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

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

CONTAINERS - FORWARD LIST

```
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”;
```

CONTAINERS - LIST

```
#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”;
}
```

CONTAINERS - VECTOR

```
#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”;
```

OTHER CONTAINERS

- 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	operator[]	front	back	data	(c)begin	(c)end	(c)rbegin	(c)rend	empty	size	max_size	reserve	capacity	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/Erase				Access	Find
	Front	Iterator	Index	Back		
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)que(ue)	log(n)	log(n)	log(n)	log(n)	1	n

ITERATORS

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

```cpp
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;
}
```

ITERATORS

- 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;
}
```

PRE-DEFINED EXCEPTIONS

- `logic_error (invalid_argument, ...)`
- `runtime_error (overflow_error, ...)`
- `bad_typeid`
- `bad_cast`
- `bad_weak_ptr`
- `bad_function_call`
- `bad_alloc`
- `bad_exception`
- `ios_base::failure`
- ...

http://en.cppreference.com/w/cpp/error/exception

CREATE YOUR OWN EXCEPTIONS

- you can derive from `std::exception`
- your class has to implement the `what()` function

```
class MyException : public std::exception {
    virtual const char* what() const override {
        return ”My exception was thrown!\n”;
    }
};
```

HANDLING DIFFERENT EXCEPTIONS

- you can use several `catch` blocks to handle different exceptions

```
void evil(int x) {
    if (x < 0)
        throw std::exception();
    else
        throw MyException();
}
/* ... */
try { evil(1); }
catch (std::exception e) { /* ... */ }
catch (MyException e) { /* ... */ }
```

THROWING ANYTHING

- you are not limited to exception objects
- you can actually throw **anything**
- (but why would you want to do that)

```
void evil() { throw 42; }
/* ... */
try { evil(); }
catch (int i) {
    cerr << i << '\n';
}
```

CATCHING ANYTHING

- because of polymorphism you can catch all objects of a class hierarchy by reference
- you can also catch anything that is thrown

```
void evil() { throw MyException(); }
/* ... */
try { evil();
} catch (std::exception& e) {
    /* also catches MyException */
} catch (...) {
    /* catches anything */
}
```

EXCEPTION SAFETY GUARANTEES

- there are 4 levels of exception guarantees in C++
- the higher safety guarantees make it easy to recover from exceptions
- levels are in decreasing order (level 1 is the highest safety guarantee)

LEVEL 1: NO-THROW GUARANTEE

- function does not throw exceptions even in exceptional situations
- occuring exceptions are handled internally
- function will success in every situation
- keyword `noexcept` can be used to mark functions

```
int f() noexcept { return 42; }
```

level 2: strong exception safety

- also known as commit or rollback semantics
- function can fail but is guaranteed to have no side effects
- if this function fails, all data will retain their original values

level 3: basic exception safety

- also known as no-leak guarantee
- failed function can have side effects but invariants are preserved and resources are not leaked

level 4: no exception safety

- no guarantees are made ™