


**Welcome to Week 3! Link to Attendance Form ↓**

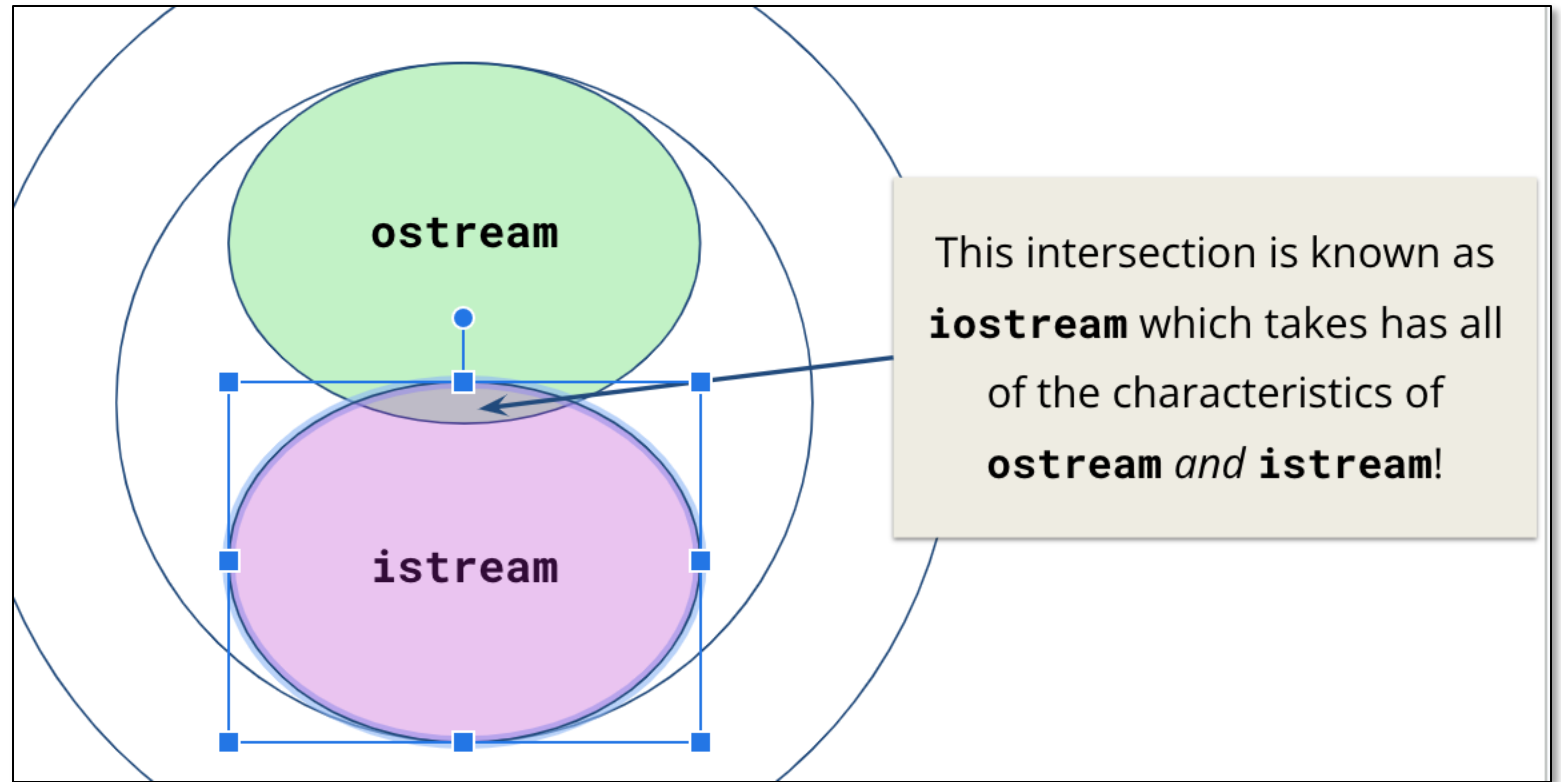


# A `stringstream` is an...

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

# A stringstream is an...

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







# A Disorganized Garage







# An Organized Garage

Item	Box
Tools	B1
DVDs	B2
Books	B3
Snacks	B4



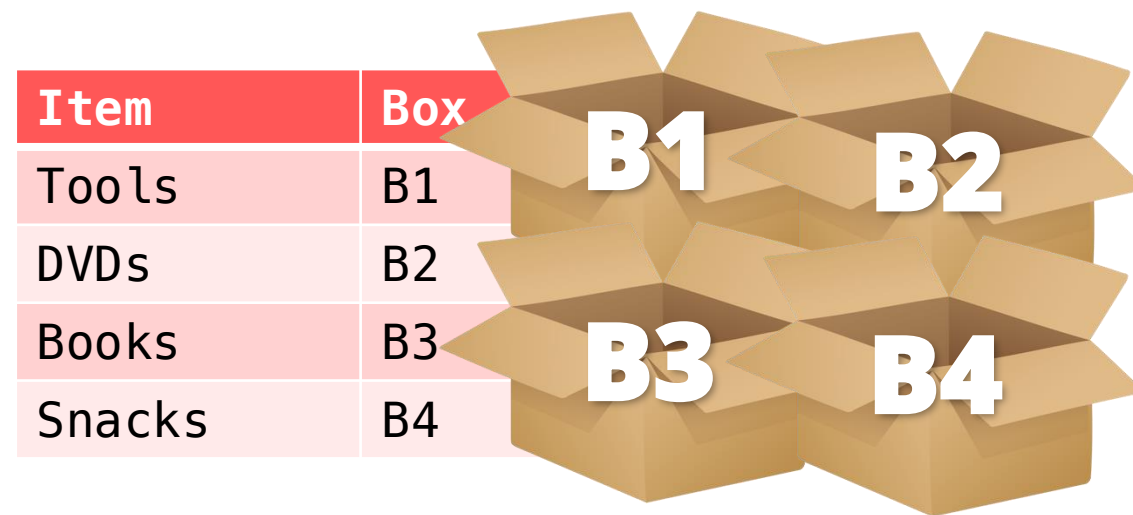


# Disorganized



- Space efficient
- Slow to lookup item
- `std::vector<T>`

# Organized

**B1****B2****B3****B4**

- Space inefficient
- Quick to lookup item
- `std::map<std::string, T>`



# Lecture 5: Containers

Stanford CS106L, Fall 2025

# Announcements

- Assignment 0 due Friday!
  - Let us know if anything looks fishy
  - (Same goes for attendance/assignment grades!)
- Assignment 1 will be released this Friday and due next Friday
  - (Trend: 1 week per assignment)

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



(But we'll still make references to them)



# The many containers of C++

## The C++ Standard Template Library (STL)

`std::vector`

`std::set`

`std::stack`

`std::queue`

`std::map`

`std::unordered_map`

`std::unordered_set`

`std::priority_queue`

`std::deque`

`std::array`

**Which container do I use?**



# The space-time tradeoff





**“Space is time”**  
— Bjarne Stroustrup

[source]

# 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?



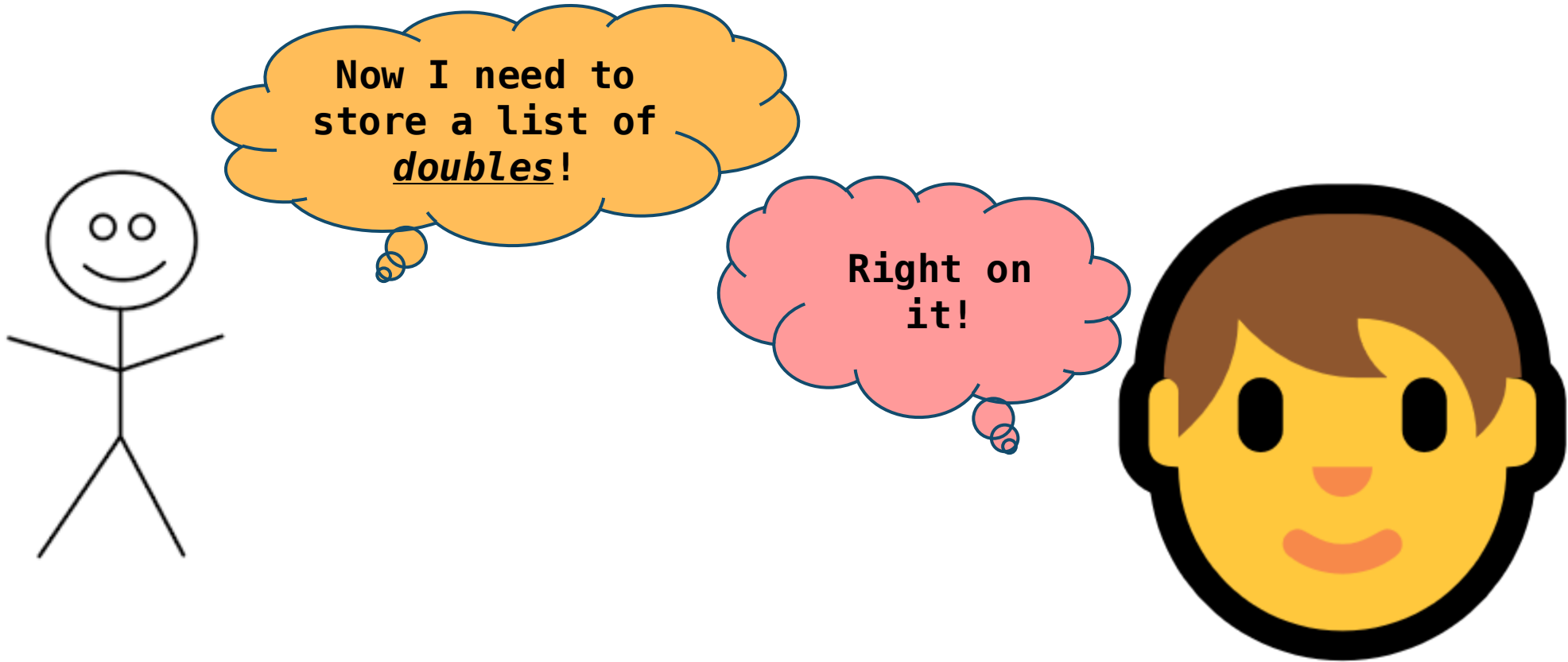


# What are templates?



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

# What are templates?



# What are templates?



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

What are tem



thats kinda sus

Delivered

Gray Brandon

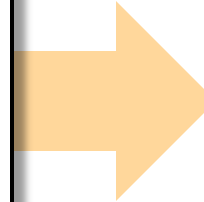


**What if we could keep the logic, but change the type?**



# What are templates?

```
class IntVector {  
    // ...  
}  
  
class DoubleVector {  
    // ...  
}  
  
class StringVector {  
    // Code to store  
    // a list of  
    // strings...  
};
```

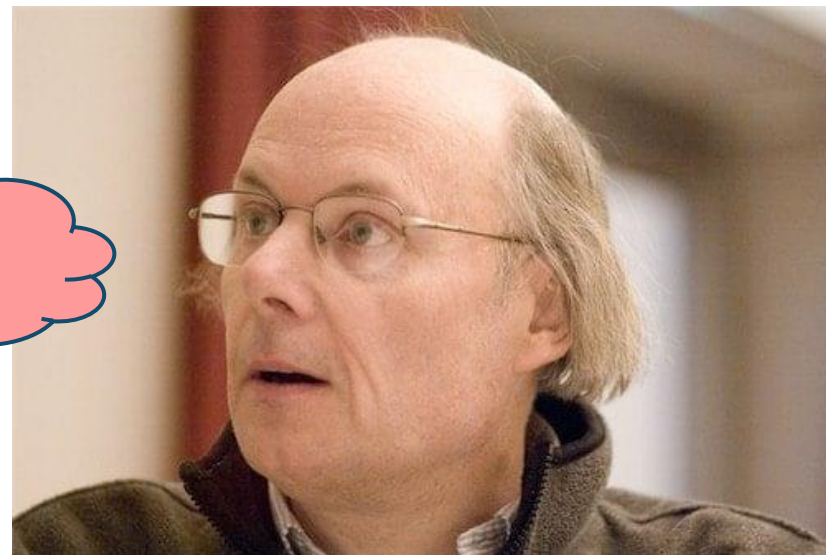
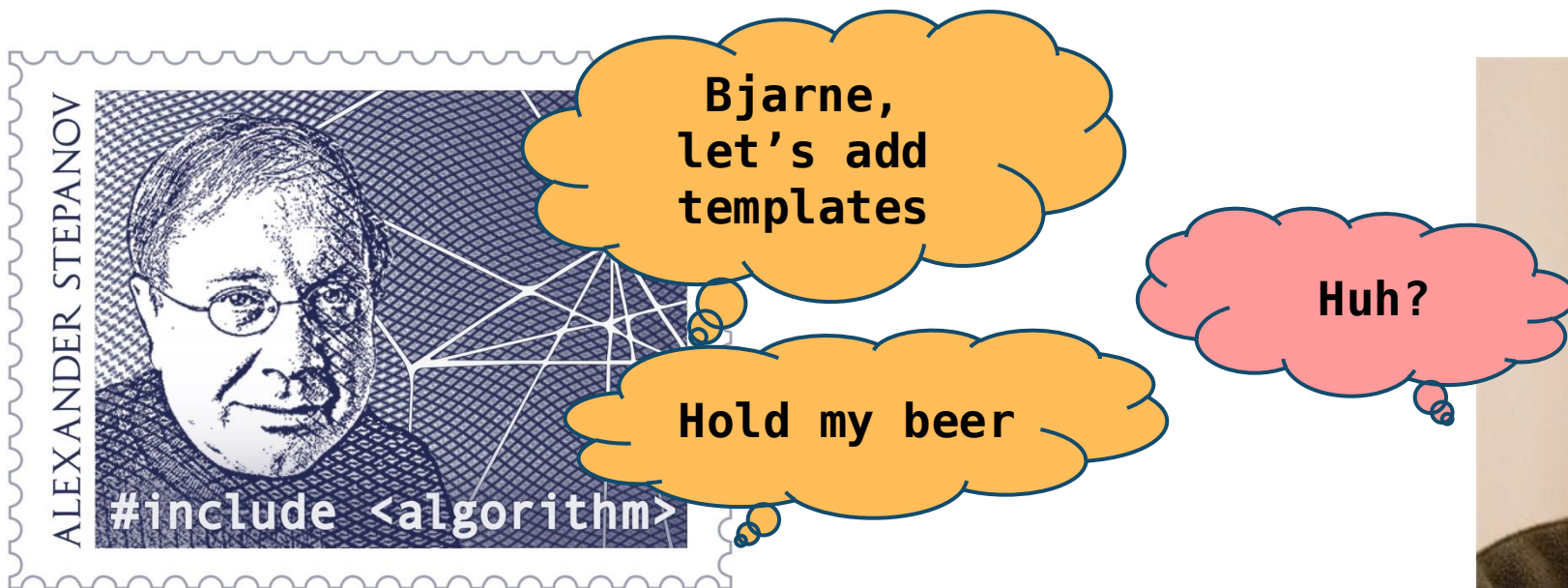


```
template <typename T>  
class vector {  
    // So satisfying.  
};  
  
vector<int> v1;  
vector<double> v2;  
vector<string> v3;
```

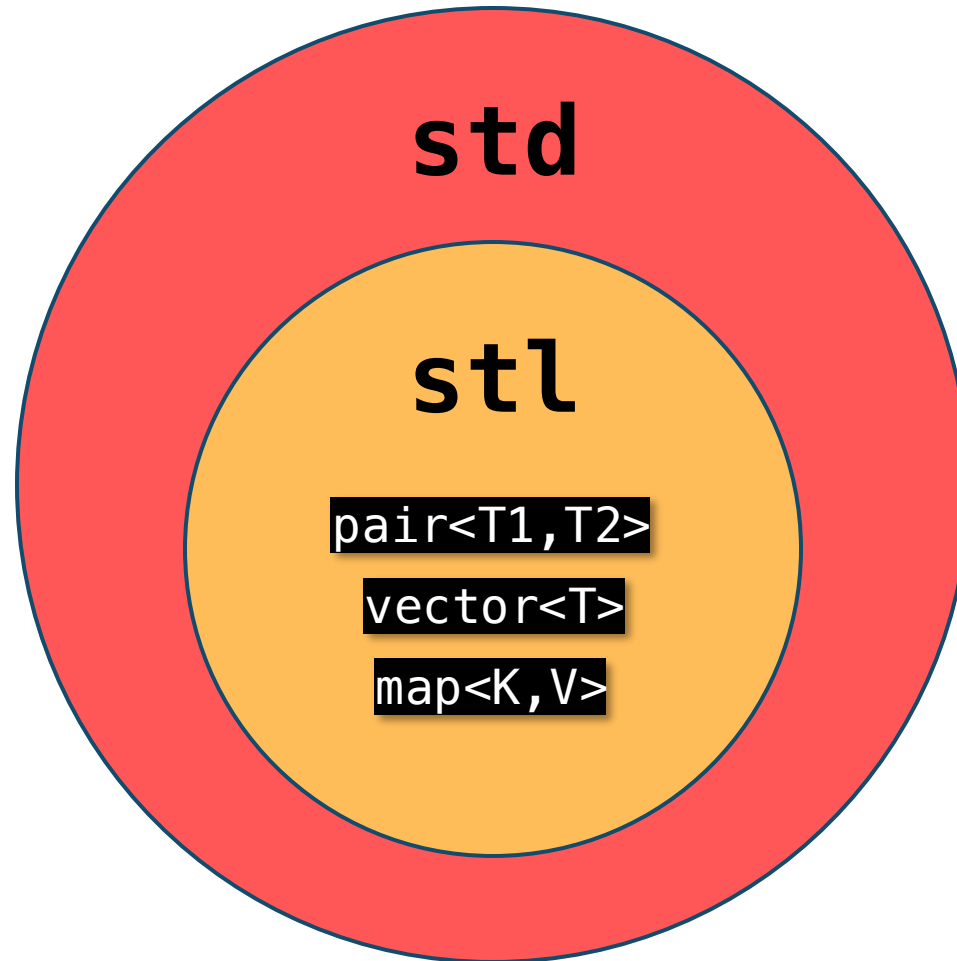
**All STL containers are templates!**

# The Standard Template Library (STL)

- Created by Alexander Stepanov
- Added templates to C++ and built a well-known library
- This library is now known as the **STL**!



# The Standard Template Library (STL)



# 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?*

## Algorithms

*How do we transform and modify containers in a generic way?*



# What questions do you have?



bjarne\_about\_to\_raise\_hand

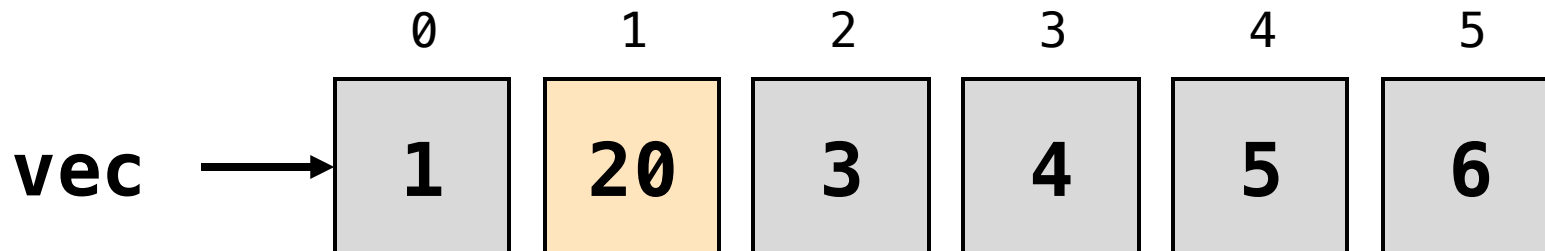
# Sequence Containers

**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] << " ";  
}
```



**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	<code>Vector&lt;int&gt; v;</code>	<code>std::vector&lt;int&gt; v;</code>
Create a vector with <b>n</b> copies of <b>0</b>	<code>Vector&lt;int&gt; v(n);</code>	<code>std::vector&lt;int&gt; v(n);</code>
Create a vector with <b>n</b> copies of value <b>k</b>	<code>Vector&lt;int&gt; v(n, k);</code>	<code>std::vector&lt;int&gt; v(n, k);</code>

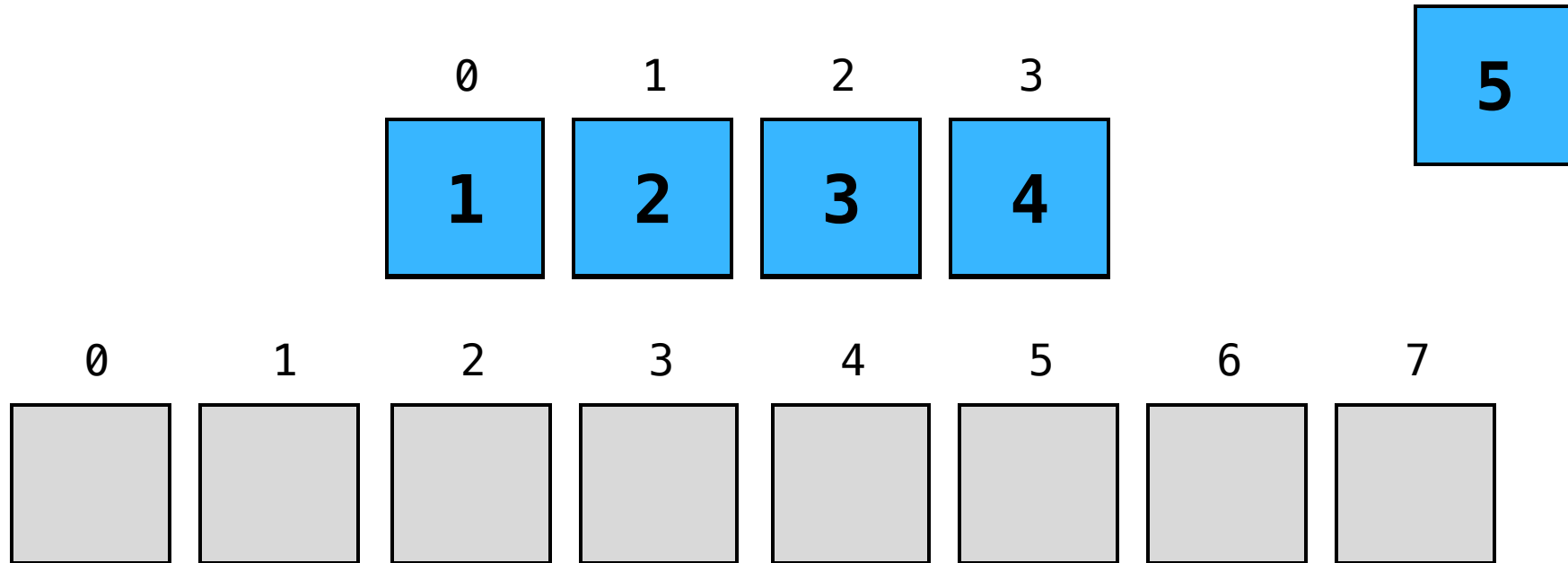
# Stanford vs. STL vector

What you want to do?	Stanford Vector<int>	std::vector<int>
Create an empty vector	<code>Vector&lt;int&gt; v;</code>	<code>std::vector&lt;int&gt; v;</code>
Create a vector with <b>n</b> copies of <b>0</b>	<code>Vector&lt;int&gt; v(n);</code>	<code>std::vector&lt;int&gt; v(n);</code>
Create a vector with <b>n</b> copies of value <b>k</b>	<code>Vector&lt;int&gt; v(n, k);</code>	<code>std::vector&lt;int&gt; v(n, k);</code>
Add <b>k</b> to the end of the vector	<code>v.add(k);</code>	<code>v.push_back(k);</code>
Clear vector	<code>v.clear();</code>	<code>v.clear();</code>
Check if <b>v</b> is empty	<code>if (v.isEmpty())</code>	<code>if (v.empty())</code>

# Stanford vs. STL vector

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

# How is vector implemented?



**size** = 5, **capacity** = 8

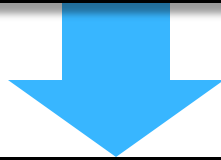
# What questions do you have?



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] << " ";  
}
```



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

- Applies for all iterable containers, not just `std::vector`



## 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	<code>Vector&lt;int&gt; v;</code>	<code>std::vector&lt;int&gt; v;</code>
Create a vector with <b>n</b> copies of <b>0</b>	<code>Vector&lt;int&gt; v(n);</code>	<code>std::vector&lt;int&gt; v(n);</code>
Create a vector with <b>n</b> copies of value <b>k</b>	<code>Vector&lt;int&gt; v(n, k);</code>	<code>std::vector&lt;int&gt; v(n, k);</code>
Add <b>k</b> to the end of the vector	<code>v.add(k);</code>	<code>v.push_back(k);</code>
Clear vector	<code>v.clear();</code>	<code>v.clear();</code>
Check if <b>v</b> is empty	<code>if (v.isEmpty())</code>	<code>if (v.empty())</code>
Get the element at index <b>i</b>	<code>int v = v.get(i); <b>int</b> k = v[i];</code>	<code>int k = v.at(i); <b>int</b> k = v[i];</code>
Replace the element at index <b>i</b>	<code>v.get(i) = k; <b>v[i] = k;</b></code>	<code>v.at(i) = k; <b>v[i] = k;</b></code>

# **operator[]** does not perform bounds checking

```
std::vector<int> vec{5, 6}; // {5, 6}
vec[1] = 3;                // {5, 3}
vec[2] = 4;                // undefined behavior
vec.at(2) = 4;             // Runtime error
```

# What questions do you have?



bjarne\_about\_to\_raise\_hand

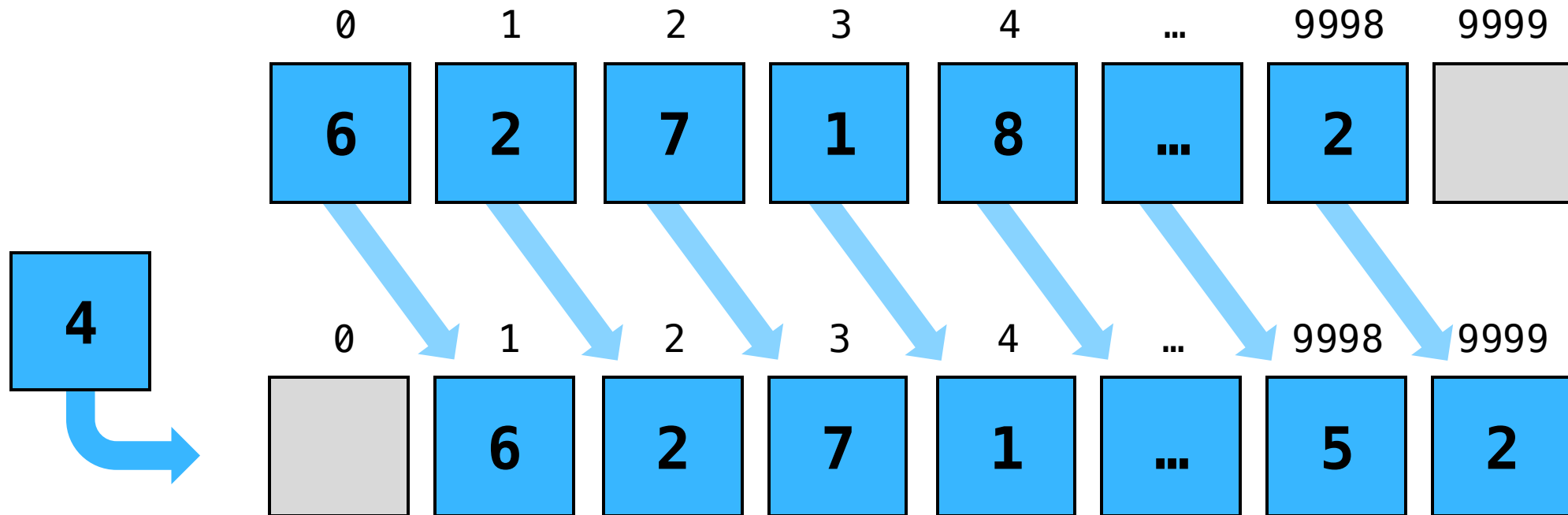




**Trick question!**

**`std::vector` has no `push_front`!**

# A hypothetical `push_front`...



**SLOW!!!**

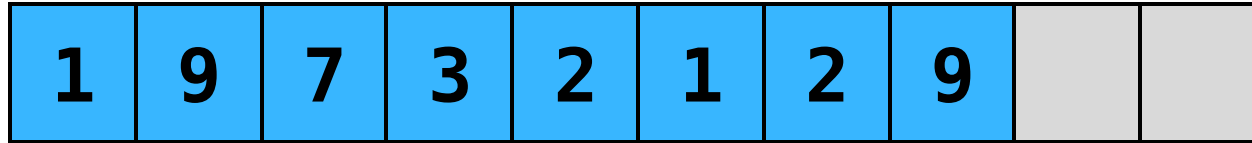


```
std::deque  
#include <deque>
```



**A deque has the same interface as vector,  
except we can push\_front / pop\_front**

# How is **deque** implemented?

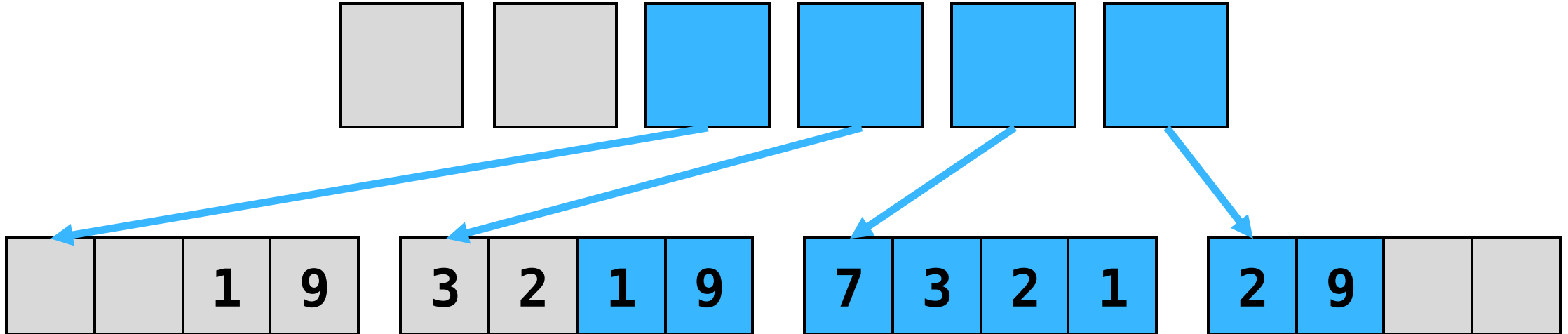


The problem with **vector** is that we have a single chunk of memory

So... let's split it up!

# How is **deque** implemented?

Array of arrays



Separate subarrays  
allocated independently

# What questions do you have?



bjarne\_about\_to\_raise\_hand

# **Associative Containers**

**Associative containers organize elements by unique keys**



```
std::map  
#include <map>
```

# `std::map` maps keys to values

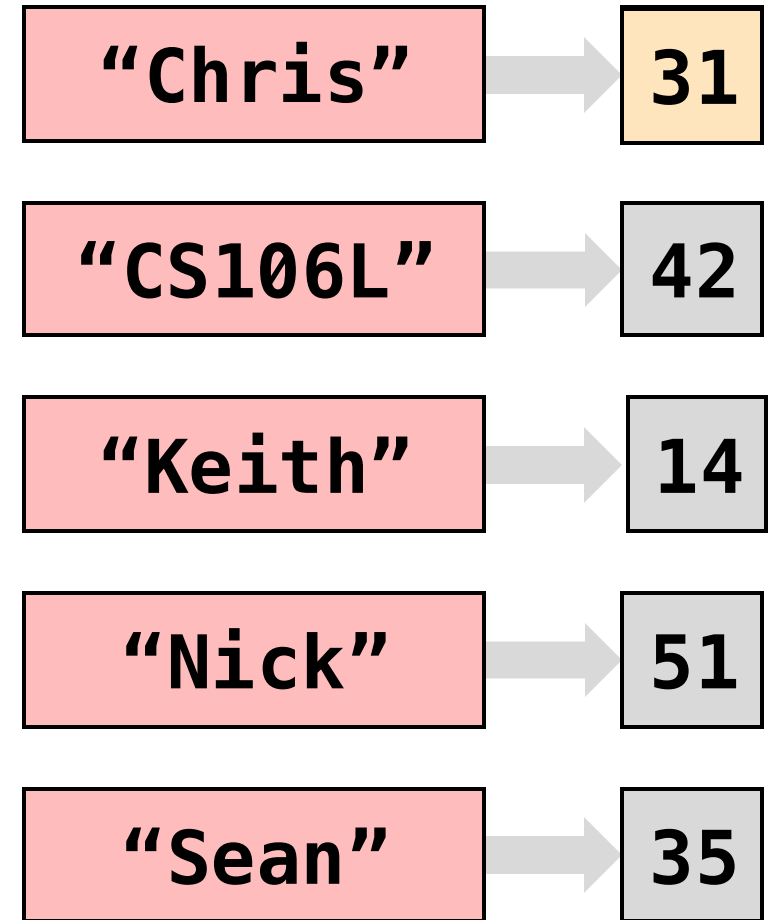


- Equivalent of a Python dictionary
- Sometimes called an **associative array**

# `std::map` maps keys to values

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

```
int sean = map["Sean"]; // 35  
map["Chris"] = 31;
```



# 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	<code>Map&lt;char, int&gt; m;</code>	<code>std::map&lt;char, int&gt; m;</code>
Add key <b>k</b> with value <b>v</b> into the map	<code>m.put(k, v);</code> <code>m[k] = v;</code>	<code>m.insert({k, v});</code> <code>m[k] = v;</code>
Remove key <b>k</b> from the map	<code>m.remove(k);</code>	<code>m.erase(k);</code>

# 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 <b>k</b> with value <b>v</b> into the map	m.put(k, v); m[k] = v;	m.insert({k, v}); m[k] = v;
Remove key <b>k</b> from the map	m.remove(k);	m.erase(k);
Check if <b>k</b> 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())

# 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 <b>k</b> with value <b>v</b> into the map	m.put(k, v); m[k] = v;	m.insert({k, v}); m[k] = v;
Remove key <b>k</b> from the map	m.remove(k);	m.erase(k);
Check if <b>k</b> 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())
Retrieve or overwrite value associated with key k (auto-insert default if doesn't exist)	int i = m[k]; m[k] = i;	int i = m[k]; m[k] = i;

```
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

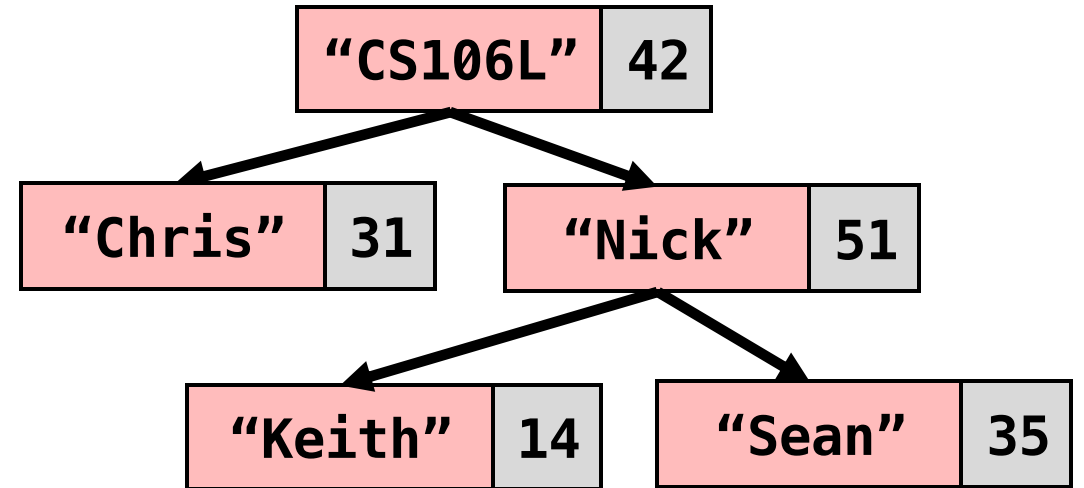
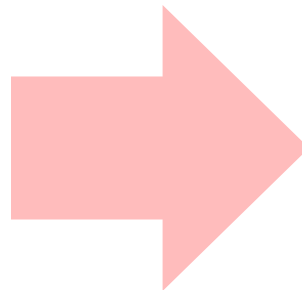
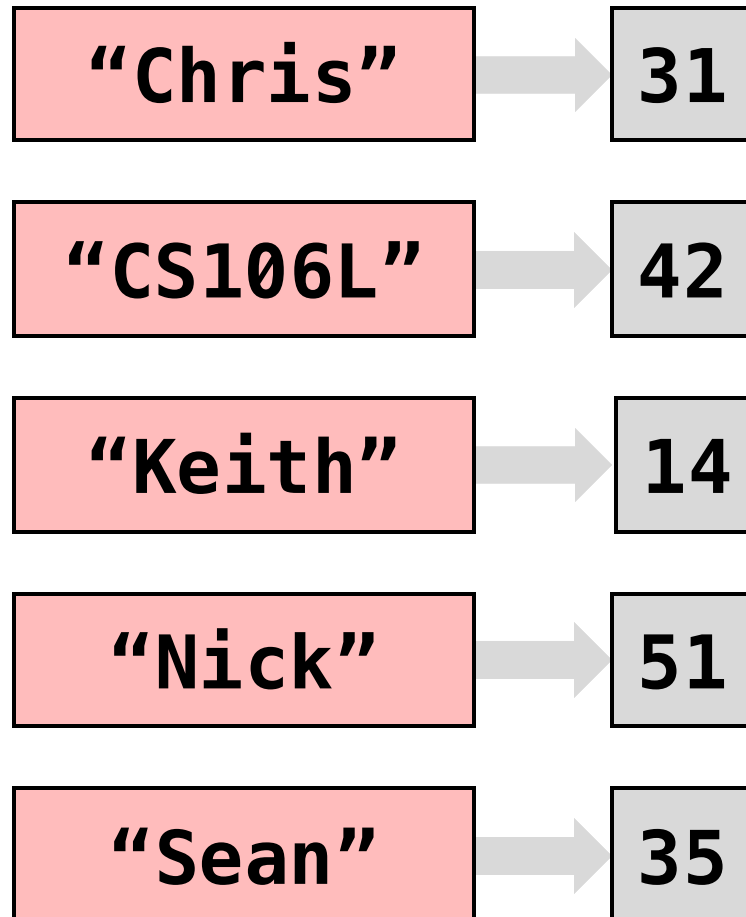
```
std::map<std::string, int> map;  
  
for (const auto& [key, value] : map) {  
    // key has type const std::string&  
    // value has type const int&  
}
```

# What questions do you have?



bjarne\_about\_to\_raise\_hand

# How is **map** implemented?

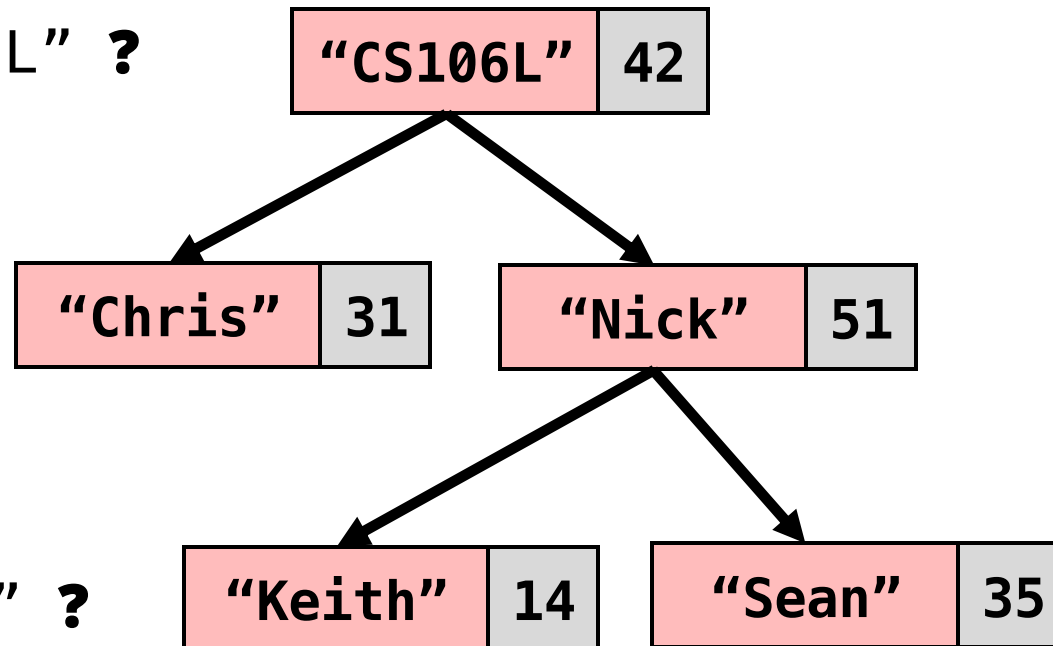


**Binary Search Tree**  
(technically a *red-black tree*)

# What is `map["Keith"]`?

"Keith" < "CS106L" ?

No! Go right



"Keith" < "Keith" ?

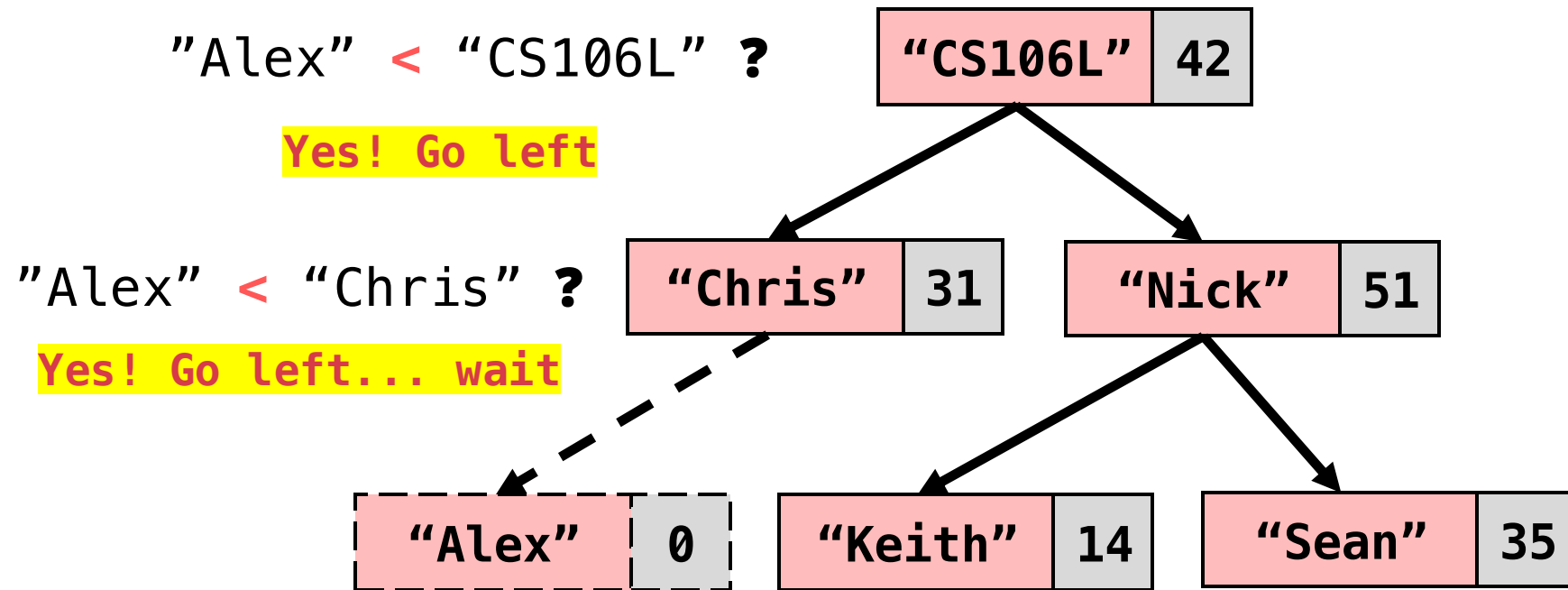
Hey look, you found me!

`map["Keith"] = 14`

"Keith" < "Nick" ?

Yes! Go left

# What is `map["Alex"]`?




`map["Alex"] = 0`

(Note: "Alex" was default-inserted into the map)

`std::map<K, V>` requires `K` to have an `operator<`

`std::map<K, V>` requires `K` to have an `operator<`

```
//  OKAY – int has operator<  
std::map<int, int> map1;
```

```
//  ERROR – std::ifstream has no operator<  
std::map<std::ifstream, int> map2;
```



# What questions do you have?



bjarne\_about\_to\_raise\_hand

```
std::set  
#include <set>
```

**std::set** stores a collection of unique items

```
std::set<std::string> set {  
    "CS106L!",  
    "Keith",  
    "Sean",  
    "Nick",  
    "Chris"  
};
```

≈

{  
 "CS106L!"  
 "Keith"  
 "Sean"  
 "Nick"  
 "Chris"  
}

# Stanford vs. STL set

What you want to do?	Stanford Set<char>	std::set<char>
Create an empty set	<code>Set&lt;char&gt; s;</code>	<code>std::set&lt;char&gt; s;</code>
Add k to the set	<code>s.add(k);</code>	<code>s.insert(k);</code>
Remove k from the set	<code>s.remove(k);</code>	<code>s.erase(k);</code>
Check if k is in the set (* C++20)	<code>if (s.contains(k))</code>	<code>if (s.count(k))</code> <code>if (s.contains(k)) (*)</code>
Check if the set is empty	<code>if (s.isEmpty())</code>	<code>if (s.empty())</code>

**std::set is an amoral std::map**

**std::set is an std::map without values**

# How is **set** implemented?

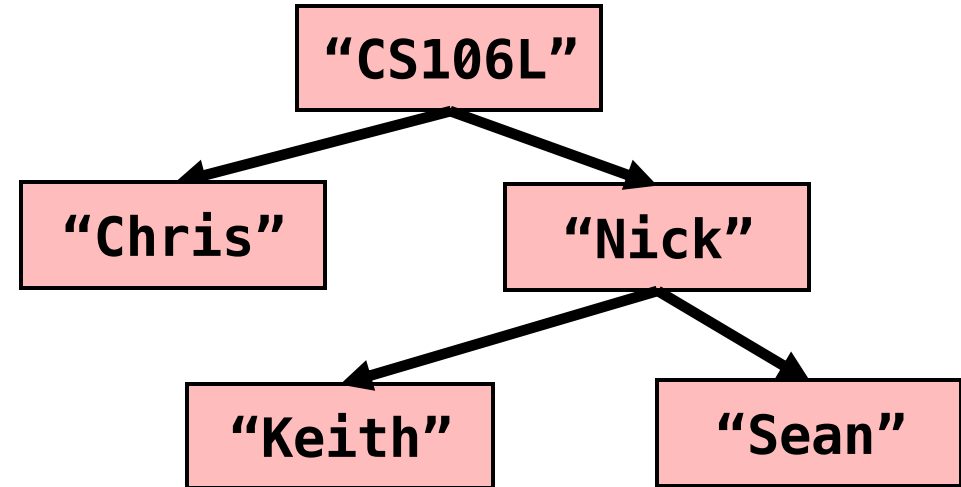
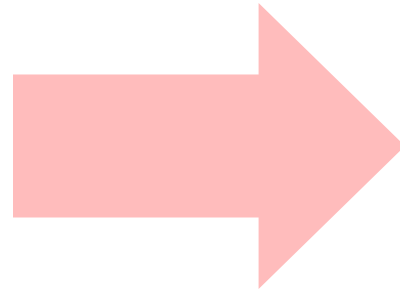
"Chris"

"CS106L"

"Keith"

"Nick"

"Sean"



**Binary Search Tree**  
(technically a *red-black tree*)

**But wait... `map` and `set` have an alter ego □ □**

**std::unordered\_map** and **std::unordered\_set**

```
#include <unordered_map>
```

```
#include <unordered_set>
```



# std::unordered\_map

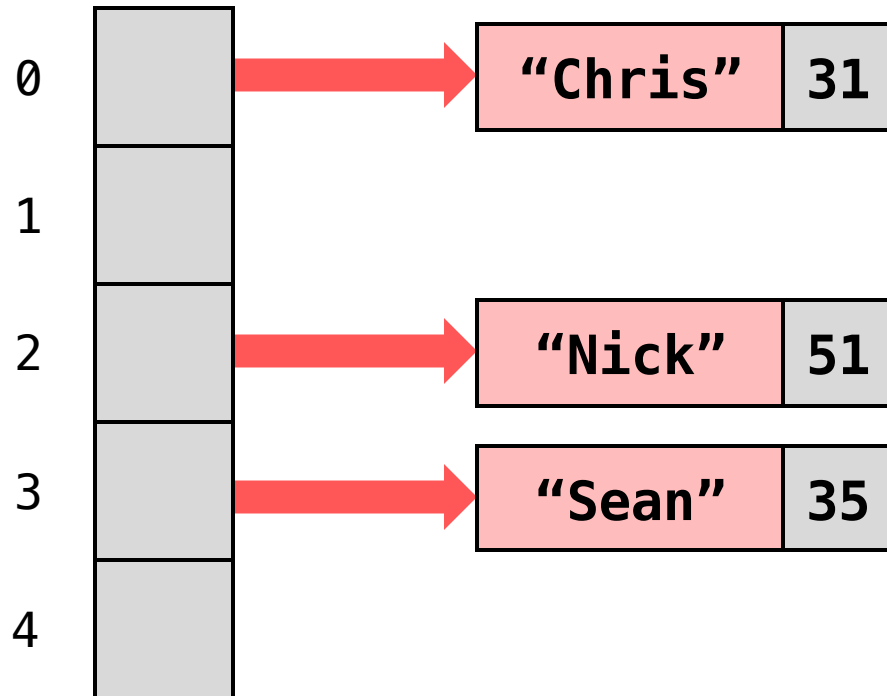
- You can think of `unordered_map` as an optimized version of `map`
- It has the same interface as `map`

```
std::unordered_map<std::string, int> map {  
    { "Chris", 2 },  
    { "Nick", 51 },  
    { "Sean", 35 },  
};
```

```
int sean = map["Sean"]; // 35  
map["Chris"] = 31;
```

# How is `unordered_map` implemented?

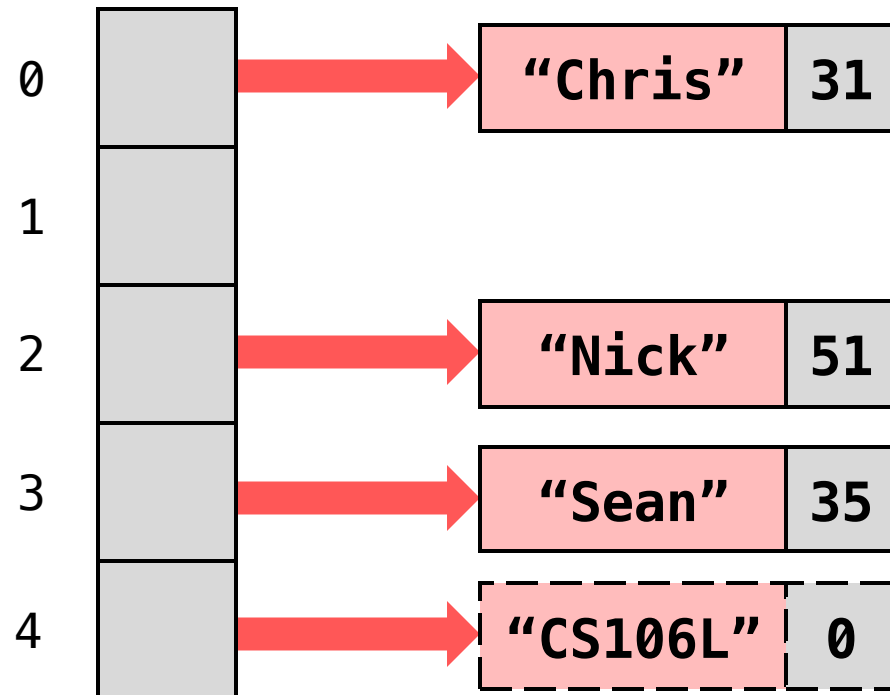
- Remember, map is a collection of `std::pair`
- `unordered_map` stores a collection of  $n$  "buckets" of pairs



```
std::unordered_map  
<std::string, int> map {  
    { "Chris", 31 },  
    { "Nick", 51 },  
    { "Sean", 35 },  
};
```

# How is `unordered_map` implemented?

- To add a key/value, we feed the key through a **hash function**
- The hash, modulo the bucket count, determines the pair's bucket no.



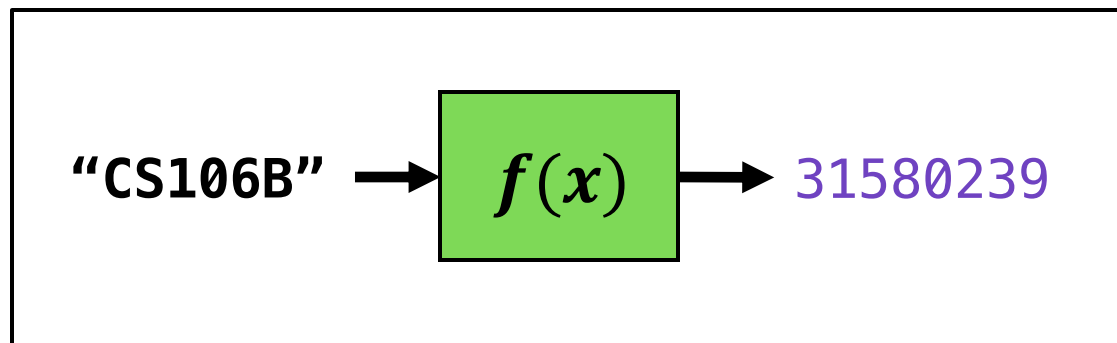
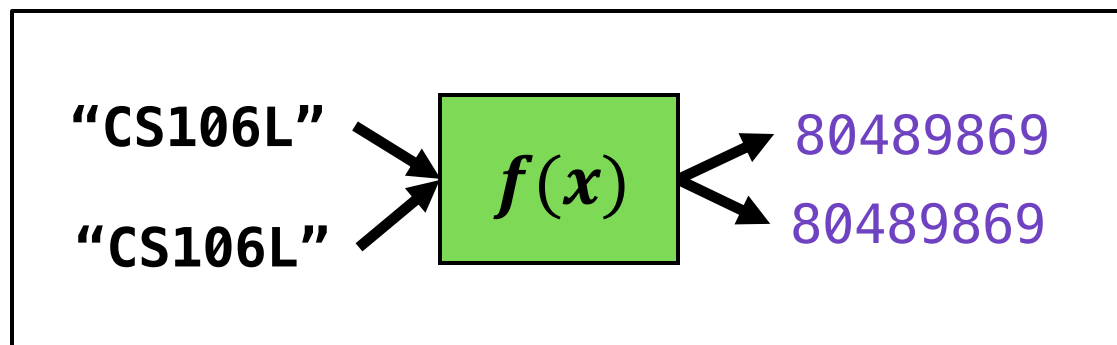
```
int x = map["CS106L"];
```

"CS106L" →  $f(x)$  → 80489869

80489869 mod 5 = 4

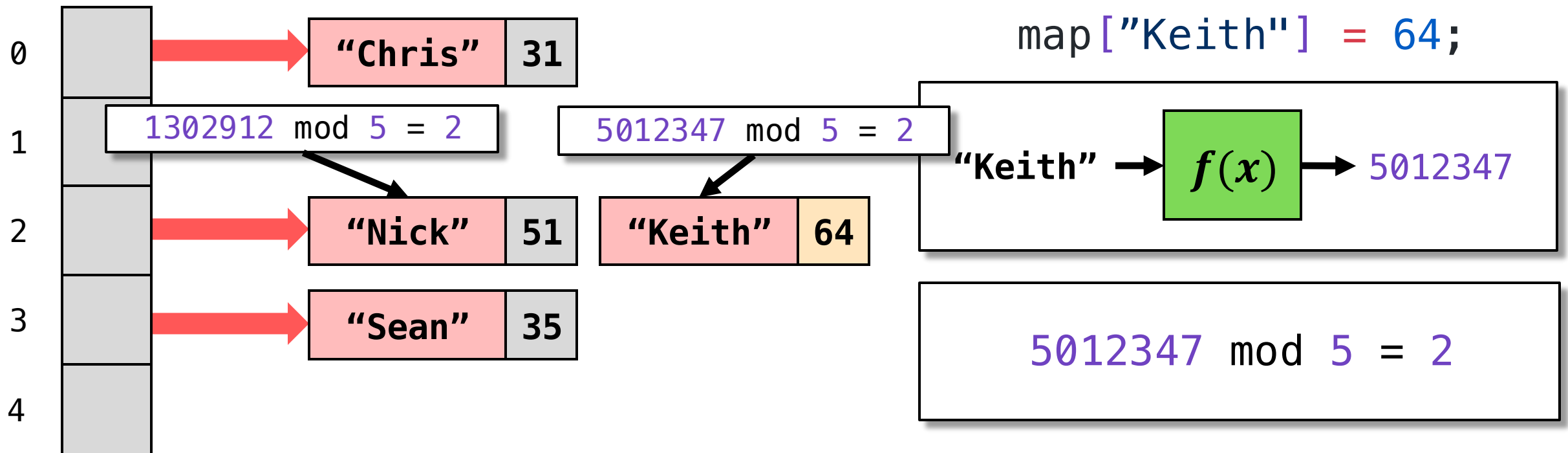
# What is a hash function?

- “Scrambles” a key into a `size_t` (64 bit)
- Small changes in the input should produce large changes in the output



# How is `unordered_map` implemented?

- If two keys hash to the same bucket, we get a **hash collision**
- During lookup, we loop through bucket and check key equality
  - Two keys with the same hash are not necessarily equal!





`std::unordered_map<K, V>` requires **K** to have a hash function (and equality)

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;
```

*(We will learn more about this syntax later!)*

`std::unordered_map<K, V>` requires **K** to be hashable

```
//  OKAY – int is hashable  
std::unordered_map<int, int> map1;  
  
//  ERROR – std::ifstream is not hashable  
std::unordered_map<std::ifstream, int> map2;
```

Most basic types (`int`, `double`, `string`) are hashable by default

# What questions do you have?

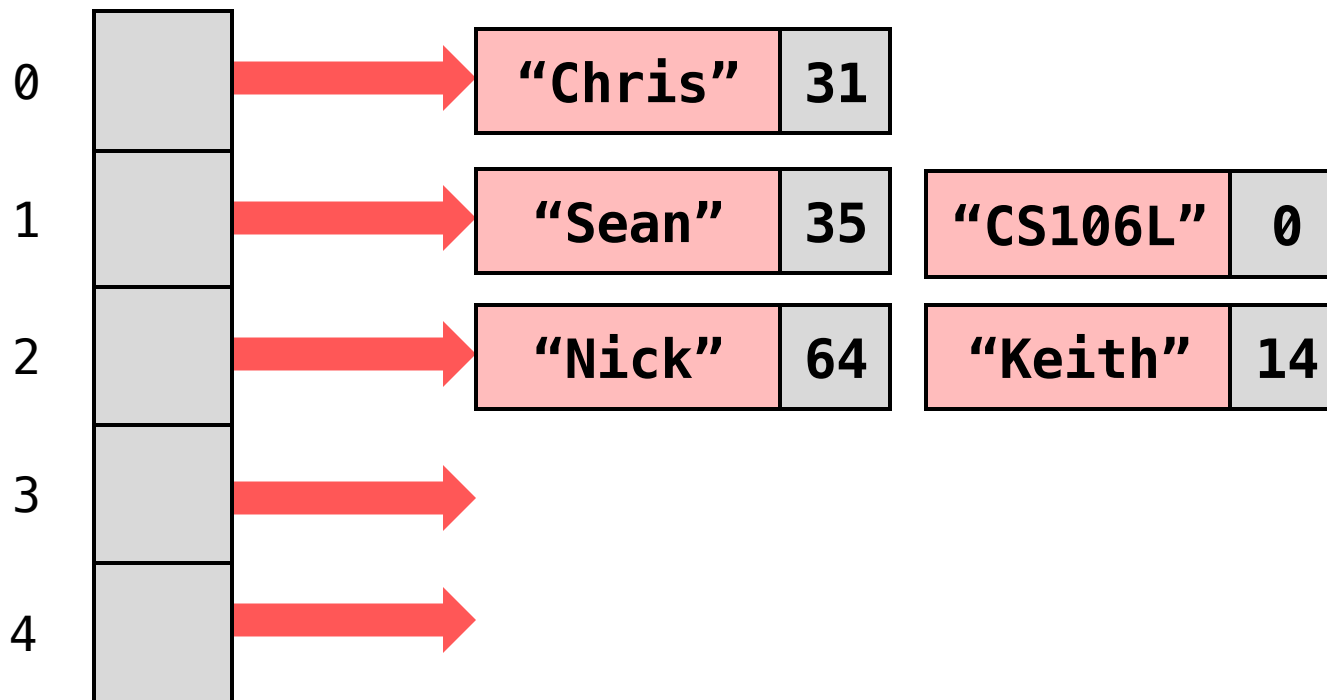


bjarne\_about\_to\_raise\_hand



# Why use `std::unordered_map`?

- **Load factor:** average number items per bucket
- `unordered_map` allows super fast lookup by keeping load factor small
- If load factor gets too large (above 1.0 by default), we **rehash**



**Load Factor: 1.66**

**Load Factor: 1.0**

# Fun C++ Trivia: `max_load_factor`

You can control the max load factor before rehashing

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

double lf = map.load_factor(); // Get current load factor
map.max_load_factor(2.0); // Set the max load factor

// Now the map will not rehash until load factor exceeds 2.0
// You should almost never need to do this,
// but it's a fun fact (good for parties!)
```

# What makes a good hash function?

A good hash function minimizes the chance of a hash collision

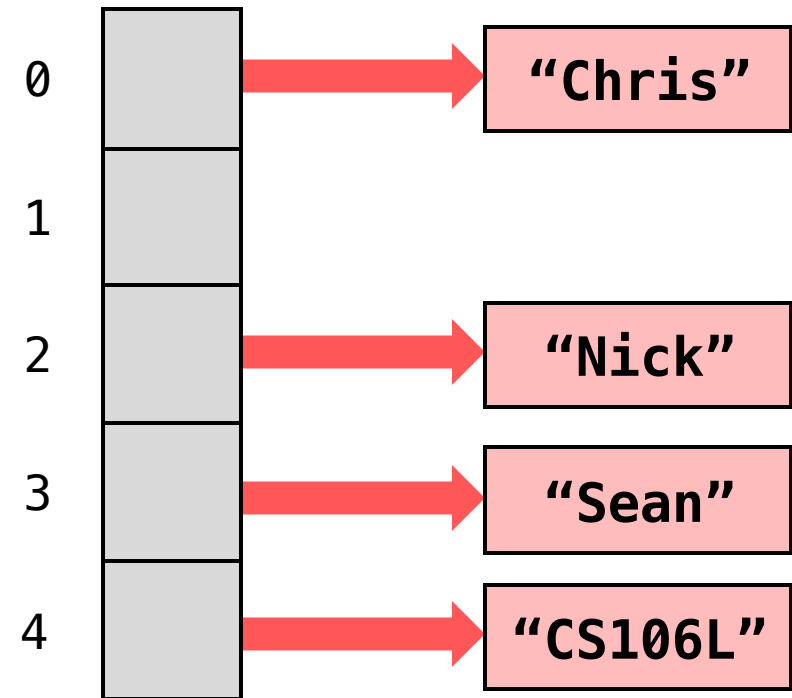
```
// ❌ The worst possible hash

template <>
struct std::hash<MyType>
{
    std::size_t operator()(const MyType& k) const
    {
        return 0;
    }
};
```

*(Don't worry too much about this syntax. We'll learn more later)*

**unordered\_set** is an **unordered\_map** without values

```
std::unordered_set  
<std::string> set {  
    "Chris",  
    "Nick",  
    "Sean",  
    "CS106L"  
};
```



# When to use `unordered_map` vs. `map`?

- `unordered_map` is *usually* faster than `map`
- However, it uses more memory (organized vs. disorganized garage)
- If your key type has no total order (`operator<`), use `unordered_map`!
- If you must choose, `unordered_map` is a safe bet

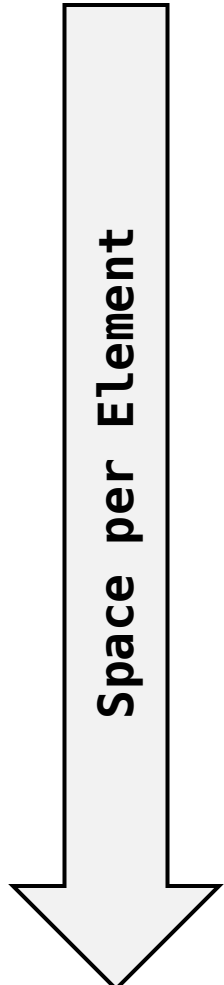
# What questions do you have?



bjarne\_about\_to\_raise\_hand

# Recap

# Summary of Data Structures



	$i^{\text{th}}$ element	Search	Insertion	Erase
<code>std::vector</code>	Very Fast	Slow	Slow	Slow
<code>std::deque</code>	Fast	Slow	Fast (front/back) Slow (all others)	Fast (front/back) Slow (all others)
<code>std::set</code>	Slow	Fast	Fast	Fast
<code>std::map</code>	Slow	Fast	Fast	Fast
<code>std::unordered_set</code>	N/A	Very Fast	Very Fast	Very Fast
<code>std::unordered_map</code>	N/A	Very Fast	Very Fast	Very Fast



# Some more containers if you're curious!

**std::array**

*A fixed-size array of items*

**std::list**

*A doubly linked list*

**std::multiset (+unordered)**

*A set that can contain duplicates*

**std::multimap (+unordered)**

*Can contain multiple values for the same key*

# Recap

- What the heck is the STL? What are templates?
  - "The Standard Template Library"
- Sequence Containers
  - A linear sequence of elements
  - `std::vector`, `std::deque`
- Associative Containers
  - A set of elements organized by unique keys
  - `std::map`, `std::set`, `std::unordered_map`, `std::unordered_set`