

KIV/CPP – Programming in C++

04. Lambda functions, exceptions, std algorithms, ranges, random

Martin Úbl

KIV ZČU

2025/2026

Lambda functions

- `#include <functional>`
- anonymous (lambda) function
- like a classic function, it has:
 - parameters
 - a return value
 - automatic or explicit
 - explicitly with the `->` type syntax
- in addition, a *capture* block (“capturing”)
 - what from the current scope will be passed and how
 - capture by value (`const!`)
 - capture by reference
- actually the `std::function` type
- called like any other function

- anonymous (lambda) function

```
auto identifier = [...] (...) -> ... {  
    ...  
};
```

- parts:
 1. capture block in []
 2. parameters in ()
 3. (return value type after ->)
 4. body in {}

Lambda functions

- empty

```
auto empty_fnc = []() {};
```

- with a return value of type `int` (explicitly)

```
auto meaning_fnc = []() -> int {  
    return 42;  
};
```

- with a return value of type `int` (explicitly II.)

```
std::function<int()> meaning_fnc = []() {  
    return 42;  
};
```

Lambda functions

- automatic return value deduction

```
auto meaning_fnc = []() {  
    return 42;  
};
```

- with the whole scope captured by value

```
auto meaning_fnc = [=]() {  
    return outer_var * another_var;  
};
```

- with the whole scope captured by reference

```
auto meaning_fnc = [&]() {  
    outer_var = 15;  
};
```

Lambda functions

- with specific variables captured by value

```
auto mod_fnc = [outer_var, another_var]() {  
    return outer_var * another_var;  
};
```

- with specific variables captured by reference

```
auto mod_fnc = [&outer_var]() {  
    outer_var = 15;  
};
```

- combination

```
auto mod_fnc = [&outer_var, another_var]() {  
    outer_var = another_var + 1;  
};
```

- capture, parameters, calls

```
auto mod_fnc = [&a, b](int c) {  
    a = b * c;  
};
```

```
mod_fnc(1);  
mod_fnc(2);  
mod_fnc(a);
```

- anonymous (lambda) function
- internally, a class is created with an overloaded operator for function calls ()
 - effectively a functor
 - captured values are attributes of the functor
- function parameters are parameters of the operator () implementation

Lambda functions

```
auto mod_fnc = [&a, b](int c) {  
    a = b * c;  
};
```

```
class mod_fnc {  
public:  
    mod_fnc(int &_a, int _b) : a(_a), b(_b) {  
    }  
    void operator()(int c) {  
        a = b * c;  
    }  
private:  
    int &a;  
    const int b;  
};
```

Lambda functions

- mutable lambda
- allows modifying a variable captured by value
- typically combined with some temporary local declaration

```
auto counter = [n = 1]() mutable {  
    return n++;  
};
```

```
std::cout << counter() << std::endl; // 1  
std::cout << counter() << std::endl; // 2  
std::cout << counter() << std::endl; // 3
```

- note: variable `n` is not declared anywhere outside, its type is deduced as `int` (from the assignment)

- constexpr (C++17) and consteval (C++20) lambda
- similarly to “ordinary” functions

```
auto calc_magic = [](int n) constexpr {  
    return n*n + 2;  
};
```

Lambda functions

- generic lambda (C++14)
- type deduction for multiple call instances

```
auto multiply_2 = [](auto a) {  
    return a + a;  
};
```

```
multiply_2(10);    // returns 20
```

```
std::string a{"knock"};  
multiply_2(a);    // returns "knockknock"
```

Lambda functions

- templated lambda (C++20)
- a more general version of a generic lambda

```
auto multiply_2 = []<typename T>(T a) {  
    return a + a;  
};
```

```
multiply_2(10);    // returns 20
```

```
std::string a{"knock"};  
multiply_2(a);    // returns "knockknock"
```

- useful, for example, for passing template classes
- more in the lecture on templates

Lambda functions

- binding
- binds a function with arguments
- so-called *placeholders* can be used for arguments that we do not want to bind
 - `std::placeholders`
 - e.g. `std::placeholders::_1, _2, ..., _20`
 - the number does not correspond to the argument, but to the order of the placeholder used

```
auto powf2 = std::bind(std::powf,  
                      std::placeholders::_1, 2.0f);
```

```
powf2(5.0f); // 25
```

Exceptions

- exceptions
- a mechanism for handling exceptional (error) events
- try, catch, throw
- but finally is missing
 - relies on proper use of static allocation, RAII, ...
- we can *throw* anything
 - `throw 20;`
- however, it is better to use descendants of `std::exception` from the STL
 - `#include <stdexcept>`
 - `throw std::runtime_error("Error!");`
- an already existing instance of anything can be thrown

```
std::runtime_error ex{"Error"};  
throw ex;
```

Exceptions

- `std::exception` contains the `what` method returning an error description
- can be caught by value or by reference
 - a reference is polymorphic (preferred)
- multiple kinds of exceptions can be caught, evaluated from the top

```
try {  
    ...  
}  
catch (std::runtime_error rex) {  
    std::cout << rex.what() << std::endl;  
}  
catch (std::exception& ex) {  
    std::cout << ex.what() << std::endl;  
}
```

- if we do not intend to handle the exception based on its kind, three dots can be used

```
try {  
    ...  
}  
catch (...) {  
    // an exception? naah...  
}
```

- note: can also be used as a fallback in the last catch block

Exceptions

- rethrow
- “rethrows” the exception to parent scopes (to outer catch blocks)

```
catch (MyException& ex) {  
    // I got this  
}  
catch (std::bad_alloc& ex) {  
    // I don't know how to handle this  
    throw;  
}  
catch (...) {  
    // I don't know what's going on  
}
```

Exceptions

- the `noexcept` keyword
 - the function will not throw an exception
- functions
 - *potentially throwing*
 - without `noexcept`
 - `noexcept(false)`
 - *non-throwing*
 - `noexcept`
- what if a *non-throwing* function throws an exception?
 - if the exception was thrown inside an inner try-catch block, handle it there
 - if it should be propagated out of the function, stop and call `std::terminate`
- useful, for example, for constructors – optimization and safety (move constructors)

```
class Circle {  
    public:  
        Circle() noexcept {  
        }  
  
        Circle(const Circle& other) noexcept {  
        }  
  
        Circle(Circle&& other) noexcept {  
        }  
};
```

- historically, the performance impact of exceptions was discussed
- compilers can now optimize
- the so-called *Zero-Cost* model
 - no added overhead when an exception does not occur
 - high overhead if an exception occurs
 - RTTI, cache miss, ...
- generally faster than many `if` blocks
- try to use the exception mechanism minimally
- definitely not for ordinary control flow

- note: exceptions between threads
- `std::exception_ptr`
- `std::current_exception()`
- `std::rethrow_exception(ex)`

- `#include <algorithm>`
- a set of standard algorithms for common operations
- vector generation, shuffling, permutations, sorting
- heap, element search, minimum and maximum search
- conditional copy and erase, swap
- and many others...
- <https://en.cppreference.com/w/cpp/algorithm>
- we will mainly show the principle of working with them
 - parameters
 - conventions

- `std::fill`
- fills a container with a specified element

```
std::array<int, 10> arr;  
std::vector<int> vec;  
vec.resize(10);
```

```
// fill with zeros  
std::fill(arr.begin(), arr.end(), 0);  
std::fill(vec.begin(), vec.end(), 0);
```

- `std::generate`
- generates container elements according to a “recipe”

```
std::vector<int> a;  
a.resize(10);
```

```
std::generate(a.begin(), a.end(),  
              [n = 0]() mutable { return n++; });
```

- this can be done better directly using `std::iota`

```
std::iota(a.begin(), a.end(), 0);
```

STD algorithms

- `std::copy`
- copies a container
- typically in combination with a so-called *inserter*
 - `std::inserter`
 - `std::back_inserter`
 - `std::front_inserter`

```
std::vector<int> a{ 1,2,3 };  
std::vector<int> b{ 7,8,9 };
```

```
std::copy(b.begin(),  
          b.end(),  
          std::back_inserter(a));
```

```
// a = { 1,2,3,7,8,9 }
```

```
std::vector<int> a{ 1,2,3 };  
std::vector<int> b{ 7,8,9 };  
  
std::copy(b.begin(),  
          b.end(),  
          std::inserter(a, a.begin()));  
  
// a = { 7,8,9,1,2,3 }
```

STD algorithms

- `std::copy_if`
- conditional copy

```
std::vector<int> a{ 1,2,3 };
```

```
std::vector<int> b{ 7,8,9 };
```

```
auto pred = [](const int& n){ return n != 7; };
```

```
std::copy_if(b.begin(),  
            b.end(),  
            std::back_inserter(a),  
            pred);
```

```
// a = { 1,2,3,8,9 }
```

STD algorithms

- `std::remove_if`
- conditional erase
- returns an iterator
- must be combined with the corresponding erase function of the container

```
std::vector<int> a{ 1,2,3,4,5 };
```

```
auto pred = [](const int& g) {  
    return g % 2 == 0;  
};
```

```
a.erase(std::remove_if(a.begin(), a.end(),  
                        pred),  
        a.end());  
// a = { 1,3,5 }
```

- heap
- not a separate data type (it is an ADT)
 - built over an existing container (vector)
- `std::make_heap`
- `std::pop_heap`, `std::push_heap`
 - only move the top heap element to the end (pop)
 - respectively insert an element into the heap (push)
- inserting an element: `push_back` + `std::push_heap`
- selecting an element: `std::pop_heap` + `back` (+ `std::pop_back`)

- other algorithms
 - <https://en.cppreference.com/w/cpp/algorithm>
- the principles are then similar

- constrained algorithms (since C++20)
 - <https://en.cppreference.com/w/cpp/algorithm/ranges>
- principles similar to ordinary algorithms, but they work over so-called ranges
- see below

- Ranges (since C++20)
- `#include <ranges>`
- literally “ranges”
- identified by a beginning and an end
 - respectively by an initial iterator (begin)
 - and a “sentinel” (end)
- they are iterable
- they can have the same variants as iterators (forward, random access, ...)
- STL containers are also ranges

- Ranges (since C++20)
- this is a generalization of the concept; at first glance, it is not really anything new
- a *view* can be built over a range
 - it is a view of data
 - a view can be modified, rearranged, filtered
 - it does not change the underlying range
- views can be chained with the *pipe* operator |
- more readable data processing

- Example: range/views for filtering even numbers and transformation (doubling)

```
std::vector<int> numbers = {1, 2, 3, 4, 5, 6};
```

```
auto results = numbers
| std::views::filter([](int n){
    return n % 2 == 0;
})
| std::views::transform([](int n){
    return n * 2;
});
```

- results is a range that can be iterated over (e.g. with a range-based for loop)

- or more readably

```
std::vector<int> numbers = {1, 2, 3, 4, 5, 6};  
auto even = [](int n) { return n%2 == 0 };  
auto mul2 = [](int n) { return n*2; };  
  
auto results = numbers  
    | std::views::filter(even)  
    | std::views::transform(mul2);
```

```
auto results = numbers
    | std::views::filter(even)
    | std::views::transform(mul2);
```

- a view is created from the numbers
- it enters as the left side of the `|` operator, the right side is the filter functor
- the output is then a view, which continues as the left side of the next `|` operator
- then the right side is again a functor and the output is a view

- *projections*
- invocation of an expression for each element
- can replace a lambda function or a custom comparator

```
struct Person {  
    std::string name;  
    unsigned short get_age();  
};
```

```
std::vector<Person> ppl = ...;
```

```
std::ranges::sort(ppl, {}, &Person::name);  
std::ranges::sort(ppl, {}, &Person::get_age);
```

Ranges

- `std::views::zip`
- “pairs” elements of multiple arrays
 - iterates “over two arrays at once”

```
std::vector<int> numbers1 = {1, 2, 3};
std::vector<int> numbers2 = {100, 200, 300};

for (auto p : std::views::zip(numbers1, numbers2))
    std::cout << std::get<0>(p)
                << " - "
                << std::get<1>(p)
                << std::endl;
}

// 1 - 100
// 2 - 200
// 3 - 300
```

- `std::views::reverse`
- reverses the order of elements

```
std::vector<int> numbers = {1, 2, 3};  
auto v = numbers | std::views::reverse;
```

```
std::cout << *v.begin() << std::endl;  
// 3
```

- `std::views::drop`
- skips elements

```
std::vector<int> numbers = {1, 2, 3, 4, 5};  
auto v = numbers | std::views::reverse  
                | std::views::drop(3);  
  
std::cout << *v.begin() << std::endl;  
// 2
```

- `std::views::take`
- limits the size

```
std::vector<int> numbers = {1, 2, 3, 4, 5};  
auto v = numbers | std::views::drop(1)  
                  | std::views::take(2);  
  
for (auto c : v)  
    std::cout << c << std::endl;  
// 2 3
```

- `std::views::iota`
- number generator

```
for (auto c : std::views::iota(0, 5))  
    std::cout << c << std::endl;  
// 0 1 2 3 4
```

- pretty useful for static work distribution with parallel STL (later lectures)

- `std::views`
- views – they do not modify the container
- they do not allocate new memory
- they only “redirect” or “adjust” the read operation
- faster, more intuitive interface

- `#include <random>`
- random number generation
- the `srand()` and `rand()` functions from C generate only a uniform distribution (LCG, MersenneTwister, ...)
- the STL in C++ additionally supports:
 - access to a source of true randomness (TRNG)
 - other pseudorandom generators (PRNG)
 - other distributions:
 - normal
 - exponential
 - and others...
 - integration into STD algorithms

- basic components
- source of true randomness (TRNG, if available)
 - `std::random_device`
 - generation can be slow (entropy, ...)
- source of a pseudorandom sequence (PRNG)
 - e.g. `std::mt19937`
 - or `std::default_random_engine` can be used
- distribution generator
 - e.g. `std::uniform_int_distribution`
- typicky:
 - TRNG is used to initialize PRNG
 - PRNG + distribution is used for generation

- TRNG, PRNG, and distributions are functors

```
// TRNG
```

```
std::random_device r;
```

```
// PRNG
```

```
std::default_random_engine e1(r());
```

```
// distribution
```

```
std::uniform_int_distribution<int> unif(1, 6);
```

```
// generate numbers
```

```
int num = unif(e1);
```

- properties of random variables
 - KMA/PSA – basic analysis
 - KMA/MSM – multidimensional problems
- internals of pseudorandom number generators
 - KIV/VSS
- and others...