

PB161 Programování v jazyce C++

Přednáška 9

Jmenné prostory

Výjimky

Nikola Beneš

22. listopadu 2016

Jmenné prostory

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

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;
public:
    Example() : x(29) {}
    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
    }
};
```

Explicitní jmenné prostory

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

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

[ukázka použití]

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;
```


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 cool_library {

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

}

#endif
```

Explicitní jmenné prostory – použití (pokr.)

Oddělení kolidujících jmen při použití cizích knihoven:

```
namespace Lib1 {  
#include "library1.h" // obsahuje třídu System  
}  
  
namespace Lib2 {  
#include "library2.h" // obsahuje třídu System  
}  
  
int main() {  
    System s; // CHYBA!  
    Lib1::System s1; // OK  
    Lib2::System s2; // OK  
}
```

Anonymní jmenné prostory

```
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 zvídání

- <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

```
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, argument-dependent lookup  
}
```

Výjimky

Windows

A fatal exception 0E has occurred at 0028:C0034B23. The current application will be terminated.

- * Press any key to terminate the current application.
- * Press CTRL+ALT+DEL again to restart your computer. You will lose any unsaved information in all applications.

Press any key to continue _

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 42;  
}
```

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

```
int main() {  
    try {  
        sillyFunction( -7 );  
    } catch (int e) {  
        cout << "exception no.: " << e << endl;  
    }  
}
```


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 **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

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";  
    }  
}
```

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

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'*

Zachycení pomocí `catch` (...)

- nemáme přístup k objektu výjimky
- používat opatrně
- v některých specifických případech se ale hodí
 - např. obalení těla destrukturu
- použití s opětovným vyhozením `throw`;
 - logování problémů
 - speciální funkce pro řešení výjimek

Zachycení libovolné výjimky (pokr.)

```
void handleException() {  
    try { throw; }  
    catch (SomeException& ex) { // ...  
    }  
    catch (OtherException& ex) { // ...  
    }  
}
```

```
int main() {  
    try {  
        // some code  
        // ...  
    }  
    catch (...) {  
        handleException();  
    }  
}
```


Je vhodné vyhazovat výjimku z konstruktoru? ANO

Kdy?

- pokud nemůžeme zaručit správný stav

Co se stane?

- *nezavolá* se destruktorka objektu
- zavolá se destruktorka 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?

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;
    }
};
```

Speciální syntax pro konstruktory

- použitelná i pro jiné metody/funkce, ale 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

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**

```
cout << boolalpha;  
cout << noexcept( 2 + 3 ) << endl;    // true  
cout << noexcept( throw 17 ) << endl; // false  
cout << noexcept( f() ) << endl;      // false  
cout << noexcept( g() ) << endl;      // true
```

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

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

Výjimky při vstupu a výstupu (pokr.)

```
#include <iostream>
#include <fstream>

using namespace std;

int main() {
    try {
        ifstream input("soubor.txt");
        input.exceptions( ifstream::failbit
                          | ifstream::badbit );
        // read from the file
        // the file is automatically closed
    }
    catch (ios_base::failure& ex) {
        cerr << "I/O exception: " << ex.what();
    }
}
```

- 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 není výjimečná situace
- házejte hodnotou, chyťte referencí
- zachytávání výjimek v inicializaci konstruktorů používejte, pokud chcete logovat nebo nějak měnit zachycenou výjimku
- nevyhazujte výjimky z destruktorů
- chyťte výjimky jen tehdy, pokud máte na výjimku jak rozumně reagovat

`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 C {  
    void fun() {  
        using namespace A;  
        A::print();  
        B::print();  
    }  
}  
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; }
}
```