

PB161 Programování v jazyce C++

Přednáška 11

Pokročilé C++(17) – povrchně

Nikola Beneš

4. prosince 2018

Dnešní přednáška

- co je nového v C++11, C++14, C++17
 - částečně rekapitulace
 - částečně výhled dál (velmi povrchně)
 - reklama na **PV264** Advanced Programming in C++

Dnešní přednáška

- co je nového v C++11, C++14, C++17
 - 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

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

- `nullptr`
- `auto`
- range-based `for`
- uniformní inicializace
- anonymní funkce – lambda
- `unique_ptr`
- `default`, `delete`
- `override`
- `noexcept`
- `using`

Rekapitulace: `nullptr`

Proč je lepší než `NULL`?

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`?

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?

```
for (const auto& word : words) {  
    std::cout << word << '\n';  
}
```

Rekapitulace: range-based for

Co se skrývá za následujícím cyklem?

```
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';  
    }  
}
```

Rekapitulace: range-based for

Co se skrývá za následujícím cyklem?

```
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';  
    }  
}
```

- od C++17 technická změna: iterátory vrácené begin a end mohou být různého typu

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>

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?

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

Sjednocení syntaxe pro inicializaci složenými závorkami

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

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

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

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

Sjednocení syntaxe pro inicializaci složenými závorkami

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

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

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

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

(Skutečně?)

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?

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?

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?

Pointa: Inicializační seznam má přednost.

Rekapitulace: anonymní funkce (lambda)

- 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?

Anonymní funkce (lambda) – 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 lambda) hodnotou
 - [&] – vše (co se vyskytuje v těle lambda) referencí
 - kombinace, např. [&, a] – vše referencí, ale a hodnotou
 - [this] – zachycení this
 - [*this] – zachycení kopie aktuálního objektu (C++17)

Anonymní funkce (lambdy) – předávání/ukládání

Co je lambda?

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

Jak předávat/ukládat lambda?

Co je lambda?

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

Jak předávat/ukládat lambda?

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

Co je lambda?

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

Jak předávat/ukládat lambda?

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

Anonymní funkce (lambdy) – C++14

Automatická dedukce typu parametrů

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

- jako obvykle, pokud chceme explicitně referenci, musíme si o to říct: `auto&`

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?

lecture11_05.cpp

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)

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*

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?

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

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?

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`, ...
- C++17
- budoucnost (C++2?)

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".);  
    // ...  
};
```

Vyhodnocení funkce během překladač

- 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 ale stále nesmí obsahovat:
 - `goto`
 - bloky `try / catch`
 - a pár dalších ...
- v C++17 i `constexpr` lambdy

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
 - proč?

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
 - proč?
 - životnost dočasného objektu končí na konci výrazu
- rvalue reference: `typ&&`

lecture11_10.cpp

Move sémantika – použití

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

```
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;  
        // ...  
    }  
}
```


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!
```

K čemu je to vlastně dobré?

- výkonostní optimalizace
 - kopírování 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)

`std::move`

- funkce ze standardní knihovny
- přetypování na `typ&&`
- „K tomuto objektu je možno se chovat, jako by byl dočasný.“
 - umožňuje (move konstruktorům) „vykrást“ vnitřnosti objektu

Move sémantika a `noexcept`

- move konstruktory by měly pokud možno být deklarovány jako `noexcept`
- proč?

Move sémantika a `noexcept`

- move konstruktory by měly pokud možno být deklarovány jako `noexcept`
- proč?

- `std::vector` od C++11 by rád používal přesouvání místo kopií při realokaci
- ale co když move konstruktor vyhodí výjimku?

Move sémantika a `noexcept`

- move konstruktory by měly pokud možno být deklarovány jako `noexcept`
- proč?

- `std::vector` od C++11 by rád používal přesouvání místo kopií při realokaci
- ale co když move konstruktor vyhodí výjimku?
 - není možno garantovat návrat do konzistentního stavu
 - proč tento problém není s kopírovacími konstruktory?

- **pointa:** `std::vector` používá move konstruktory, jen pokud
 - jsou `noexcept` nebo
 - neexistuje kopírovací konstruktor
(v tom případě neplatí garance návratu do konzistentního stavu)

Problém: v době psaní funkce nevím, kolik dostane parametrů

- řešení

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

`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`

Nové části standardní knihovny od C++11

- type traits (užitečné šablonové triky používající SFINAE)
- regulární výrazy <regex>
- podpora paralelního programování
 - vlákna, mutexy, podmínkové proměnné, asynchronní volání, ...
- lepší generování náhodných čísel <random>
- lepší zacházení s časem <chrono>
- unordered_* asociativní kontejnery
 - implementované pomocí hashovacích tabulek

Nové části standardní knihovny od C++17

- optional
- variant – typově bezpečný **union**
- any
- string_view – efektivní způsob předávání podřetězců bez kopírování
- nové metody existujících kontejnerů
- ...

Nové části jazyka C++17

- automatická dedukce parametrů pro třídy
 - `std::pair p{ 1, 3.14 };`
- tzv. *folding* výrazů – lepší způsob zpracování variadických parametrů
- *structured bindings* – rozbalování tuple apod. při inicializaci
 - `auto [iter, inserted] = m.emplace(key, value);`
- *constexpr if*
- ...

C++2?

- koncepty
- ranges (algoritmy/iterátory s líným vyhodnocováním)
- porovnávací operátor $\lt\Rightarrow$
- moduly (?)
- korutiny (coroutines) (?)
- https://en.cppreference.com/w/cpp/compiler_support
- <https://en.wikipedia.org/wiki/C++20>

PV264 Advanced Programming in C++

- přednášky, materiály k předmětu apod. v angličtině
- dva domácí úkoly v první polovině semestru
- skupinový projekt v druhé polovině semestru
- <https://www.fi.muni.cz/pv264>

PV264 Advanced Programming in C++

- přednášky, materiály k předmětu apod. v angličtině
- dva domácí úkoly v první polovině semestru
- skupinový projekt v druhé polovině semestru
- <https://www.fi.muni.cz/pv264>

Přeji úspěšné zkouškové!