KIV/CPP – Programování v jazyce C++

04. Lambda funkce, výjimky, std algoritmy, ranges, random

Martin Úbl

KIV ZČU

2023/2024

Lambda funkce

- `#include <functional>`
- anonymní (lambda) funkce
- jako klasická funkce má:
  - parametry
  - návratovou hodnotu
    - automaticky nebo explicitně
    - explicitně syntaxe `s -> type`
- navíc *capture* blok („zachytávací“)
  - co z aktuální scope se bude předávat a jak
  - zachycení hodnotou (`const`!)
  - zachycení referencí
- ve skutečnosti typ `std::function`
- volá se jako každá jiná funkce

Lambda funkce

- anonymní (lambda) funkce

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

- části:
  1. capture blok v []
  2. parametry v ()
  3. (typ návratové hodnoty za ->)
  4. tělo v {}

Lambda funkce

- prázdná

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

- s návratovou hodnotou typu `int` (explicitně)

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

- s návratovou hodnotou typu `int` (explicitně II.)

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

Lambda funkce

- automatická dedukce návratové hodnoty

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

- se zachycením celé scope hodnotou

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

- se zachycením celé scope referencí

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

Lambda funkce

- se zachycením konkrétních proměnných hodnotou

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

- se zachycením konkrétních proměnných referencí

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

- kombinace

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

Lambda funkce

- capture, parametry, volání

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

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

Lambda funkce

- anonymní (lambda) funkce
- interně se vytvoří třída s přetíženým operátorem pro funkční volání ()
  - tedy vlastně funktor
  - zachycené hodnoty jsou atributy funktoru
- parametry funkce jsou parametry implementace operátoru ()

Lambda funkce

```
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 funkce

- `mutable` lambda
- dovoluje modifikovat proměnnou zachycenou hodnotou
- typicky v kombinaci s nějakou dočasnou lokální deklarací

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

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

- pozn.: proměnná `n` není vně nikde deklarována, její typ je dedukován na `int`

Lambda funkce

- `constexpr` (C++17) a `consteval` (C++20) lambda
- obdobně jako u „běžných“ funkcí

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

Lambda funkce

- generic lambda (C++14)
- dedukce typů pro více instancí volání

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

multiply_2(10);  // vraci 20

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

Lambda funkce

- templated lambda (C++20)
- obecnější verze generic lambdy

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

multiply_2(10);  // vraci 20

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

- hodí se např. pro předávání šablonových tříd
- více v přednášce o šablonách

Lambda funkce

- binding – svázání
- sváže funkci s argumenty
- lze použít tzv. *placeholdery* pro argumenty, které nechceme svázat
  - `std::placeholders`
  - např. `std::placeholders::_1`, `_2`, ..., `_20`
  - číslo nekoresponduje s argumentem, ale s pořadím použitého placeholderu

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

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

Výjimky

- výjimky
- mechanismus pro ošetření výjimečných (chybových) událostí
- `try`, `catch`, `throw`
- ale chybí `finally`
  - spoléhá se na správné použití statické alokace, RAII, ...
- můžeme *vyhodit* cokoliv
  - `throw 20;`
- lepší je však použít potomky `std::exception` z STL
  - `#include <stdexcept>`
  - `throw std::runtime_error("Chyba!");`
- lze vyhodit již existující instanci čehokoliv

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

Výjimky

- `std::exception` obsahuje metodu `what` vracející popis chyby
- odchytávat lze hodnotou i referencí
  - reference je polymorfní (preferujeme)
- lze odchytávat více druhů výjimek, vyhodnocení shora

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

Výjimky

- pokud nemáme v úmyslu výjimku nijak ošetřovat na základě jejího druhu, lze použít tři tečky

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

- pozn. lze i jako fallback v posledním `catch` bloku

Výjimky

- rethrow
- „znovu vyhodí“ výjimku do nadřazených scope (do vnějších `catch` bloků)

```
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
}
```

Výjimky

- klíčové slovo `noexcept`
  - funkce nebude házet výjimku
- funkce
  - *potentially throwing*
    - bez `noexcept`
    - `noexcept(false)`
  - *non-throwing*
    - `noexcept`
- co když *non-throwing* funkce vyhodí výjimku?
  - pokud byla výjimka vyhozena v rámci vnitřního try-catch bloku, ošetřit tam
  - pokud by měla být propagována z funkce ven, zastavení a volání `std::terminate`
- hodí se např. pro konstruktory – optimalizace

Výjimky

```
class Circle {
  public:
    Circle() noexcept {
    }

    Circle(const Circle& other) noexcept {
    }

    Circle(Circle&& other) noexcept {
    }
};
```

Výjimky

- historicky se u výjimek diskutoval dopad na výkon
- nyní kompilátory zvládají optimalizovat
- tzv. *Zero-Cost* model
  - bez přidané režie, když výjimka nenastane
  - s velkou režií, pokud výjimka nastane
    - RTTI, cache miss, ...
- v zásadě rychlejší než spousty `if` bloků
- snaha používat mechanismus výjimek minimálně
- rozhodně ne k běžnému řízení toku instrukcí

Výjimky

- pozn.: výjimky mezi vlákny
- `std::exception_ptr`
- `std::current_exception()`
- `std::rethrow_exception(ex)`

STD algoritmy

- `#include <algorithm>`
- sada standardních algoritmů pro obvyklé operace
- generování vektorů, zamíchání, permutace, řazení
- halda, hledání prvků, hledání minima a maxima
- podmíněná kopie a mazání, swap
- a spousty dalších...
- `https://en.cppreference.com/w/cpp/algorithm`
- ukážeme si hlavně princip práce s nimi
  - parametry
  - konvence

STD algoritmy

- `std::fill`
- vyplní kontejner zadaným prvkem

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

// vyplnit nulami
std::fill(arr.begin(), arr.end(), 0);
std::fill(vec.begin(), vec.end(), 0);
```

STD algoritmy

- `std::generate`
- vygeneruje prvky kontejneru podle „návodu“

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

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

- tohle lze lépe přímo pomocí `std::iota`

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

STD algoritmy

- `std::copy`
- zkopíruje kontejner
- typicky v kombinaci s tzv. *inserterem*
  - `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 algoritmy

```
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 algoritmy

- `std::copy_if`
- podmíněná kopie

```
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 algoritmy

- `std::remove_if`
- podmíněné mazání
- vrací iterátor
- nutná kombinace s příslušnou mazací funkcí kontejneru

```
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 }
```

STD algoritmy

- halda
- není samostatný datový typ (je to ADT)
  - tvoří se nad existujícím kontejnerem (vector)
- `std::make_heap`
- `std::pop_heap`, `std::push_heap`
  - pouze přesouvají vrchní prvek haldy na konec (pop)
  - resp. zatřizují prvek do haldy (push)
- vložení prvku: `push_back` + `std::push_heap`
- výběr prvku: `std::pop_heap` + `back` (+ `std::pop_back`)

STD algoritmy

- další algoritmy
  - `https://en.cppreference.com/w/cpp/algorithm`
- principy jsou pak už podobné

STD algoritmy

- constrained algoritmy (od C++20)
  - `https://en.cppreference.com/w/cpp/algorithm/ranges`
- principy podobné jako obyčejné algoritmy, ale fungují nad tzv. ranges
- viz dále

Ranges

- Ranges (od C++20)
- `#include <ranges>`
- dosl. „rozsahy“
- identifikované počátkem a koncem
  - resp. počátečním iterátorem (begin)
  - a „zarážkou“ (end)
- jsou iterovatelné
- mohou být stejných variant, jako iterátory (forward, random access, ...)
- STL kontejnery jsou také ranges

Ranges

- Ranges (od C++20)
- jde o zobecnění konceptu, vlastně nejde na první pohled o nic nového
- nad range lze vystavět *view*
  - jde o pohled na data
  - view lze modifikovat, přeházet, promazat
  - nemění podlehlý range
- view lze řetězit *pipe* operátorem |
- čitelnější zpracování dat

Ranges

- Příklad: range/views pro filtraci sudých čísel a transformaci (zdvojnásobení)

```
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` je range, nad kterou lze iterovat (např. range-based for cyklem)

Ranges

- nebo čitelněji

```
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);
```

Ranges

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

- z čísel se vytvoří view
- vstupuje jako levá strana operátoru |, pravou stranou je funktor filtru
- výstupem je pak view, ten dále pokračuje jako levá strana dalšího operátoru |
- pak je identicky pravou stranou funktor a výstupem view

Ranges

- *projekce*
- invokace výrazu pro každý prvek
- může nahradit lambda funkci nebo vlastní komparátor

```
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`
- „páruje“ prvky více polí

```
std::vector<int> cisla1 = {1, 2, 3};
std::vector<int> cisla2 = {100, 200, 300};

for (auto p : std::views::zip(cisla1, cisla2)){
    std::cout << std::get<0>(p)
              << " - "
              << std::get<1>(p)
              << std::endl;
}
// 1 - 100
// 2 - 200
// 3 - 300
```

Random

- `#include <random>`
- generování náhodných čísel
- funkce `srand()` a `rand()` z C generují pouze rovnoměrné rozložení (LCG, MersenneTwister, ...)
- STL v C++ zvládá navíc:
  - přístup ke zdroji skutečné náhody (TRNG)
  - jiné generátory pseudonáhody (PRNG)
  - další rozdělení:
    - normální
    - exponenciální
    - a další...
  - integraci do STD algoritmů

Random

- základní složky
- zdroj skutečné náhody (TRNG, pokud je dostupný)
  - `std::random_device`
  - generování může být pomalé (entropie, ...)
- zdroj pseudonáhodné posloupnosti (PRNG)
  - např. `std::mt19937`
  - nebo lze použít `std::default_random_engine`
- generátor rozdělení
  - např. `std::uniform_int_distribution`
- typicky:
  - TRNG se použije pro inicializaci PRNG
  - pro generování se použije PRNG + rozdělení

- TRNG, PRNG a rozdělení jsou funktory

```
// 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);
```

Random

- vlastnosti náhodných veličin
  - KMA/PSA – základní analýza
  - KMA/MSM – vícerozměrné problémy
- vnitřnosti generátorů pseudonáhodných čísel
  - KIV/VSS
- a jiné...