NPRG041 – C++

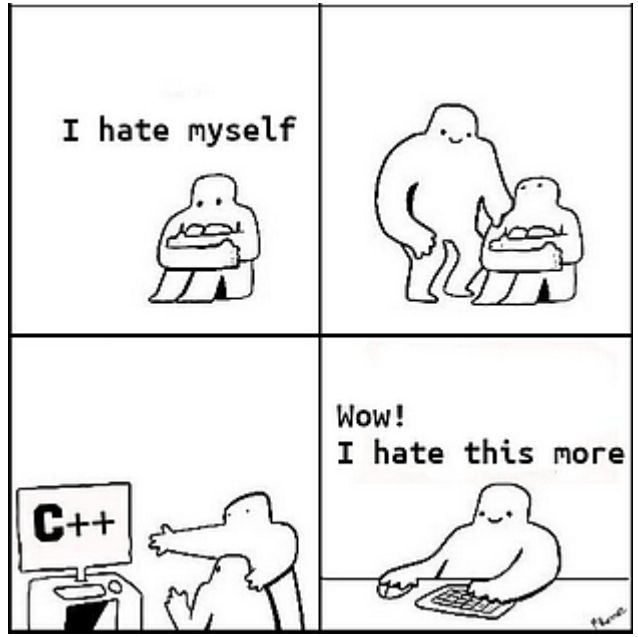
cvičení – Jiří Klepl

mattermost: ulita/2526ZS: nprg041-cpp-klepl (inv na SIS nástěnce)

klepl@d3s.mff.cuni.cz

Agenda

- Iterátory
- Ranges



Iterátory

- Základ: podporují ***it** a ***++it***
  - Dále se děli na Ouput a Input iterátory (a postupně zesilované varianty input it.)

Iterator category	Operations and storage requirement						
	Write *it = x	Read x = *it	Increment ++it		Decrement --it	Random access it ± n	Contiguous storage (adresy za sebou)
			w/o multiple passes	with multiple passes			
Iterator							
OutputIterator							
InputIterator							
ForwardIterator							
BidirectionalIterator							
RandomAccessIterator							
ContiguousIterator							

Příklad Output iterátoru a Input iterátoru

```
std::istringstream stream("1 2 3 4 5");
std::copy(
    std::istream_iterator<int>(stream),
    std::istream_iterator<int>(),
    std::ostream_iterator<int>(std::cout, " ")
);
```

Příklad Output iterátoru a Input iterátoru

```
std::istringstream stream("1 2 3 4 5");
std::copy(
    std::istream_iterator<int>(stream),
    std::istream_iterator<int>(),
    std::ostream_iterator<int>(std::cout, " ")
);
```

istream_iterator mění stav streamu
→ nejde dělat víc passů „jen tak“

Příklad forward iterátoru

```
std::forward_list<int> nums{1, 2, 4, 8, 16};

// Print forward_list.
std::for_each(nums.begin(), nums.end(), [](const int n) {
    std::cout << n << ' ';
});

std::cout << '\n';
```

forward_list::iterator nemění stav listu,
můžeme iterovat opakovaně
(kopie iterátoru by iterovaly to samé)

Další příklady kategorií iterátorů

- Bidirectional iterátor (dovoluje **++it** i **--it**)
  - **std::list<T>::iterator**
  - **std::set<K>::iterator**, **std::map<K, V>::iterator**
  - ⚠️ naopak ne pro **std::unordered_set** a **std::unordered_map**
- RandomAccess iterátor
  - **std::deque<T>::iterator**
- Contiguous iterátor
  - **std::vector<T>::iterator**

Vlastní iterátor (jednoduchý generátor čísel)

1. Určíme jeho kategorii (např. ForwardIterator)
   - Korektnost implementace kontrolujeme conceptem **std::forward_iterator**
2. Definujeme typedefy popisující iterátor
   - **difference_type**, **value_type**, **pointer**, **reference**, **iterator_category**
   - **value_type**, **point**, **reference** popisují výsledek dereference
   - Do iterator_category je uložen tag popisující kategorii (**forward_iterator_tag**)
3. Definujeme patřičné metody definované danou kategorií a všemi parent kategoriemi
   - Pro ForwardIterator: **++it**, **it++**, ***it**, **it->m**, **it1 == it2**

Ranges

#include <ranges>

- Range ≡ **[begin, end)** – typický range (všechny standardní kontejnery)
  - nebo **begin + [0, size)** – range definovaný rozsahem
  - nebo **[begin, pred)** – range zakončený podmínkou
  - nebo **[begin, ...)** – range bez konce
- Velké množství standardních algoritmů **(begin, end, ...)** má ranges:: variantu **(range, ...)**

https://en.cppreference.com/w/cpp/algorithm/ranges

- Lazy evaluace a kompozabilita (typicky pomocí | operátoru)

```
std::vector<int> numbers{ ... };
auto view = numbers
    | std::views::filter([](int i) { return i % 2 == 0; })
    | std::views::take(5);
for (auto&& item : view) {
    // Process item.
}
```

Standard nabízí velké množství utilit pro ranges

1. Funkce na získávání informací o rangích
   - **ranges::begin(range)**, **ranges::end(range)**, **ranges::size(range)**, …
2. Koncepty
   - **ranges::range** (requireuje jen aby struktura podporovala **ranges::begin/end**)
   - **ranges::sized_range** (požaduje podporu **begin**, **end** a **ranges::size**)
   - **ranges::view**
   - **ranges::input_range**, **ranges::output_range**, **ranges::forward_range**, …
3. Factories
   - **ranges::views::iota** (umí iterovat čísla z určeného intervalu; jako pythoní range)
4. Adaptéry (typické views)
   - **ranges::views::filter**, **ranges::views::take**, **ranges::views::zip**, …
5. Konverze: **std::views::iota(0, 42) | std::ranges::to<std::vector>()**

Příklad: Generátor Fibonacciho čísel

1. Třída Fibonacci má konstruktor s dvěma parametry (**a_0**, **a_1**)
2. Dané parametry řídí iniciální hodnoty posloupnosti **a_0**, **a_1**, **a_2**, ...
3. Posloupnost je definovaná **$a_{n+1} = a_n + a_{n-1}$**
4. Třída Fibonacci modeluje range s hodnotami posloupnosti
   - **it = std::ranges::begin(fib)** je iterátor odpovídající hodnotě **a_0**, **++it** odpovídá **a_1**, atd.
   - **std::ragnes::end(fib)** je „sentinel“ na konec. Pro tuto úlohu nebude nikdy roven platnému iterátoru (posloupnost je nekonečná)