TDDE18 & 726G77

Standard Templated Library – Iterator and Containers

Lab 5 – wordlist

- No loops in your code (neither for-loop, while-loop nor do-while-loop)
- No Range-based for loop
- No regex solutions allowed for this lab
- Use algorithms and containers in the Standard Templated Library (STL)

Standard Template Library

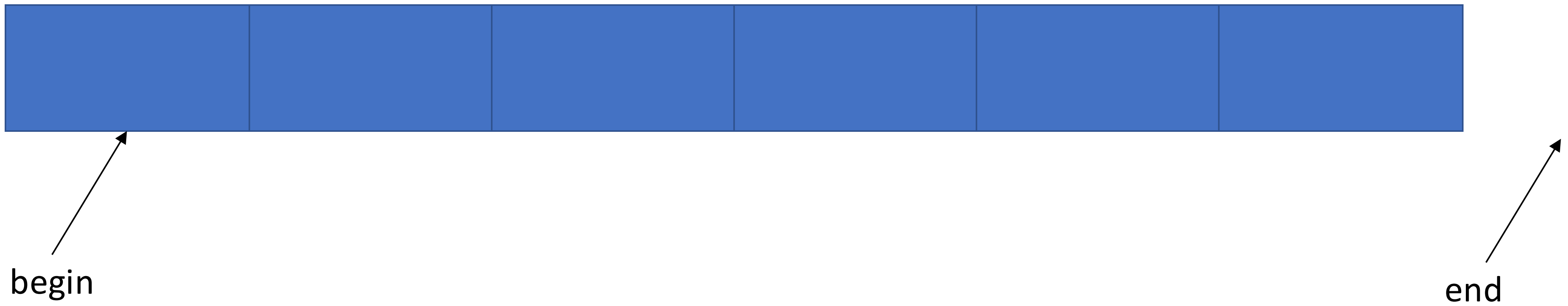
- The C++ Standard Library is a collection of classes and functions, which are written in the core language.
- Provides several generic containers with different strength and weakness
- A general way to iterate over all element in a container
- Algorithms to process data in the container in different ways.
- Everything is templated – works on all datatypes

Iterator concept

- Describes types that can be used to identify and traverse the elements of a container (eg. vector and list)
- Iterator can be dereferenced to get the object
- Iterator can be used with the pre- and post-increment to get to the next element in a container
- You can think of iterators as pointers, which are used in the Standard Library

Iterator concept

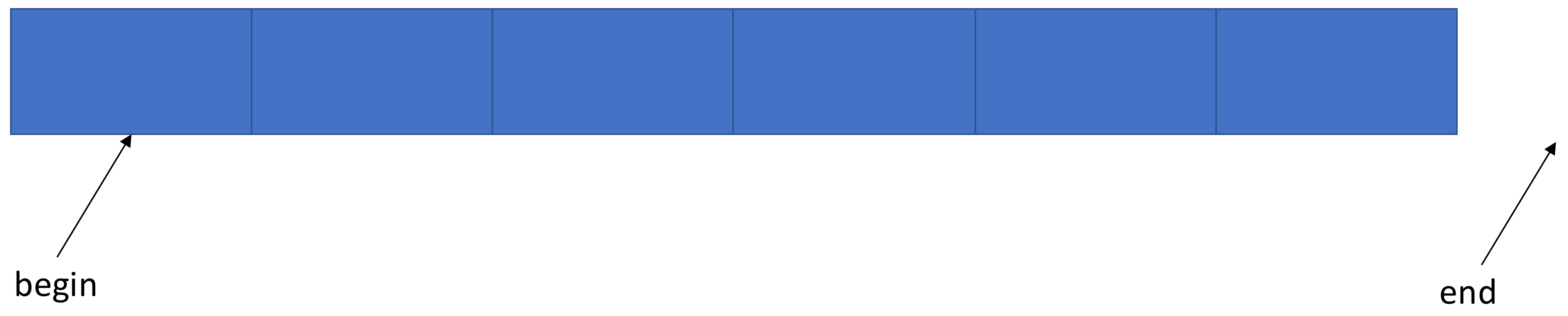
- To iterate a collection of data we need
  - A starting point (begin)
  - Some way to get to the next data in the collection (++)
  - Some way to get from the iterator to the actual data (*)
  - An ending point (end)



Forward iterator

- Begin
  - Refer to first element of container
  - Valid to dereference on non-empty container
  - Increment toward last element (forward iteration)
- End
  - Refer to just after last element of container
  - Invalid to dereference
- Data type

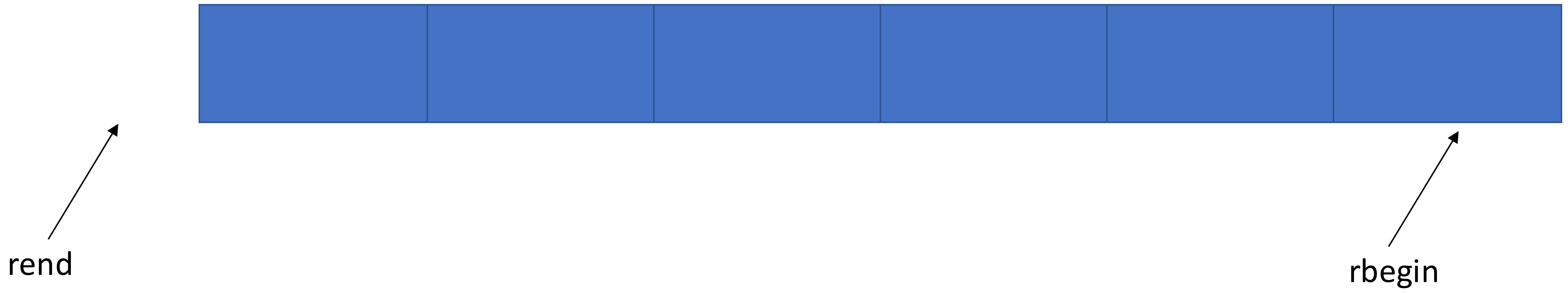
  ::iterator



Reverse iterator

- Rbegin
  - Refer to last element of container
  - Valid to dereference on non-empty container
  - Increment toward first element (backward iteration)
- Rend
  - Refer to just before first element of container
  - Invalid to reference
- Datatype

  ::reverse_iterator



Iterator over constant data

- begin(), end(), rbegin(), rend()
  - return mutable (non-const) iterator
  - data in container can be modified through iterator
  - None of the above refer to same position
- cbegin(), cend(), crbegin(), crend()
  - return immutable (const) iterators
  - data in container can only be read
  - type ::const_iterator or ::const_reverse_iterator

Which iterator to use

- Depend on what you want to do
- A good safe default

  ::const_iterator, cbegin(), cend()
- If you really need to change data

  ::iterator, begin(), end()
- If you really need to go backwards

  ::const_reverse_iterator

  ::reverse_iterator (if you need mutable access)

Containers

- pair
- tuple
- vector
- string
- list
- set
- map
- array

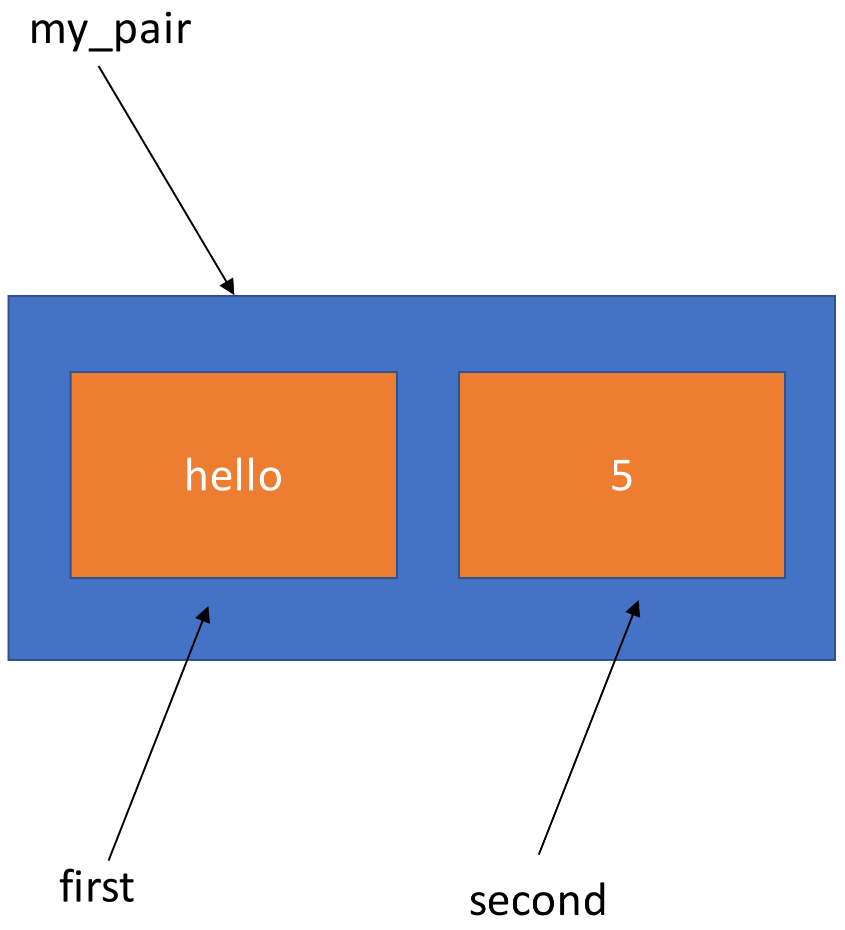
(Many more!)

std::pair

- Store two data items
- They do not have to be of the same type

std::pair

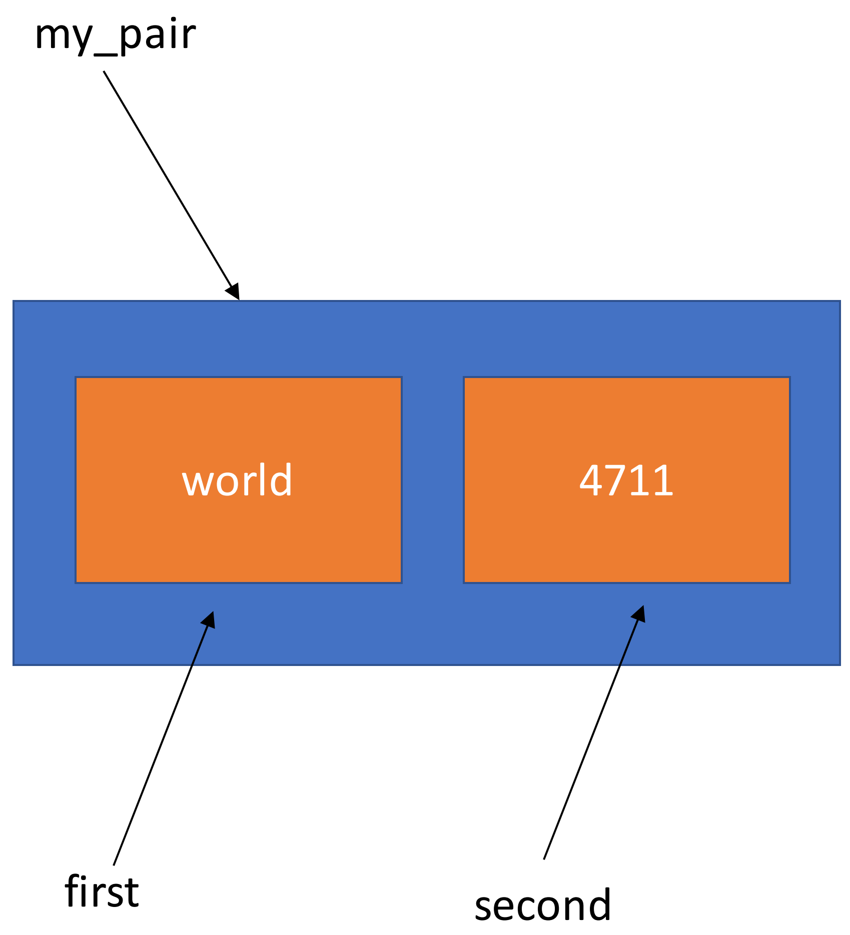
```
#include <utility>
pair<string, int> my_pair{“hello”, 5};

my_pair.first;      // returns “hello”
my_pair.second;     // return 5
```



std::make_pair

- `Creates a std::pair object, deducing the target type from the types of arguments`

```
pair<string, int> my_pair;
my_pair = make_pair("world", 4711);
```

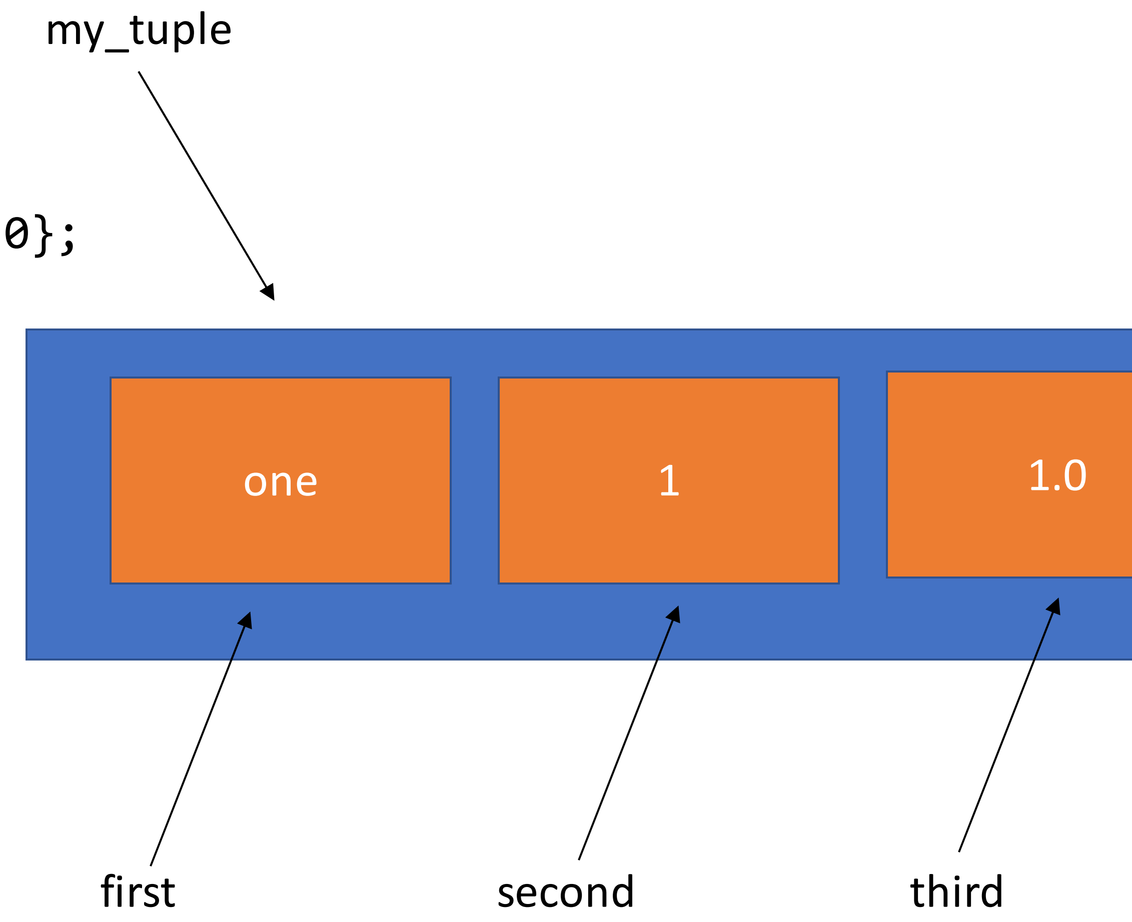


std::tuple

- Stores (groups) any fix number of data items
- They do not have to be of same type

std::tuple

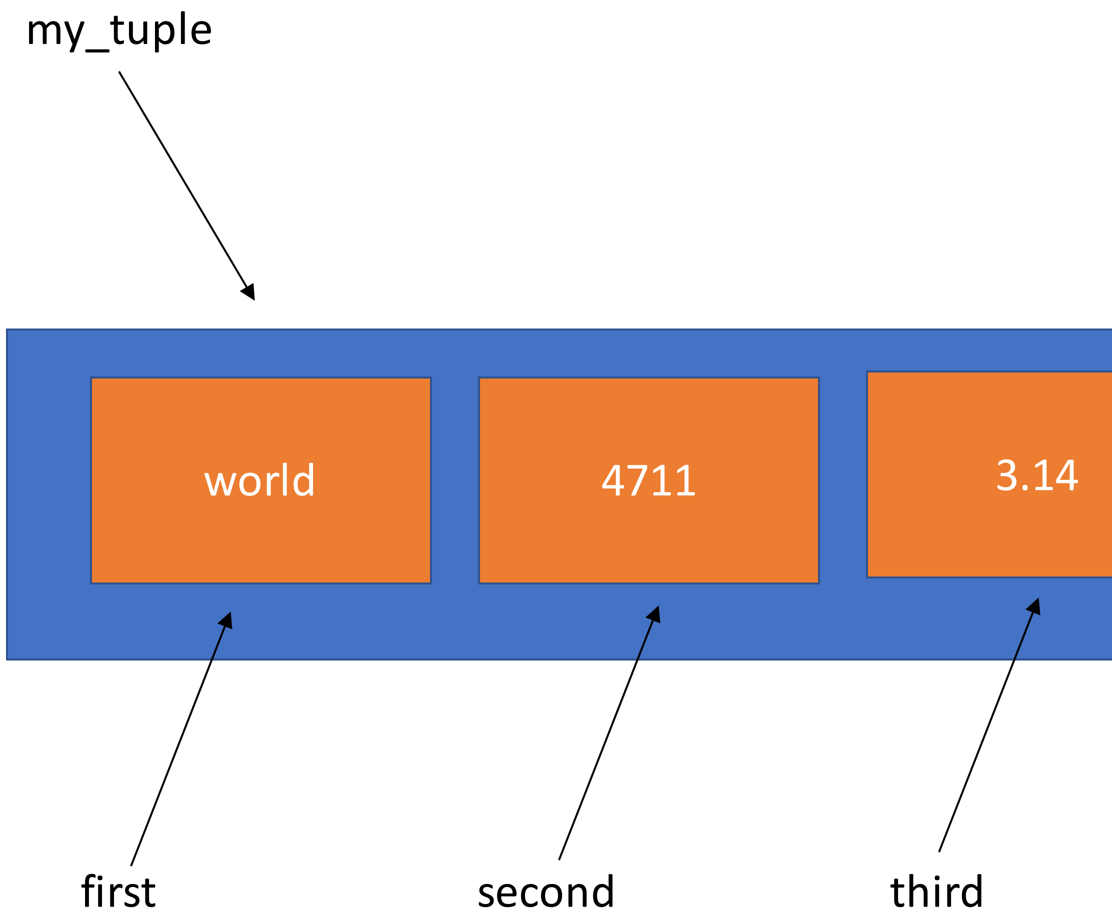
```
#include <tuple>
tuple<string, int, float> my_tuple{“one”, 1, 1.0};

get<0>(my_tuple);     // return “one”
get<1>(my_tuple);     // return 1
get<2>(my_tuple);     // return 1.0
```



std::make_tuple

- `Creates a std::tuple object, deducing the target type from the types of arguments`

```
my_tuple = make_tuple(“world”, 4711, 3.14);
```



std::vector

- vector is a sequence container that encapsulates dynamic size arrays
- The elements are stored contiguously, which means that elements can be access by using offsets
- The storage of vector is handled automatically, being expanded and contracted as needed

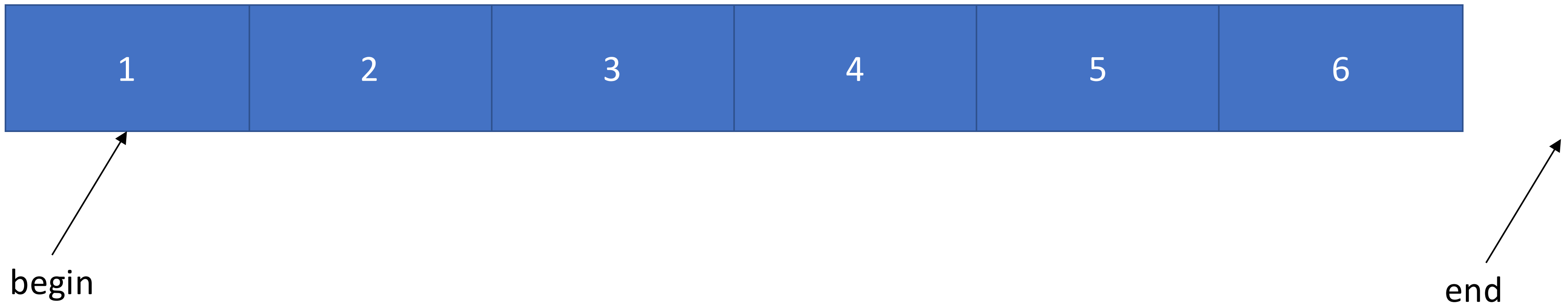
std::vector - constructor

```
vector<int> v1{};                  // default constructor
vector<int> v2{v1};                // copy constructor
vector<int> v3{1, 2, 3, 4, 5};     // initializer list
vector<int> v4(5);                 // size is 5, all element are initialized to 0
vector<int> v5(5, 1);              // size is 5, all element are initialized to 1

vector<int> v6{begin(v2), end(v2)}; // using iterators to initialize the vector
vector<int> v7{begin(v2) + 3, end(v2)};     // will have 2 elements: 4 and 5
```

There are more at http://en.cppreference.com/w/cpp/container/vector/vector

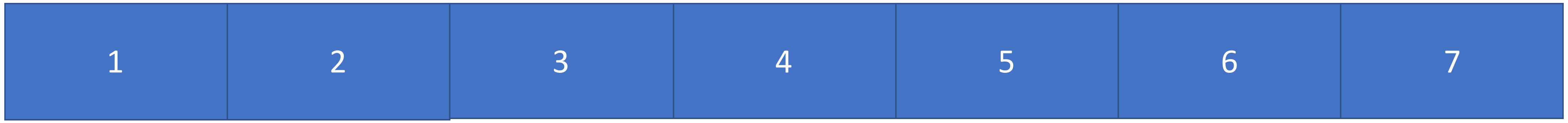
std::vector – begin- and end-iterator

```
vector<int> v{1, 2, 3, 4, 5, 6};
v.begin();                  // begin(v) returns v.begin()
v.end();                    // end(v) returns v.end()
```



std::vector – size vs capacity

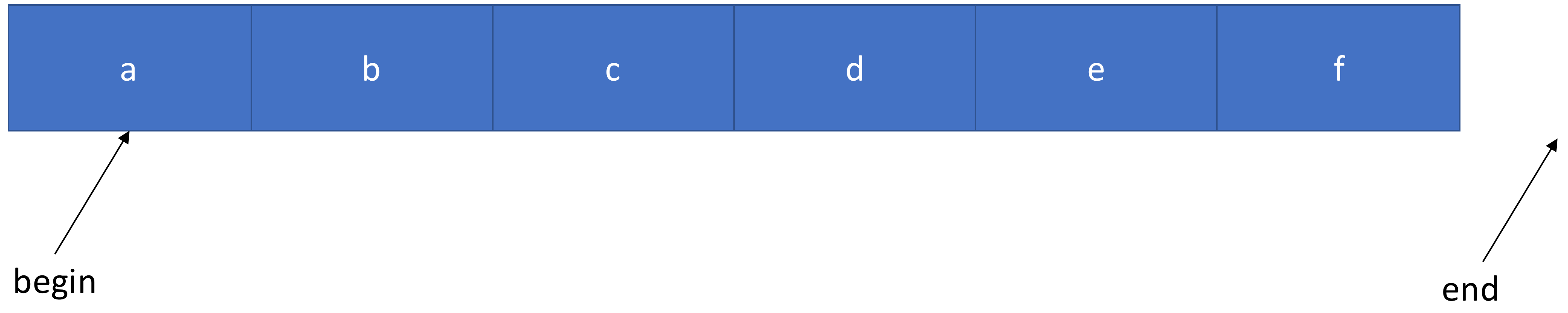
```
vector<int> v{1, 2, 3, 4, 5, 6};
v.push_back(7);
v.size();                   // return 7
v.capacity();               // return 12
```



std::string

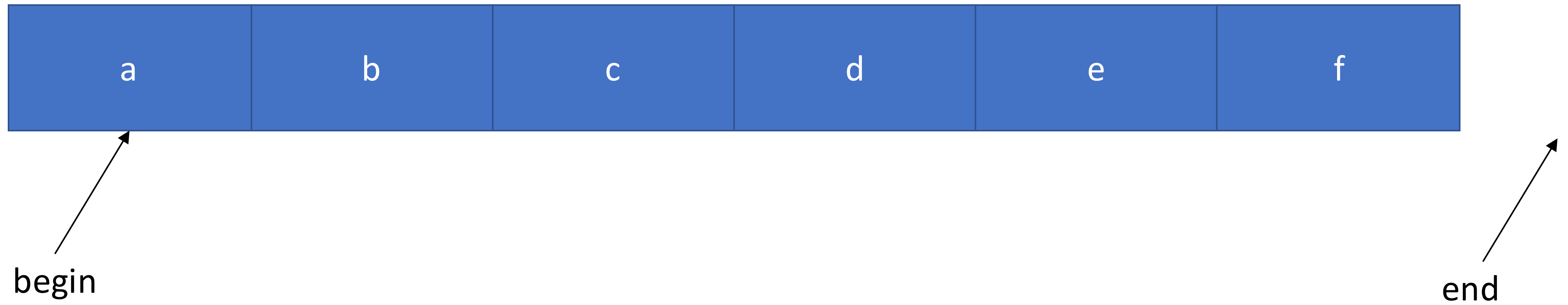
- Store and manipulates sequences of char-like objects
- The elements are stored contiguously, and can be accessed by offset
- strings in C++ are mutable (they can be changed)
- You can think of string as basically a vector<char>

```
string s{“abcdef”};
```

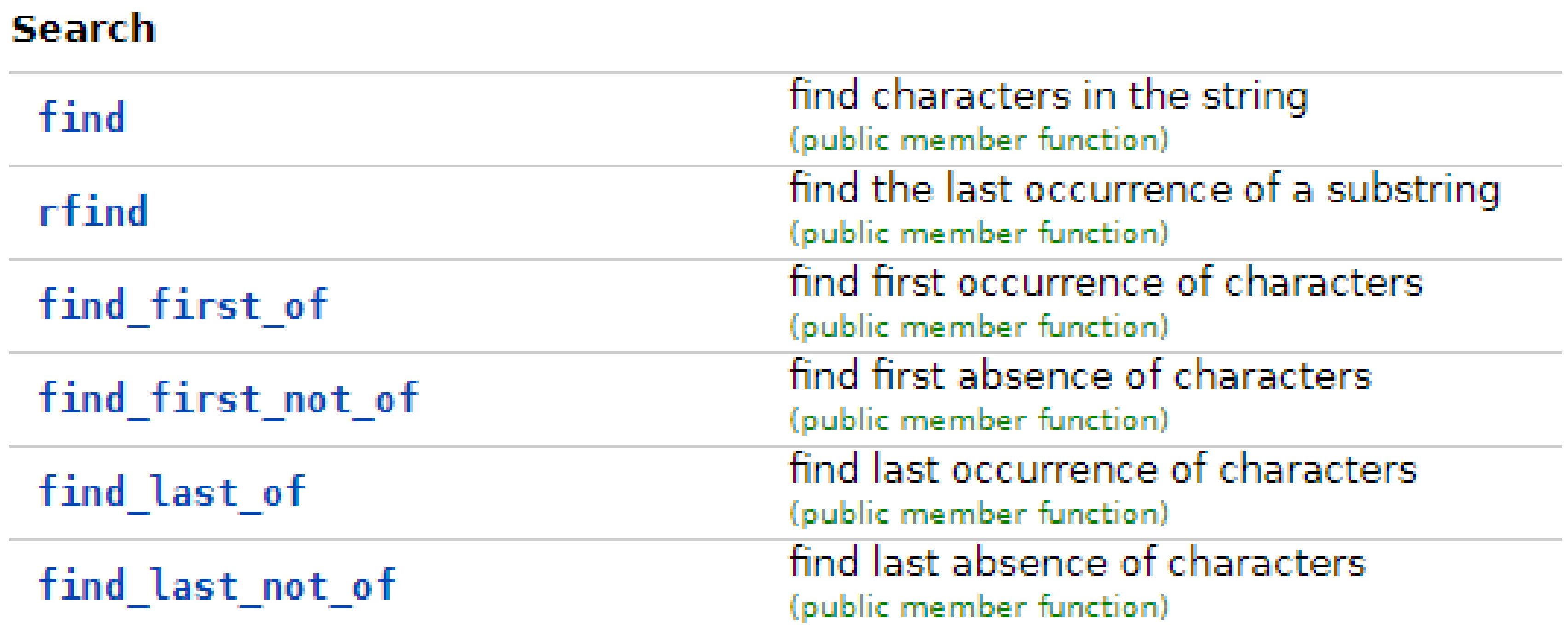


Range based for loops for string

```
string s{“abcdef”};
for (char c : s) {
    cout << c;
}

for (auto it{begin(s)}; it != end(s); it++) {
    cout << *it;
}
```



std::string – search



Search

`find`	find characters in the string (public member function)
`rfind`	find the last occurrence of a substring (public member function)
`find_first_of`	find first occurrence of characters (public member function)
`find_first_not_of`	find first absence of characters (public member function)
`find_last_of`	find last occurrence of characters (public member function)
`find_last_not_of`	find last absence of characters (public member function)

std::string - search

- return the position of the first character
- return string::npos if such substring is not found
- return type is size_type

```
string s{“abcabcd”};
size_type index1{s.find(“bc”)};       // index1 is 1
auto index2{s.find(“bc”, 2)};                 // index2 is 4
auto index3{s.find_first_not_of(“abc”)};      // index3 is 6
```

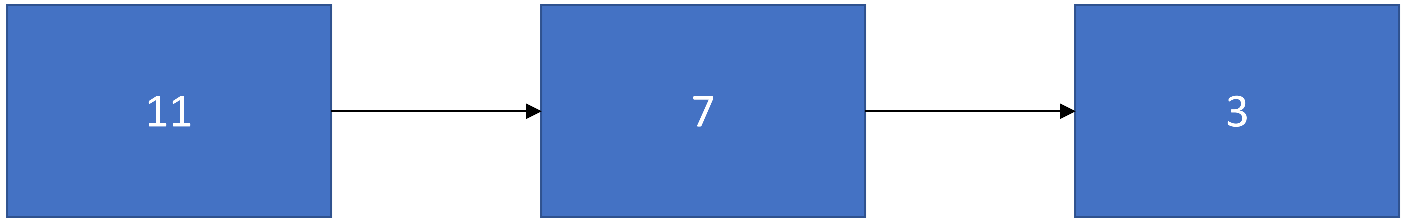
a	b	c	a	b	c	f

std::forward_list

- Is a container that supports fast insertion and removal of elements from anywhere in the container.
- Stores a dynamic length sequence
- All elements must be of same type
- Not optimized for random access
- Forward list iterates only one way
- Implemented as a singly-linked list (your lab4)

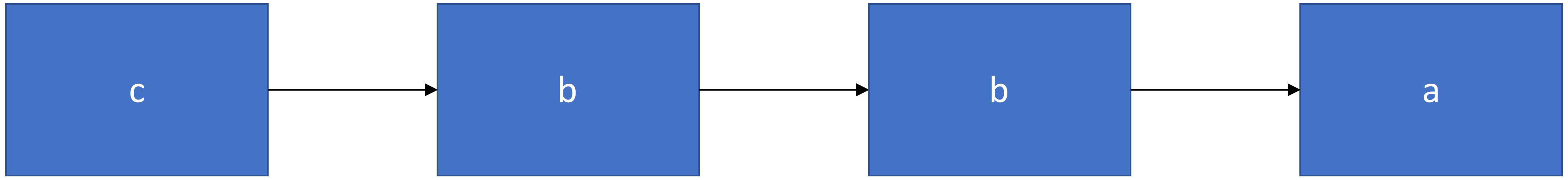
std::forward_list – push_front / front

```
#include <forward_list>
forward_list<int> my_forward_list{};

my_forward_list.push_front(3);
my_forward_list.push_front(7);
my_forward_list.push_front(11);

my_forward_list.front();     // return 11
```



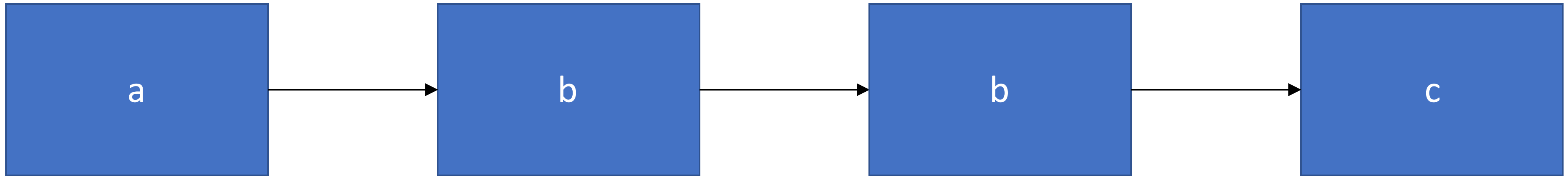
std::forward_list – initialize with string

```
string s{“cbba”};
forward_list<char> my_forward_list{begin(s), end(s)};
```



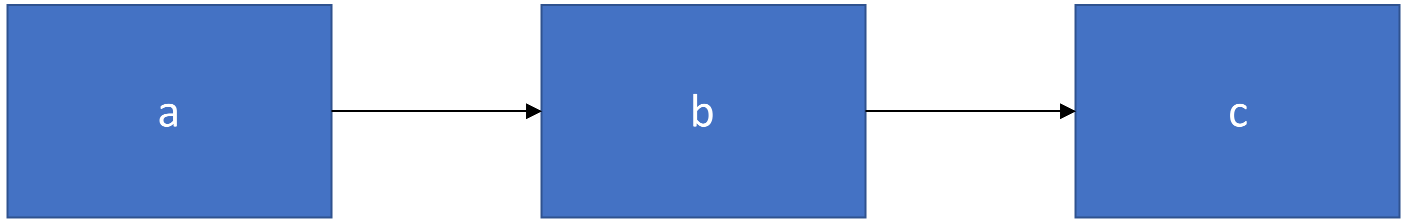
std::forward_list – sort

```
string s{“cbba”};
forward_list<char> my_forward_list{begin(s), end(s)};
my_forward_list.sort();
```



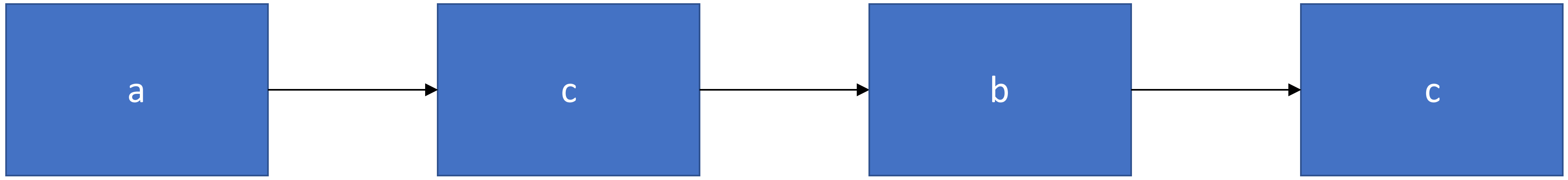
std::forward_list – unique

```
string s{“cbba”};
forward_list<char> my_forward_list{begin(s), end(s)};
my_forward_list.sort();
my_forward_list.unique();
```



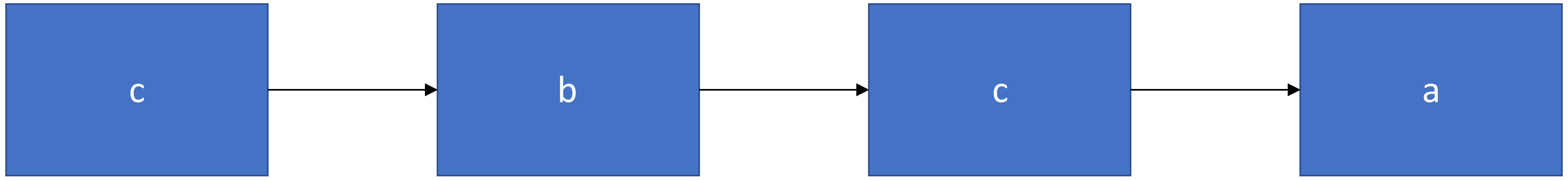
std::forward_list – insert_after

```
string s{“cbba”};
forward_list<char> my_forward_list{begin(s), end(s)};
my_forward_list.sort();
my_forward_list.unique();
my_forward_list.insert_after(begin(s), “c”);
```



std::forward_list – reverse

```
string s{“cbba”};
forward_list<char> my_forward_list{begin(s), end(s)};
my_forward_list.sort();
my_forward_list.unique();
my_forward_list.insert_after(begin(s), “c”);
my_forward_list.reverse();
```

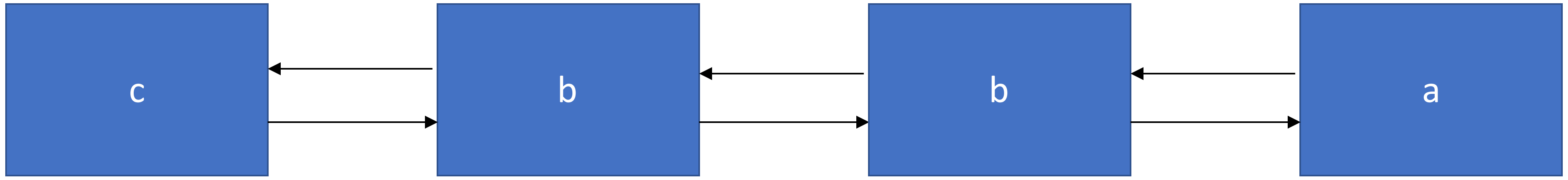


std::list

- Is a container that supports fast insertion and removal of elements from anywhere in the container.
- Stores a dynamic length sequence
- All elements must be of same type
- Not optimized for random access
- List iterates both ways, from begin to end and the other way around
- List uses more memory than forward_list

std::list – graphical representation

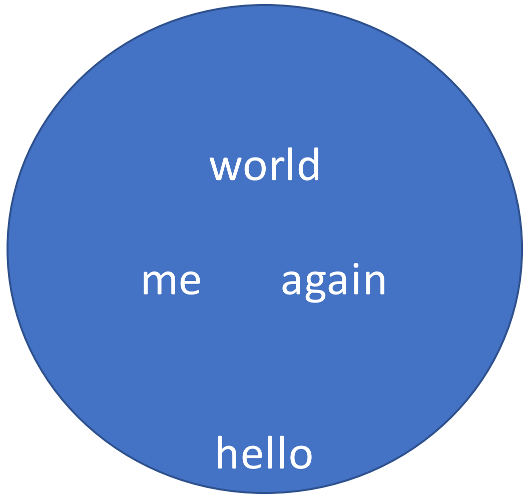
```
#include <list>
using namespace std;

string s{“cbba”};
list<char> list{begin(s), end(s)};
```



std::set

- Stores a collection of unique immutable values

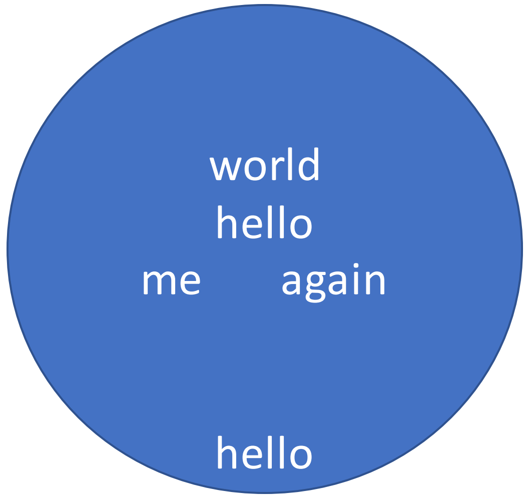
```
#include <set>
set<string> s{“hello”, “hello”, “world”, “me”, “again”};
```



std::unordered_set

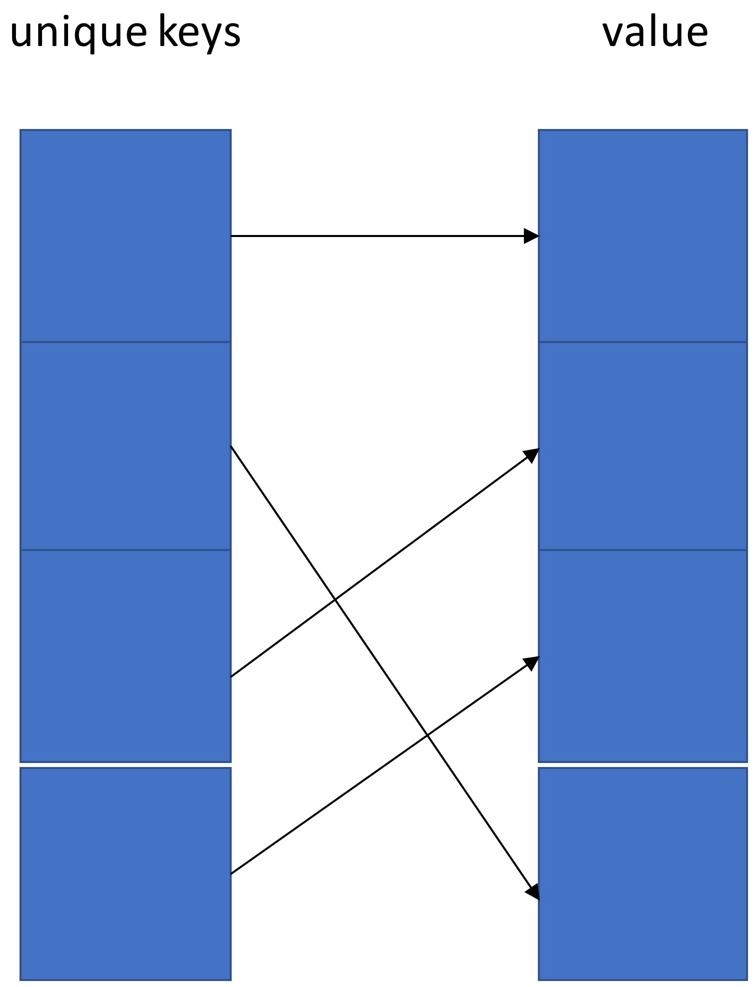
- Stores a collection of immutable values

```
#include <unordered_set>
unordered_set<string> s{“hello”, “hello”, “world”, “me”, “again”};
```



std::map

- Associative container
- Stores a collection of unique keys
- Each key is associated with a value
- Think of a set that stores pair<key, value>
- Key are sorted



std::map – constructor

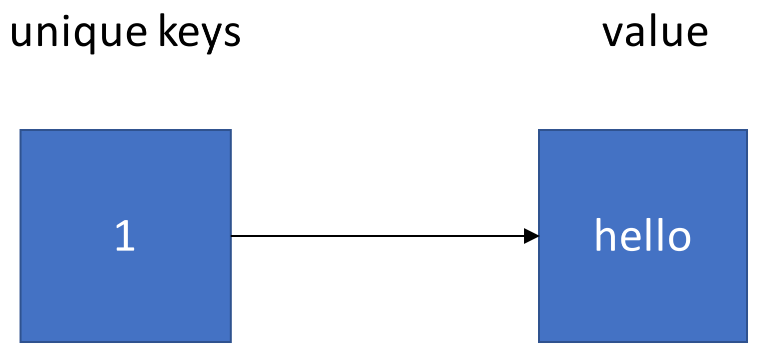
unique keys

value

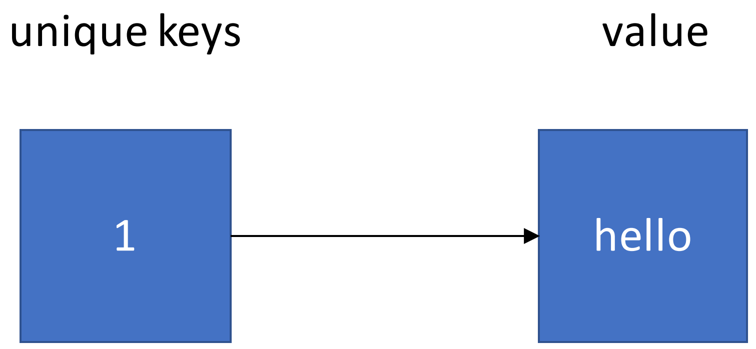
```
#include <map>
map<int, string> m{};
```

std::map – insert

```
#include <map>
map<int, string> m{};
m.insert(make_pair(1, “hello”));
```



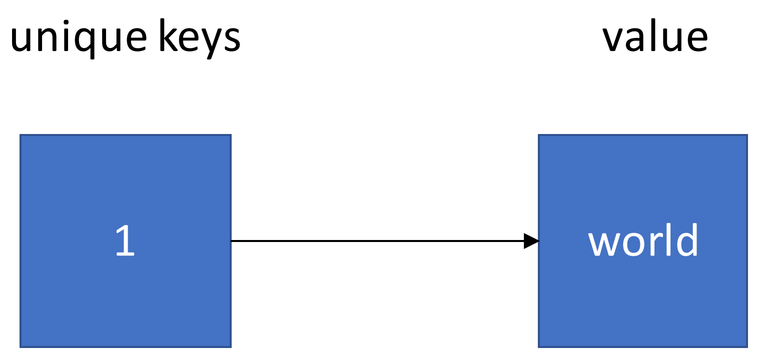
std::map – insert



```
#include <map>
map<int, string> m{};
m.insert(make_pair(1, “hello”));
// equivalent
m.insert({1, “hello”});
// compiler will deduce that its a pair<int, string> object
```

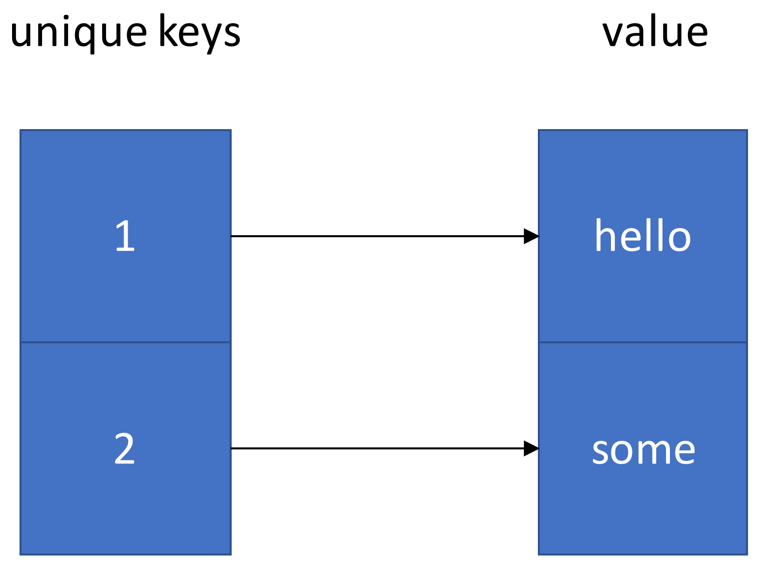
std::map – operator[] or at

```
#include <map>
map<int, string> m{};
m.insert(make_pair(1, “hello”));
m[1] = “world”;      // equivalent to m.at(1) = “world”
```



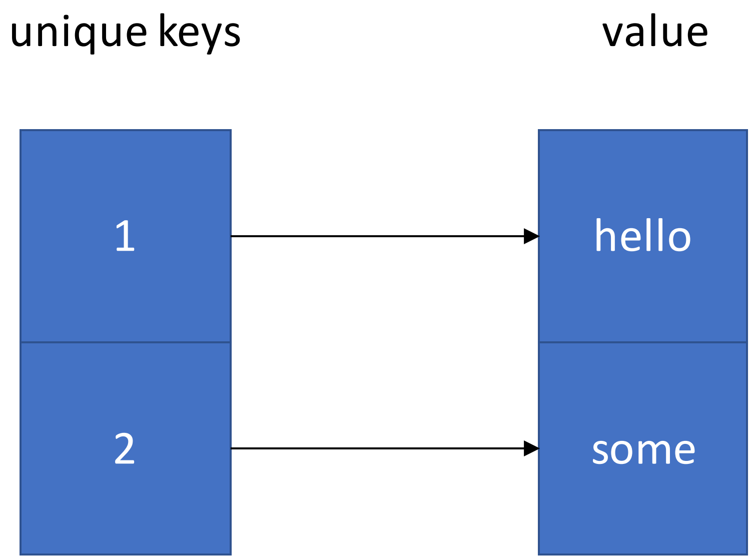
std::map – operator[]

```
#include <map>
map<int, string> m{};
m.insert(make_pair(1, “hello”));
m[1] = “world”;       // equivalent to m.at(1) = “world”
m[2] = “some”;
```



std::map – count

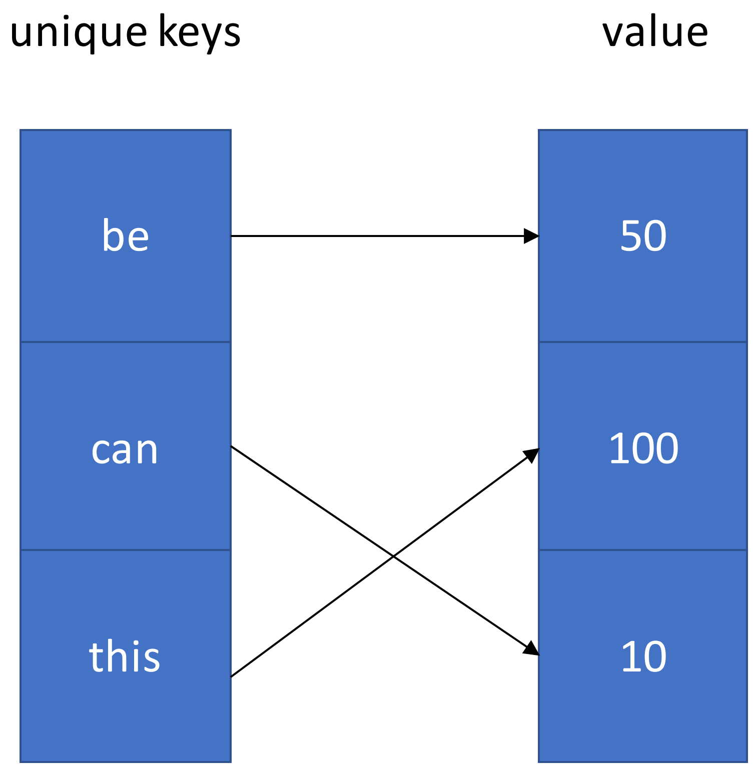
- There are no function that check if a key exists or not
- But you can use count instead



```
#include <map>
map<int, string> m{};
m.insert(make_pair(1, “hello”));
m[1] = “world”;       // equivalent to m.at(1) = “world”
m[2] = “some”;
m.count(1);           // return 1
m.count(14);          // return 0
```

std::map – iterating elements

```
map<string, int> m {
   {“this”, 1},
   {“can”, 10},
   {“be”, 50} };
for (auto it{begin(m)}; it != end(m); it++) {
   cout << it->first << “ “ << it->second << endl;
}
// equivalent
for (auto p : m) {
   cout << p.first << “ “ << p.second << endl;
}
```



std::unordered_map

- Associative container
- Stores a collection of keys
- Each key is associated with a value
- Think of a set that stores pair<key, value>
- Key are sorted