PB161 Programování v jazyce C++
Přednáška 11
Pokročilé C++(14) – povrchně

Nikola Beneš

6. prosince 2016

Organizační

Dnešní přednáška

- co je nového v C++11, C++14
  - částečně rekapitulace
  - částečně výhled dál (velmi povrchně)
  - reklama na **PV264** Advanced Programming in C++

Příští přednáška

- zvaná přednáška – zkušenosti z praxe
- přednášející? nechte se překvapit

Zápočet

- v poslední týdnu semestru během cvičení
  - přihlašování v ISu

Domácí úkoly

- hw04 – bude zveřejněno po dnešní přednášce
- hw05 (bonusový) – bude zveřejněn do konce týdne

Moderní C++ zařazené do PB161 (rekapitulace + něco navíc)

- `nullptr`
- `auto`
- range-based `for`
- uniformní inicializace
- anonymní funkce – lambdy (povrchně)
- `unique_ptr` (zlehka)
- `default`, `delete`
- `override`, (`final`)
- `noexcept`
- `using`

Rekapitulace: `nullptr`

Proč je lepší než `NULL`?

- typově bezpečné
- není to makro
- implicitně se konvertuje na
  - ukazatele
  - `bool`

Rekapitulace: `auto`

- klíčové slovo žádající překladač o **automatické** odvození typu
  - stejná pravidla jako pro odvozování šablonových parametrů
  - … s malou výjimkou pro `std::initializer_list`
    - za chvíli

Kdy je vhodné používat `auto`?

- víme, jaký bude výsledný typ, a ten je buď
  - příliš škaredý (iterátory)
  - jasně čitelný z okolního kódu, např. je někde přímo uvedený

```
auto printer = myFactory.createInkPrinter();
auto ptr = std::make_unique<ListElement>();
```

- v rozumné míře ve hlavičce range-based `for`

```
for (const auto& person : people) { /* ... */ }
```

- nebo nevíme, jaký bude výsledný typ (a nemáme to jak vědět)
  - jak se tohle může stát?

Rekapitulace: range-based `for`

Co se skrývá za následujícím cyklem?

```
std::vector<std::string> words;
// ...
for (const auto& word : words) {
    std::cout << word << '\n';
}
```

```
{
    auto&& __l = words;                 // ???
    auto __i = std::begin(__l),
         __e = std::end(__l);
    for(; __i != __e; ++__i) {
        const auto& word = *__i;
        std::cout << word << '\n';
    }
}
```

Vlastní iterovatelné objekty

Co musí objekt splňovat, abych přes něj mohl iterovat pomocí `for`?

- musí mít iterátory
  - ty musí umět minimálně operátor ++ (prefixový), * (unární, dereference) a porovnávání pomocí ==
- musí mít metodu `begin()`, která vrací iterátor
- musí mít metodu `end()`, která vrací iterátor

Iterátorů je mnoho druhů

- už jste mohli vidět na `http://en.cppreference.com/w/cpp/iterator`

Uniformní inicializace

Rekapitulace (viděli jsme)

- inicializace v C++03: použitím kulatých závorek nebo = u deklarace
- inicializace v C++11: kromě výše uvedeného navíc ještě složené závorky
- proč?

Co se stane?

```
std::string line();
Person person(std::string());
```

- oba tyto řádky *deklarují funkce*
- toto je známo jako **most vexing parse**

Inicializace složenými závorkami – snaha o sjednocení syntaxe

Uniformní inicializace (pokr.)

Sjednocení syntaxe pro inicializaci složenými závorkami

- lze použít složené závorky
- navíc: limituje přetypování
  - pouze takové, které neomezuje rozsah typu [ukázka]
- navíc: v některých případech není nutno použít jméno typu [ukázka]

```
std::string line{};
Person person{ std::string{} };
```

- … a tím je problém vyřešen, všechny objekty můžeme inicializovat stejnou syntaxí.

(Skutečně?)

Inicializační seznam

Statická inicializace (Céčkových) polí

```
int array[] = { 1, 2, 3, 4, 0 };
```

Chceme totéž pro vektory

```
std::vector<int> data{ 1, 2, 3, 4, 0 };
```

- už víme, že tohle funguje, ale jak to funguje?
- typ `std::initializer_list<T>`, který umí chytat seznamy ve složených závorkách, má metody `begin()` a `end()`
- `std::vector<T>` má konstruktor `vector(std::initializer_list<T>)`
  - bere se hodnotou (neznamená kopii seznamu!)
- můžete použít i v konstruktorech vlastních tříd

Inicializační seznam (pokr.)

```
for (int i : { 1, 2, 3 }) { /* ... */ }
```

- jaktože to funguje? výjimka pro `auto`

Kdo má přednost (uniformní inicializace vs. inicializační seznam)?

```
std::string s1("text");
std::string s2({ 't', 'e', 'x', 't' });
std::string s3(65, 't');
std::string s4{ "text" };
std::string s5{ 't', 'e', 'x', 't' };
std::string s6{ 65, 't' };
```

- co bude obsahem řetězců?

[ukázka]

Pointa: Inicializační seznam má přednost.

Rekapitulace: anonymní funkce (lambdy)

- funkce, kterou můžeme definovat lokálně, v místě výrazu
- syntaktická zkratka za funkční objekt
  - ve skutečnosti to je tzv. uzávěra (closure)
  - může zachytávat proměnné z okolí

```
std::vector<int> v;
// ...
int sum = 0;
std::for_each(v.begin(), v.end(), [&](int i) { sum += i; });
```

- co se zde skrývá?

[ukázka]

Anonymní funkce (lambdy) – syntax

```
[capture] (parameters) -> return_type { body }
[capture] (parameters) { body }
[capture] { body }
```

- nepovinné části
  - lze vynechat návratový typ (dedukuje se automaticky)
  - lze vynechat seznam parametrů (pak je prázdný)
- zachytávání
  - `[]` – nic
  - `[a]` – a hodnotou (konstantní
  - `[&b]` – b referencí
  - `[=]` – vše (co se vyskytuje v těle lambdy) hodnotou
  - `[&]` – vše (co se vyskytuje v těle lambdy) referencí
  - kombinace, např. `[&,a]` – vše referencí, ale a hodnotou
  - `[this]` – zachycení `this`

Anonymní funkce (lambdy) – předávání/ukládání

Co je lambda?

- lambda je dočasný objekt – instance anonymní třídy
- každá lambda má vlastní třídu

Jak předávat/ukládat lambdu?

- pokud nic nezachytává, lze ji přetypovat na ukazatel na funkci
- lze ji předat/uložit do
  - `auto`
  - šablonového parametru
  - `std::function<signatura funkce>`
    - používejte jen, pokud je to nutné
  - proč? je to dražší a neumožňuje to inlining

[ukázka]

Anonymní funkce (lambdy) – C++14

Automatická dedukce typu parametrů

```
[](auto x){ return ++x; }
```

Možnost inicializace v sekci `capture`

```
int x = 4;
int y = [&r = x, x = x + 1] {
    r += 2;
    return x + 2;
}();
```

- jaká bude na konci hodnota proměnných `x` a `y`?
- a co když přehodíme pořadí uvnitř sekce `capture`?

[ukázka]

Rekapitulace: `std::unique_ptr`

- chytrý ukazatel
- myšlenka: unikátní vlastník alokované paměti
- automaticky dealokuje paměť v destruktoru
- nelze kopírovat, ale lze předávat vlastnictví pomocí `move`
  - (o tom za chvíli)

Upozornění:

- nenahrazuje běžné ukazatele, jen jedno z jejich použití
- nenahrazuje vytváření lokálních objektů (na zásobníku / uvnitř třídy)

[ukázka toho, co strašného jsme viděli v domácích úlohách]

Rekapitulace: `default`, `delete`

`default`

- explicitní vynucení automaticky generovaných metod
  - konstruktory (včetně kopírovacích)
  - destruktory
  - kopírovací přiřazovací operátor

`delete`

- explicitní zabránění automatickému vytváření metod/funkcí
  - konstruktory
  - destruktory
  - přiřazovací operátory
  - *libovolné funkce*

[ukázka]

Rekapitulace: `override`, `final`

`override`

- sdělujeme překladači, že tato metoda překrývá virtuální metodu předka
- ochrana před nedodržením signatury (typu, `const`)

`final` (o tomto jsme nemluvili)

- totéž, co override + navíc ochrana proti překrytí virtuální metody
- tuto metodu už potomci nesmí překrýt
- lze použít i s třídami: od třídy `final` se nesmí dědit
- raději moc nepoužívat, proč?
  - porušuje to *Open-Closed Principle*
  - neumožňujete rozšiřování funkcionality
  - pokud nechcete umožnit překrývání metod, proč jsou vůbec virtuální?

[ukázka]

Rekapitulace: `noexcept`

- operátor i specifikátor
  - specifikátor může mít parametr typu `bool` (konstantní výraz)
  - samotný `noexcept` je totéž, co `noexcept(true)`
- označuje metodu/funkci, že nebude vyhazovat výjimky
  - pokud přesto něco vyhodí, volá se `std::terminate`
- přidání kvůli `move` sémantice (později)

```
template<typename T>
void doSomething(T& obj) noexcept(noexcept(obj.run())) {
    obj.run();
}

template<typename T>
void safeWithSmallThings(T t) noexcept(sizeof(T) < 4) {
    // ...
}
```

Rekapitulace: `using`

„Hezčí“ `typedef`

```
typedef int (*funcPtr)(int, const char*);
using func = int(int, const char*);
using funcPtr = func*;
```

Umí být šablonované

```
template<typename T>
using Matrix = std::vector<std::vector<T>>;

template<std::size_t N>
using IntArray = int[N];

template<typename T>
using Ptr = T*;
```

Další témata moderního C++

- `static_assert`
- zobecněné konstantní výrazy (`constexpr`)
- `move` sémantika
- variadické šablony
- další chytré ukazatele
- SFINAE a jiné šablonové triky
- asynchronní zpracování, vlákna
- knihovny `random`, `chrono`, `regex`, ...
- budoucnost (C++17)

Statický assert

- kontrola předpokladů během překladu
- vyhodnocení konstantního výrazu typu `bool`
- případné zahlášení chyby
- užitečné zejména v kombinaci se šablonami a tzv. type traits

```
static_assert(sizeof(int) == 4,
    "This program only works with 32-bit int type.");

template<typename T>
class Container {
    static_assert(std::is_trivial<T>::value,
    "This class is only designed to work for trivial types".);
    // ...
};
```

Zobecněné konstantní výrazy

Vyhodnocení funkce během překladu

- specifikátor `constexpr`
- funkci lze volat i v době provádění programu
- v C++11 velmi omezené
  - pouze jeden `return`
  - podmínka řešitelná pouze pomocí operátoru `?:`
- v C++14 rozšířeno; funkce nesmí obsahovat:
  - `goto`
  - výjimky (`try` / `catch`)
  - a pár dalších ...

[ukázka]

Move sémantika

L-hodnoty (lvalues)

- původně: „to, co může stát vlevo od přiřazení“
- v C++: cokoli, co má adresu (trochu zjednodušeně)
  - proměnná, reference, dereferencovaný ukazatel, ...

P-hodnoty (rvalues)

- „to ostatní“
- dočasné objekty, číselné konstanty, ...

Základní myšlenka move sémantiky

- když vím, že to, co jsem dostal je p-hodnota (dočasný objekt), můžu si s ní dělat, co chci
- rvalue reference: `typ&&`

[ukázka]

Move sémantika – použití

Move konstruktor (pro třídy, které drží nějaký zdroj – RAII)

```
class IntArray {
    int* array;
    size_t size;
public:
    // ...
    IntArray(IntArray&& other)
      : array(other.array), size(other.size) {
        other.array = nullptr;
        other.size = 0;
        // we steal other's resources
    }
};
```

Move přiřazovací operátor

```
    IntArray& operator=(IntArray&& other) { /* ... */ }
```

Move sémantika – Rule of Three/Four and a Half

- běžná třída s pravidlem 3,5 se změní v třídu s pravidlem 4,5:
  - stačí přidat konstruktor

```
class A {
public:
    A(const A& other) { /* ... */ }
    ~A() { /* ... */ }
    A& operator=(A other) {
        swap(other);
        return *this;
    }
    void swap(A& other) {
        using std::swap;
        // ...
    }

    A(A&& other) { /* ... */ } // NEW!
```

Move sémantika

K čemu je to vlastně dobré?

- výkonostní optimalizace
  - kopírovaní je drahé a my se mu takto umíme občas vyhnout
- kde se používá?
  - kontejnery ve standardní knihovně (např. `std::vector`)
  - tam, kde se spravují zdroje (RAII)
  - i jinde (perfect forwarding)

[ukázka]

Variadické šablony

Problém: v době psaní funkce nevím, kolik dostane parametrů

- řešení (C/C++03):
  - `void foo(int, ...)`
  - použití `va_args`
- řešení C++11:
  - variadické šablony
  - typově bezpečné
  - často používáno ve standardní knihovně

[ukázka]

Chytré ukazatele

`std::shared_ptr<T>`

- více vlastníků, počítání referencí
- „poslední zhasne“
- **podstatně** dražší než použití `std::unique_ptr`
- používejte jen, pokud skutečně **potřebujete** reference counting
  - to většinou nepotřebujete
  - na běžné použití stačí `std::unique_ptr`
  - dost často stačí **vůbec nealokovat paměť dynamicky** (+ používat prostředků standardní knihovny)

`std::weak_ptr<T>`

- doplněk k `std::shared_ptr`
- nepočítá se k referencím
- před použitím třeba konvertovat na `std::shared_ptr`

… a další

SFINAE (známo už v C++03)

- užitečné šablonové triky (mimo jiné i *type traits*)

Paralelní programování

- vlákna, mutexy, podmínkové proměnné, asynchronní volání, …

Součásti standardní knihovny a jiné

- `random`, `chrono`, `regex`, …

C++17

- `optional`, `variant`, folds, …

Závěrem

Pokud vás C++ zajímá podrobněji, zapište si:
PV264 Advanced Programming in C++

Přeji úspěšné zkouškové!