# Výjimky; princip RAII

## PB161 Programování v jazyce C++

Nikola Beneš

28. března 2023

# 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 _

# Motivace

## Obsluha chyb / výjimečných situací za běhu programu

- speciální chybová hodnota, globální příznak
  - horší čitelnost kódu
  - implicitní ignorování chyb
  - lokalita obsluhy chyby (výhoda i nevýhoda)
- výjimky (*exceptions*)
  - signál, že funkce (metoda, …) nemůže skončit běžným způsobem
  - vyhození výjimky (*throw*)
  - zachycení výjimky (*catch*) a reakce na ni –
    v libovolné nadřazené funkci (z hlediska zásobníku volání)

*When was the last time you checked the return value of printf?*

*(Bjarne Stroustrup)*

# Syntax výjimek

exceptions.cpp

## Vyhození výjimky `throw`

- výjimkou smí být libovolná hodnota
- hierarchie výjimek ve standardní knihovně: `std::exception`

```
void communicate() {
    if (fail) {
        throw std::runtime_error("disconnected");
    }
    std::cout << "communicating...\n";
}
```

# Syntax výjimek

exceptions.cpp

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

- bloků `catch` může být i více (pro různé typy výjimek)
- možné reakce v bloku `catch`
  - vyřešení problému
  - opětovné vyhození stejné výjimky `throw;`
  - vyhození jiné výjimky (nahrazení, ne řetězení)

```
try {
    connection con;
    con.communicate();
} catch (std::runtime_error& ex) {
    std::cerr << "Runtime error: "
              << ex.what() << '\n';
}
```

# 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`:
   - **konec života lokálních objektů**
     - ve všech blocích, které jsme opustili
   - 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

# Výjimky a efektivita

**Moderní implementace** – tzv. „zero-cost exceptions“

- optimalizace pro „šťastnou cestu“ (*happy path*) bez chyb
- režie pro blok `try` je (skoro) nulová
- výkonost programu, pokud k žádné výjimce nedojde, je (prakticky) stejná, jako by se výjimky vůbec nepoužily
- ale mašinérie zachytávání je náročná a pomalá

**Důsledek:** výjimky používejte jen k řešení *výjimečných* situací

# Standardní a vlastní výjimky

## Hierarchie výjimek standardní knihovny

- https://en.cppreference.com/w/cpp/error/exception
- používá podtypový polymorfismus (virtuální metody)
- metoda `what()` vrací popis výjimky
- vyhazovány standardní knihovnou, jazykovými konstrukcemi
  - metoda `at()` u kontejnerů
  - operátor `new`
  - …

## Vlastní výjimky

- výjimkou může být libovolná hodnota
- bývá zvykem pro vlastní výjimky používat
  - standardní výjimky
  - objekty vlastních typů
  - objekty vlastních typů, které dědí ze standardních výjimek

# Zachytávání výjimek

exceptions.cpp

- bloky **catch** se prochází postupně dle *pořadí v kódu*
- první použitelný **catch** se použije
- důsledek pro hierarchie výjimek:
  pořadí musí jít od konkrétních (potomků) k obecným (předkům)

```
try {
    // ...
} catch (std::runtime_error& ex) {
    std::cerr << "Runtime error: "
              << ex.what() << '\n';
} catch (std::exception& ex) {
    std::cerr << "Generic exception: "
              << ex.what() << '\n';
}
```

# Zachytávání výjimek

catch_all.cpp

## `catch (...)`

- zachycení libovolné výjimky
- nemáme (snadný) přístup k objektu výjimky
  - odvážní mohou hledat `std::current_exception()`
- můžeme znovu vyhodit stejnou výjimku pomocí `throw;`
  - logování problémů
  - speciální funkce pro zpracování výjimek

```
try {
    do_something();
} catch (...) {
    handle_exception();
}
```

# Životní cyklus výjimky

## Objekt výjimky

- vzniká při `throw výraz` zkopírováním výrazu
  - může jít o přesun, je-li výraz *r-hodnota*
  - kopie/přesun se smí úplně vynechat (*copy elision*)
    podobně jako u `return`
- zaniká[1] při opuštění posledního bloku `catch`, který končí jinak než opětovným vyhozením `throw;`
  - opětovné vyhození použije týž objekt (nekopíruje, nepřesouvá)

- běžné použití: *házejte hodnotou, chytejte referencí*
  - `throw` s dočasnou hodnotou
  - `catch` s referencí na typ výjimky (může a nemusí být `const`)

[1] pokud jsme nepoužili `std::current_exception()`, jinak je život objektu výjimky prodloužen – mimo záběr předmětu

# Specifikace noexcept

- úmysl nevyhazovat z funkce / metody žádnou výjimku

```
void f();             // can throw whatever
void g() noexcept;    // promises not to throw
```

- kompilátor může použít pro optimalizace
- standardní knihovna může změnit chování
- pokud funkce s **noexcept** vyhodí výjimku → `std::terminate`

# Úrovně bezpečnosti s výjimkami *(Exception safety)*

**Základní záruka** *(basic exception guarantee)*

- vyhodí-li funkce výjimku, program zůstane ve validním stavu
- nedošlo k žádným únikům paměti ani jiných zdrojů
- invarianty datových struktur zůstaly v platnosti

**Silná záruka** *(strong exception guarantee)*

- vyhodí-li funkce výjimku, stav programu se vrátí do okamžiku před jejím voláním („transakční“ chování)
- např. `push_back` pro `std::vector`

**Záruka neselhání** *(nofail / nothrow exception guarantee)*

- funkce nevyhazuje výjimky
- např. `pop_back` pro `std::vector`

# Odbočka: `std::vector` a silná záruka

`push_back` apod.

- realokace vektoru při znamená *přesun* prvků
- pokud přesouvací konstruktor vyhodí výjimku, není cesty zpět
  - původní objekt už může být „vykradený“
  - nový objekt nevznikl
- realokace použije přesun jen pokud je přesouvací konstruktor `noexcept`
  - nebo pokud neexistuje kopírovací konstruktor (pak ale nedává žádnou záruku)

ctor_exc.cpp

## Vyhození výjimky z konstruktoru

- pokud nemůžeme zaručit správný (konzistentní) stav objektu
  - např. platnost *invariantů* datové struktury
  - *jediný rozumný* způsob jak oznámit chybu z konstruktoru
- život objektu ani nezačne ani neskončí
  - tj. *destruktor objektu se nezavolá*
- život položek a předků skončí (při odvinování zásobníku)
  - tj. zavolají se jejich destruktory

## Výjimky během inicializace

- selže-li inicializace položky / předka
- další položky / předkové už se neinicializují
- skončí život úspěšně inicializovaných položek / předků

ctor_exc.cpp

## Chycení výjimky během inicializace

- speciální syntax: **`try`** před inicializační sekcí
- před vstupem do bloku **`catch`** už skončil život položek / předků
  - tj. v tomto bloku už s nimi nemůžeme nic dělat
- blok **`catch`** vždy znovu vyhodí výjimku (implicitní **`throw;`**)
- použití: logování, úprava výjimek (vyhození jiné)

https://en.cppreference.com/w/cpp/language/function-try-block

# Výjimky a destruktory

## Vyhození výjimky z destruktoru

- téměř vždy špatný nápad
- důvod – destruktory se volají v průběhu odvinování zásobníku
  - výjimka vyhozená během odvinování zásobníku způsobí zavolání `std::terminate`
- destruktory jsou implicitně **`noexcept`**
  - i když je definujeme explicitně a nic k nim nenapíšeme

# Řešení chybových stavů

`std::abort` / `std::terminate`

- není-li možno dál pokračovat
- není-li žádný způsob, jak se ze situace zotavit

`assert`

- tvrzení, která *nutně musí platit*
  - vstupní podmínky funkcí
  - invarianty (cyklů, datových struktur) apod.
- selhání znamená chybu programu (resp. programátora)

# Řešení chybových stavů

## Výjimky

- výjimečné situace, kdy nemůžeme dále pokračovat
  - není možné splnit výstupní podmínku
  - není možné splnit / zachovat invariant datové struktury (konstruktory, operátory)
- chytejte jen tehdy, máte-li jak reagovat
- výjimky (v C++) nejsou nástroj toku řízení!

**Alternativa** – chybový stav v návratové hodnotě

- často formou součtového typu
- (Haskell: `Either`, Rust: `Result`)
- od C++23 `std::expected`

# Úklid po vyhození výjimky

`finally`

- v C++ nic takového nemáme!
- úklid zajistí destruktory lokálních objektů
  - spuštěné při odvinování zásobníku
- RAII

# Princip RAII


*RAII is the greatest contribution C++ has made to software development.* *(Russel Winder)*

# Motivace

```
void do_something(int size) {
    char *mem = malloc(size);

    // ... do something with mem ...

    free(mem);
}
```

- co když funkci opustíme před tím, než se provede řádek s `free`?
  - memory leak!
- podobné problémy?
  - otevření a zavření souboru
  - různé druhy zámků (mutex)
  - vytvoření a uzavření síťového přípojení
  - ...

# Motivace

## Zdroj (*resource*)

- něco, co je třeba získat (*acquire*)
- potom to používáme (*use*)
- nakonec to musíme vrátit (*release*)

## Příklady zdrojů

- dynamicky alokovaná paměť
- soubory
- zámky, mutexy, semafory, …
- síťová připojení, připojení k databázi, …
- grafické prvky, okna, textury, …

Správa všech zdrojů funguje v C++ na stejném principu – **RAII**.

## Resource Acquisition is Initialisation

- někdy též *scope-based resource management*
- správa zdroje spjatá s životním cyklem objektu
  - třída reprezentující druh zdroje
  - jeden objekt spravuje jednu instanci zdroj
- inicializace objektu – získání zdroje (*acquire*)
- konec života objektu – uvolnění zdroje (*release*)

## Kde se používá RAII?

- skoro všude!
- kontejnery
- vstupně/výstupní proudy (uvidíme později)
- chytré ukazatele
- zamykání, mutexy (nad rámec kurzu)

## Třída pro správu zdroje

- jedna třída – jeden druh zdroje
- konstruktor získá zdroj
  - objekt třídy *vlastní* jednu instanci zdroje
  - alternativně: konstruktor vytvoří „prázdný“ objekt, zdroj se získá později explicitním voláním některé metody
- kopírovací konstruktor
  - často nemá smysl (zdroj nemusí být kopírovatelný)
  - je-li zdroj kopírovatelný, pak získá kopii
- přesouvací konstruktor
  - přebere vlastnictví zdroje
  - původní objekt zanechá v „prázdném stavu“
  - alternativně: nepřesouvatelný objekt

# Správa zdrojů

## Třída pro správu zdroje

- přiřazení
  - uvolní aktuálně držený zdroj
  - získá zdroj z „pravé strany“ (kopírováním, přesunem)
- destruktor
  - uvolní aktuálně držený zdroj
  - (je-li objekt „prázdný“, nedělá nic)

## Co když chceme spravovat více zdrojů?

- kompozice!

# RAII a kompozice

composition.cpp

## RAII třída

- jeden druh zdroje
- *rule of five*

## Třída využívající zdroje

- drží položky RAII typů
  - nebo jiných typů využívajících zdroje
  - nebo kontejnerů RAII objektů
- nestarají se o správu zdrojů
- *rule of zero*
  - o všechno se postarají implicitně vygenerované metody

# Příklad: Obal (*wrapper*) pro Céčkové soubory

cfile.cpp

```
class c_file {
    std::FILE* ptr;

public:
    c_file(c_file&& other) noexcept
      : ptr(std::exchange(other.ptr, nullptr)) {}

    c_file& operator=(c_file&& other) noexcept {
        close();
        ptr = std::exchange(other.ptr, nullptr);
        return *this;
    }

    ~c_file() { close(); }
    // ...
};
```

# Idiom „Scope Guard“

timer.cpp

## Stráž bloku

- jednoduchý objekt ve stylu RAII pro lokální použití
- typicky nekopírovatelný, nepřesouvatelný
- příklady:
  - náhrada `defer` z jiných jazyků
  - zamknutí / odemknutí mutexu
  - měření času

```
struct timer {
    std::time_t start;
    int& result;
    timer(int& result)
      : start(std::time(nullptr)), result(result) {}
    ~timer() { result = std::time(nullptr) - start; }
};
```

# Správa zdrojů v jiných jazycích

Některé jazyky mají RAII

- **C++**, D (částečně), Ada, Rust mají RAII
- tak trochu: Perl, Python (CPython), PHP mají *reference counting* a něco jako destruktory (v Pythonu nedoporučováno používat)

Jiné způsoby správy zdrojů

- garbage collector: správa paměti, typicky ne jiných zdrojů (není záruka, kdy a jestli vůbec se zavolají destruktory)
- Java (od 1.7)
  - `synchronized` (specificky pro zámky)
  - `try (Resource res = new Resource(...)) { ... }`
- Python (od 2.5)
  - `with Resource(...) as resource:`
- C#
  - `using (Resource res = new Resource(...)) {...}`

# Srovnání RAII s jinými způsoby správy zdrojů

## Úklid v bloku `finally`

- je třeba napsat v každém místě použití zdroje
- RAII: destruktor píšeme jednou pro každý *druh* zdroje
- v reálných systémech je méně druhů zdrojů než míst použití

```
try {
    // work with one file
} finally {
    // close the file
}
try {
    // work with another file
} finally {
    // close the file
}
```

# Srovnání RAII s jinými způsoby správy zdrojů

Kontextové manažery (`with`, `try(...)`, `using`)

- podobné RAII, ale nedají se snadno komponovat
- kompozice zdrojů s RAII je prakticky zadarmo
- životnost RAII objektu nemusí nutně být jen lokální blok
  - může být položkou dynamicky alokovaného objektu
  - funkce mohou vracet zdroje (a volající nemusí přemýšlet o tom, že volání funkce musí zavřít do `with` apod.)
  - move sémantika umožňuje objekt přesunout jinam

```
class App {
    Renderer renderer;
    std::vector<Window> windows;
    std::map<std::string, Texture> textures;
    // ...
};
```

*Kontrolní otázka: Jak se v C++ správně uvolňuje zámek nebo paměť, zavírá soubor nebo síťové spojení apod.?*

*Odpověď:* `}`

*(zdroj: okoun.cz/boards/programovani)*