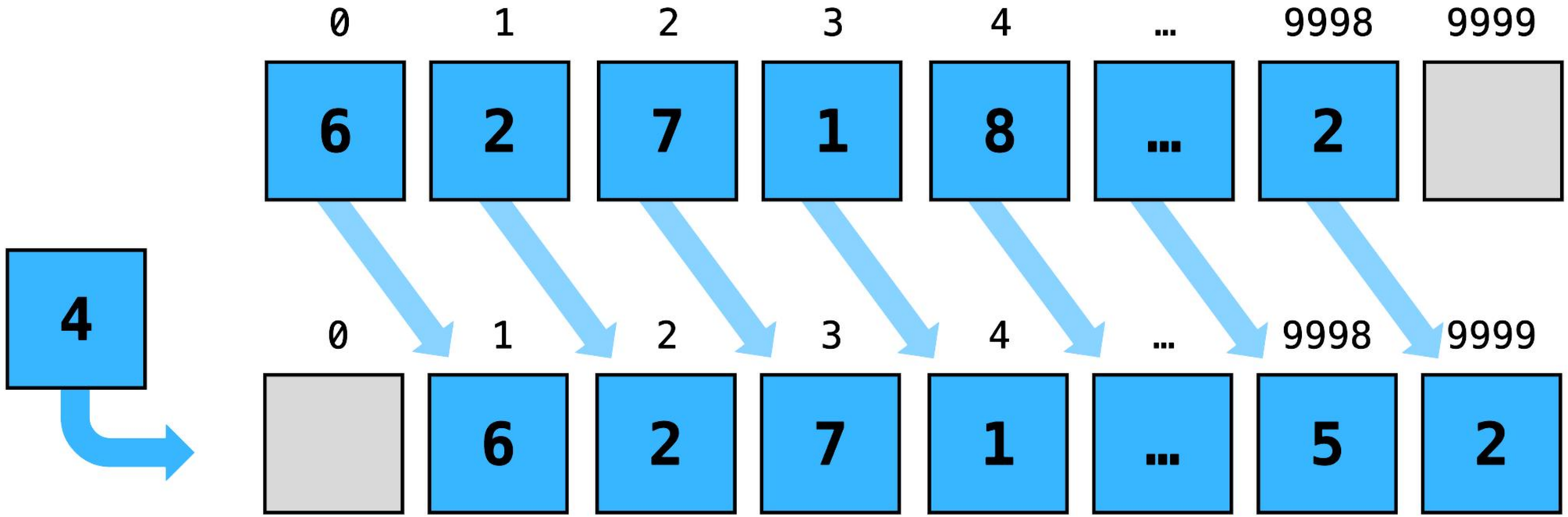
```
void receivePrice(vector<double>& prices, double price)
{
   prices.push_front(price);
   if (prices.size() > 10000)
      prices.pop_back();   // Remove last price
                           // so we don't exceed 10k
}
```


It's a
Trap!

A hypothetical `push_front`...



A hypothetical `push_front`...



std::deque

```
#include <deque>
```

`std::deque`

- A deque (“deck”) is a double-ended queue

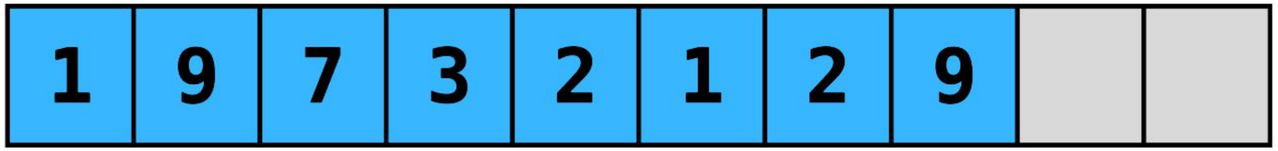
std::deque

- A deque (“deck”) is a double-ended queue
- Allows efficient insertion/removal at either end

```
void receivePrice(deque<double>& prices, double price)
{
   prices.push_front(price);   // Super fast
   if (prices.size() > 10000)
      prices.pop_back();       // Remove last price
                               // so we don't exceed 10k
}
```

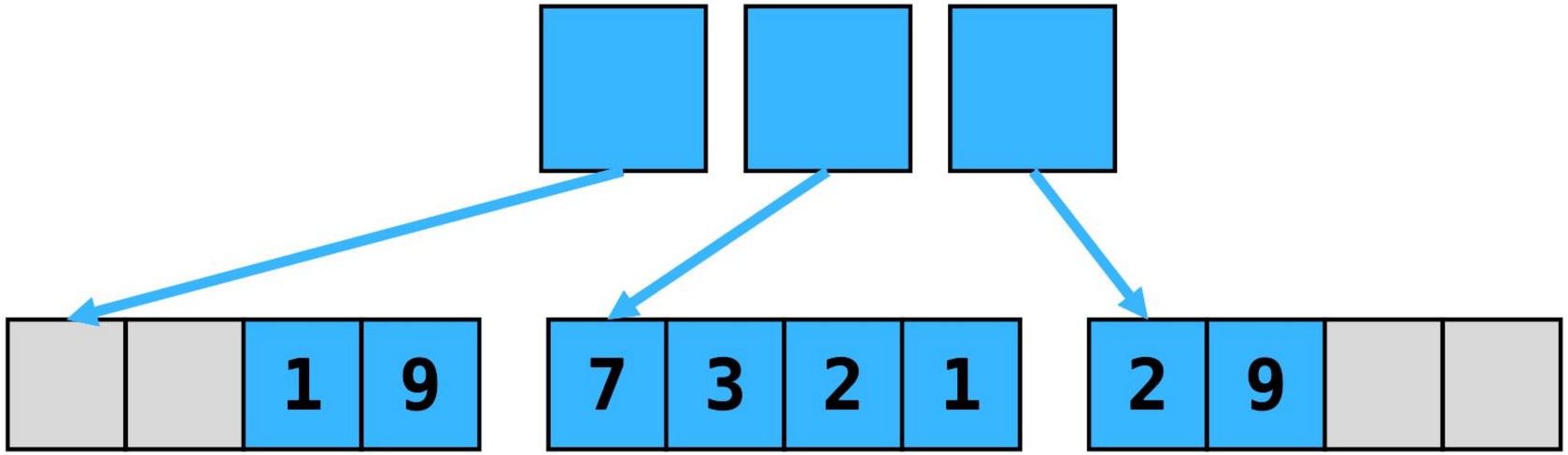
A `deque` has the same interface as `vector`,
except we can `push_front` / `pop_front`

How is deque implemented?



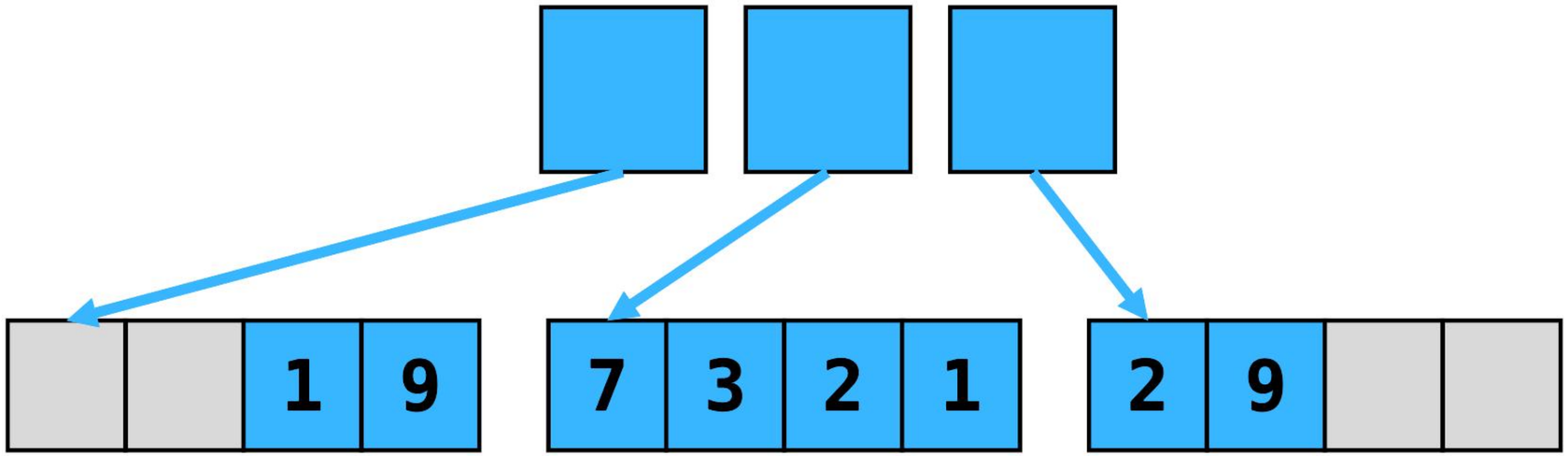
The problem with **vector** is that we have a single chunk of memory

How is deque implemented?



How is deque implemented?

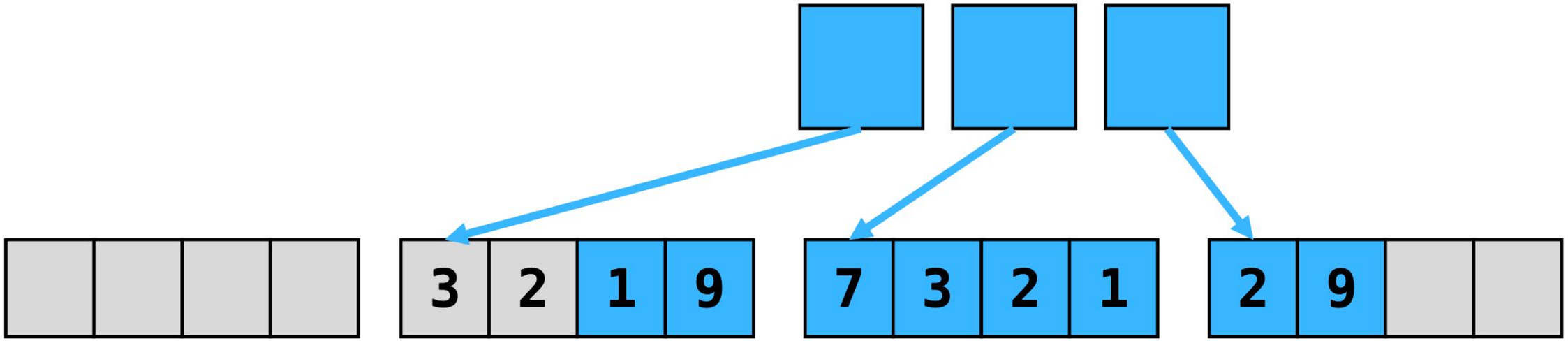
Array of arrays



Separate subarrays
allocated independently

How is deque implemented?

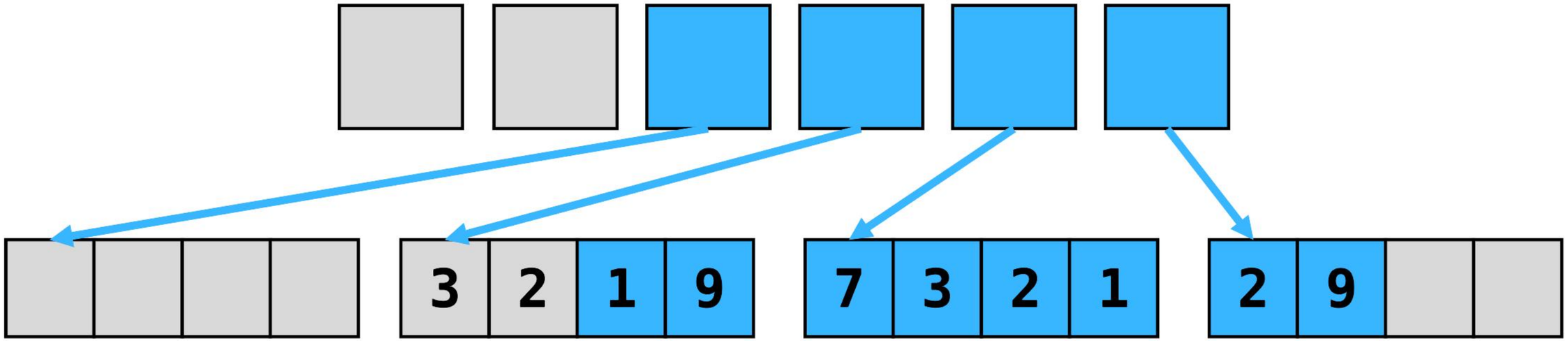
Array of arrays



Separate subarrays
allocated independently

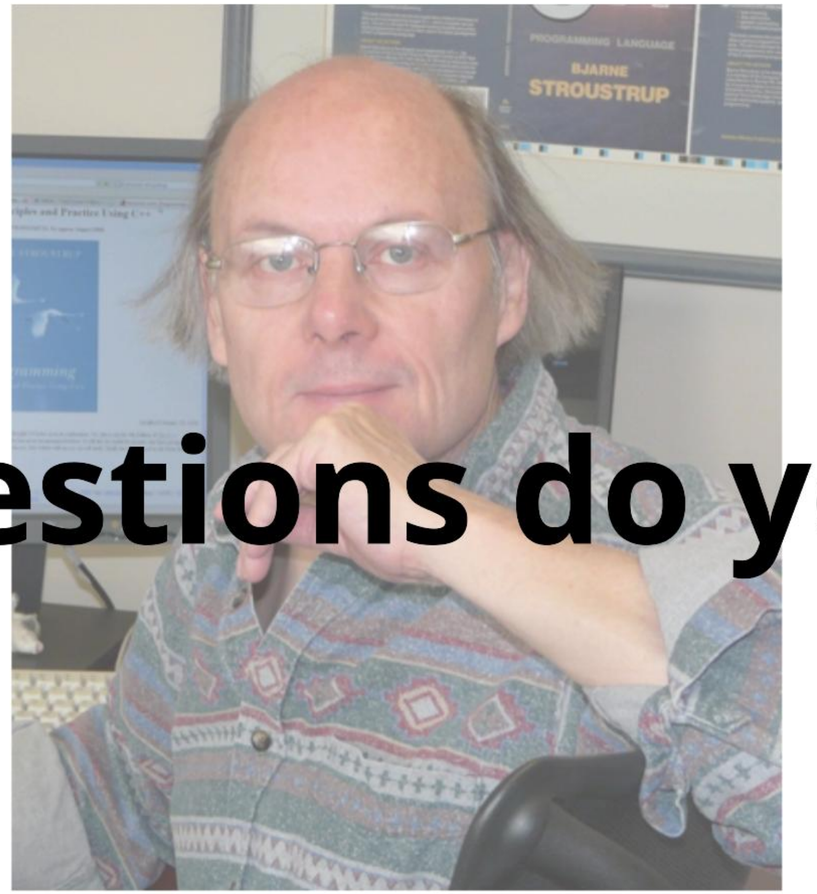
How is deque implemented?

Array of arrays



Separate subarrays
allocated independently

What questions do you have?



bjarne_about_to_raise_hand

Announcements

Announcements

- Assignment 1 due Friday!
- A0 grades are out!
  - Let us know if anything looks fishy
  - Same goes for attendance grades!
- Watch for OH announcements

Associative Containers

Associative containers organize elements by unique keys

std::map

```
#include <map>
```

`std::map` maps keys to values

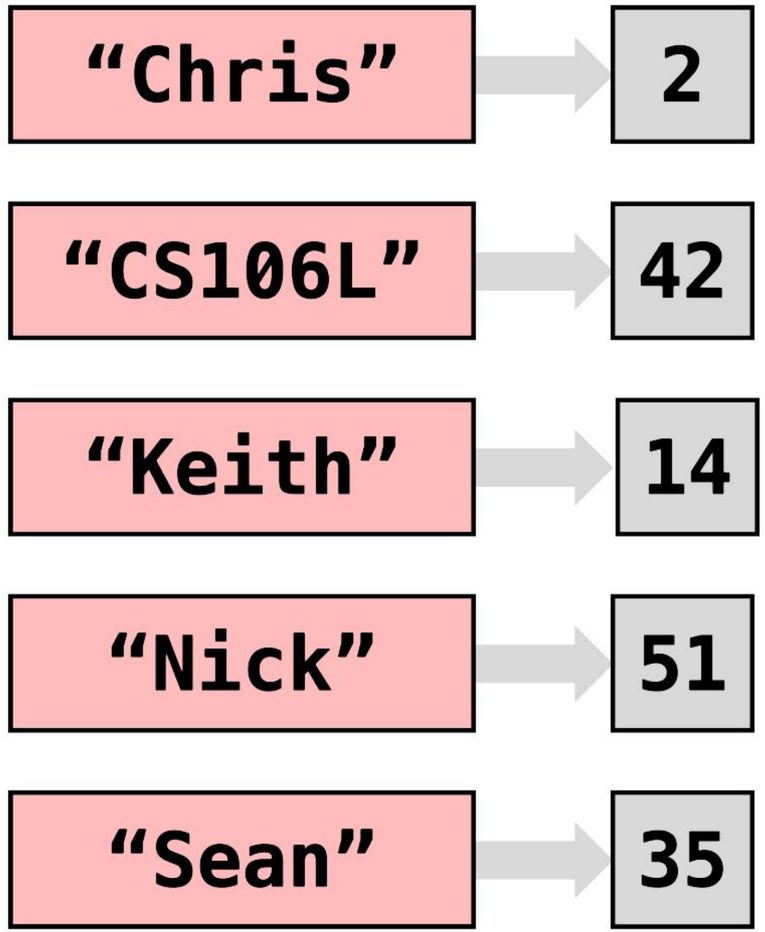
```
std::map<std::string, int>
```

`std::map` maps keys to values

```
std::map<std::string, int> map {




};
```

std::map maps keys to values

```
std::map<std::string, int> map {
   { "Chris", 2 },
   { "CS106L", 42 },
   { "Keith", 14 },
   { "Nick", 51 },
   { "Sean", 35 },
};
```



Stanford vs. STL map

What you want to do?	Stanford `Map<char, int>`	`std::map<char, int>`
Create an empty map	`Map<char, int> m;`	`std::map<char, int> m;`

Stanford vs. STL map

What you want to do?	Stanford Map<char, int>	std::map<char, int>
Create an empty map	`Map<char, int> m;`	`std::map<char, int> m;`
Add key **k** with value **v** into the map	`m.put(k, v);` `m[k] = v;`	`m.insert({k, v});` `m[k] = v;`
Remove key **k** from the map	`m.remove(k);`	`m.erase(k);`
Check if **k** is in the map ***(* C++20)***	`if (m.containsKey(k))`	`if (m.count(k))` `if (m.contains(k))` ***(*)***
Check if the map is empty	`if (m.isEmpty())`	`if (m.empty())`

```
std::map<K, V>
```

stores a collection of

```
std::pair<const K, V>
```

(I encourage you to think about why K is const.
What would happen if we could modify a key?)

`map` as a collection of `pair`

We can iterate through the key-value pairs using a range based for loop

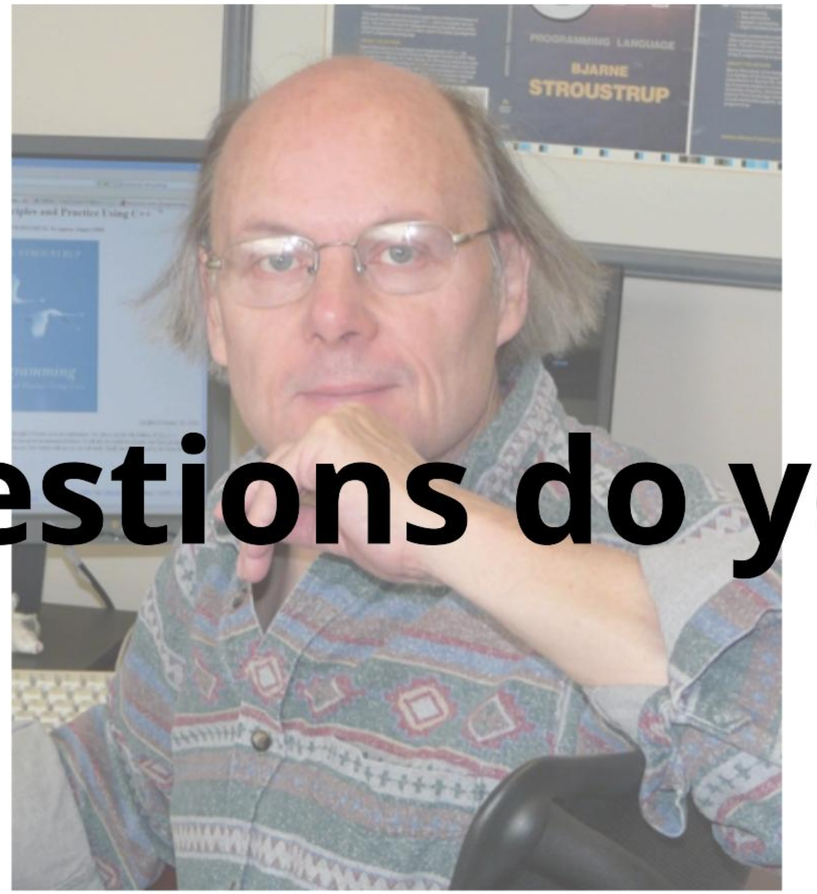
```
std::map<std::string, int> map;

for (auto kv : map) {
   // kv is a std::pair<const std::string, int>
   std::string key = kv.first;
   int value = kv.second;
}
```

`map` as a collection of `pair`

Structured bindings come in handy when iterating a map

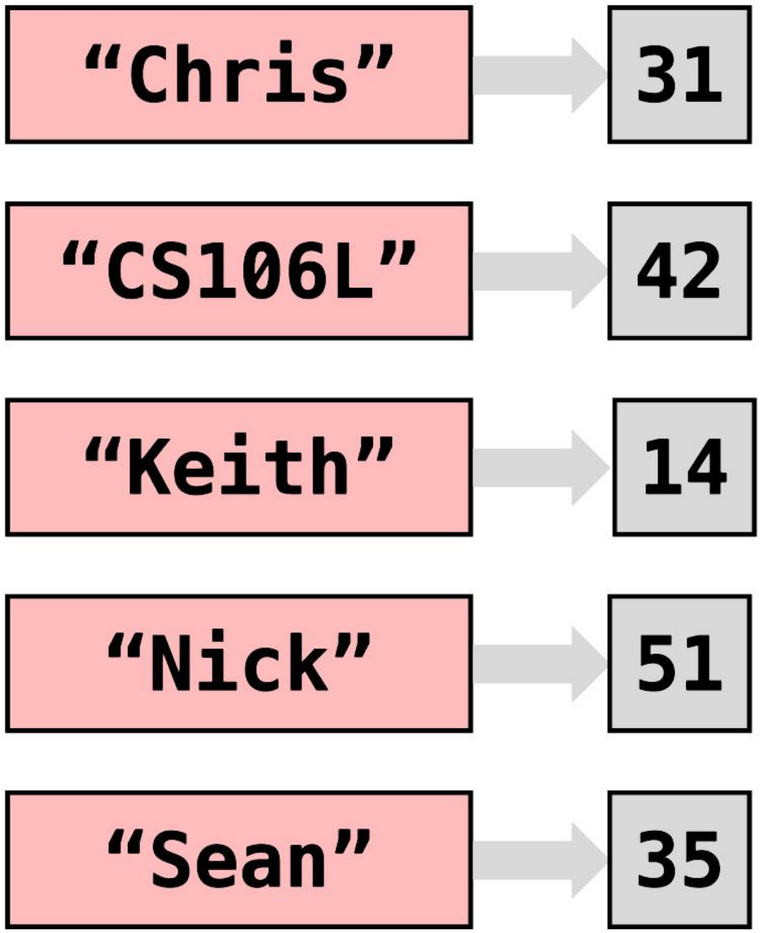
```
std::map<std::string, int> map;

for (const auto& [key, value] : map) {
   // key has type const std::string&
   // value has type const int&
}
```

What questions do you have?

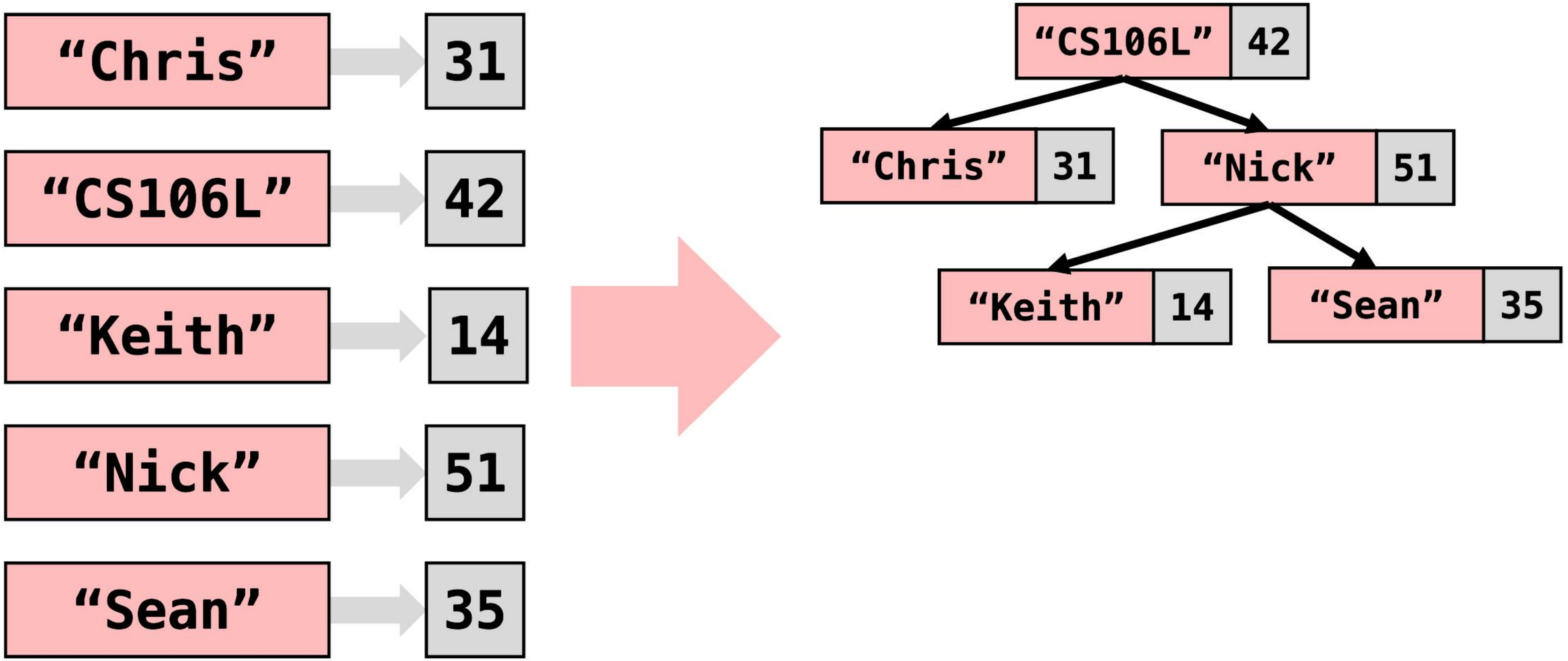


bjarne_about_to_raise_hand

How is `map` implemented?



How is `map` implemented?



What is `map["Keith"]`?