

PB161 Programování v jazyce C++

Přednáška 12

C++11 podrobněji

Jiří Weiser

7. prosince 2015

Vypněte, prosím, Ad-Block a podobná rozšíření.

Vypněte, prosím, Ad-Block a podobná rozšíření.

Pohodlně se usadte.

Vypněte, prosím, Ad-Block a podobná rozšíření.

Pohodlně se usadte.

Z dnešní přednášky vás *skoro* nikdo nebude zkoušet.

(Ale může se vám to hodit.)

Novinky z C++11 zařazené do PB161

- `nullptr`
- `using`
- `noexcept`
- `default`, `delete`
- `final`, `override`
- `auto`
- `range-for`
- `(lambda)`

Proč se hodí namísto `NULL`?

Proč se hodí namísto NULL?

- jednoznačný typ
- není to makro
- implicitně konvertibilní na
 - ukazatele
 - `bool`

Rekapitulace: using

„Hezčí“ typedef

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

```
funcPtr p1 = foo;  
func *p2 = foo;
```


Rekapitulace: using

„Hezčí“ typedef

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

```
funcPtr p1 = foo;  
func *p2 = foo;
```

Umožňuje brát šablonové parametry.

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

```
template< typename N >  
using IntArray = int[N];
```

Rekapitulace: `noexcept`

- operátor i specifikátor
- označuje metodu/funkci, že nebude vyhazovat výjimky
 - pokud ano, zavolá se `std::terminate()`
- nepřiliš podařená implementace
 - “šita horkou jehlou”
 - přidáno kvůli move sémantice (později)

Rekapitulace: default, delete

default

- explicitní vynucení automaticky generovaných metod
 - konstruktory
 - destruktory
 - přiřazovací operátor

Rekapitulace: default, delete

default

- explicitní vynucení automaticky generovaných metod
 - konstruktory
 - destruktory
 - přiřazovací operátor

delete

- explicitní zabránění vytvoření metod/funkcí
 - konstruktory
 - destruktory
 - přiřazovací operátor
 - *libovolná funkce*

Rekapitulace: default, delete

default

- explicitní vynucení automaticky generovaných metod
 - konstruktory
 - destruktory
 - přiřazovací operátor

delete

- explicitní zabránění vytvoření metod/funkcí
 - konstruktory
 - destruktory
 - přiřazovací operátor
 - *libovolná funkce*

[ukázka – delete.cpp]

Rekapitulace: `final`, `override`

`final`

- ochrana proti překrytí a proti nedodržení signatury
- platí pro metody i třídy

[ukázka – `finalClass.cpp` `finalMethod.cpp`]

Rekapitulace: `final`, `override`

`final`

- ochrana proti překrytí a proti nedodržení signatury
- platí pro metody i třídy

[ukázka – `finalClass.cpp` `finalMethod.cpp`]

`override`

- ochrana pouze proti nedodržení signatury

Rekapitulace: **auto**

- klíčové slovo žádající překladač o **automagické** odvození typu
 - stejná pravidla jako pro odvozování šablonových parametrů
 - ...s malou výjimkou pro `std::initializer_list`
 - více později

Kdy byste mohli **auto** použít?

Rekapitulace: **auto**

- klíčové slovo žádající překladač o **automagické** odvození typu
 - stejná pravidla jako pro odvozování šablonových parametrů
 - ...s malou výjimkou pro `std::initializer_list`
 - více později

Kdy byste mohli **auto** použít?

- jednoznačně víte, jaký bude výsledný typ
 - iterátory
 - v hlavičce range-foru
- nebo naopak nevíte, co bude výsledný typ
 - pouze v situacích, kdy není možné typ vědět

Rekapitulace: range-for

Co se skrývá za následujícím cyklem?

```
std::vector< std::string > words;  
...  
for ( const auto &word : words )  
    std::cout << word << std::endl;
```

Rekapitulace: range-for

Co se skrývá za následujícím cyklem?

```
std::vector< std::string > words;
...
for ( const auto &word : words )
    std::cout << word << std::endl;

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

Rekapitulace: range-for

Požadavky na vaši třídu:

1. implementovat iterátor
2. implementovat metodu `begin`
3. implementovat metodu `end`

Lambdy: představení (C++11)

- anonymní funkce
- syntaktická zkratka za funkční objekt

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

Lambdy: syntax (C++11)

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

- ne všechny části jsou povinné
 - lze vynechat návratový typ
 - lze vynechat seznam parametrů

Lambdy: zachytávání (C++11)

Co a jak zachytávat?

- `[]` – nic
- `[a]` – a hodnotou
- `[&b]` – b referencí (teoreticky)
- `[=]` – vše hodnotou (konstantní)
- `[&]` – vše referencí (teoreticky)
- `[this]` – `this` hodnotou
- `[&,this]` – vše referencí, ale `this` hodnotou

Lambdy: ukládání (C++11)

Co je lambda?

Lambdy: ukládání (C++11)

Co je lambda?

Lambda je třída.

- pokud nic nechytá, lze přetypovat na ukazatel na funkci
- lze uložit do
 - `auto`
 - `std::function< signatura >`
 - šablonového parametru

Lambdy: ukládání (C++11)

Co je lambda?

Lambda je třída.

- pokud nic nechytá, lze přetypovat na ukazatel na funkci
- lze uložit do
 - `auto`
 - `std::function< signatura >`
 - šablonového parametru

[ukázka – lambda.cpp]

Lambdy (C++14)

Zavedení šablonových parametrů

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

Lambdy (C++14)

Zavedení šablonových parametrů

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

Rozšíření o možnost inicializace v capture sekci.

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

Jaká bude hodnota proměnných `x` a `y`?

Lambdy (C++14)

Zavedení šablonových parametrů

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

Rozšíření o možnost inicializace v capture sekci.

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

Jaká bude hodnota proměnných x a y?

[ukázka – lambdaTest.cpp]

Neprobádané vody C++11/C++14

- `static_assert`
- uniformní inicializace
- zobecněné konstantní výrazy
- move sémantika
- variadické šablony
- práce s pamětí à la C++14
- SFINAE
- asynchronní zpracování

Statický assert

- kontrola předpokladů během překlada
- vyhodnocení výrazu
- případně zhlášení chyby

```
static_assert(  
    sizeof( bool ) > 1,  
    "bool is greater than one byte" );
```

Co se stane?

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


Co se stane?

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

Deklarace funkce `line`, která vrací `std::string`.

- známé jako most-vexing-parse
- bez parametrů → bez závorek
- jeden parametr
 - → závorky
 - → rovnítko
- více parametrů → závorky

Snaha o sjednocení syntaxe vytváření instancí.

- lze použít složené závorky
- limitace pro přetypování
 - pouze neomezuující přetypování

```
std::string line{};  
std::string name{ "Yoda" };
```

Použitím složených závorek se problém vyřeší.

Snaha o sjednocení syntaxe vytváření instancí.

- lze použít složené závorky
- limitace pro přetypování
 - pouze neomezuující přetypování

```
std::string line{};  
std::string name{ "Yoda" };
```

Použitím složených závorek se problém vyřeší.

Mezitím v paralelním vesmíru...

Umíme staticky inicializovat pole.

```
int array[] = { 1, 2, 3, 4, 5 };
```

Umíme staticky inicializovat pole.

```
int array[] = { 1, 2, 3, 4, 5 };
```

Chceme umět obdobně inicializovat vektor.

```
std::vector< int > data{ 1, 2, 3, 4, 5 };
```

Inicializační seznam

- složené závorky mají svůj typ (lži dětem)
 - `std::initializer_list< T >`
 - lze tak použít v konstruktoru vlastní třídy

Jak je možné, že funguje toto?

```
for ( int i : { 1, 2, 3 } )  
    std::cout << i;
```

Inicializační seznam

- složené závorky mají svůj typ (lži dětem)
 - `std::initializer_list< T >`
 - lze tak použít v konstruktoru vlastní třídy

Jak je možné, že funguje toto?

```
for ( int i : { 1, 2, 3 } )  
    std::cout << i;
```

Dříve deklarovaná výjimka pro `auto`.

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?

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 – uniformInitialization.cpp]

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 – uniformInitialization.cpp]

Inicializační seznam má přednost.

Zobecněné konstantní výrazy

Možnost nechat si vyhodnotit funkci během překladu.

- `constexpr` specifikátor
- funkci lze volat i v době provádění programu
- v C++11 poněkud bezzubé
 - pouze jeden `return` příkaz
 - podmínka řešena pomocí ternárního operátoru

Zobecněné konstantní výrazy

Možnost nechat si vyhodnotit funkci během překladu.

- `constexpr` specifikátor
- funkci lze volat i v době provádění programu
- v C++11 poněkud bezzubé
 - pouze jeden `return` příkaz
 - podmínka řešena pomocí ternárního operátoru
- v C++14 rozšířena paleta nástrojů
 - funkce nesmí obsahovat
 - `goto`
 - výjimky (`try-catch` bloky)
 - inline assembler
 - a ještě pár dalších. . .
 - celý strom výpočtu musí být konstantně vyhodnotitelný

[ukázka – `constexpr.cpp`]

- L-hodnota
 - “To, co může být na levé straně od rovnítka.”
 - proměnná, reference, bitfield, dereferencovaný ukazatel
- R-hodnota
 - “To ostatní.”
 - dočasné objekty, konstanty

Příklad použití:

```
void grabber( std::string &&text ) {  
    std::cout << text << std::endl;  
    text.clear();  
}  
  
std::string s = "text";  
  
grabber( "blaabol" );  
grabber( std::string( "kecy v kleci" ) );  
  
//grabber( s );  
  
grabber( std::move( s ) );  
std::cout << '[' << s << ']' ;
```

Move sémantika: The Rule of Three and Half

Běžná třída A s implementovaným pravidlem 3,5.

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

Move sémantika: The Rule of Four and Half

Stačí přidat konstruktor, zbytek se vyřeší sám.

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


Move sémantika: ukázka

- výkonnostní optimalizace
 - přepojení ukazatelů namísto kopírování
- přímo se moc nesetkáte
 - vnitřně používají kontejnery z STD
 - move konstruktor
 - `std::move`

[ukázka – move.cpp]

Problém: v době psaní funkce nevím, kolik dostane parametrů.

Problém: v době psaní funkce nevím, kolik dostane parametrů.

- řešení C/C++
 - `void foo(int, ...);`
 - uvnitř použít `va_args`
- řešení C++11
 - variadické šablony
 - poměrně mnoho uplatnění ve standardní knihovně

Ukázka využití za malý moment.

Přístupy práce s pamětí v programovacích jazycích.

- C
 - ruční kontrola práce s pamětí
- Java, PHP, Javascript, Python, C#, ...
 - garbage collector
- Haskell
 - paměť se nějak děje. ... se stim smíř

Přístupy práce s pamětí v programovacích jazycích.

- C
 - ruční kontrola práce s pamětí
- Java, PHP, Javascript, Python, C#, ...
 - garbage collector
- Haskell
 - paměť se nějak děje. ... se stim smíř
- C++
 - RAI
 - lze využít pro chytré ukazatele
 - moderní C++14 nepoužívá **new** ani **delete**
 - (C++11 dovoluje přilinkovat GC)

Chytré ukazatele.

- `std::unique_ptr< T >`
 - symbolizuje unikátní vlastnictví paměti
 - lze pouze přesouvat
 - dealokuje v destruktoru
 - jako doplněk slouží běžný ukazatel

Chytré ukazatele.

- `std::unique_ptr< T >`
 - symbolizuje unikátní vlastnictví paměti
 - lze pouze přesouvat
 - dealokuje v destruktoru
 - jako doplněk slouží běžný ukazatel
- `std::shared_ptr< T >`
 - symbolizuje vlastnictví paměti
 - počítá reference
 - poslední destruktork dealokuje

Chytré ukazatele.

- `std::unique_ptr< T >`
 - symbolizuje unikátní vlastnictví paměti
 - lze pouze přesouvat
 - dealokuje v destruktoru
 - jako doplněk slouží běžný ukazatel
- `std::shared_ptr< T >`
 - symbolizuje vlastnictví paměti
 - počítá reference
 - poslední destruktork dealokuje
- `std::weak_ptr< T >`
 - doplněk k `std::shared_ptr< T >`
 - před použitím nutné konvertovat na `std::shared_ptr< T >`

Práce s pamětí à la C++11

```
std::weak_ptr< int > wp;
void foo() {
    // auto == std::shared_ptr< int >
    if ( auto spt = wp.lock() )
        std::cout << *spt << std::endl;
    else
        std::cout << "wp expired" << std::endl;
}
int main() {
    {
        auto sp = std::make_shared< int >( 42 );
        wp = sp;
        foo(); // 42
    }
    foo(); // wp expired
}
```

Práce s pamětí à la C++11

```
template< typename T, typename... Args >
std::unique_ptr< T > make_unique( Args &&... args ) {
    return { new T( std::forward< Args >( args )... ) };
}

void foo( int *ptr ) {
    *ptr = 42;
    throw std::runtime_error( "whatever" );
}

int main() {
    try {
        auto handle = make_unique< int >();
        *handle = 10;
        foo( handle.get() ); // zpřístupni ukazatel
    } catch( const std::exception &ex ) {
        std::cout << ex.what() << std::endl;
    }
}
```

SFINAE (popsáno už v roce 2002)

Substitution **F**ailure **I**s **N**ot **A**n **E**rror.

```
void foo( double f ) {  
    std::cout << "double: " << f << std::endl;  
}  
void foo( int i ) {  
    std::cout << "int: " << i << std::endl;  
}  
int main() {  
    foo( 3.14 );  
    foo( 1 );  
    unsigned u = 42;  
    foo( u );  
}
```

Co se vypíše?

SFINAE (popsáno už v roce 2002)

Substitution Failure Is Not An Error.

```
void foo( double f ) {  
    std::cout << "double: " << f << std::endl;  
}  
  
void foo( int i ) {  
    std::cout << "int: " << i << std::endl;  
}  
  
int main() {  
    foo( 3.14 );  
    foo( 1 );  
    unsigned u = 42;  
    foo( u );  
}
```

Co se vypíše?

[ukázka – numbers.cpp]

```
template< typename F >
typename std::enable_if<
    std::is_floating_point< F >::value
>::type foo( F f ) {
    std::cout << "double: " << f << std::endl;
}

void foo( int i ) {
    std::cout << "int: " << i << std::endl;
}

int main() {
    foo( 3.14 );
    foo( 1 );
    unsigned u = 42;
    foo( u );
}
```

Nepřehledný zápis návratového typu.

SFINAE

```
template< typename F >
auto foo( F f )
    -> typename std::enable_if<
        std::is_floating_point< F >::value
    >::type
{
    std::cout << "double: " << f << std::endl;
}

void foo( int i ) {
    std::cout << "int: " << i << std::endl;
}

int main() {
    foo( 3.14 );
    foo( 1 );
    unsigned u = 42;
    foo( u );
}
```

Složitější příklad.

- chcete pracovat s libovolným kontejnerem

Složitější příklad.

- chcete pracovat s libovolným kontejnerem
- pro efektivnější práci chcete volat metodu `reserve`

Složitější příklad.

- chcete pracovat s libovolným kontejnerem
- pro efektivnější práci chcete volat metodu `reserve`
- metodu `reserve` mají pouze tyto kontejnery:
 - `std::string`
 - `std::vector`
 - `std::unordered_set`, `std::unordered_multiset`
 - `std::unordered_map`, `std::unordered_multimap`

[ukázka – `sfinae.cpp`]

- podpora vláken
 - vlákna
 - mutexy
 - podmínkové proměnné
 - **asynchronní volání**
 - zabalení vlákna do zadám-práci & vyzvednu-výsledek
- může být potřeba přilinkovat pthread knihovnu
- o paralelním programování více v
 - **IB109** Návrh a implementace paralelních systémů
 - **PV192** Paralelní technické systémy
 - **IV112** Projekt z paralelních aplikací

Asynchronní volání

```
auto handle = std::async(  
    std::launch::async,  
    fibonacci,  
    100  
);  
...  
int result = handle.get();
```

- možné strategie zpracování
 - asynchronní (samostatné vlákno)
 - odložené (líné vyhodnocení)
 - implementace STD si může přidávat další strategie
- vrací `std::future`
 - pomocí metody `get` lze počkat na výsledek
 - vyhozená výjimka se propaguje

[ukázka – `async.cpp`]

Líbilo se vám to?

Líbilo se vám to?

Zaujalo vás to?

Líbilo se vám to?

Zaujalo vás to?

Příští semestr se v předmětu PB173 Tematické programování v C/C++ otevírají skupiny pro pokročilé C++.