

PB161 – 8. přednáška (9. listopadu 2015)

Výjimky

Motivace

- hlášení a obsluha chyb za běhu programu
 - nečekaná situace, kterou je možno nějak řešit
 - nedostatek paměti při alokaci
 - dělení nulou
 - nedovolené argumenty funkce
 - apod.
- možná řešení (znáte z C)
 - vracet speciální chybovou hodnotu (př. `malloc` v C může vrátit `NULL`)
 - nastavovat globální příznak (př. v C `errno`)
- problémy těchto řešení
 - použití globálního příznaku vede k tomu, že chyby jsou implicitně ignorovány
 - navíc nemusí být jasné, které volání selhalo
 - vracení hodnoty má zase problém v tom, když je třeba chybu propagovat skrze víc funkcí
 - kód se někdy hůře píše a čte (opakované testy, jestli nenastala chyba)
- výjimky
 - objekty, které reprezentují výjimečné situace za běhu programu
 - vyhození výjimky (*throw*) označuje, že došlo k chybovému stavu
 - výjimky je možno zachytávat (*catch*) a reagovat na ně – i jinde než v místě výskytu výjimky
 - používáno ve velké řadě jazyků

Syntaxe pro výjimky v C++

- vyhození výjimky – **throw**
- výjimkou může být libovolná hodnota
 - je ale rozumným zvykem používat speciálně k tomu určené objekty
 - ve standardní knihovně – hierarchie výjimek `std::exception`
 - <http://en.cppreference.com/w/cpp/error/exception>

```
void sillyFunction(int x) {  
    if (x < 0)  
        throw 42;  
}
```

- zachytávání výjimek – **try** { ... } **catch** (...) { ... }
- blok **try** označuje oblast, v níž se má zachytit výjimka
- jeden nebo více bloků **catch** pak obsahuje kód, který se má spustit při zachycení výjimky

```

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

```

- v závorkách za klíčovým slovem `catch` je formální parametr (podobně jako v deklaraci funkce), kterým se má výjimka zachytit
 - zachycení hodnotou – nepříliš doporučované, může znamenat kopii (a to může znamenat vyhození další výjimky!)
 - zachycení referencí – doporučovaný způsob
 - zachytávání libovolné výjimky (*catch-all*) pomocí `catch (...)` (ano, to jsou skutečně tři tečky)
- v bloku `catch` se můžeme rozhodnout zachycenou výjimku znova vyhodit (poslat dál) pomocí `throw`; bez parametrů
- můžeme ovšem také vyhodit nějakou jinou výjimku
- silně doporučované pravidlo pro házení a chytání výjimek: *throw by value*, *catch by reference* (házejte hodnotou, chytejte referencí)

Mechanismus zachytávání výjimek

- v případě nečekané situace (chyby) se vyvolá výjimka
 - obsahuje informaci o chybě
- výjimka postupně prochází skrz zásobník funkcí, dokud nenarazí na blok `try`
 - dochází k odvinování zásobníku (*stack unwinding*), při kterém se volají destruktory objektů na zásobníku
- po nalezení bloku `try` se procházejí příslušné bloky `catch` a hledá se takový, který může výjimku zachytit
 - stejný typ výjimky a parametru
 - parametr je reference na typ výjimky
 - parametr je (reference na) předka typu výjimky
 - (...) chytá vše
 - podrobně: http://en.cppreference.com/w/cpp/language/try_catch
- pokud se najde správný blok `catch`, provede se jeho kód; pokud se žádný nenašel, pokračuje se s odvinováním zásobníku
- pokud se výjimka nezachytí nikde, zavolá se `std::terminate` a program je ukončen

[ukázka chytání výjimek]

Standardní výjimky

- standardní knihovna obsahuje hierarchii výjimek
 - základní třída je `std::exception`
 - virtuální metoda `what()` vrací popis výjimky (typu `const char *`)

- vyhazovány standardní knihovnou
 - př. metoda `at()` kontejnerů
- vyhazovány některými konstrukcemi jazyka C++
 - operátor `new` může vyhodit `std::bad_alloc`
 - operátor `dynamic_cast` při použití s referencí může vyhodit `std::bad_cast`
 - a další ...

```
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

- doporučený způsob: zdědit od některé ze standardních výjimek
 - typicky z `std::logic_error` nebo `std::runtime_error` nebo některých jejich potomků

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

```

    }
};

```

Dědičnost při zachycování výjimek

- víme, že výjimky mohou tvořit hierarchii
- víme, že při zachytávání můžeme použít typ předka
- zachytávání probíhá v pořadí bloků **catch** v kódu
 - bloky „výše“ jsou vyhodnoceny dříve
- proto je třeba řadit **catch** od konkrétních k obecným (od potomků k předkům)
- naštěstí nás na to kompilátor upozorní

```

try {
    // ...
}
catch (std::invalid_argument &) {
    // ...
}
catch (WrongNameException &) {
    // ...
}
// warning: exception of type 'WrongNameException' will be
// caught by earlier handler for 'std::invalid_argument'

```

Zachycení libovolné výjimky

- zachytit libovolnou výjimku umíme pomocí **catch (...)**
 - některé (ale zdaleka ne všechny) kombinace překladač/běhové prostředí/knihovna umí zachytávat i SIGSEGV apod.
- nemáme ale přístup k objektu výjimky
 - v C++11 k tomu nějaké způsoby jsou, to je ale mimo záběr tohoto předmětu
- používat opatrně
 - v běžném kódu bychom měli zachytávat jen výjimky, které umíme zpracovat
 - okolní kód může být schopen reagovat na výjimku lépe
- v některých specifických případech se ale hodí
 - např. obalení těla destruktora (z destruktora bychom neměli vyhazovat výjimku, viz dále)
- používá se často spolu s opětovným vyhozením téže výjimky pomocí **throw;**
 - logování problémů
 - speciální funkce pro řešení výjimek

```

void handleException() {
    try {

```

```

        throw;
    }
    catch (SomeException & ex) {
        // ...
    }
    catch (OtherException & ex) {
        // ...
    }
}

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

```

Výjimky a konstruktory

- je vhodné vyhazovat výjimku z konstrukturu?
 - ano, pokud nemůžeme zaručit, aby byl objekt ve správném stavu
 - konstruktor nic nevrací, proto je výjimka často jedinou (rozumnou) možností
- co se stane, pokud je vyhozena výjimka z konstrukturu?
 - *nezavolá* se destruktory objektu
 - zavolá se destruktory všech věcí, jejichž inicializace už úspěšně proběhla (předci, atributy)
- jak zachytit výjimku v konstrukturu?
 - pokud je výjimka vyvolána až v těle konstrukturu, normálně pomocí `try ... catch`
 - ale co když je výjimka vyvolána při inicializaci (předka, atributů)?
- speciální syntax pro konstruktory (dá se použít i pro ostatní metody a funkce, ale tam nemá moc význam)

```

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;
    }
};

```

- předtím, než se vejde do bloku **catch**, všechny atributy a předci už byli zrušeni (pomocí svých destruktůrů)
 - takže k nim už není možno přistupovat!
- blok **catch** v tomto případě *musí* znovu vyhodit výjimku
 - pokud tak neučiníme explicitně, zavolá se automaticky na jeho konci **throw**;
 - toto platí jen pro tuto speciální syntax a jen pro konstruktory!
- hlavní použití: logování a případně úprava výjimek

Výjimky a destruktory

- je vhodné vyhazovat výjimku z destrukturu?
 - NE
- problém: co když v průběhu zachycení výjimky vznikne další výjimka?
 - jak se to mohlo stát?
 - jediné tak, že nějaký destruktör vyhodil výjimku
- silné doporučení: NIKDY nevyhazovat výjimky v destrukturu
 - jistě se najdou velmi specifické případy, kdy je toto doporučení vhodné porušit, ale v tomhle předmětu se s nimi zcela jistě nesetkáte

Specifikace **noexcept**

- můžeme kompilátoru sdělit úmysl nevyhazovat z funkce/metody žádnou výjimku
 - pomocí specifikátoru **noexcept**

```

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
 - souvisí s *move semantics* v C++11, více později
- je možno použít operátor **noexcept** na zjištění, zda daný výraz může vyhodit výjimku

```

cout << boolalpha;
cout << noexcept( 2 + 3 ) << endl;    // true

```

```
cout << noexcept( throw 17 ) << endl; // false
cout << noexcept( f() ) << endl;      // false
cout << noexcept( g() ) << endl;      // true
```

- pokud funkce označená jako `noexcept` i přesto vyhodí výjimku, volá se rovnou `std::terminate()`, tj. nedojde k žádnému zachytávání
- destruktory jsou automaticky `noexcept`, u ostatních metod a funkcí je třeba tento specifikátor psát explicitně

Poznámka: V C++03 existovaly ještě tzv. dynamické specifikace výjimek, zapisovaly se pomocí `throw(seznam_vyjimek)` za seznamem parametrů funkce. Tyto specifikace jsou nyní nedoporučované (*deprecated*).

RAII – motivace

- jak uvolnit prostředky (paměť, otevřené soubory, zámky, síťové spojení, apod.)?
 - možné řešení: uvolňovat prostředky jak v bloku `try`, tak v bloku `catch` – nešikovné
 - řešení v jiných jazycích pomocí bloku `finally`
 - řešení v C++: idiom RAII (*Resource Acquisition Is Initialisation*)

RAII

- správa prostředku spjatá s životním cyklem objektu
- při inicializaci objektu se prostředek získá (*acquire*)
- destruktory objektu prostředek opět uvolní (*release*)
- ideálně by měl jeden objekt spravovat vždy jen jeden prostředek
 - obecnější princip dobrého programovacího návrhu: každá entita (objekt, funkce, ...) dělá jen o jednu věc
- klíčovou součástí RAII je to, že víme (deterministicky), kdy a jak se volají destruktory
 - na konci bloku
 - při odvinování zásobníku po vyvolání výjimky
 - destruktory potomků se volají před destruktory předků
 - destruktory se volají v opačném pořadí než konstruktory
- toto spolu s principem RAII zajišťuje korektní uvolňování prostředků
- někdy se též používá termín *scope-based resource management* (SBRM)
- rozdíly oproti `finally`:
 - lokalita kódu: kód pro získávání/uvolňování prostředků je poblíž
 - stručnost: kód pro uvolnění jednoho druhu prostředku píšeme jenom jednou
- s RAII souvisí Rule of three
 - pokud máme spravovat prostředky, je typicky třeba buď zakázat kopírování nebo definovat explicitní kopírovací konstruktor/operátor=
- třídy, které nespravují prostředky, by měly dodržovat *Rule of zero*

- jen implicitní destruktory a kopírování
- důsledek používání RAI – v běžném kódu se téměř nikdy nestaráme o (de)alokaci paměti

Kde se používá RAI?

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

Výjimky a knihovna `iostream`

- knihovna vstupně/výstupních proudů implicitně nepoužívá výjimky, ale nastavuje příznaky
- jednak historické důvody, jednak ne vždy je vhodné používat výjimky pro vstup a výstup (ne všechny příznaky reprezentují výjimečné stavy)
- objektům z hierarchie `iostream` je možno říct, ať používají výjimky

```
#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();
    }
}
```

Souhrn doporučení pro používání výjimek

- výjimkami řešte výjimečné situace
 - chyby, porušení dohody mezi programátorem a funkcí (špatné parametry) apod.
 - zejména na místech, kde je jiné řešení nemožné/nevhodné – konstruktory, operátory
- nepoužívejte výjimky pro vrácení hodnot z funkcí a metod
 - nenalezení prvku v poli není výjimečná situace (podívejte se na standardní knihovnu a funkce `find` apod.)

- házejte hodnotou, chytejte referenci
- 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