KIV/TSI - Seminář C++

02. STL kontejnery, RAII, chytré ukazatele

Martin Úbl

KIV ZČU

2020/2021

Iterátor

- 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
```

Iterátor

- *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

Iterátor

- po změně podlehlého kontejneru nemusí být platný!
  - neexistuje zpětná reference kontejner $\rightarrow$ 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();
```

Iterátor

- typicky např. iterace od `begin()` k `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

Iterátor

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

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

- *ForwardIterator*
  - `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`

```
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[1]

Tabulka: Operace nad `std::array`

[1] neexistuje standardní metoda

```
#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]

Tabulka: Operace nad `std::vector`

[2] neexistuje standardní metoda

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

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

vložení	přístup	smazání	vyhledání
`push_back`, $O(1)$	iterátor, $O(n)$	`erase`, $O(1)$	NE[3]
`insert`, $O(1)$			

Tabulka: Operace nad `std::list`

[3] neexistuje standardní metoda

```
#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
`mp[i]`, $O(logn)$	`mp[i]`, $O(logn)$	`erase`, $O(logn)$	`find`, $O(logn)$
`insert`, $O(logn)$	iterátor, $O(logn)$		

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`, $O(logn)$	iterátor, $O(logn)$	`erase`, $O(logn)$	`find`, $O(logn)$

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

Initializer listy a kontejnery

- 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"}};
```

Range-based for

- 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 (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!
}
```

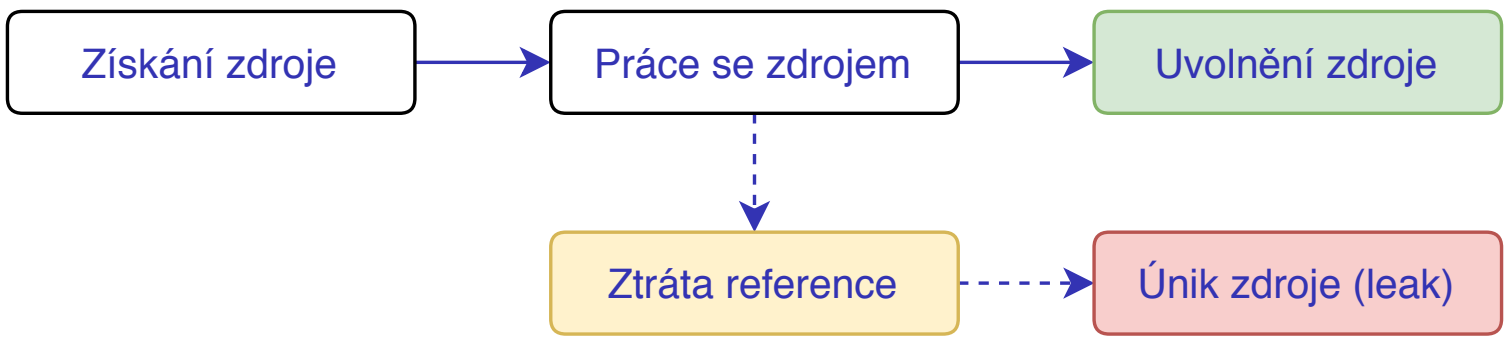
- pozn.: `auto` dedukuje automaticky typ s `const`

Kontejnery

- příklady:
  - kontejnery
  - iterátory
  - řetězce

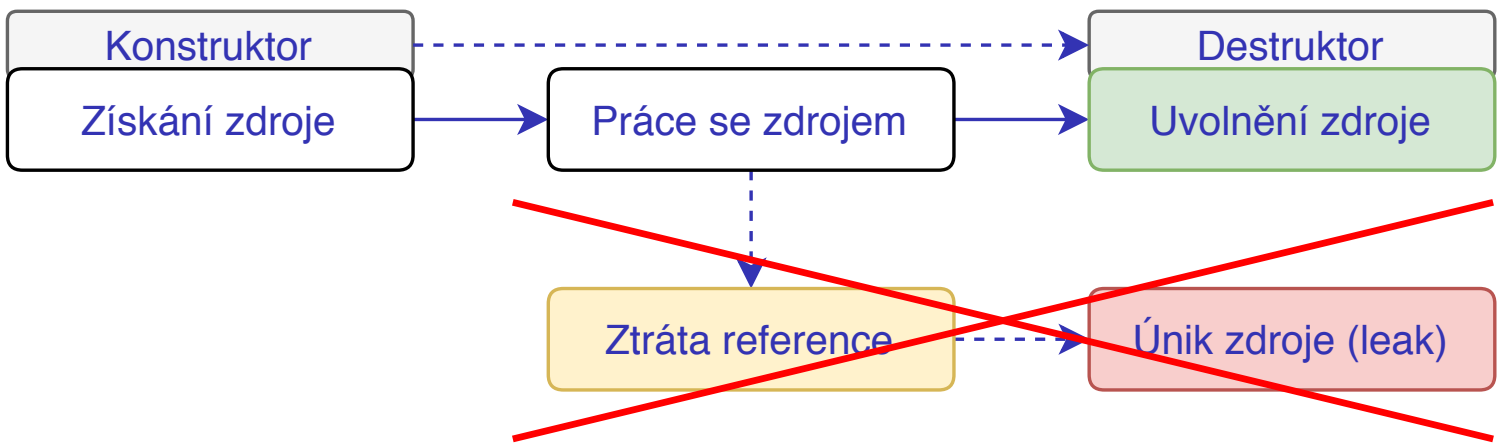
RAII

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



RAII

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



RAII

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

Chytré ukazatele

- smart pointery
- 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);
}
```

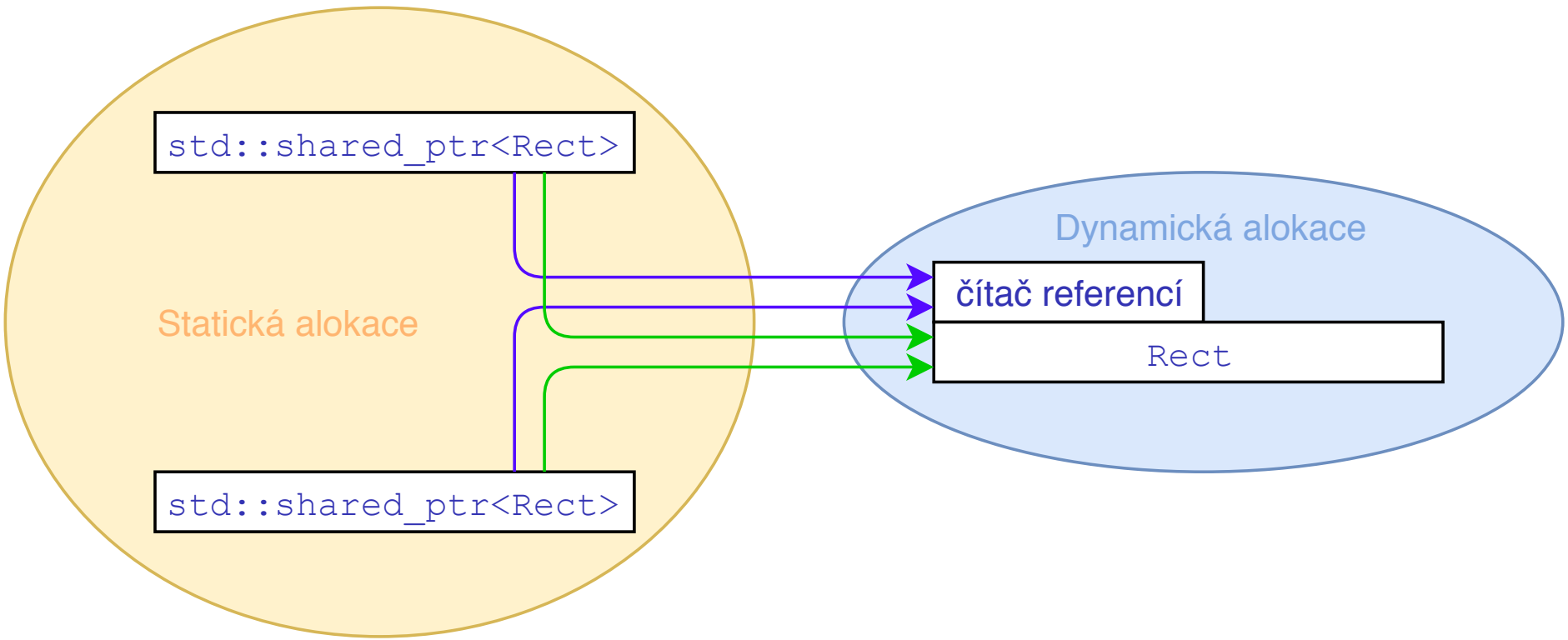
- std::shared_ptr
- RAII struktura
- nejméně jeden vlastník
- interní čítač referencí
  - konstruktor zvyšuje
  - destruktor snižuje

```
// 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 již není problém

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

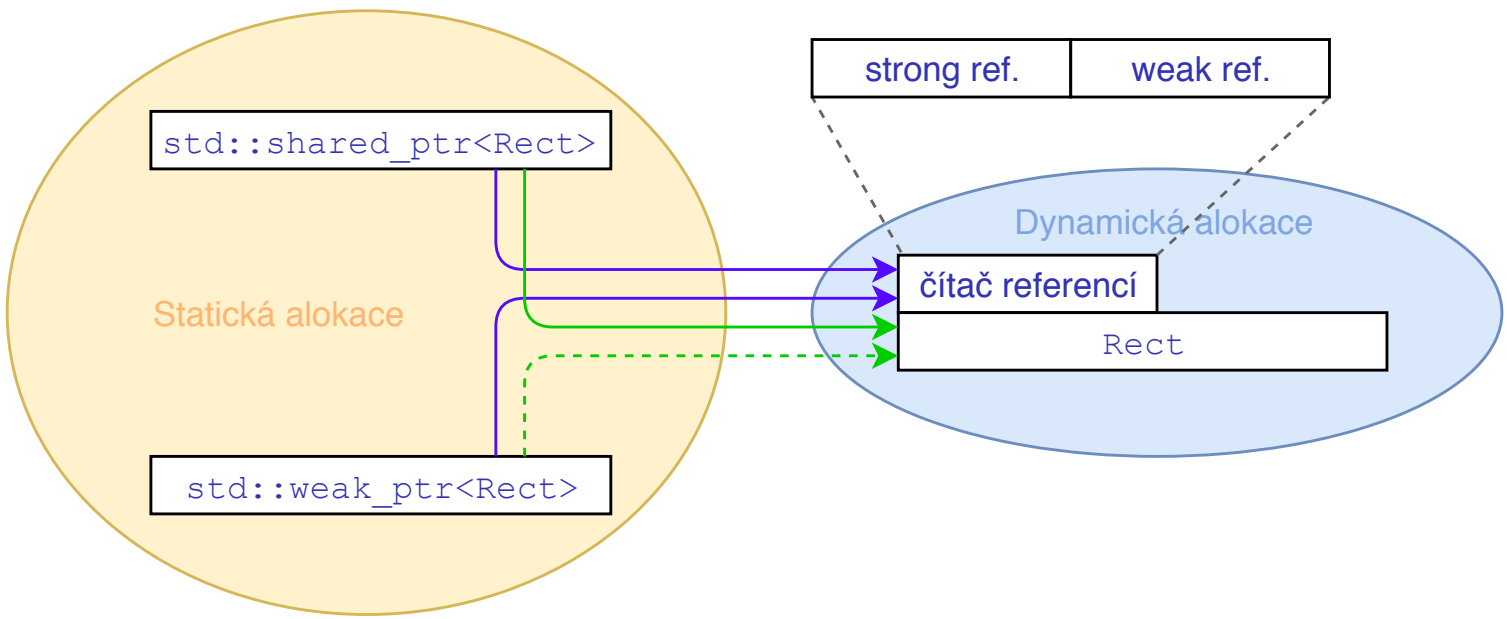
Chytré ukazatele

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

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

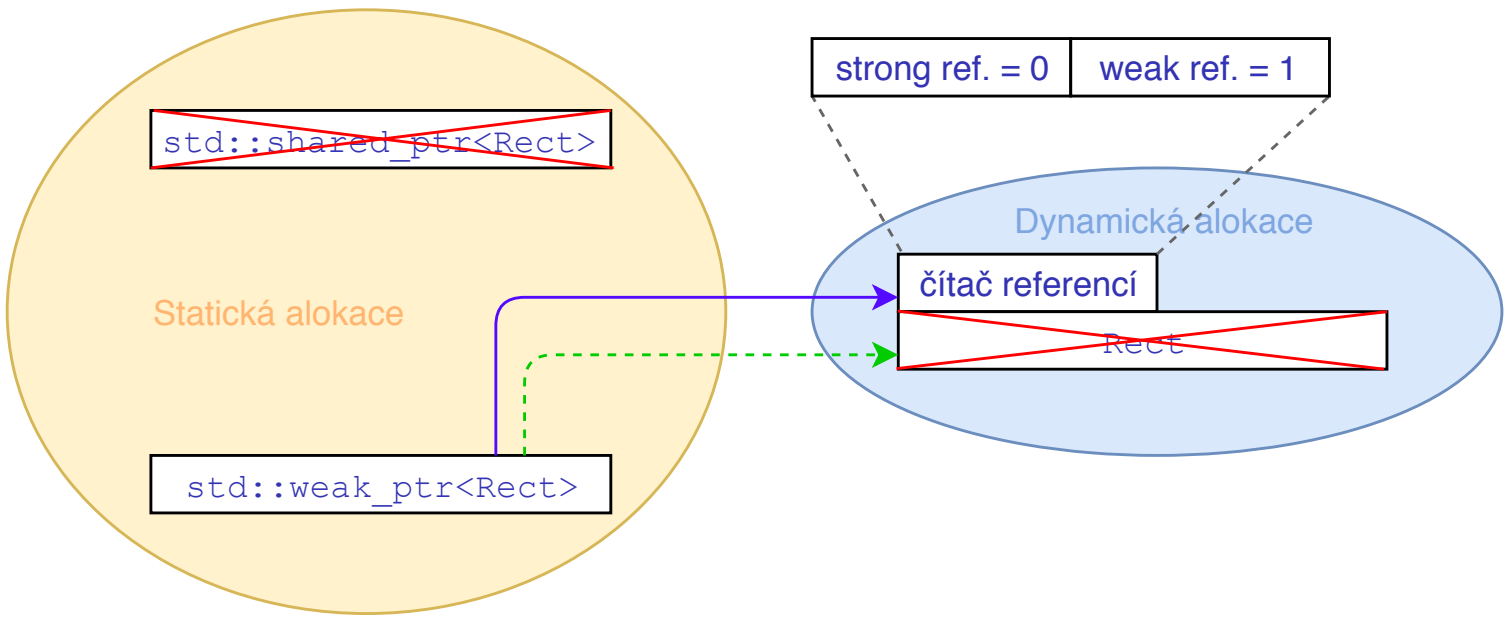
- std::weak_ptr
- slabá reference, vždy vázaná na std::shared_ptr

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



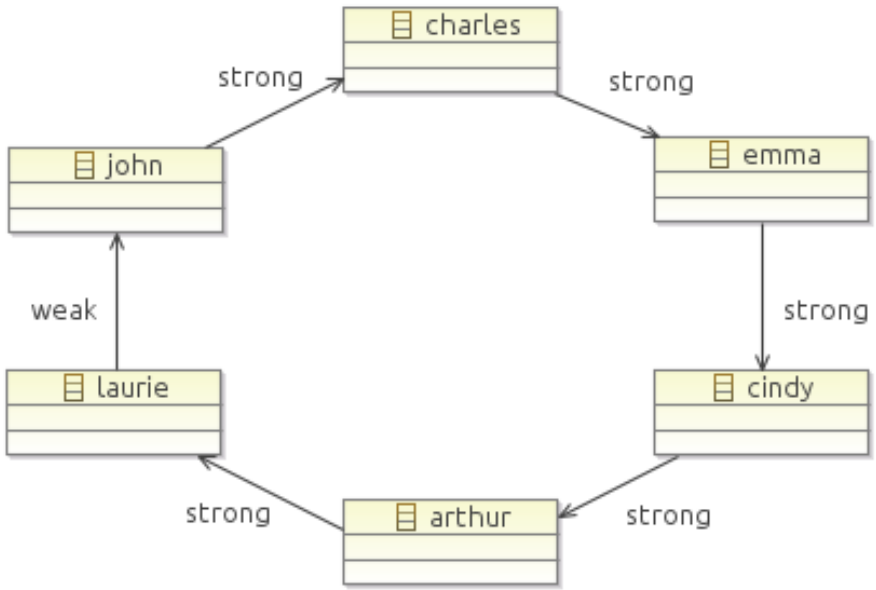
- nikdy ne práce přímo, vždy získat instanci `std::shared_ptr`

```
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: Převzato z https://visualstudiomagazine.com/articles/2012/10/19/circular-references.aspx

Chytré ukazatele

- příklady:
  - užití `unique_ptr`
  - užití `shared_ptr`
  - kombinace s `weak_ptr`