NPRG041 – 2025/26 ZS – Cvičení MS – **Malý úkol C12**

Databáze filmů IV

Účelem posledního úkolu v tématu databáze filmů bude další rozšíření stávajícího projektu o několik nových dotazů, v rámci kterých budeme používat nejrůznější standardní algoritmy (knihovny `<algorithm>` a `<numeric>`), funktory, lambda výrazy, falešné iterátory (knihovna `<iterator>`) a také rozsahy a pohledy (knihovna `<ranges>`).

Jelikož už máme z předchozích úkolů implementovaný náš vlastní kontejner gumového pole, byla by škoda jej nepoužít. Jinými slovy současné pojetí databáze v podobě `std::vector<std::shared_ptr<Title>>` nahradíme za `lib::Array<std::shared_ptr<Title>>`. Tedy alespoň v případě, že gumové pole opravdu máte a máte jej dostatečně odladěné. Jinými slovy tato změna povinná není.

Následně do projektu přidáme následujících šest nových databázových dotazů. Všechny předepsané konstrukty i postupy je potřeba věrně dodržet.

- `database_t db_query_14(`
  `const database_t& database,`
  `const Actor& actor`
  `)`: najdeme tituly, ve kterých hrál uvedený herec `actor`; toho dosáhneme tak, že nejprve úplně všechny tituly z databáze (myšleno sdílené chytré ukazatele na ně) překopírujeme do námi připraveného výstupního kontejneru našeho typu `database_t`, a to pomocí funkce `std::copy`; následně pomocí funkce `std::remove_if` odebereme ty tituly, které nám nevyhovují, a to pomocí predikátu v podobě funktoru `Predicate_Q14_Actor`; nakonec všechny nalezené tituly seřadíme primárně sestupně podle roku natočení a sekundárně vzestupně podle jejich jména, a to pomocí funkce `std::sort` a funktoru `Comparator_Q14_Years` simulujícího chování operátoru `<`

- `void db_query_15(`
  `const database_t& database,`
  `int year,`
  `std::ostream& stream = std::cout`
  `)`: najdeme všechny filmy (tituly mající typ `Type::MOVIE`) natočené v roce `year`, a to překopírováním tentokráte přímo jen vyhovujících titulů do námi vytvořeného prázdného pomocného kontejneru našeho typu `database_t`; použijeme k tomu funkci `std::copy_if`, predikát implementovaný pomocí lambda výrazu a instanci falešného iterátoru pro výstup získanou voláním `std::back_inserter`; následně tituly seřadíme, a to opět pomocí lambda výrazu simulujícího chování operátoru `<`, přičemž tituly chceme řadit vzestupně podle jejich názvů; nakonec všechny tituly vypíšeme v transformované podobě do předaného výstupního streamu, a to pomocí funkce `std::transform`; vlastní transformaci pak provedeme pomocí lambda výrazu, který pro daný titul vrátí řetězec `{ name: "`*title*`", year: `*year*` }`, kde *title* a *year* nahradíme jeho názvem resp. rokem natočení; pro výstup použijeme falešný iterátor `std::ostream_iterator<std::string>`, přičemž každý záznam ještě ukončíme vypsáním konce řádku

- `std::optional<int> db_query_16(`
  `const database_t& database,`
  `Type type,`
  `std::string_view genre`
  `)`: spočítáme celočíselné průměrné hodnocení všech titulů majících náš enumerační typ `type` a žánr `genre`; pro iteraci nad jednotlivými tituly použijeme funkci `std::for_each`, přičemž zpracování konkrétního titulu provedeme pomocí funktoru s názvem `Visitor_Q16_Rating`; ten naprogramujeme tak, aby veškerá logika počítání tohoto průměru byla schována uvnitř něho; pro nulový počet titulů vrátíme dle očekávání `std::nullopt`; dodejme dále, že funkce `std::for_each` si vytvoří a následně používá vlastní kopii předaného funktoru, tu pak vrátí pomocí návratové hodnoty

- `size_t db_query_17(`
  `const database_t& database,`
  `int rating`

`)`: spočítáme celkový součet počtů herců hrajících v titulech s hodnocením alespoň `rating`; očekává se použití funkce `std::for_each` a lambda výrazu

- ```
  std::vector<std::string> db_query_18(
    const database_t& database,
    std::string_view genre,
    std::string_view pattern,
    const std::variant<int, std::string>& rating,
    int year
  )
  ```
  : najdeme všechny tituly, které mají žánr `genre`, jejich název odpovídá regulárnímu výrazu `pattern`, jejich hodnocení je alespoň `rating`, mají nadprůměrný počet herců a všichni jejich herci jsou narozeni před rokem `year`; pro spočítání součtu počtů herců přes všechny jednotlivé tituly použijeme funkci `std::accumulate`, následný průměr spočítáme jako desetinné číslo `double`; pro práci s regulárními výrazy je potřeba použít knihovnu `<regex>`; nejprve se vytvoří instance regulárního výrazu pomocí konstruktoru `std::regex(pattern, flags)`, kde pomocí příznaku `std::regex_constants::icase` aktivujeme case-insensitive chování; vlastní testování shody provedeme přes funkci `std::regex_match(string, regex)`; kromě klasického číselného hodnocení `0` až `100` v rámci tohoto dotazu povolíme i hodnocení pomocí hvězdiček, konkrétně v podobě řetězce `std::string` obsahujícího 0 až 5 hvězdiček `*`, přičemž číselná hodnota každé hvězdičky je `20` bodů; pro reprezentaci takového dvojího hodnocení použijeme typ `std::variant` z knihovny `<variant>`; pro zjištění skutečně použité varianty použijeme funkci `std::holds_alternative<type>(variant)`, pro následnou extrakci příslušné hodnoty funkci `std::get<type>(variant)`; pro nalezení hledaných titulů opět využijeme funkci `std::for_each`; pro kontrolu podmínky na věk herců pak použijeme funkci `std::all_of`, která umožňuje simulovat chování všeobecného kvantifikátoru; všechny dílčí podmínky, akce a operace budeme tentokrát řešit pomocí lambda výrazů; pro každý nalezený titul vložíme do výstupního kontejneru řetězec s JSON objektem `{ name: "`*title*`", year: `*year*` }`

- ```
  void db_query_19(
    const database_t& database,
    int begin,
    int end,
    std::ostream& stream = std::cout
  )
  ```
  : najdeme všechny tituly natočené během let [`begin`, `end`); k jejich nalezení použijeme pohledy `std::views::filter` a `std::views::transform`, které zřetězíme pomocí pipe operátoru `|`, přičemž filtrovací predikát i transformační akci naprogramujeme v podobě lambda výrazů; cílem zmíněné transformace je pro každý titul získat trojici v podobě `std::tuple<std::string, int, size_t>`, kde postupně uložíme název titulu, rok jeho natočení a počet herců v něm hrajících; získané záznamy následně uložíme do kontejneru `std::vector` pomocí jeho rozsahového konstruktoru; všechny záznamy dále uspořádáme pomocí funkce `std::ranges::sort`, a to opět pomocí lambda výrazu řadícího tituly podle roků natočení a následně jejich názvů; nakonec záznamy transformujeme pomocí `std::views::transform`, tentokráte do řetězců v podobě `{ name: "`*title*`", year: `*year*`, actors: `*actors*` }`, kde *title*, *year* a *actors* nahradíme názvem titulu, rokem natočení a počtem herců; takto získané řetězce nakonec vypíšeme do uvedeného výstupního streamu pomocí funkce `std::ranges::copy` a falešného iterátoru `std::ostream_iterator`

- ```
  template <typename Selector, typename Comparator, typename Serializer>
  void db_query_20(
    const database_t& database,
    Selector selector,
    Comparator comparator,
    Serializer serializer,
    std::ostream& stream = std::cout
  )
  ```
  : najdeme všechny tituly splňující obecnou vyhledávací podmínku `selector` definovanou jako `bool operator()(const std::shared_ptr<Title>& title_ptr)`, takto nalezené tituly následně seřadíme pomocí porovnávání `comparator` s rozhraním `bool operator()(const std::shared_ptr<Title>& title_1, const std::shared_ptr<Title>& title_2)` a nakonec je serializované vypíšeme pomocí `serializer` v podobě `std::string operator()(const std::shared_ptr<Title>& title_ptr)` do
  ```

uvedeného výstupního streamu `stream`

Odevzdejte opět jen modul dotazů, tedy soubory `Queries.h` a `Queries.cpp`, a také hlavičkový soubor `Storage.h` s volbou typu úložiště pro naši databázi filmů. Konkrétně tedy odevzdejte jen dotazy Q14 až Q20, nikoli starší dotazy Q1 až Q13 nebo indexy. Odevzdáte-li je, nic se nestane, testovány ale nebudou. Pokud jste se rozhodli používat gumové pole, odevzdejte i jeho implementaci. Zbytek projektu databáze tedy ani tentokrát odevzdávat nebudete.

Cílem úkolu je naučit se pracovat s vybranými standardními algoritmy (`std::copy`, `std::copy_if`, `std::remove_if`, `std::sort`, `std::transform`, `std::accumulate`, `std::for_each`, `std::all_of`), dále falešnými iterátory (`std::back_inserter`, `std::ostream_iterator`), obecnými lambda výrazy, rozsahy (`std::ranges::copy`, `std::ranges::sort`) a pohledy (`std::views::filter`, `std::views::transform`), a to včetně zřetězování pomocí operátoru `|`. Také se seznámíme s pomocnou datovou strukturou `std::variant` a základy práce s regulárními výrazy `std::regex`.

Na úplném konci se ještě krátce vraťme k práci s regulárními výrazy. V rámci dotazu Q18 je dostanete již formulované prostřednictvím parametru dotazu. Pokud byste si je však v průběhu ladění chtěli vytvářet sami, určitě se vám budou hodit tzv. raw string literály. Jde o alternativní způsob zápisu céčkových řetězců ve tvaru `R"`*delim*`(`*text*`)`*delim*`"`, kde je možné si oddělovač *delim* zvolit dle potřeby (klidně jako prázdný řetězec) a skutečný výsledný řetězec pak tvoří jen obsah části *text*. Podstatné je, že uvnitř této části *text* se neaplikuje standardní mechanismus escape sekvencí (např. `\n` zůstane prostým `\n` aneb na symbol nového řádku se nezmění), což přináší uživatelsky výrazně pohodlnější způsob zápisu např. právě regulárních výrazů. Ty totiž často obsahují řadu konstrukcí vyjadřovaných pomocí zpětných lomítek `\` (např. `\b` pro označení hranice slov, `\w` pro jakýkoli alfanumerický znak nebo podtržítko `_` atp.). Díky tomu pak jejich výskyty nemusíme pracně ošetřovat a celkový zápis se stane přehlednějším. Např. `R"(\b\w*bobule\b)"` odpovídá klasickému řetězci `"\\b\\w*bobule\\b"`.