PB161 Programování v jazyce C++
Přednáška 9

Jmenné prostory
Výjimky podrobně

Nikola Beneš

20. listopadu 2018

Jmenné prostory

Motivace

Problém: výskyt dvou entit se stejným jménem

```
// library1.h
class Object { /* ... */ };

// library2.h
class Object { /* ... */ };

// main.cpp
#include "library1.h"
#include "library2.h"
```

Při překladu main.cpp dojde k chybě:

```
error: redefinition of ‘class Object’
```

Implicitní jmenné prostory

```
int x;
// double x; // CHYBA!
class Example {
    float x;
public:
    void method() const {
        double x;
        for (int i = 0; i < 3; ++i) {
            std::string x;
        }
        {
            char x;
        }
    }
};
```

Implicitní jmenné prostory

globální jmenný prostor

```
int x;
// double x; // CHYBA!
class Example {
    float x;
public:
    void method() const {
        double x;
        for (int i = 0; i < 3; ++i) {
            std::string x;
        }
        {
            char x;
        }
    }
};
```

Implicitní jmenné prostory

globální jmenný prostor

```
int x;
// double x; // CHYBA!
class Example {
    float x;
public:
    void method() const {
        double x;
        for (int i = 0; i < 3; ++i) {
            std::string x;
        }
        {
            char x;
        }
    }
};
```

jmenný prostor třídy Example

Implicitní jmenné prostory

globální jmenný prostor

```
int x;
// double x; // CHYBA!
class Example {                          jmenný prostor třídy Example
    float x;
public:
    void method() const {  jmenný prostor metody method
        double x;
        for (int i = 0; i < 3; ++i) {
            std::string x;
        }
        {
            char x;
        }
    }
};
```

Implicitní jmenné prostory

globální jmenný prostor

```
int x;
// double x; // CHYBA!
class Example {                          jmenný prostor třídy Example
    float x;
public:
    void method() const { jmenný prostor metody method
        double x;
        for (int i = 0; i < 3; ++i) {
            std::string x;
        }                                jmenný prostor cyklu for
        {
            char x;
        }                                jmenný prostor bloku
    }
};
```

Jmenné prostory

Přístup ke jmenným prostorům – operátor `::`

```
int x = 17;
class Example {
    int x = 29;
public:
    void print() const {
        int x = 3;
        {
            int x = 9;
            cout << x << endl; // 9
            cout << Example::x << endl; // 29
            cout << ::x << endl; // 17
        }
        cout << x << endl; // 3
    }
};
```

lecture09_01.cpp

Explicitní jmenné prostory

Pojmenované jmenné prostory – syntax: `namespace jmeno { ... }`

```
namespace MyLib {
void print();
namespace Experimental { // možno i vnořovat
    void print();
} // namespace Experimental
} // namespace MyLib

namespace MyLib {
class Example {
public:
    void print() const;
};
} // namespace MyLib
```

lecture09_02.cpp

Explicitní jmenné prostory – zpřístupnění

Zpřístupnění jmenného prostoru

- plná kvalifikace – `std::string`
- direktiva `using namespace jmeno_prostoru;`

```
#include <string>

string s; // CHYBA!

void print() {
    using namespace std;
    string s; // OK
}

int main() {
    string s; // CHYBA!
}
```

Explicitní jmenné prostory – zpřístupnění (pokr.)

- deklarace `using jmeno_prostoru::jmeno_entity`
  - má prioritu před `using namespace`

```
#include "libAdam.h"
#include "libEve.h"
using namespace Adam; // obsahuje funkci getApple();
using namespace Eve;  // taky obsahuje getApple();

getApple(); // CHYBA!

using Eve::getApple;

getApple(); // OK, volá se Eve::getApple();
```

- alias jmenného prostoru

```
namespace SysWinWidget
    = System::Window::Widget;
```

lecture09_03.cpp

Explicitní jmenné prostory – zpřístupnění

Používání `using` a `using namespace`

- v globálním prostoru
  - užívejte rozumně
  - **nikdy v hlavičkových souborech**
  - vždy až po všech `#include`
- lokálně
  - ve funkcích/metodách
  - ve vnořených blocích
  - není možno používat přímo uvnitř třídy (class scope)

Explicitní jmenné prostory – použití

Použití jmenných prostorů ve vlastních knihovnách:

```
// cool_library.h
#ifndef COOL_LIBRARY_H
#define COOL_LIBRARY_H

namespace CoolLibrary {

class Cool { /* ... */ };

} // namespace CoolLibrary

#endif
```

Anonymní jmenné prostory

```
namespace {
// ...
} // namespace
```

- chová se jako by se vytvořil jmenný prostor unikátního jména, za kterým by okamžitě následovalo `using namespace`
- k čemu je to dobré?

Anonymní jmenné prostory

```
namespace {
// ...
} // namespace
```

- chová se jako by se vytvořil jmenný prostor unikátního jména, za kterým by okamžitě následovalo `using namespace`
- k čemu je to dobré?

- zapouzdření identifikátorů uvnitř jednoho zdrojového souboru (resp. překladové jednotky)
  - při linkování nejdou vidět z ostatních překladových jednotek
- podobné jako globální `static` v C, ale lepší – proč?

Anonymní jmenné prostory

```
namespace {
// ...
} // namespace
```

- chová se jako by se vytvořil jmenný prostor unikátního jména, za kterým by okamžitě následovalo `using namespace`
- k čemu je to dobré?

- zapouzdření identifikátorů uvnitř jednoho zdrojového souboru (resp. překladové jednotky)
  - při linkování nejdou vidět z ostatních překladových jednotek
- podobné jako globální `static` v C, ale lepší – proč?

- globální `static` funguje pouze pro proměnné a funkce, do anonymního namespace můžeme ale zavřít libovolná jména (např. deklarace typů)

Jmenné prostory – další informace

Další čtení pro zvídavé

- http://en.cppreference.com/w/cpp/language/namespace
- http://en.cppreference.com/w/cpp/language/lookup
  - qualified name lookup
  - unqualified name lookup
  - argument-dependent lookup (ADL)

```
namespace Test {
    int x;
    void print(int y) {}
}

int main() {
    print(x); // CHYBA!
    Test::print(x); // CHYBA!
    Test::print(Test::x); // OK
    print(Test::x); // taky OK, ADL
}
```

lecture09_04.cpp

Výjimky

Motivace

Obsluha chyb za běhu programu

- možná řešení:

Motivace

Obsluha chyb za běhu programu

- možná řešení: speciální chybová hodnota, globální příznak

Motivace

Obsluha chyb za běhu programu

- možná řešení: speciální chybová hodnota, globální příznak
  - kód se hůře čte a píše
  - chyby jsou implicitně ignorovány
  - které volání selhalo?
  - co když je třeba chybu propagovat skrze víc funkcí?

- výjimky
  - výjimečné situace za běhu programu
  - vyhození výjimky (*throw*)
  - zachycení výjimky (*catch*) a reakce – i jinde než v místě výjimky
  - používáno ve velké řadě jazyků

Syntaxe výjimek v C++

Vyhození výjimky `throw`

- výjimkou může být libovolná hodnota
  - raději však používáme speciální objekty
  - ve standardní knihovně – `std::exception`

```
void sillyFunction(int x) {
    if (x < 0) throw std::runtime_error("x is negative");
}
```

Zachycení výjimky `try { ... } catch ( ... ) { ... }`

```
int main() {
    try {
        sillyFunction(-7);
    } catch (std::runtime_error& ex) {
        std::cout << "Runtime error: " << ex.what() << '\n';
    }
}
```

lecture09_05.cpp

Syntaxe výjimek v C++ (pokr.)

Zachycení výjimky

- `catch` má formální parametr, kterým se má výjimka zachytit
  - zachycení hodnotou – nedoporučované, může znamenat kopii
  - zachycení referencí – doporučovaný způsob
  - zachytávání libovolné výjimky pomocí `catch (...)`
- reakce v bloku `catch`
  - vyřešení problému
  - znovu vyhození stejné výjimky pomocí `throw;`
  - vyhození jiné výjimky

Throw by value, catch by reference.

Mechanismus zachytávání výjimek

1. vyhodí se výjimka
2. prochází se skrz zásobník funkcí, dokud se nenarazí na blok `try`
3. hledá se související blok `catch`, který může výjimku zachytit
   - stejný typ výjimky a parametru
   - parametr je reference na typ výjimky
   - parametr je předek typu výjimky (reference, ukazatel)
   - `(...)` chytá vše
4. pokud se najde správný blok `catch`:
   - **volají se destruktory lokálních objektů** na zásobníku
   - tzv. odvinování zásobníku (*stack unwinding*)
   - nakonec se provede tělo bloku catch
5. pokud se správný blok `catch` nenajde, pokračuje se s hledáním od bodu 2
6. pokud se výjimka nezachytí nikde, zavolá se `std::terminate`
   - v tom případě se destruktory *nemusí* zavolat

`lecture09_06.cpp`

Standardní výjimky

Hierarchie výjimek standardní knihovny

- základní `std::exception`
- virtuální metoda `what()` vrací popis výjimky
- vyhazovány standardní knihovnou
  - př. metoda `at()` kontejnerů
- vyhazovány některými konstrukcemi jazyka C++
  - operátor `new` může vyhodit `std::bad_alloc`
  - operátor `dynamic_cast` může vyhodit `std::bad_cast`

Standardní výjimky (pokr.)

Výjimka při nepodařené alokaci

```
int main() {
    const size_t SIZE = 1000;
    try {
        auto array = std::make_unique<int[]>(SIZE);
        // není třeba testovat na nullptr
        for (int i = 0; i < SIZE; ++i) {
            array[i] = i*i;
        }
        // atd ...
    }
    catch (std::bad_alloc&) {
        std::cerr << "Failed to allocate memory.\n";
    }
}
```

lecture09_07.cpp

Vlastní výjimky

- doporučeno: dědit ze standardních výjimek

```
class WrongNameException : public std::invalid_argument {
    std::string name;
public:
    WrongNameException(const std::string& reason,
                       const std::string& n)
        : std::invalid_argument(reason), name(n) {}
    const std::string& getName() const { return name; }
};
class Person() {
    std::string name;
    static bool isValidName(const std::string&);
public:
    Person(const std::string& n) : name(n) {
        if (!isValidName(name))
            throw WrongNameException("invalid name", name);
        // ...
    }
```

lecture09_08.cpp

Výjimky a dědičnost

- při zachytávání můžeme použít typ předka
- zachytávání probíhá v pořadí bloků `catch` v kódu
- doporučení: řadit bloky `catch` od konkrétních k obecným

```
try {
    // ...
} catch (std::invalid_argument&) {
    // ...
} catch (WrongNameException&) {
    // ...
}
```

```
// warning: exception of type 'WrongNameException' will be
// caught by earlier handler for 'std::invalid_argument'
```

lecture09_08.cpp

Zachycení libovolné výjimky

Zachycení pomocí `catch (...)`

- nemáme přístup k objektu výjimky
  - alespoň ne úplně snadný
  - odvážní mohou zkusit hledat `std::current_exception` (C++11)
- používat opatrně
- v některých specifických případech se ale hodí
  - např. obalení těla destruktoru
- použití s opětovným vyhozením `throw;`
  - logování problémů
  - speciální funkce pro řešení výjimek

`lecture09_09.cpp`

Výjimky a konstruktory

Je vhodné vyhazovat výjimku z konstruktoru?

Výjimky a konstruktory

Je vhodné vyhazovat výjimku z konstruktoru? ANO

Kdy?

- pokud nemůžeme zaručit správný (konzistentní) stav objektu

Co se stane?

- *nezavolá* se destruktor objektu
- zavolá se destruktor všeho, co už bylo inicializováno (předci, atributy)
  - v opačném pořadí inicializace

Jak zachytit výjimku v konstruktoru?

`lecture09_10.cpp`

Výjimky a konstruktory

Je vhodné vyhazovat výjimku z konstruktoru? ANO

Kdy?

- pokud nemůžeme zaručit správný (konzistentní) stav objektu

Co se stane?

- *nezavolá* se destruktor objektu
- zavolá se destruktor všeho, co už bylo inicializováno (předci, atributy)
  - v opačném pořadí inicializace

Jak zachytit výjimku v konstruktoru?

- normálně pomocí `try ... catch`

Co když je výjimka vyvolána při inicializaci?

`lecture09_10.cpp`

Výjimky a konstruktory (pokr.)

Speciální syntax pro konstruktory

```
class Person {
    std::string name;
public:
    Person(const std::string& n) : name(n) {}
};
class Teacher : public Person {
    std::vector< Course > courses;
    Person& departmentBoss;
public:
    Teacher(const std::string& name, Person& boss)
    try : Person(name), departmentBoss(boss) {
        courses.reserve(5);
    } catch (std::exception& ex) {
        std::cerr << "Teacher constructor failed: " << ex.what()
                  << std::endl;
    }
};
```

lecture09_11.cpp

Výjimky a konstruktory (pokr.)

Speciální syntax pro konstruktory

- použitelná i pro jiné metody/funkce, ale tam nemá moc význam

- destruktory předků a atributů se volají před blokem `catch`
- blok `catch` *musí* znovu vyhodit výjimku
  - implicitní `throw;` na konci bloku
- hlavní použití: logování nebo úprava výjimek

Výjimky a destruktory

Je vhodné vyhazovat výjimku z destruktoru?

Výjimky a destruktory

Je vhodné vyhazovat výjimku z destruktoru? NE

- když v průběhu zachycení výjimky vznikne další výjimka, zavolá se `std::terminate`

Specifikace `noexcept`

- úmysl nevyhazovat z funkce/metody žádnou výjimku

```
void f();           // může vyhodit libovolnou výjimku
void g() noexcept; // slibuje, že nebude vyhazovat výjimky
```

- kompilátor může tuto informaci použít pro optimalizace
- standardní knihovna může tuto informaci použít pro volbu chování
- operátor `noexcept`

```
std::cout << boolalpha;
std::cout << noexcept( 2 + 3 )     << '\n'; // true
std::cout << noexcept( throw 17 ) << '\n'; // false
std::cout << noexcept( f() )       << '\n'; // false
std::cout << noexcept( g() )       << '\n'; // true
```

- co když `g()` přesto vyhodí výjimku?
- destruktory jsou automaticky `noexcept`

Specifikace `noexcept`

- úmysl nevyhazovat z funkce/metody žádnou výjimku

```
void f();           // může vyhodit libovolnou výjimku
void g() noexcept; // slibuje, že nebude vyhazovat výjimky
```

- kompilátor může tuto informaci použít pro optimalizace
- standardní knihovna může tuto informaci použít pro volbu chování
- operátor `noexcept`

```
std::cout << boolalpha;
std::cout << noexcept( 2 + 3 )     << '\n'; // true
std::cout << noexcept( throw 17 ) << '\n'; // false
std::cout << noexcept( f() )       << '\n'; // false
std::cout << noexcept( g() )       << '\n'; // true
```

- co když `g()` přesto vyhodí výjimku? `std::terminate`
- destruktory jsou automaticky `noexcept`

Výjimky při vstupu a výstupu

Knihovna `iostream`

- implicitně nepoužívá výjimky, ale nastavuje příznaky
- důvody
  - historické
  - ne vždy je vhodné používat výjimky pro vstup a výstup
- použití výjimek je možno vynutit
  - pomocí metody `exceptions`
  - příznaky typu `std::ios_base::iostate`, kombinace pomocí |
  - výjimka typu `std::ios_base::failure`

`lecture09_12.cpp`

Doporučení

- výjimkami řešte výjimečné situace
  - chyby, špatné parametry, apod.
  - tam, kde je jiné řešení nemožné/nevhodné (konstruktory, operátory)
- nepoužívejte výjimky pro vracení hodnot z funkcí a metod
  - nenalezení prvku v poli nemusí být výjimečná situace
- házejte hodnotou, chytejte referencí
- zachytávání výjimek v inicializaci konstruktorů používejte, pokud chcete logovat nebo nějak měnit zachycenou výjimku
- nevyhazujte výjimky z destruktorů
- chytejte výjimky jen tehdy, pokud máte na výjimku jak rozumně reagovat

Závěrečný kvíz

https://kahoot.it

Závěrečný kvíz (kód č. 1)

```
void print() { std::cout << "x"; }
namespace A {
    void print() { std::cout << "y"; }
    namespace B {
        void print() { ::print(); }
    } // namespace B
} // namespace A
namespace C {
    void fun() {
        using namespace A;
        A::print();
        B::print();
    }
} // namespace C
int main() {
    using namespace C;
    print();
    fun();
}
```

Závěrečný kvíz (kód č. 2)

```
class A { /* ... */ };
class B : public A { /* ... */ };
class C : public B { /* ... */ };
class D { /* ... */ };
void foo() {
    D d;
    throw B();
}
int main() {
    try {
        foo();
    }
    catch (C& ex) { cout << 1; }
    catch (A& ex) { cout << 2; }
    catch (B& ex) { cout << 3; }
}
```