**NPRG041** – 2025/26 ZS – Cvičení MS – **Malý úkol C04**

# Databáze filmů I

Cílem tohoto úkolu je naprogramovat první část aplikace, pomocí které budeme schopni simulovat práci s databází filmů, čímž myslíme schopnost jednotlivé filmy reprezentovat a ukládat (vytvářet jejich instance a uchovávat je ve standardním vektoru) a následně se nad nimi dotazovat (hledat filmy odpovídající určitým vyhledávacím kritériím a následně je vypisovat na standardní výstup).

U každého filmu potřebujeme evidovat následující datové položky: název filmu (typu `std::string`), rok natočení (`int`), žánr (`std::string`), hodnocení (`int`) a množinu jmen herců (`std::string`), kteří v daném filmu hráli. Množinu těchto herců realizujeme pomocí dalšího typu kontejneru, konkrétně `std::set`, který najdeme v knihovně `<set>`. Všechny údaje jsou povinné, počet herců ale může být zcela libovolný, tedy klidně i nulový.

Pro reprezentaci popsaných filmů vytvoříme obyčejnou třídu s názvem `Movie`, každý konkrétní film bude reprezentován jednou instancí této třídy. Všechny výše uvedené údaje budou realizovány jako privátní datové položky, přístup k nim zvenku bude možný jen přes veřejné bezparametrické getter metody `name`, `year`, `genre`, `rating` resp. `actors` vracející nemodifikovatelné reference na příslušné položky (v případě názvu, žánru a herců) resp. příslušné hodnoty kopií (u roku a hodnocení). Tyto metody navíc musí být deklarované jako konstantní (abychom je mohli volat i na nemodifikovatelných instancích filmů) a z důvodu efektivity je naprogramujeme jako inline metody.

Třída `Movie` nabídne následující dva konstruktory. V případě prvního z nich si u sebe uložíme kopie předaných položek, v případě druhého si položky předané rvalue referencemi přivlastníme. Pro uložení jednotlivých položek využijeme výhradně mechanismu inicializačních seznamů.

- `Movie(const std::string& name, int year, const std::string& genre, int rating, const std::set<std::string>& actors)`
- `Movie(std::string&& name, int year, std::string&& genre, int rating, std::set<std::string>&& actors)`

Poslední metodou třídy `Movie` bude metoda `void print_json(std::ostream& stream = std::cout)`. Jejím prostřednictvím budeme schopni daný film vypsat do uvedeného výstupního streamu. Konkrétně jej vypíšeme v podobě JSON objektu, jeho struktura musí odpovídat následující šabloně:

```
{ name: "Bobule", year: 2008, genre: "comedy", rating: 65, actors: [ "Krystof Hadek", "Te
reza Voriskova" ] }
```

Kromě mezer a celkové struktury musíme dodržet i pořadí a názvy jednotlivých položek, textové hodnoty budeme zapouzdřovat do uvozovek. Jména herců budeme vypisovat přesně v tom pořadí, v jakém nám je vrátí příslušný iterátor. Pokud film nemá ani jednoho herce, pak položku `actors` vůbec nevypíšeme. V opačném případě za posledním z nich uvnitř pole už oddělovací čárku samozřejmě nevypisujeme. Součástí výstupu nejsou žádné konce řádků, a to ani na úplném konci za uzavírací závorkou celého JSON objektu. Dále předpokládáme, že hodnoty názvů a žánrů filmů a jmen herců neobsahují uvozovky. Jejich případné výskyty tak kvůli zajištění korektní struktury výstupu ošetřovat nemusíme.

Samotnou databázi reprezentujeme pomocí standardního kontejneru vektoru `std::vector`, do kterého postupně budeme jednotlivé instance filmů vkládat. Z důvodu přípravy na změny, které nás budou čekat v dalších navazujících úkolech, však nebudeme pracovat s tímto typem napřímo. Namísto toho budeme uvažovat typový alias v podobě `database_t = std::vector<Movie>`. To nám hned v dalším úkolu umožní snadno změnit reprezentaci filmů jako takových a na konci semestru dokonce nahradit kontejner vektoru za jiný. Definici tohoto typového aliasu (jako jedinou věc, samozřejmě kromě potřebných `#include` direktiv) umístíme do hlavičkového souboru s názvem `Storage.h`.

Pro vkládání jednotlivých filmů do naší aktuální databáze můžeme využít např. metody `push_back` nebo `emplace_back`. Uživatelům však nabídneme i možnost hromadného importu filmů ze vstupních souborů resp. obecného vstupního streamu. Za tímto účelem navrhneme třídu `Database`, její definici umístíme do hlavičkového souboru `Database.h`. Cílem této třídy není ukládání obsahu naší databáze, ale jen zapouzdření funkcionality importu, a to prostřednictvím následujících dvou veřejných statických metod:

- `static void import(const std::string& filename, database_t& database)`:
  otevře uvedený vstupní soubor, provede import filmů a jejich instance vloží do připravené databáze (se zachováním jejího případného aktuálního obsahu)
- `static void import(std::istream& stream, database_t& database)`:
  provede import filmů z uvedeného vstupního streamu a instance filmů opět vloží do připravené databáze

V obou případech předpokládáme vstupní data ve formátu CSV (a to bez hlavičky aneb bez prvního řádku obsahujícího názvy sloupců). To znamená, že na každém jednotlivém řádku budou údaje o jednom filmu, případné zcela prázdné řádky budeme přeskakovat. Pro daný film postupně očekáváme jeho název, rok natočení, žánr, hodnocení a množinu jmen herců. Pořadí těchto položek je fixní a implicitní, a proto je v našem kódu zadrátujeme napevno. Jednotlivé položky budou odděleny (nikoli ukončeny) pomocí středníku, jména jednotlivých herců pak pomocí čárky. Uvnitř hodnot položek se používané oddělovače vyskytovat nemohou, přebytečný obsah řádků budeme ignorovat.

Položky názvu filmu a jeho žánr musí být neprázdné řetězce. Pokud jde o povolené hodnoty číselných položek, rok natočení se očekává z uzavřeného intervalu `1900` až `2100`, hodnocení filmu `0` až `100`. Jelikož v navazujících úkolech přidáme několik dalších položek, je nutné kód vhodně dekomponovat a důsledně se vyhnout jakémukoli opakování kódu. To minimálně znamená dvě funkce na zpracování textových a číselných položek (včetně nízkoúrovňového získání jejich hodnoty, řešení chybových situací a následné kontroly uvedených integritních omezení). Herce naopak bude vhodnější zpracovat specificky (v příštím úkolu totiž změníme jejich vnitřní strukturu), prázdné herce budeme přeskakovat. Připomeňme také, že počet herců může být i nulový, jak je ostatně vidět i na následujícím příkladu u druhého filmu:

```
Dira u Hanusovic;2014;comedy;49;Tatiana Vilhelmova,Ivan Trojan,Klara Meliskova
Vlastnici;2019;comedy;74;
```

Pro samotné parsování vstupu se očekává použití globální funkce `std::getline`, tentokráte i ve variantě s volitelným třetím parametrem, pomocí kterého můžeme určit i jiný separátor než výchozí konce řádků. Za tímto účelem budeme potřebovat vytvořit si stream nad řetězcem, a to pomocí `std::istringstream` z knihovny `<sstream>`. Instanci filmu budeme do kontejneru databáze vkládat co nejefektivněji, tedy za použití již zmíněné metody `emplace_back` (která očekává parametry nějakého konstruktoru našeho filmu, nikoli již vytvořenou instanci takového filmu) v kombinaci s konstrukcí `std::move`.

Veškeré chybové situace budeme řešit pomocí strukturovaných výjimek, a to v podobě struktury `struct Exception { int code; std::string text; }`, kde první položka bude obsahovat číselný kód typu chyby a druhá textový řetězec blíže vysvětlující příčinu chyby. Konkrétně očekáváme následující chování:

- Kód `1` (chyby vstupů)
  - `Unable to open input file <`*filename*`>`:
    nepodaří se otevřít vyžádaný vstupní soubor s názvem *filename*
- Kód `2` (parsovací chyby)
  - `Missing field <`*name*`> on line <`*line*`>`:
    chybí příslušná položka aneb ve vstupním řetězci už ani žádná další položka není (např. položka `rating` ve vstupu `Pres prsty;2019;comedy`)
  - `Empty string in field <`*name*`> on line <`*line*`>`:
    prázdný řetězec u textových položek, které prázdné hodnoty nepřipouští (např. položka `genre` ve vstupu `Pres prsty;2019;;56;`)
  - `Invalid integer <`*value*`> in field <`*name*`> on line <`*line*`>`:
    neplatná hodnota *value* u číselných položek jakožto reakce na výjimku `std::invalid_argument` z funkce `std::stoi` (např. položka `year` ve vstupu `Pres prsty;NaN;comedy;56;`)
  - `Overflow integer <`*value*`> in field <`*name*`> on line <`*line*`>`:
    analogicky přetečená číselná hodnota *value* jakožto reakce na výjimku `std::out_of_range`
  - `Malformed integer <`*value*`> in field <`*name*`> on line <`*line*`>`:
    analogicky špatně formovaná číselná hodnota *value* jakožto reakce na nesplněný poziční test (např. položka `year` ve vstupu `Pres prsty;2019ad;comedy;56;`)
  - `Integer <`*value*`> out of range <`*min*`, `*max*`> in field <`*name*`> on line <`*line*`>`:
    platná číselná hodnota *value* nepatřícího do uzavřeného intervalu povolených hodnot [*min*, *max*]

Parametry v lomených závorkách se nahradí konkrétními hodnotami, závorky samotné se ponechají: *filename* je jméno požadovaného souboru, *value* původní parsovaná hodnota, *name* jméno problematické položky (tedy `name`, `year`, `genre`, `rating` nebo `actors`), *line* číslo řádku ve vstupním souboru (počítáno od `1`, včetně prázdných řádků), a *min* a *max* je interval povolených hodnot pro číselné položky.

Pokud dojde k problémům při parsování záznamu filmu, instanci daného filmu nevytvoříme, vyhodíme příslušnou výjimku, a ve zpracování případných dalších filmů na vstupu tedy už nepokračujeme. Všechny filmy, které se nám už podařilo do databáze vložit, v něm zůstanou nedotčené. Pro sestavení textové hlášky výjimky stačí použít operátor konkatenace `+` nad řetězci (alespoň jeden operand musí být typu `std::string`, pro převod čísel se použije funkce `std::to_string` z knihovny `<string>`).

Nakonec ještě implementujeme dvě globální funkce, pomocí kterých budeme simulovat vyhodnocování konkrétních dotazů nad naší databází. V obou případech budeme jednoduše iterovat přes všechny filmy v předaném kontejneru a ty hledané vypíšeme v požadované podobě do uvedeného výstupního streamu, přičemž výpis každého takového filmu ještě ukončíme koncem řádku. Obě funkce deklarujeme v hlavičkovém souboru `Queries.h`.

- `void db_query_1(const database_t& database, std::ostream& stream = std::cout)`: najdeme všechny filmy (tedy žádné filtrování neprovádíme); každý z nich vypíšeme jako kompletní JSON objekt (pomocí naší připravené funkce)
- `void db_query_2(const database_t& database, std::ostream& stream = std::cout)`: najdeme všechny komedie (žánr *comedy*) natočené před rokem *2010*, ve kterých hrál *Ivan Trojan* nebo *Tereza Voriskova*; pro každou z nich vypíšeme jen příslušný název filmu; všechny uvedené parametry dotazu zadrátujeme přímo do kódu, možnost jejich případné změny se vzhledem k experimentální povaze této funkce neočekává (specificky tedy nebudeme vytvářet pojmenované konstanty, nedávalo by to smysl); vyhodnocení podmínky pro herce je nutné zvládnout v logaritmickém čase, není tedy možné procházet všechny herce sekvenčně a jednotlivě je porovnávat pomocí manuálního testu rovnosti

Odevzdejte všechny vytvořené zdrojové soubory (`*.cpp` a `*.h`) kromě hlavního souboru s funkcí `main`. I tentokráte ji totiž připravený test již obsahuje a bude průběh celého testu řídit. Veškerý kód rozdělte do vhodně oddělených modulů. Jak již bylo řečeno, třídu `Database` je nutné definovat v hlavičkovém souboru `Database.h`, funkce `db_query_*` je nutné deklarovat v souboru `Queries.h`. Veškerá očekávaná funkcionalita pak musí být přímo či nepřímo dostupná právě skrze tyto dva hlavičkové soubory.

Konkrétním cílem úkolu je ověřit schopnost práce s následujícími konstrukty: návrh a použití obyčejných datových tříd, návrh jejich parametrických konstruktorů v kombinaci s inicializačními seznamy, návrh inline metod, použití konstruktu `std::move` a práce s rvalue referencemi, seznámení se s kontejnerem `std::set` a pokročilejšími způsoby vkládaní prvků do kontejneru `std::vector` pomocí obecného mechanismu `emplace` a konečně také práce s funkcí `std::getline` s jiným než výchozím separátorem a v kombinaci se streamy `std::istringstream`. Všechny tyto konstrukty je nutné v řešení úkolu použít.

Předpokládá se dodržení všech obecných požadavků, které už na úkoly máme. Specificky nezapomeňte na pojmenované konstanty pro názvy položek filmů (jsou stejné v JSON objektech i textech výjimek), pro kódy výjimek a pro námi používané oddělovače středníků a čárek ve vstupních CSV souborech (aby je v případě potřeby bylo možné snadno změnit). Naopak nevytvářejte konstanty pro metasymboly používané v JSON objektech (jsou fixní a nedává smysl je měnit). Nezapomeňte také na používání konvence postfixování názvů privátních datových položek v našich třídách pomocí symbolu `_`.

Každou lokální proměnnou deklarujeme v nejvíce specifickém bloku, tedy jen v takovém bloku (který je vždy definován párem složených závorek `{}`), kde ji opravdu potřebujeme používat. Případné další pomocné metody pro parsování filmů je opět vhodné přidávat jako privátní statické členské funkce třídy `Database`, nikoli jako samostatné globální funkce. Inline funkce musí být naprogramovány tak, aby v místě jejich použití byly známé nejenom jejich deklarace, ale také plné definice. To v našem případě znamená, že jejich těla musíme implementovat přímo v hlavičkových souborech.

Během parsování jednotlivých položek konkrétního filmu ze vstupu není vhodné použít cyklus (ať už `while` nebo `for`), protože každý údaj potřebujeme zpracovat jinak. Jinými slovy se cykly obecně hodí jen v takových situacích, kdy každá iterace probíhá řekněme podobně nebo má alespoň nějaký dostatečně významný společný základ. V našem případě bychom ale tělo cyklu museli zase zpětně rozvětvit na každý jednotlivý případ (třeba pomocí konstrukce `switch` nebo jinak) a řešit je samostatně. Bez ohledu na zjevnou neefektivitu by takové řešení především značilo špatný návrh. Vyvarujeme se i postupu, kdy bychom nejprve

vyextrahovali všechny údaje a v podobě řetězců si je uložili do nějakého pracovního kontejneru před jejich dalším zpracováním.

Obecně předpokládáme, že kontroly vstupních údajů provádíme během importu, takže konstruktory filmů už očekávají platné hodnoty a žádné kontroly neprovádí. Výjimky reagující na chybové situace je vhodné házet rovnou v místě detekce a nebalit je do samostatných funkcí. Ty by totiž čtenáře kódu jen mátly (záměr vyhození výjimky by totiž zakrývaly) a každou bychom stejně patrně volali jen jednou. Nakonec si uvědomme, že rovněž není vhodné vymýšlet nějaké systematické funkce na vypisování JSON objektů, polí a hodnot, protože ty se do sebe obecně mohou libovolně rekurzivně zanořovat.