NPRG041 – C++

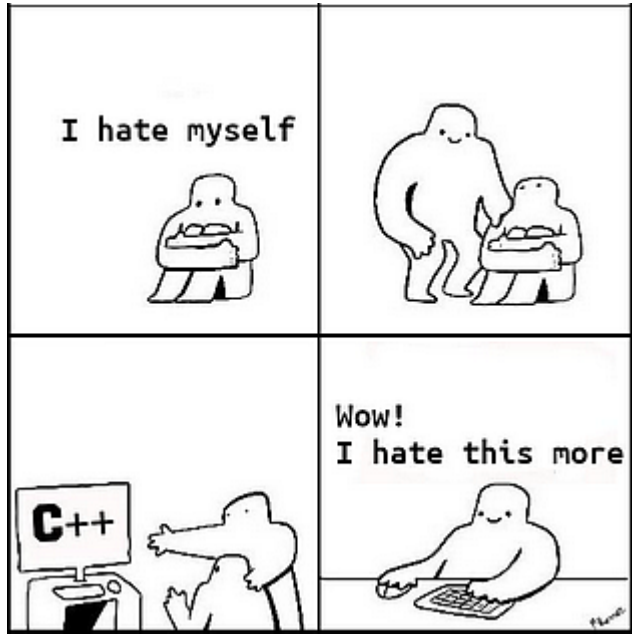
cvičení – Jiří Klepl

mattermost: ulita/2425ZS: nprg041-cpp-klepl (inv na SIS nástěnce)

Klepl@d3s.mff.cuni.cz

Agenda

- Třídění
- Funktory, lambda funkce
- Algoritmy: std::sort, std::find, ...
- Speciální iterátory: std::back_inserter, ...



Uspořádání set/map podle operator<

- Máme třídu s 3 položkami: std::string name, int age, double height
- Zadáno na recodexu
- Upravte soubor https://www.ksi.mff.cuni.cz/teaching/nprg041-klepl-web/data/05/user_data.hpp
- Minipříklad 1:
  - Chceme definovat operator< s následucímími vlastnostmi:
    1. Obě porovnávaná čísla pouze čte
    2. Prvním kritériem je věk, potom délka jména a nakonec jméno lexikograficky
- Minipříklad 2:
  - Chceme definovat komparátor tak, aby porovnával podle jména/věku/výšky; kritérium zadáno v konstruktoru komparátoru

Useful triky

Zkrácená definice pomocí std::tie

```
struct XY {
  bool operator<(const XY& rhs)
                 const {
    return tie(x, y)
         < tie(rhs.x, rhs.y);
  }

  int x, y;
};

set<XY> points{
  {0, 5}, {3, 5}, {5, 2},
  {3, 4}, {1, 4}, {2, 2}
};

for (auto&& [x, y] : points)
  cout << x << ", " << y << endl;
```

Implicitně definovaný structured binding pro struct

Požádáme compiler (c++20) implementuje vše za nás

```
struct XY {
  auto operator<=>(const XY& rhs) const
      = default;

  int x, y;
};

set<XY> points{
  {0, 5}, {3, 5}, {5, 2},
  {3, 4}, {1, 4}, {2, 2}
};

for (auto&& [x, y] : points)
  cout << x << ", " << y << endl;
```

Přátelská funkce

⚠ není metoda

```
struct XY {
  friend bool operator<(const XY& lhs, const XY& rhs)
  {
    return tie(lhs.x, rlhs.y) < tie(rhs.x, rhs.y);
  }
private:
  int x, y;
};
set<XY> points{
  {0, 5}, {3, 5}, {5, 2},
  {3, 4}, {1, 4}, {2, 2}
};

for (auto&& [x, y] : points)
  cout << x << ", " << y << endl;
```

Komparátor (ukázka funktoru)

```
struct XY { int x, y; };

struct MyCmp {
  bool operator()(const XY& lhs, const XY& rhs) const {
    return tie(lhs.x, lhs.y) < tie(rhs.x, rhs.y);
  }
};

set<XY, MyCmp> points{
  {0, 5}, {3, 5}, {5, 2},
  {3, 4}, {1, 4}, {2, 2}
};

for (auto&& [x, y] : points)
  cout << x << ", " << y << endl;
```

Parametry

Volatelný objekt

Lambda: compilerem napsaný funktor

```
struct XY { int x, y; };

auto cmp = [](const XY& lhs, const XY& rhs) {
  return std::tie(lhs.x, lhs.y) < std::tie(rhs.x, rhs.y);
};

set<XY, decltype(cmp)> points{
  {0, 5}, {3, 5}, {5, 2},
  {3, 4}, {1, 4}, {2, 2}
};

for (auto&& [x, y] : set) {
  cout << x << ", " << y << endl;
}
```

Lambda

```
[ captures ] ( params )opt mutableopt -> rettypeopt { body }
```

- **[captures]**: Seznam hodnot (inicializace), které lambda používá:
  - **[x]** zkopíruje hodnotu proměnné x
  - **[&x]** vytvoří referenci na proměnnou x
  - První můžeme použít **[=]**, **[&]** pro imlicitní copy/ref
  - this (capture objektu v metodě) je výjimka:
    [this] -> reference; **[*this]** -> kopie
  - Zobecněný capture **[nazev=hodnota]**
    - Vytvoří úplně novou proměnnou
- **(parameters)**: běžné parametry jako u funkcí
- **mutable**
  - příznak, kterým označíme,
    že nakopírované hodnoty lze měnit, jinak jsou const
- **-> rettype**: explicitní zadání návratového typu

```
int x = 3;

auto lambda1 =
       [x](int n){return x*n;};
auto lambda2 =
       [&x](int n){return x*n;};

x = 5;

int y = lambda1(2); // y == 6 ⚠
int z = lambda2(2); // z == 10 ⚠
```

Lambda přeložena na funktor

Lambda je zkratka za funktor → ⚠️ Dvě stejné lambdy mají různé typy

```
[ captures ] ( params ) -> rettype { statements; }
```

```
class ftor {
private:
  CaptTypes captures_;
public:
  ftor( CaptTypes captures ) : captures_( captures ) {}
  rettype operator() ( params ) { statements; }
};
```

#include <algorithm>

Standardní algoritmy

- **it std::find(it beg, it end, const T& val)**
  - **std::find_if(it beg, it end, pred(const T&))**
- **int std::count(it beg, it count, T& val)**
  - **std::count_if(it beg, it end, pred(const T&))**
- **std::for_each(it beg, it end, fnc(T&))**
- **std::sort(it beg, it end, cmp(const T&, const T&))**
- **std::copy(it beg, it end, it out_it), std::move(it beg, it end, it out_it)**
  - **std::transform(it beg, it end, it out_it, T' fnc(const T&))**
  - **std::transform(it beg, it end, it beg2, it out_it, T' fnc(const T1&, const T2&))**

Spojování kontejnerů

- **remove**, **remove_if** – ⚠️ přesun (move) nálezů na konec, vrátí it na první
  - Skutečné smazání až zkrácením kontejneru (např. **vector.erase(it_returned, end)**)
- Mnoho dalších na https://en.cppreference.com/w/cpp/algorithm

Pokročilé druhy iterátorů

#include <iterator>

Jde je například používat v algoritmech

- Adaptéry iterátorů
  - **std::back_inserter(container&)**, **std::front_inserter(container&)**
    - Přídává na konec/začátek zadaného kontejneru
  - **std::inserter(container&)** – to samé, ale např. u map
- **std::istream_iterator<T>(in)**
  - iteruje vstup, jako bychom dělali: T value; while(in >> value) …
- **std::ostream_iterator<T>(out[, delim])**
  - Do streamu out píše prvky (oddělené delim – typicky ' ' nebo '\n')

Příklad: algorithms

- **Zadání**: vyplňte soubor algorithms.cpp https://www.ksi.mff.cuni.cz/teaching/nprg041-klepl-web/data/05/algorithms.cpp
- Výsledek nahrajte do **labs/algorithms**