NPRG 041 – cvičení 5
Programování v C++

Jiří Klepl

mail:
klepl@d3s.mff.cuni.cz

mattermost:
https://ulita.ms.mff.cuni.cz/mattermost/ar2324zs/channels/nprg041-cpp-klepl

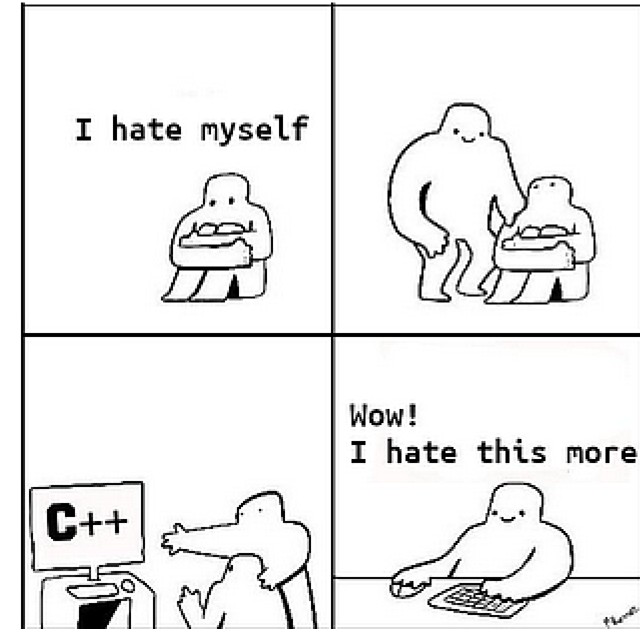
Feedback k úloze Práce s kontejnery

TODO

Prostor na otázky

Agenda

- **Čtvrtá úloha: Práce se standardními algoritmy**



Adaptéry standardních kontejnerů

Nejsou reference, obalují nějaký kontejner a nabízí upravený interface

- **std::stack** – definuje interface stacku nad std::deque
  - Podporuje push(item), emplace(args…) a pop() – odebírá poslední přidaný
  - Získávání vrchního prvku: top()

#include <stack>

- **std::queue** – definuje interface fronty nad std::deque
  - Podporuje push(item), emplace(args…) a pop() – zde odebírá předek
  - Získávání prvního prvku: front()

#include <queue>

- **std::priority_queue** – definuje interface max-haldy nad std::vectorem
  - push(item), emplace(args…) a pop() – odebírá největší prvek
  - Získávání nejvyššího prvku: top()
  - Pořadí prvků jde změnit zadáním jiného komparátoru

#include <queue>

Třídění

Určení pořadí v set/map: operator< - metoda

- V kontejnerech (multi)map a (multi)set třídění defaultně podle operátoru <
  - Ten Může být definován jako metoda **bool operator<(const T& rhs) const**
  - Vrací bool se sémantikou „lhs je menší než rhs“
  - Pokud lhs == rhs, tak v obou směrech false
    - Pak prvkům lhs a rhs říkáme *ekvivalentní*

```
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;
```

Zkrácená definice pomocí std::tie

inicializace přímo s prvky pomocí std::initializer_list

Implicitně definovaný structured binding pro struct

```
Program output:
0, 5
1, 4
2, 2
3, 4
3, 5
5, 2
```

Určení pořadí v set/map: operator< - přátelská funkce

- Operátor < může být i externí funkce
  - Hodí se, aby byla deklarovaná jako **friend** naší třídy (nemusí)
    - Pokud je **friend**, můžeme ji rovnou i definovat uvnitř
    - Ale stále je vnímána, jako by byla definovaná vedle té třídy
    - Akorát může libovolně přistupovat k private/protected

```
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;
```

```
Program output:
0, 5
1, 4
2, 2
3, 4
3, 5
5, 2
```

Určení pořadí v set/map: operator< - přátelská funkce

- Operátor < může být i externí funkce

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

```
endl;
```

```
2, 2
3, 4
3, 5
5, 2
```

Co když chci jiné pořadí než je default?

- **Řetězce primárně podle délky**
- **Nejde udělat obecné < pro ten typ**
- **Chci jiné pořadí v různých mapách**

Určení pořadí v set/map: Komparátory: funktor

- Lze zadat komparátor
  - Předáme jako druhý typový argument
  - Libovolný objekt, který jde volat s argumenty **(const T& lhs, const T& rhs)**
    - Funkce nebo tzv. „*function object*" s metodou: **bool operator()(const T& lhs, const T& rhs) const**
      - Objekt, který jde volat – říkáme tomu ***Funktor***

```cpp
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;
```

Zde zadáme typ funktoru
kontejner vyrobí default

```
Program output:
0, 5
1, 4
2, 2
3, 4
3, 5
5, 2
```

Určení pořadí v set/map: Komparátory: lambda

- Komparátor můžeme nahradit lambdou
- Lambda je zkrácený způsob jak definovat **function object** (neboli **functor**)
- Zatím stačí chápat základní notaci, kdy definuje jednoduchou funkci:
  [](const T& lhs, const T& rhs) {
  // tělo, návratový typ odvozen z returnu
  }
- Nebo, pokud chceme explicitní návratový typ:
  [](const T& lhs, const T& rhs) -> bool {
  // tělo
  }

```
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;
}
```

Odvození typu objektu

```
Program output:
0, 5
1, 4
2, 2
3, 4
3, 5
5, 2
```

Každá lambda má vlastní typ

Třídění

<algorithm> callout: #include <algorithm>

Všechny následující kontejnery/funkce mohou brát komparátor jako extra argument: např. **std::sort(it_beg, it_end, my_sort)**

bool my_sort(const T&, const T&)

Může být i funktor nebo lambda

1. **Setříděné kontejnery** (např. std::set)
2. **std::sort(it_beg, it_end)** (např. na vectoru)
   - Setřídí celý range mezi zadanými iterátory, může přehazovat ekvivalentní
3. **std::partial_sort(it_beg, it_middle, it_end)** (zase, např. na vectoru)
   - Přeuspořádá pole tak, že po iterátor it_middle jsou prvky setříděné
   - Následující jsou pak >= *it_middle
4. **std::stable_sort(it_beg, it_end)** (zase, např. na vectoru)
   - Jako sort, ale zachovává pořadí ekvivalentních prvků

Standardní algoritmy

Standardní algoritmy

#include <algorithm>

- **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&))**
- **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

Pokud berou funkce,
tak mohou brát i funktory a lambdy

Typicky použijeme na vectoru a dequeu
některé i na (forward_)listu nebo arrayi

U (unordered, multi) setů a map použijeme:
kontejner.find

Vrátí funktor po poslední aplikaci
Můžeme třeba zaznamenávat statistiku

Funkce kopírující/přesouvající prvky
z jednoho kontejneru do druhého

vrací modifikovaný (transformed) prvek

Spojování kontejnerů

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

Compiler interpretuje lambdu jako zjednodušený zápis funktoru

Proto mají dvě totožné lambdy různé typy compiler totiž napíše dva různé funktory

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

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

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::vector<int> in{1, 2, 3, 4, 5};
      std::vector<int> out;
      std::copy(in.begin(), in.end(), std::back_inserter(out));
      ```
  - 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')