

PB161 Programování v jazyce C++

Přednáška 8

Výjimky
Správa prostředků (RAII)

Nikola Beneš

9. listopadu 2015

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 _

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 skrz zásobník funkcí, dokud nenarazí na blok **try**
 - odvinování zásobníku (*stack unwinding*)
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**, provede se;
pokud se nenajde, pokračuje se s odvinováním zásobníku (bod 2.)
5. pokud se výjimka nezachytí nikde, zavolá se `std::terminate`

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 int SIZE = 1000;  
    try {  
        int * pole = new int[SIZE];  
        // není třeba testovat na nullptr  
        for (int i = 0; i < SIZE; ++i)  
            pole[i] = i*i;  
        // atd ...  
        delete [] pole;  
    }  
    catch (std::bad_alloc &) {  
        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, 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(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 destruktoru
- 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?

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?

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(std::string n) : name(n) {}
};

class Teacher : public Person {
    std::vector< Course > courses;
    Person & departmentBoss;
public:
    Teacher(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`;
- hlavní použití: logování nebo úprava výjimek

Je vhodné vyhazovat výjimku z destruktoru?

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?
- 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**

```
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**

Uvolnění prostředků

- paměť, otevřené soubory, zámky, síťová spojení, ...
- možné řešení: uvolňovat prostředky jak v bloku `try`, tak v bloku `catch`
- řešení v jiných jazycích: `finally`
- řešení v C++: *Resource Acquisition Is Initialisation* (RAII)
 - používá se i v jiných jazycích (D, Ada, Rust)

Princip RAII

- správa prostředku spjatá s životním cyklem objektu
- inicializace: získání prostředku
- destruktory: uvolnění prostředku
- ideálně: jeden objekt spravuje jeden prostředek

Deterministické volání destruktorků

- na konci bloku
- při odvinování zásobníku
- destruktory potomků před destruktory předků
- opačné pořadí než konstruktory

Jiný název: *scope-based resource management* (SBRM)

Rozdíly proti finally

- lokalita: získávání/uvolňování prostředků na jednom místě
- stručnost: kód pro jeden druhu prostředku píšeme jednou

Destruktory/kopírovací konstruktory/kopírovací operátor =

- třída spravující prostředek: **Rule of three**
 - buď nekopírovat vůbec nebo definovat explicitní kopírování
- třídy, které nespravují prostředky: **Rule of zero**
 - implicitní destruktory a kopírování

V běžném kódu se téměř nikdy nestaráme o (de)alokaci paměti.

Kde se používá RAII?

- skoro všude
- standardní knihovna
 - `string`, `vector` a jiné kontejnery
 - vstupně/výstupní proudy
 - chytré ukazatele (C++11) – `unique_ptr` a spol.
 - zamykání, mutexy (C++11)

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();
    }
}
```

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 není výjimečná situace
- házejte hodnotou, chytějte referencí
- zachytávání výjimek v inicializaci konstruktorů používejte, pokud chcete logovat nebo nějak měnit zachycenou výjimku
- nevyhazujte výjimky z destruktorů
- používejte RAII
 - ideálně: jeden objekt – jeden prostředek