

STDLIB

The standard library

Florian Warg, Max Staff

June 1, 2017

A FEW WORDS ON TEMPLATES

- Classes and functions depending on compile-time defined types or values
- template-arguments given in <>, other arguments as usual

```
std::array<int, 50> test;
```

- arguments can sometimes be deduced from context
- since C++17 class arguments can sometimes be deduced:

```
std::sort(test.begin(), test.end());  
std::array test{1, 2, 3};  
std::array test{1, 2, 3.0}; // error
```

```
struct ListElem {  
    int content;  
    ListElem* next;  
}  
// Why reinvent the wheel?  
  
#include <forward_list>  
  
std::forward_list<char> letters {'H', 'i'};  
std::forward_list x {'H', 'i'}; // since C++17  
  
x.insert_after(x.begin() + 1, 's');  
std::cout << letters.front() << "\n";
```

```
#include <list>

std::list<char> letters {'H', 'i'};
std::list x {'e', 'l', 'l', 'o'};

x.insert(x.begin(), 'H');
if (!letters.empty()) {
    std::cout << x.front() << "\n";
}
```

```
#include <vector>

std::vector<char> l
std::vector x {'H', 'i'}; // since C++17

letters[1] = 'e';

letters.push_back('l');
letters.push_back('l');
letters.push_back('o');

std::cout << letters[4] << "\n";
```

- array, vector
- queue, deque, stack
- list, forward_list
- set, multiset
- map, multimap
- unordered_set, unordered_multiset
- unordered_map, unordered_multimap
- string, basic_string

COMPARISON: FUNCTIONALITY

Container	Element access	Iterators	Capacity	Modifiers/Operations	Comparison
	at front back data operator[]	(c)begin (c)end (c)begin	capacity reserve max_size size empty	shrink_to_fit clear insert emplace erase {push,pop}_back {push,pop}_front resize fill swap	<, <=, >, >=, ==, !=
vector					
basic_string					
array					
deque					
list					
forward_list					
(multi)set					
(multi)map					
u_(multi)set					
u_(multi)map					
stack					
queue					
priority_queue					

COMPARISON: COMPLEXITY

- always $O(1)$: `begin()`, `end()`, `empty()`, `size()`, `push_back()`

Container	Insertion/Erased				Access	Find
vector	n	n	n	1	1	n
string	n	n	n	1	1	n
list	1	1	n	1	←	n
forward_list	1	1	n	n	←	n
set/map	log(n)	log(n)	log(n)	log(n)	1	log(n)
unordered set/map	1 (n)	1 (n)	1 (n)	1 (n)	1 (n)	n
(de)queue	log(n)	log(n)	log(n)	log(n)	1	n
	Front	Iterator	Index	Back		

- what happens internally in a range-based for-loop?
- what if we want to traverse backwards?

```
std::vector fib {1, 1, 2, 3, 5, 8};  
for (auto i : fib) { std::cout << i; }
```

```
for (auto i = fib.begin(); i != fib.end(); ++i) {  
    std::cout << *i;  
}
```

- what happens internally in a range-based for-loop?
- what if we want to traverse backwards?

```
std::vector fib {1, 1, 2, 3, 5, 8};  
for (auto i : fib) { std::cout << i; }
```

```
for (auto i = fib.begin(); i != fib.end(); ++i) {  
    std::cout << *i;  
}
```

```
for (auto i = fib.rbegin(); i != fib.rend(); ++i) {  
    std::cout << *i;  
}
```

- a lot of commonly used algorithms and functionality is already given:

```
std::vector fib {1, 8, 1, 2, 34, 5, 21, 13, 3};
```

```
std::is_sorted(fib.begin(), fib.end()); // false  
std::is_sorted_until(fib.begin(), fib.end());  
// returns fib.begin() + 2
```

```
std::sort(fib.begin(), fib.end());  
std::replace(fib.begin(), fib.end(), 1, 0);
```

```
std::vector fib {1, 8, 1, 2, 34, 5, 21, 13, 3};
```

```
bool f(int i) { return i % 2 == 0; }
```

```
bool g(int i) { return i > 5; }
```

```
int h(int i) { return i * 2; }
```

```
bool k(int i, int j) { return i > j; }
```

```
std::partition(fib.begin(), fib.end(), f);
```

```
// fib: 34 8 2 1 1 5 21 13 3
```

```
std::replace_if(fib.begin(), fib.end(), g, 0);
```

```
// fib: 0 0 2 1 1 5 0 0 3
```

```
std::for_each(fib.begin(), fib.end(), h);
```

```
// fib: 0 0 4 2 2 10 0 0 6
```

```
std::sort(fib.begin(), fib.end(), k);
```

```
std::vector fib {1, 8, 1, 2, 34, 5, 21, 13, 3};

std::for_each(fib.begin(), fib.end(),
    [](int i){ return i * 2; }
);
```

```
std::vector fib {1, 8, 1, 2, 34, 5, 21, 13, 3};
```

```
std::for_each(fib.begin(), fib.end(),  
    [](int i){ return i * 2; }  
);
```

```
int a = 0, b = 1;  
std::vector<int> fib = {0, 1};  
fib.resize(20);  
std::generate(fib.begin() + 2, fib.end(),  
    [&]() { int c = a + b; a = b; b = c; return c; }  
);
```

```
std::vector fib {1, 8, 1, 2, 34, 5, 21, 13, 3};
```

```
std::all_of(fib.begin(), fib.end(),  
    [](int i) {return i < 10;});
```

```
std::any_of(fib.begin(), fib.end(),  
    [](int i) {return i < 10;});
```

```
std::none_of(fib.begin(), fib.end(),  
    [](int i) {return i < 10;});
```
