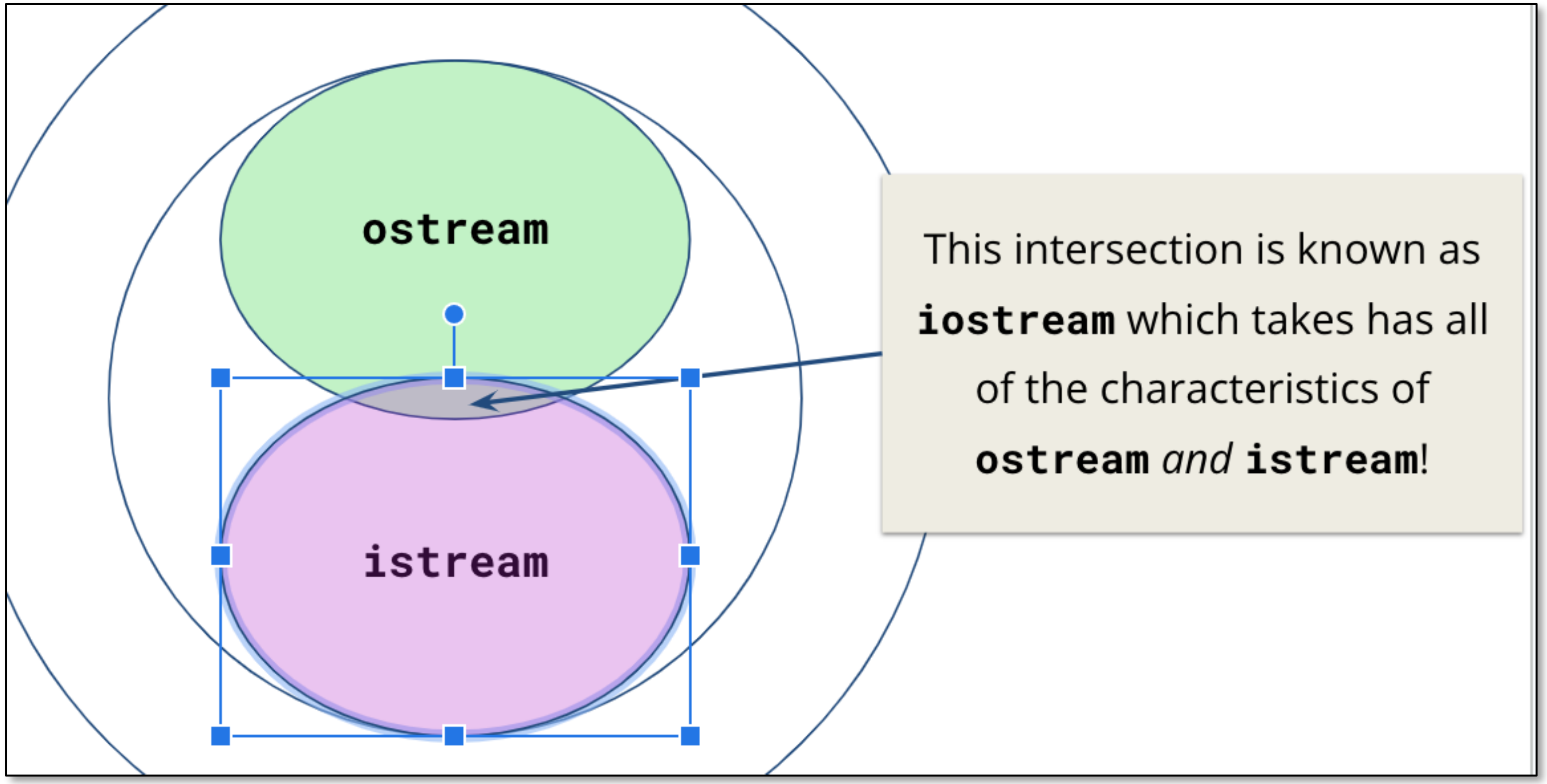
What type of initialization can all types use?

- 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!



The many containers of C++

The C++ Standard Template Library (STL)

std::vector	std::unordered_map
std::set	std::unordered_set
std::stack	std::priority_queue
std::queue	std::deque
std::map	std::array

What are templates?

```
class IntVector {

}
```

```
class DoubleVector {

}
```

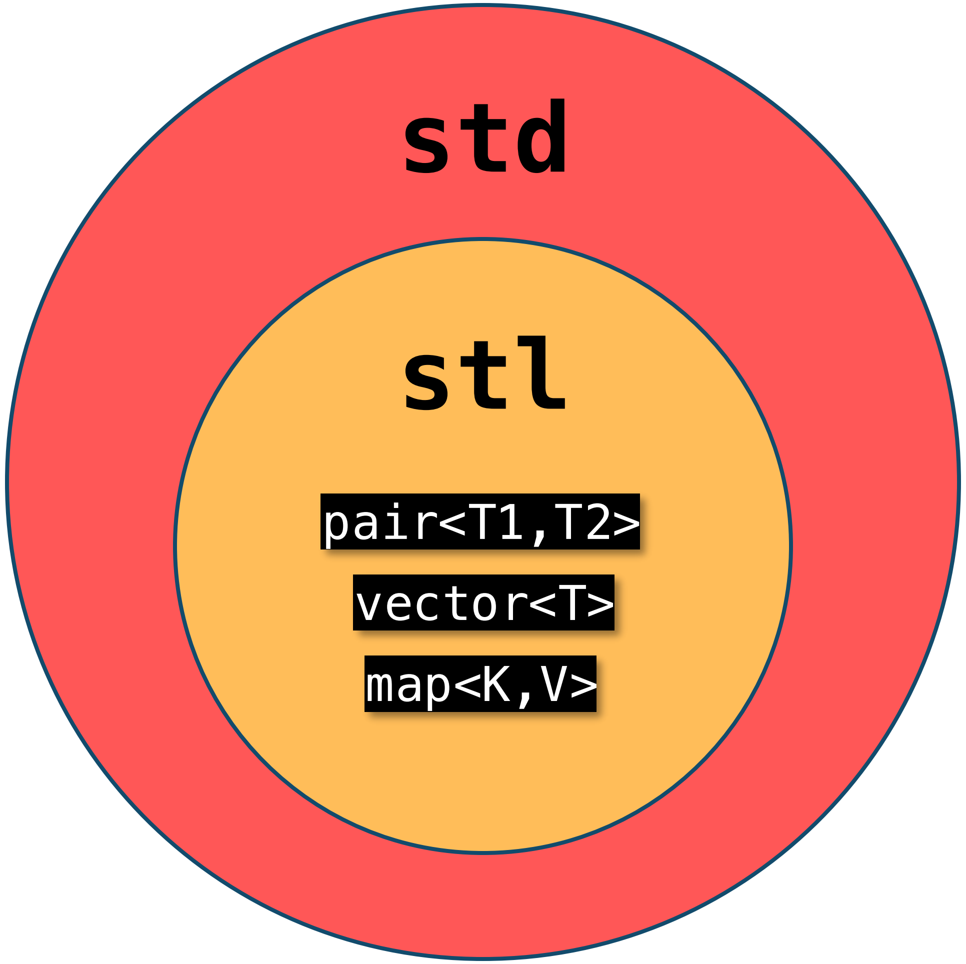
```
class StringVector {
  // Code to store
  // a list of
  // strings…
};
```

```
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)



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?

Sequence containers store a linear sequence of elements

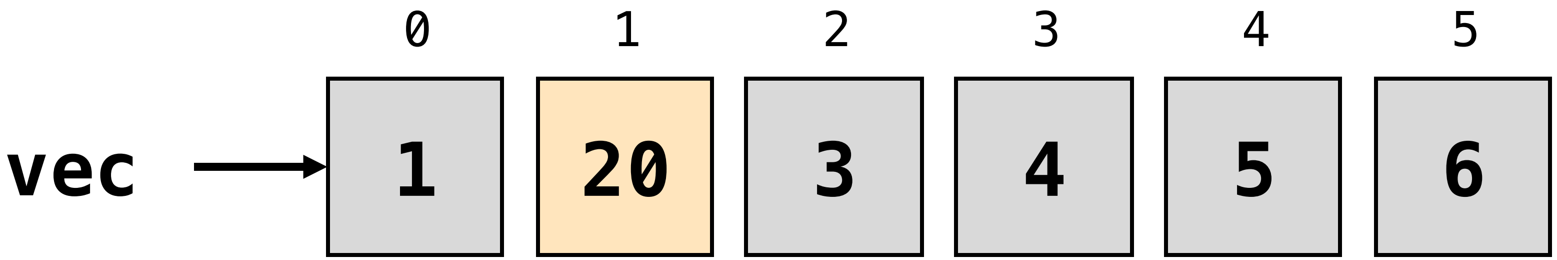
std::vector

```
#include <vector>
```

std::vector stores a list of elements

```cpp
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] << " ";
}
```

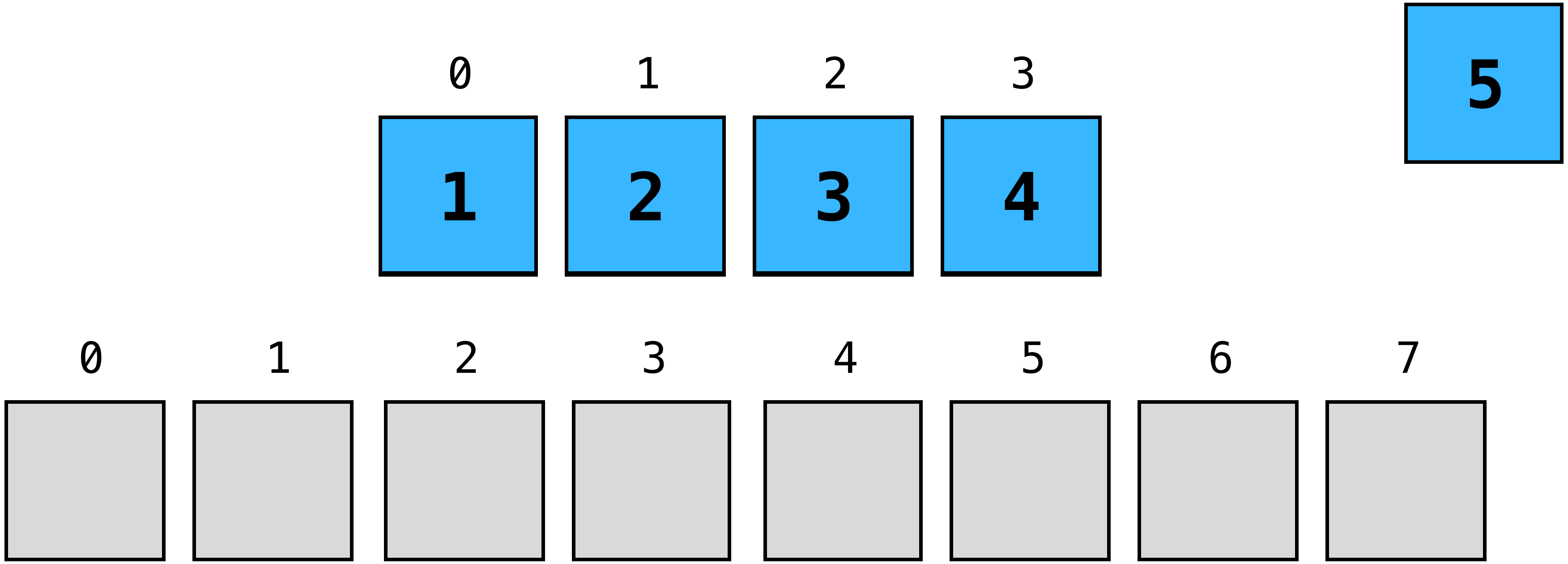


Output:
1 20 3 4 5 6

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);`
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 = 4, capacity = 8

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

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
```

std::vector is not the best for all cases...

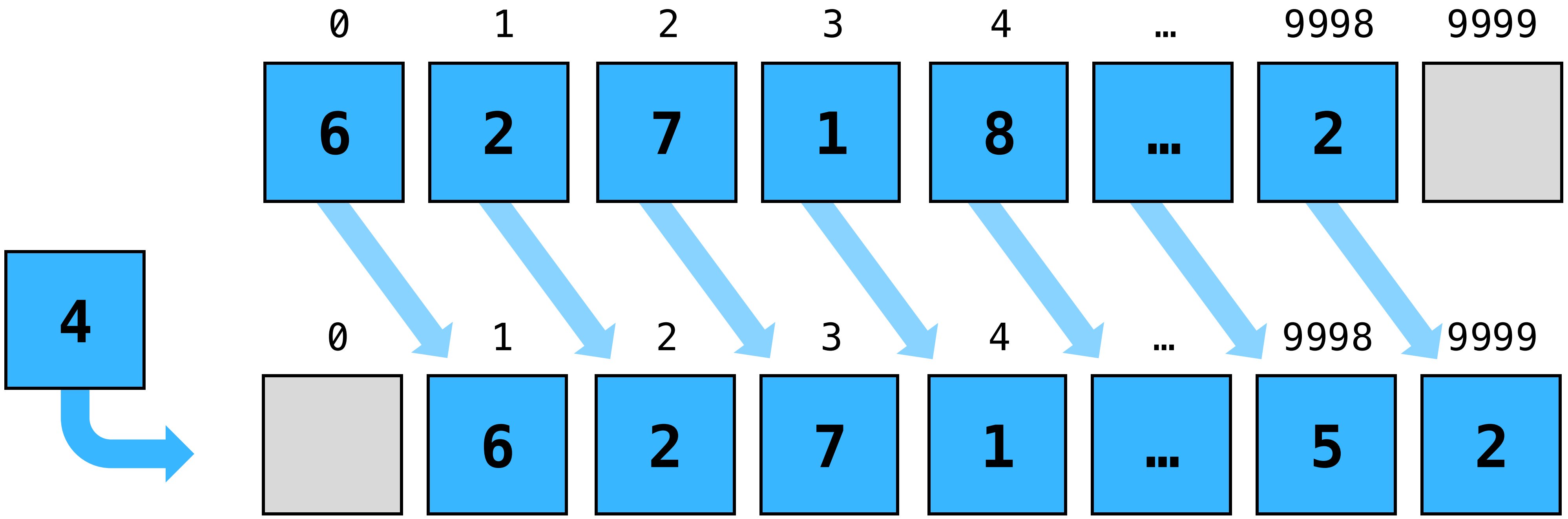
- Suppose we need to observe the last 10,000 prices of a stock
- What might be concerning about the code below?

```
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
}
```

Trick question!

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

A hypothetical `push_front`...



SLOW!!!

`std::deque`

`#include <deque>`

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?

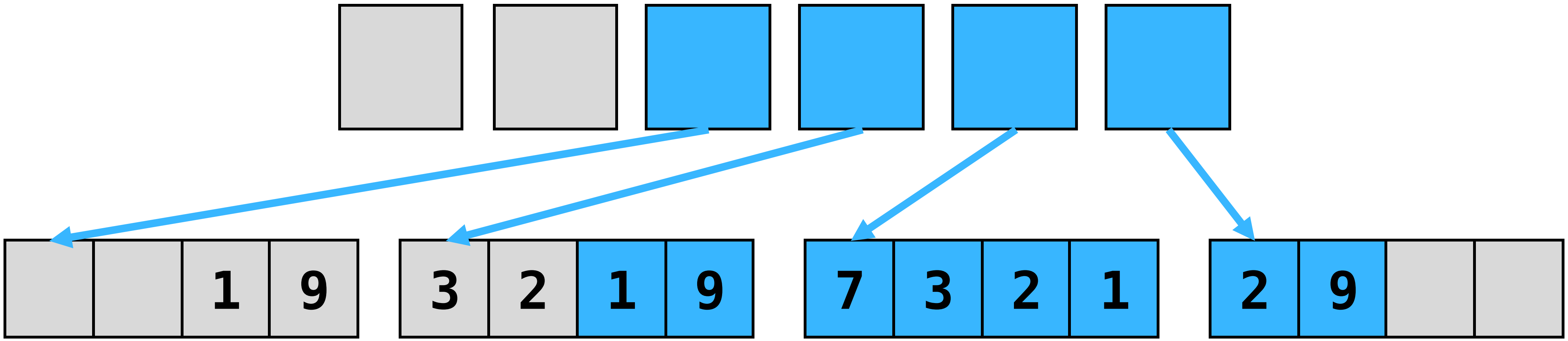
1	9	7	3	2	1	2	9		

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

Associative containers organize elements by unique keys

std::map

```
#include <map>
```

`std::map` maps keys to values

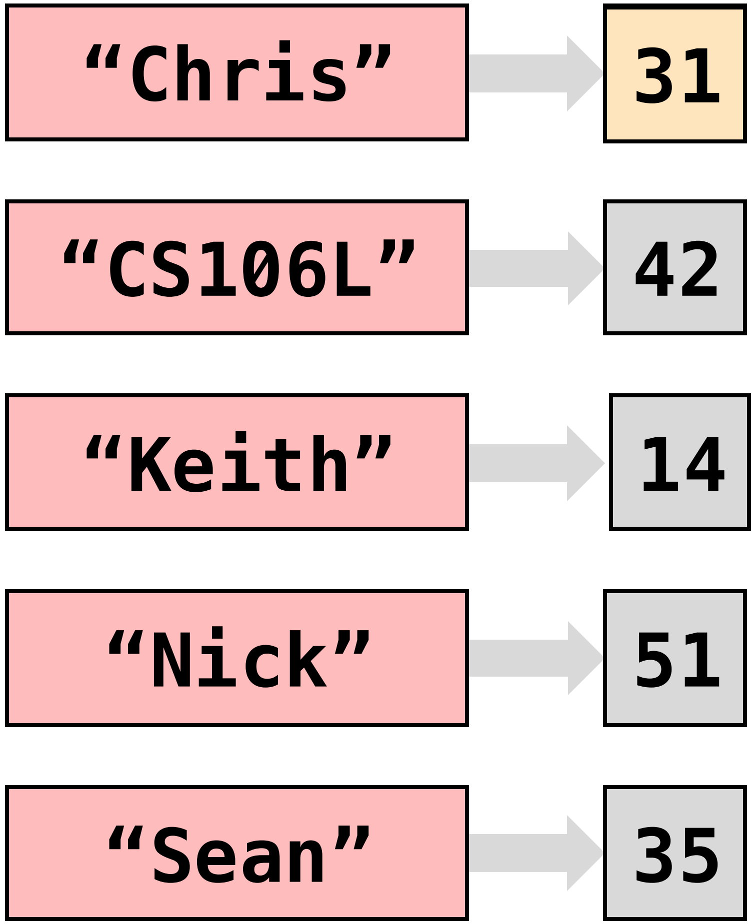
```
"CS106L!" → std::map<std::string, int> → 42
```

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

std::map maps keys to values

```cpp
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;`
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())`
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;
}
```

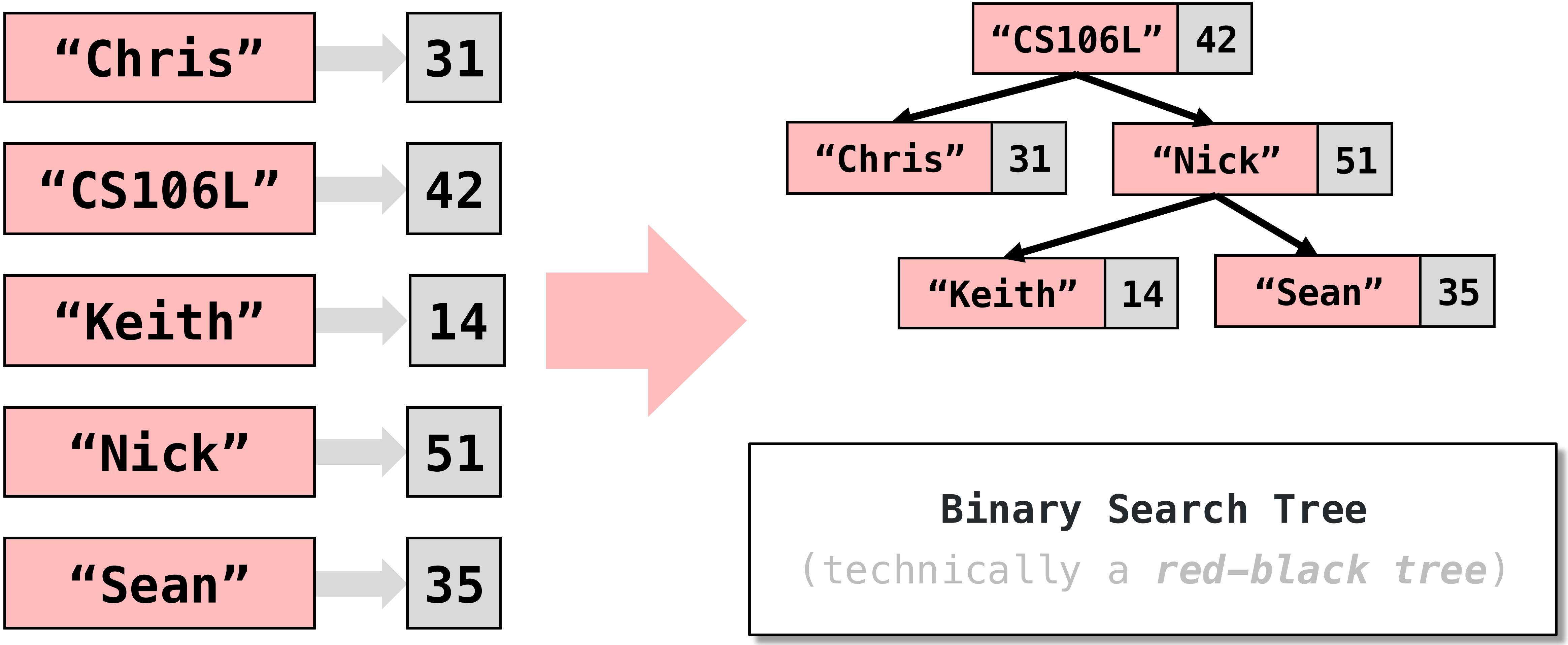
map as a collection of pair

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&
}
```

How is map implemented?



`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;
```

std::set

```
#include <set>
```

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

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

$$\approx \left\{ \begin{array}{c} \text{“CS106L!”} \\ \text{“Keith”} \\ \text{“Sean”} \\ \text{“Nick”} \\ \text{“Chris”} \end{array} \right\}$$

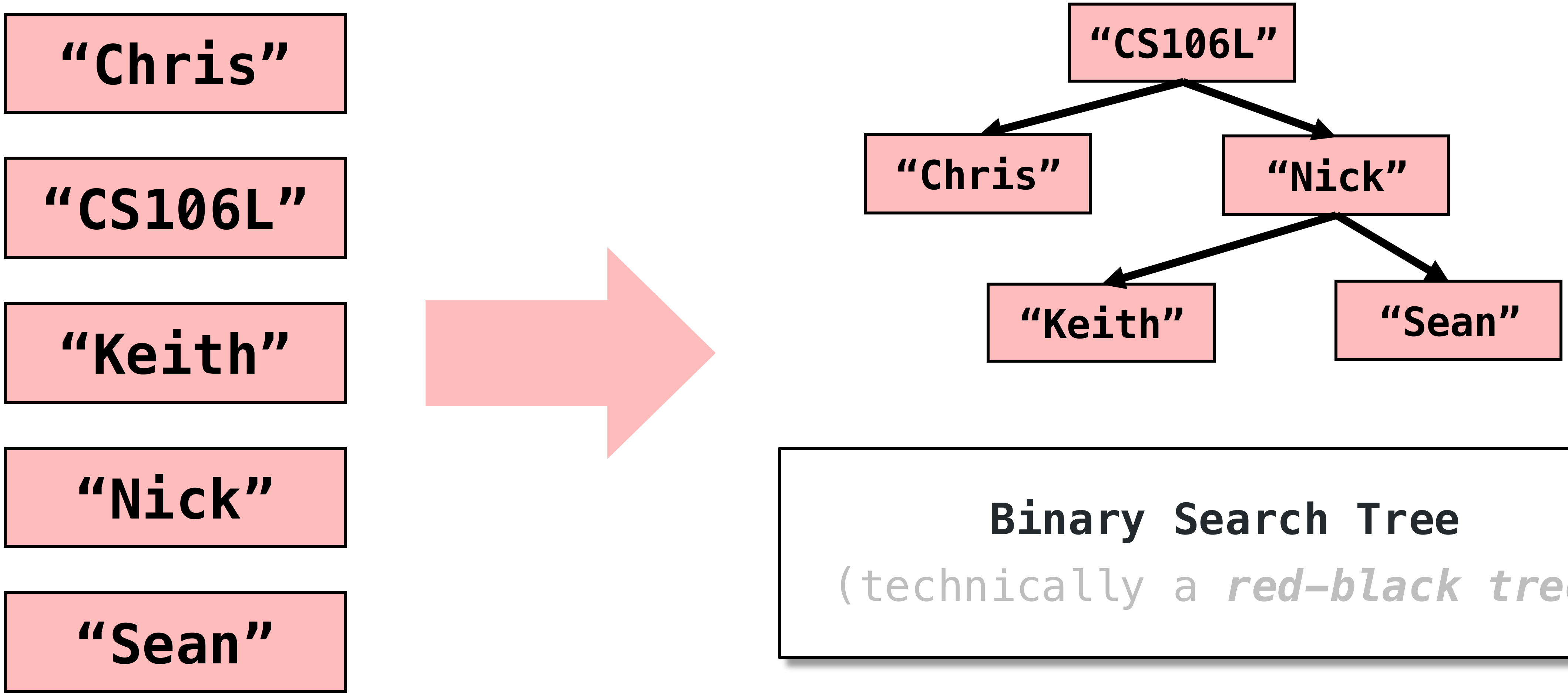
Stanford vs. STL set

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

`std::set` is an amoral `std::map`

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

How is set implemented?



Binary Search Tree

(technically a *red-black tree*)

`std::unordered_map` and `std::unordered_set`

```
#include <unordered_map>
#include <unordered_set>
```

`std::unordered_map`

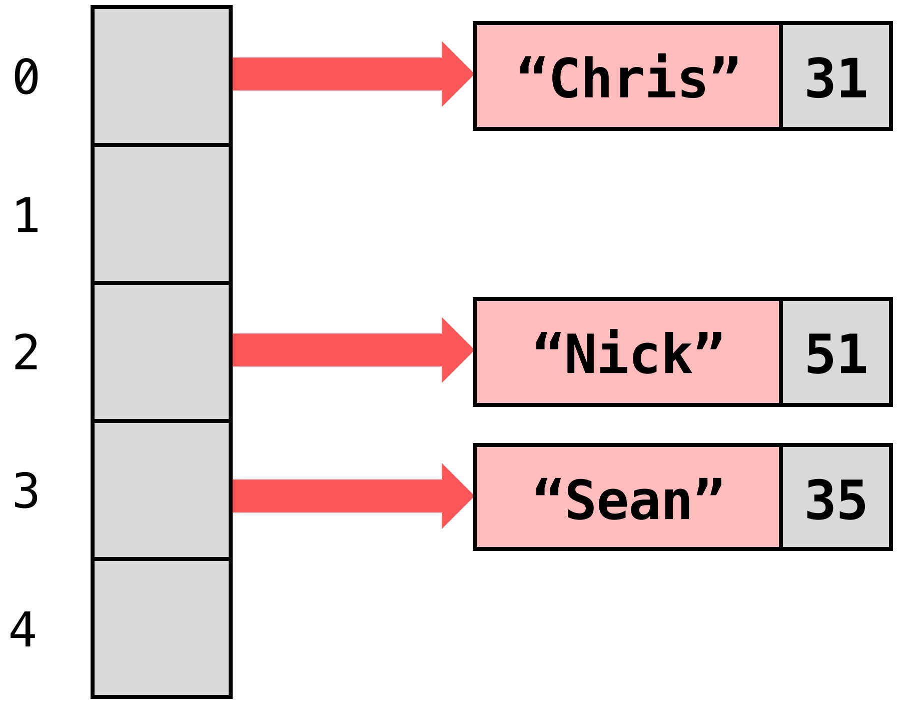
- You can think of `unordered_map` as an optimized version of `map`
- It has the same interface as `map`

```
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?

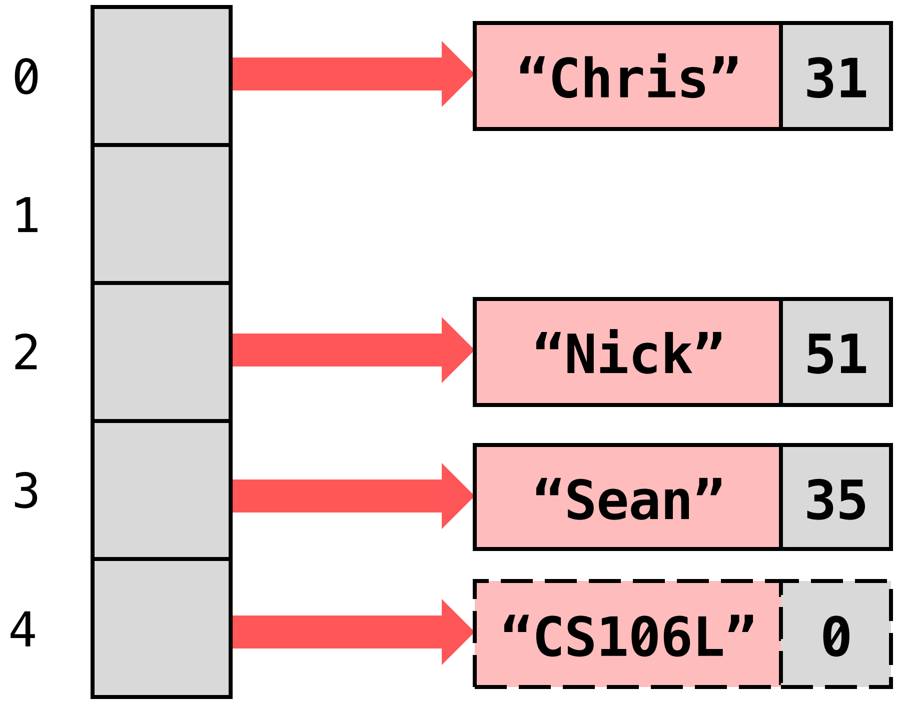
- Remember, map is a collection of `std::pair`
- `unordered_map` stores a collection of n "buckets" of pairs



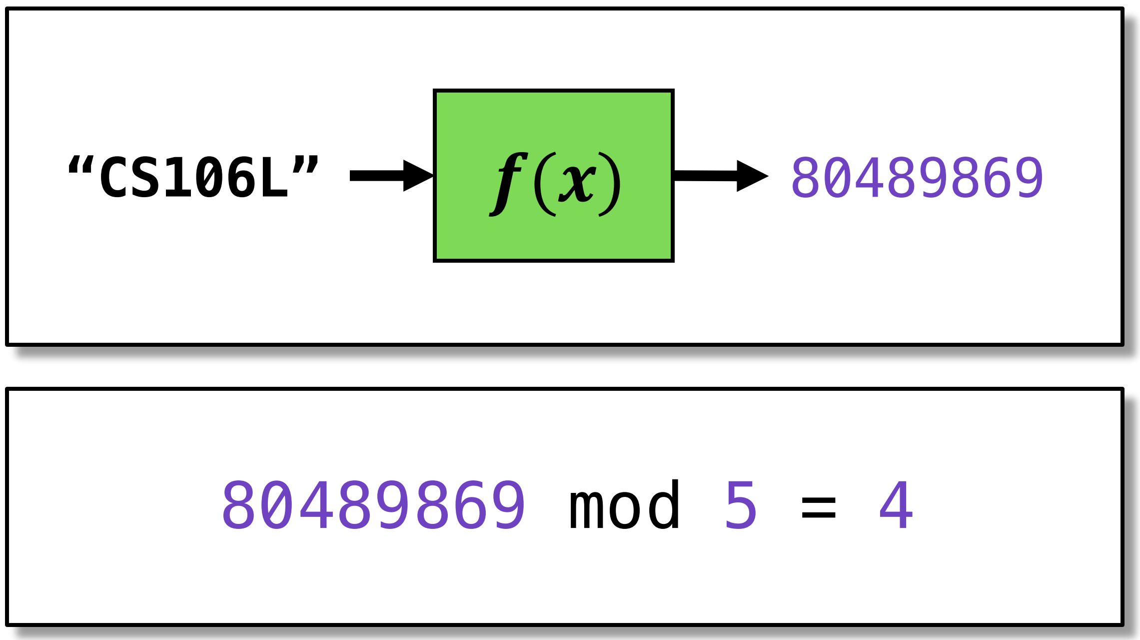

```
std::unordered_map
<std::string, int> map {
   { "Chris", 31 },
   { "Nick", 51 },
   { "Sean", 35 },
};
```

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.

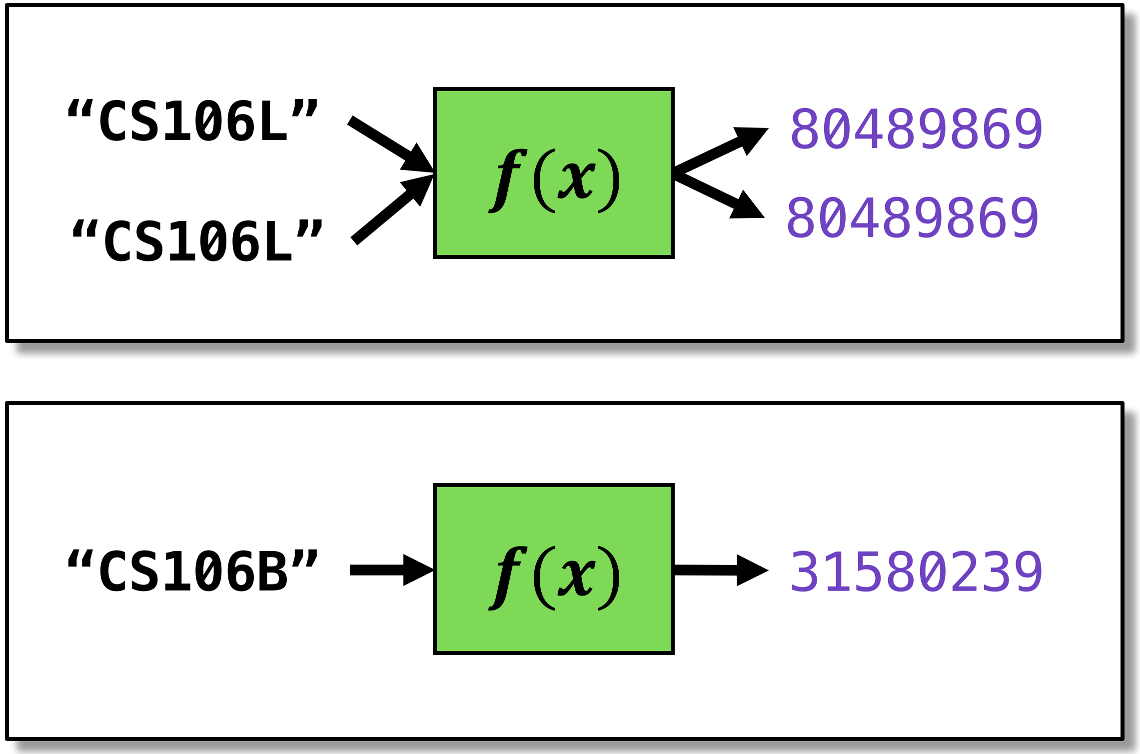



```
int x = map["CS106L"];
```



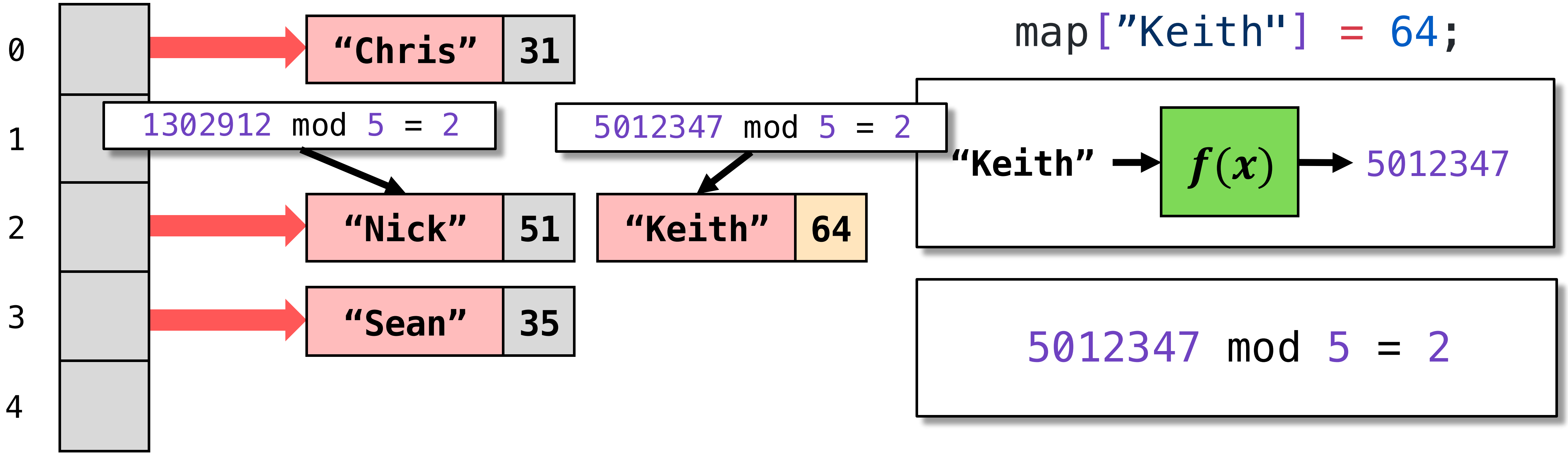
What is a hash function?

- “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

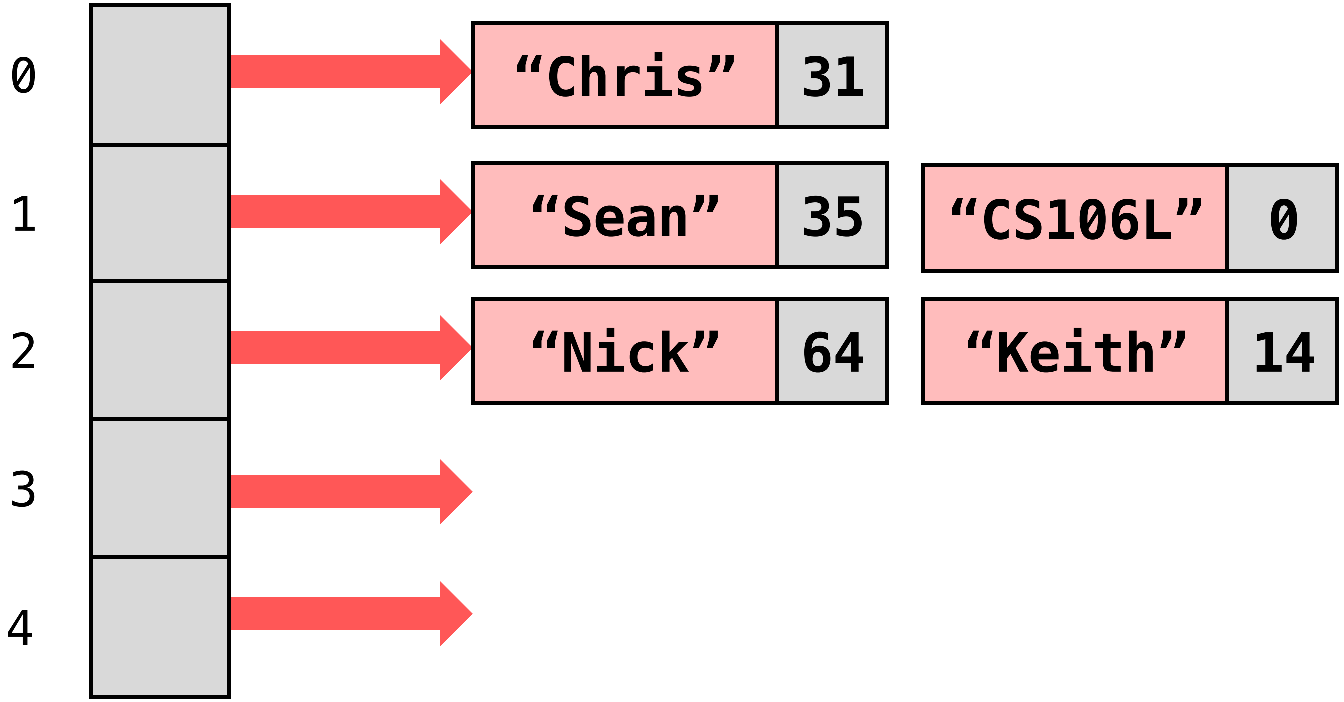
```
// ✅ OKAY - int is hashable
std::unordered_map<int, int> map1;

// ❌ ERROR - std::ifstream is not hashable
std::unordered_map<std::ifstream, int> map2;
```

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

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

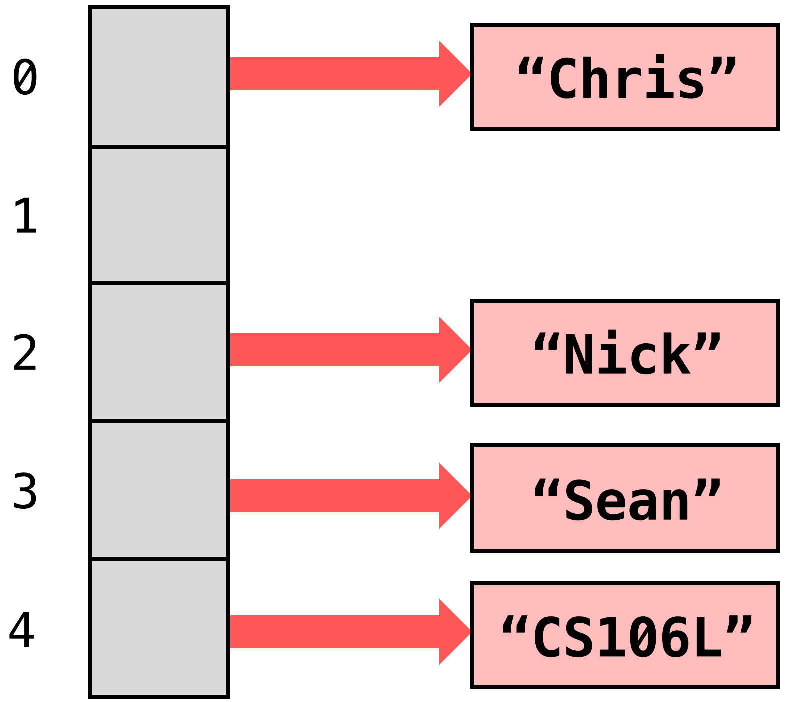
```
// ❌ The worst possible hash

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

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



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

- `unordered_map` is <u>*usually*</u> 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

Summary of Data Structures

Space per Element

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