

# KIV/CPP – Programování v jazyce C++

## 02. STL kontejnery, RAII, chytré ukazatele

Martin Úbl

KIV ZČU

2023/2024

- návrhový vzor
- pevně vázán ke kontejneru
- „ukazuje“ na právě jeden prvek
- každý kontejner poskytuje typ iterátoru, např.

```
std::vector<int>::iterator
```

- případně const varianta, např.

```
std::vector<int>::const_iterator
```

- někdy k dispozici reverse varianta, např.

```
std::vector<int>::reverse_iterator
```

- před C++20 - *named requirements*, nyní s prefixem *Legacy*
- *Iterator*
  - obecný princip; dereference, inkrementace
- *InputIterator*, *OutputIterator*
  - iterátor pro čtení / zápis
- *ForwardIterator*
  - dopředný iterátor, víceprůchodový
- *BidirectionalIterator*
  - obousměrný iterátor; navíc dekrementace
- *RandomAccessIterator*
  - náhodný přístup; přičtení, odečtení
- a další...

- po C++20 - *concepts*, iterator library
- `std::input_iterator`, `std::output_iterator`
  - iterátor pro čtení / zápis
- `std::forward_iterator`
  - dopředný iterátor, víceprůchodový
- `std::bidirectional_iterator`
  - obousměrný iterátor; navíc dekrementace
- `std::random_access_iterator`
  - náhodný přístup; přičtení, odečtení
- a další...
- jediný named requirement zůstává - *ConstexprIterator*

- ukázka konceptu
- `std::random_access_iterator`

```
template<class I>
concept random_access_iterator =
    std::bidirectional_iterator<I> &&
    std::totally_ordered<I> &&
    std::sized_sentinel_for<I, I> &&
    requires(I i, const I j, const std::iter_difference_t<I> n) {
        { i += n } -> std::same_as<I&&>;
        { j + n } -> std::same_as<I>;
        { n + j } -> std::same_as<I>;
        { i -= n } -> std::same_as<I&&>;
        { j - n } -> std::same_as<I>;
        { j[n] } -> std::same_as<std::iter_reference_t<I>>;
    };
```

- koncepty budeme brát na pozdějších přednáškách
- ve zkratce jde o „výpis“ všech požadavků tak, aby jim rozuměl kompilátor

- po změně podlelého kontejneru nemusí být platný!
  - neexistuje zpětná reference kontejner → iterátor
  - <https://en.cppreference.com/w/cpp/container>
- důvod, proč např. mazání (erase) vrací nový iterátor, např.

```
itr = myList.erase(itr);
```

- získání iterátoru – begin, rbegin, find, ...

```
auto itr = myList.begin();
```

- typicky např. průchod kontejnerem od začátku (`begin()`) do konce (`end()`)

```
for (auto itr = myList.begin();
     itr != myList.end();
     ++itr) ...
```

- nebo pozpátku od `rbegin()` k `rend()` (pokud kontejner podporuje)
- `end()` (`rend()`) je speciální iterátor
  - značí konec kontejneru
  - `find()` vrací při nenalezení prvku

- *RandomAccessIterator*, `std::random_access_iterator`
  - `std::array`, `std::vector`

```
// 5. prvek
auto itr = myVector.begin() + 4;
std::cout << *itr;
```

- *ForwardIterator*, `std::forward_iterator`
  - `std::list`

```
// 5. prvek
auto itr = myList.begin();
std::advance(itr, 4);
std::cout << *itr;
```



## Initializer list

---

- typ `std::initializer_list`
- v určitých kontextech lze psát bez typu `s { a }`
  - konstruktory tříd
  - inicializace parametrů
  - inicializace „vnitřností“ kontejnerů
- zabraňuje tzv. *narrowing* konverzi typů (podmnožina implicitní)
  - např. `double` na `float`
  - celočíselné typy na číslo s plovoucí des. tečkou a naopak
- bezpečnější
- [https://en.cppreference.com/w/cpp/language/list\\_initialization](https://en.cppreference.com/w/cpp/language/list_initialization)

```
std::array<int, 3> arr{1,2,3};      // direct
std::array<int, 3> arr = {1,2,3}; // copy
Player plr{"Adam"};
Player* pplr = new Player{"Adam"};
```

```
#include <array>
std::array<int, 3> arr = {1, 2, 3};
```

- generický typ (šablony), reprezentace pole pevné délky

vložení	přístup	smazání	vyhledání
NE	arr[i], $O(1)$	NE	NE <sup>1</sup>
	iterátor, $O(1)$		

Tabulka: Operace nad std::array

---

<sup>1</sup>pouze generická

```
#include <vector>
```

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

- generický typ (šablony), reprezentace pole proměnné délky

vložení	přístup	smazání	vyhledání
push_back, $O(1)$	vec[i], $O(1)$	erase, $O(n)$	$NE^2$
insert, $O(n)$	iterátor, $O(1)$		

Tabulka: Operace nad std::vector

---

<sup>2</sup>pouze generická

```
#include <list>
std::list<int> lst = {1, 2, 3};
```

- generický typ (šablony), reprezentace zřetěženého seznamu (obousměrně)
- varianta `std::forward_list` – jednosměrné řetězení

vložení	přístup	smazání	vyhledání
<code>push_back</code> , $O(1)$	iterátor, $O(n)$	<code>erase</code> , $O(1)$	$NE^3$
<code>insert</code> , $O(1)$			

Tabulka: Operace nad `std::list`

---

<sup>3</sup>pouze generická

```
#include <map>
std::map<int, double> mp
    = {{2, 2.5},{3, 5.5},{5, 12.95}};
```

- generický typ (šablony), reprezentace asociativního úložiště
- interně často např. *red-black* strom

vložení	přístup	smazání	vyhledání
<code>mp[i], <math>O(\log n)</math></code>	<code>mp[i], <math>O(\log n)</math></code>	<code>erase, <math>O(\log n)</math></code>	<code>find, <math>O(\log n)</math></code>
<code>insert, <math>O(\log n)</math></code>	<code>iterátor, <math>O(\log n)</math></code>		

Tabulka: Operace nad `std::map`

```
#include <set>
std::set<int> st = {2, 3, 5};
```

- generický typ (šablony), reprezentace množiny s unikátními prvky
- interně často např. *red-black* strom

vložení	přístup	smazání	vyhledání
insert, $O(\log n)$	iterátor, $O(\log n)$	erase, $O(\log n)$	find, $O(\log n)$

Tabulka: Operace nad std::set

- konkrétní typ `std::basic_string<T>` pro typ `char`
- varianty pro `wchar_t` (`std::wstring`), `char16_t` (`std::u16string`), ...
- interně vektor znaků
- také kontejner (má iterátor, `begin()` a `end()`, ...)

```
#include <string>
```

```
std::string str1("Retezec_1");  
std::string str2{"Retezec_2"};  
std::string str3 = "Retezec_3";  
std::string str4 = {"Retezec_3"};
```

- přetížený operátor == (viz další semináře) pro porovnání i např. s const char\*

```
const char* cstr = "hello";
std::string str("hello");

if (cstr == "hello") {
    // nejspis selze... a nebo ne?
}

if (str == "hello") {
    // bude fungovat
}
```



- inicializace mapy a vkládání

```
std::map<int, double> mp{{2, 5.1}};  
mp.insert({1, 2.5});
```

- vnořená inicializace (např. vektor řetězců)

```
std::vector<std::string>  
    vec1{"Hello", "World"};
```

```
std::vector<std::string>  
    vec2{{"Hello"}, {"World"}};
```

- speciální syntaxe cyklu for pro kontejnery s iterátorem
- často kombinace s auto a referencí
- někdy žádoucí const
- analogie foreach z jiných jazyků
- iterace od begin() k end() s dereferencí

```
std::list<int> mujList = {2, 4, 8, 16, 32};
```

```
for (const auto& prvek : mujList)
    std::cout << prvek << std::endl;
```

## Range-based for

---

- u asociativních úložišť vždy buď
  - reference s const klíčem
  - const reference
  - kopie
- klíč určuje fyzickou polohu v paměti

```
std::set<int> mnozina = {1, 5, 10};
```

```
for (auto& prvek : mnozina) {  
    std::cout << prvek << std::endl;  
    prvek = 5; // nelze!  
}
```

- pozn.: auto dedukuje automaticky typ s const

## Range-based for

---

- od C++20 lze dodat inicializační výraz

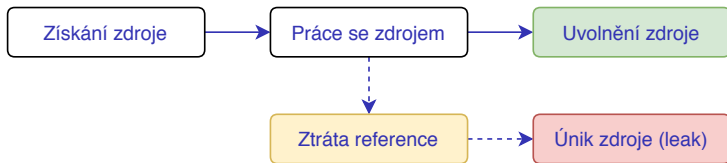
```
for (int i = 5; auto& prvek : mnozina) {  
    //  
}
```

- hodí se spíš pro iteraci skrz obsah dočasného objektu

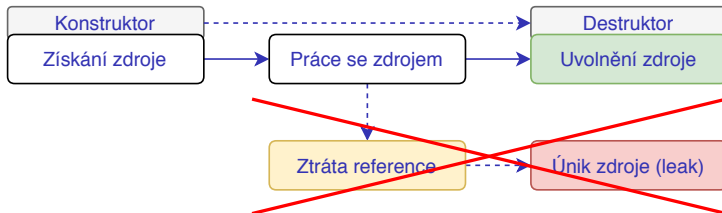
```
for (Obj a = getObj();  
     auto& p : a.items()) {  
    //  
}
```

- Na cvičení:
  - kontejnery
  - iterátory
  - řetězce

- *Resource Acquisition Is Initialization*
- `resource = zdroj`
  - paměť
  - soubor
  - zámek (mutex, ...)
  - vlákno
  - a další...



- *Resource Acquisition Is Initialization*
- využití vlastností jazyka
- deterministická životnost objektu ohraničená konstruktorem a destruktorem



- typická aplikace:
  - práce se soubory (`std::fstream, ...`)
  - chytré ukazatele (`std::unique_ptr, ...`)
  - zámky (`std::unique_lock, ...`)
  - STL kontejnery



- *smart pointers*
- několik variant
  - `std::unique_ptr`
  - `std::shared_ptr`
  - `std::weak_ptr`
- inicializátory
  - `std::make_unique`
  - `std::make_shared`

- std::unique\_ptr
- RAII struktura
- právě jeden vlastník

```
// obdelnik
```

```
std::unique_ptr<Rect> rect  
    = std::make_unique<Rect>(2, 3);
```

```
rect->CalcSurface();
```

- nelze kopírovat

```
auto rect = std::make_unique<Rect>(a, a);  
auto rect2 = rect; // chyba!
```

- předávání pomocí std::move

```
auto rect = std::make_unique<Rect>(2, 3);  
// ...  
auto rect2 = std::move(rect);  
rect2->CalcSurface();  
rect->CalcSurface(); // chyba!
```

- předávání pomocí std::move
- návratová hodnota funkce

```
std::unique_ptr<Rect> VytvorCtverec(int a) {  
    auto rect = std::make_unique<Rect>(a, a);  
    // ...  
    return std::move(rect);  
}
```

- nemusí být nutné, C++ dovoluje aplikaci tzv. *copy elision* na vracení lokálních objektů (NRVO)

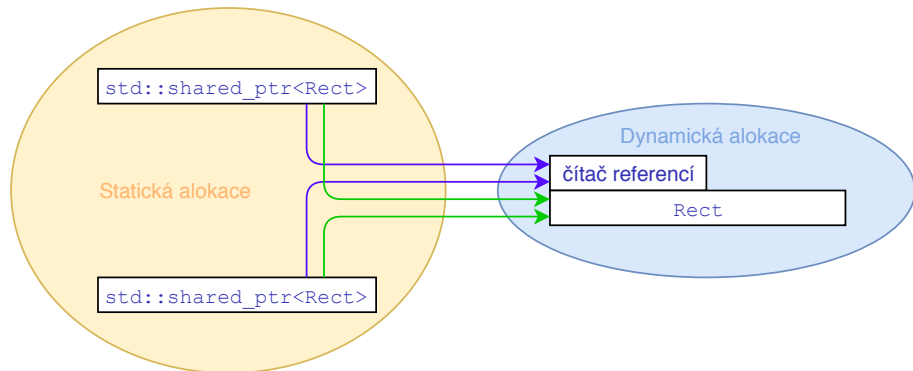
- std::shared\_ptr
- RAII struktura
- nejméně jeden vlastník
- interní čítač referencí
  - konstruktor zvyšuje
  - destruktor snižuje

```
// obdelnik
std::shared_ptr<Rect> rect
    = std::make_shared<Rect>(2, 3);

rect->CalcSurface();
```

- lze kopírovat

```
auto rect = std::make_shared<Rect>(2, 3);  
auto rect2 = rect;
```



- návratová hodnota není problém

```
std::shared_ptr<Rect> VytvorCtverec(int a) {  
    auto rect = std::make_shared<Rect>(a, a);  
    // ...  
    return rect;  
}
```

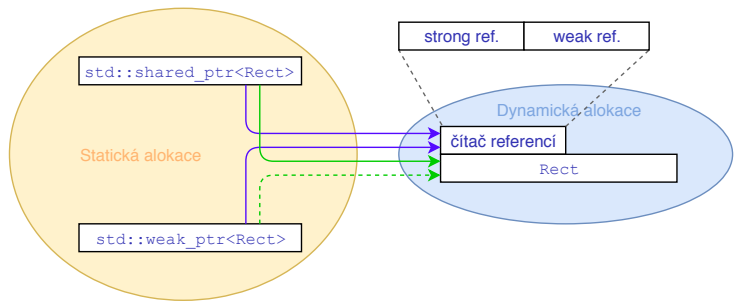


- oba lze zneplatnit
  - nastavením na `nullptr`
  - metodou `reset()`

```
ptr = nullptr;  
ptr.reset();  
ptr.reset(nullptr); // pouze unique_ptr
```

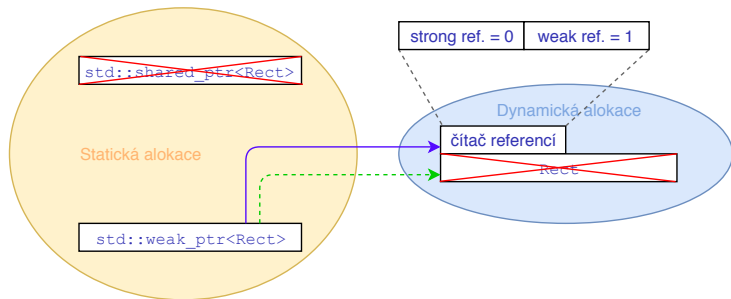
- std::weak\_ptr
- slabá reference, vždy vázaná na std::shared\_ptr

```
auto rect = std::make_shared<Rect>(2, 3);  
std::weak_ptr<Rect> rect_weak = rect;
```



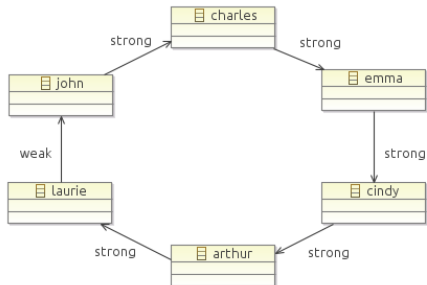
- nikdy ne práce přímo, vždy získat instanci std::shared\_ptr

```
std::weak_ptr<int> rect_weak = rect;  
if (auto shared = rect_weak.lock()) {  
    // ...  
}
```



Obrázek: Situace, kdy již žádný `shared_ptr` neodkazuje na objekt

- hodí se např. na práci s kruhovými seznamy



Obrázek: Cyklická závislost; převzatý obrázek<sup>4</sup>

---

<sup>4</sup><https://visualstudiomagazine.com/articles/2012/10/19/circular-references.aspx>

- Na cvičení:
  - užití `unique_ptr`
  - užití `shared_ptr`
  - kombinace s `weak_ptr`