http://www.ksi.mff.cuni.cz/~svoboda/courses/221-NPRG041/

Cvičení

NPRG041: Programování v C++

2022/23 ZS

Martin Svoboda
martin.svoboda@matfyz.cuni.cz

Univerzita Karlova, Matematicko-fyzikální fakulta

Cvičení 1: **Úvod**

Funkce main
Standardní výstup
Dekompozice
Pole

Používané nástroje

Visual Studio Community / Enterprise 2022

- https://visualstudio.microsoft.com/vs/community/
- https://portal.azure.com/

Gitlab

- https://gitlab.mff.cuni.cz/
  - .../teaching/nprg041/2022-23/svoboda-1040/
  - .../teaching/nprg041/2022-23/svoboda-1220/

TortoiseGit

- https://tortoisegit.org/

Používané nástroje

Mattermost

- https://ulita.ms.mff.cuni.cz/mattermost/
  - .../zs2223/channels/nprg041-cpp-svoboda

ReCodEx

- https://recodex.mff.cuni.cz/

P1: Hello World

Vytvořte standardní Hello World aplikaci

- Aneb na standardní výstup vypište uvedený pozdrav
- Vytvoření nového projektu ve Visual Studiu
  - Jazyk: *C++*
  - Typ projektu: *Empty Project*
- Menší nápověda
  - `#include <iostream>`
  - `int main(int argc, char** argv) { ... }`
  - `std::cout << "..." << std::endl;`

P2: Hledání podmnožin

Najděte a vypište **všechny podmnožiny dané množiny** na vstupu

- Vstup zatím nasimulujeme jen pomocí lokální proměnné
  - `const char items[] = { 'A', 'B', 'C', 'D' };`
  - `const size_t count = sizeof(items) / sizeof(items[0]);`
- Celý problém vhodně dekomponujte do jednotlivých funkcí
- Každou nalezenou podmnožinu vypište na standardní výstup
  - Na každý řádek vypište jen jednu podmnožinu
    - Zachovejte pořadí jednotlivých prvků
    - Přítomnost prvku má přednost před jeho nepřítomností
  - Formát výstupu: `{ A, C, D }`
- Nápověda pro alokaci pole s dynamickou velikostí
  - `bool* signature = new bool[count];`
  - `delete[] signature;`

Cvičení 2: **Argumenty I**

Hlavičkové soubory
Argumenty programu
Řetězce std::string
Kontejner std::vector
Typové aliasy
Předávání parametrů
Iterace
Pojmenované konstanty

P1: Vypsání argumentů

Na standardní výstup vypište všechny předané **vstupní argumenty**

- Argumenty nejprve konvertujte do řetězců `std::string` a vložte je do kontejneru `std::vector`
  - `#include <string>`
  - `#include <vector>`
  - `using args_t = std::vector<std::string>;`
  - `args_t arguments(argv + 1, argv + argc);`
- Výkonný kód zabalte do samostatné funkce
  - Kontejner s argumenty předejte pomocí reference
  - Přes jednotlivé argumenty iterujte následujícím způsobem
    - `for (auto&& item : arguments) { ... }`

P1: Vypsání argumentů

Pokračování…

- Oddělte definice od deklarací pomocí hlavičkového souboru
  - `#ifndef`, `#define`, `#endif`
  - `#include "..."`
- Nastavení vstupních argumentů ve VS
  - (Project) *Properties* $\rightarrow$ *Debugging* $\rightarrow$ *Command Arguments*

P2: Detekce přepínačů

Detekujte předem očekávané **krátké i dlouhé přepínače**

- Konkrétně očekávejte následující parametry
  - `-t`, `-x`, `-y`
  - `--grayscale`, `--transparent`
- Jména přepínačů definujte pomocí globálních konstant
  - `constexpr char OPTION_TRANSPARENT_SHORT = 't';`
  - `constexpr char OPTION_TRANSPARENT_LONG[] = "transparent";`
- Umožněte seskupování krátkých přepínačů
  - Např.: `-xy`
- Rozpoznané přepínače vypište na standardní výstup
  - Flag option <x> detected
  - Unknown option <something> found!

P2: Detekce přepínačů

Pokračování…

- Pro iteraci nad argumenty použijeme iterátory
  - Získáme tím možnost manuální kontroly nad průběhem iterace
  - `for (auto it = arguments.begin(); it != arguments.end(); ++it) { ... }`
    - Vlastní iterátor je typu `args_t::const_iterator`
    - A tedy `std::vector<std::string>::const_iterator`
- Dereferencování iterátoru
  - `const std::string& item = *it;`

P2: Detekce přepínačů

Pokračování…

- Užitečné metody nad řetězci
  - `std::string substr(size_t pos, size_t len)`
    - Druhý parametr může být vynechán
  - `size_t size()`
- Návratový kód určete podle celkové úspěšnosti detekce
  - `0` při úspěchu, `1` jinak

P3: Hodnotové přepínače

Rozšiřte náš program o **rozpoznávání hodnotových přepínačů**

- Konkrétně očekávejte následující nové hodnotové přepínače
  - `-r`, `-g`, `-b`, `-a`
  - `--red`, `--green`, `--blue`, `--alpha`
- Podporujte následující způsoby předání hodnoty
  - `-xy -r 255`, `-xyr255`, `-xyr 255`
  - `-xy --red 255`
- Detekujte chybějící hodnoty i nevázané hodnoty navíc
  - `-r`, `-x something`
- Vše opět vypisujte na standardní výstup
  - Value option <r> detected with value <255>
  - Value option <r> detected but its value is missing!
  - Standalone value detected <something>

Cvičení 3: **Argumenty II**

Parsování čísel
Funkce std::stoi a std::stof
Odchycení výjimek
Struktura struct

P1: Parsování čísel

Rozšiřte náš program o **parsování číselných hodnot**

- Rozpoznání celých čísel / čísel s plovoucí desetinnou čárkou
  - `int stoi(const std::string& s, size_t* p)`
    - Knihovna `<string>`
    - `std::invalid_argument`, `std::out_of_range`
  - `float stof(const std::string& s, size_t* p)`
- Konkrétně očekávejte následující chování
  - Celá čísla: `-r`, `--red`, `-g`, `--green`, `-b`, `--blue`
  - Desetinná čísla: `-a`, `--alpha`
- Zajistěte korektní ošetření chybových situací
- Na standardní výstup vypisujte vhodné chybové hlášky
  - Value <text> is not a valid integer number!
  - Value <text> is not a valid floating point number!

P2: Uložení přepínačů

Uložte rozpoznané přepínače i jejich hodnoty do vhodné **struktury**

- Strukturu definujte v hlavičkovém souboru
  - ```
    struct options_t {
      bool flag_x = false;
      bool flag_r = false; int value_r;
      ...
    }
    ```
- Nevázané hodnoty uložte do vektoru řetězců
  - `values.push_back(…)`
- Na konci programu uložené parametry pro kontrolu vypište
  - Flag option <x> is <disabled>
  - Value option <r|red> is <enabled> and associated with value <255>
  - Standalone value <something>
    ```

Cvičení 4: **Počítadlo**

Streamy std::istream, std::ostream
Souborové streamy std::ifstream, std::ofstream
Funkce std::getline
Třídy se statickými metodami
Ukazatele

P1: Vypsání souboru

Vypište obsah existujícího textového souboru na standardní výstup

- Využijte následující konstrukce
  - Knihovny `<iostream>`, `<fstream>`, `<string>`
  - `std::ifstream`
    - `void open(const char* filename);`
    - `bool good();`
    - `void close();`
  - `std::istream& std::getline(std::istream& is, std::string& line);`

P2: Počítání znaků

Rozšiřte a upravte předchozí kód následujícím způsobem

- Obsah vstupního souboru už na výstup nevypisujte
- Umožněte zpracování většího množství vstupních souborů
  - Alternativně umožněte čtení i ze standardního vstupu
- Spočítejte a vypište celkový počet znaků přes všechny vstupy
  - Získané číslo umožněte vypsat i do výstupního souboru
- Kód umístěte do vhodné třídy a jejích statických metod
  - `bool process(const std::string& filename, size_t* letters);`
  - `bool process(std::istream& is, size_t* letters);`
  - `bool print(const std::string& filename, const size_t* letters);`
  - `bool print(std::ostream& os, const size_t* letters);`

P3: Rozšířené počítadlo

Rozšiřte předchozí kód o **počítání vybraných statistik**

- Mějme následující předpoklady o textovém vstupu
  - Vstup obsahuje libovolný počet vět
  - Věty jsou ukončeny znaky `.!?` a případně odděleny mezerami
  - Věta obsahuje slova nebo čísla oddělená mezerami
  - Slovo obsahuje jen písmena, číslo jen číslice `0` až `9` nebo tečku `.`
- Přes všechny vstupy detekujte a ve třídě uložte tyto údaje
  - Celkový počet řádků, vět, slov, čísel
  - Celkový počet písmen, číslic, mezer, symbolů
  - Součet hodnot všech celých a odděleně desetinných čísel
- Využijte následující funkce
  - `int isdigit(int c); int isalpha(int c);`
- Spočtené statistiky na konci běhu vypište na standardní výstup

Cvičení 5: **Databáze I**

Streamy std::stringstream
Funkce std::getline se separátorem
Třída s datovými položkami
Konstruktory a inicializátory
Inline členské funkce
Funkce std::move a rvalue reference
Mechanismus emplace
Kontejner std::set

P1: Reprezentace filmů

Navrhněte třídu pro reprezentaci databázového záznamu filmu

- Každý film má následující privátní datové položky
  - Jméno (`std::string`)
  - Rok natočení (`unsigned short`)
  - Žánr (`std::string`)
  - Hodnocení (`unsigned short`)
  - Množinu jmen herců (`std::set<std::string>`)
- Nejprve implementujte následující funkce
  - Parametrický konstruktor
  - Funkce pro přístup k jednotlivým datovým položkám
    - A to v podobě `inline` funkcí

P1: Reprezentace filmů

Pokračování…

- Implementujte i funkci na vypsání filmu formou JSON objektu
  - `void print_json(std::ostream& os = std::cout) const;`
    - `{ name: "Bobule", year: 2008, genre: "comedy", rating: 65, actors: [ "Krystof Hadek", "Tereza Voriskova" ] }`
  - Není-li žádný herec uvedený, položku s polem herců vůbec nevypisujte
- Kód experimentálně otestujte přímo v `main` funkci
  - Nejprve vytvořte kontejner pro záznamy filmů
    - `std::vector<Movie> db;`
  - Následně ručně přidejte několik filmů a iterací přes kontejner je vypište na standardní výstup

P2: Konstrukce filmů

Přidejte další možnosti na **efektivnější vytváření objektů filmů**

- Implementujte parametrický konstruktor přijímající rvalue reference
  - A to u položek jména, žánru a množiny herců
- Následně vyzkoušejte následující možnosti vytvoření a vložení nového filmu
  - Standardní `push_back`
  - Vylepšený `push_back` v kombinaci s funkcí `std::move`
  - Mechanismus `emplace_back`

P3: Import filmů

Rozšiřte naši databázi o **import filmů z CSV souborů**

- Fukcionalitu implementujte pomocí statických funkcí třídy `Database`
  - `bool import(const std::string& filename, std::vector<Movie>& db);`
  - `bool import(std::istream& is, std::vector<Movie>& db);`
- Pro parsování CSV záznamů použijte následující konstrukty
  - `std::istringstream` (knihovna `<sstream>`)
  - `istream& std::getline(istream& is, string& line, char delimiter);`
- Konkrétně předpokládáme následující oddělovače
  - Středník `;` pro údaje a čárka `,` pro herce

P4: Vyhledávání filmů

Připravte následující dva jednoduché **databázové dotazy**

- **Q1**: všechny filmy
  - `void db_query_1(const std::vector<Movie>& db);`
  - Vypište celé JSON objekty nalezených filmů
- **Q2**: názvy *komedií* natočených před rokem *2010*, ve kterých hrál *Ivan Trojan* nebo *Tereza Voriskova*
  - `void db_query_2(const std::vector<Movie>& db);`
  - Vypište jen názvy nalezených filmů

Cvičení 6: **Výrazy I**

Třídy s dědičností
Virtuální a čistě virtuální funkce
Enumerační třídy
Dynamická alokace

P1: Aritmetické výrazy

Předpokládejme jednoduché celočíselné **aritmetické výrazy**

- Tyto výrazy mohou obsahovat pouze…
  - Základní binární operace
    - Sčítání `+`, odčítání `-`, násobení `*` a dělení `/`
  - Přirozená čísla včetně nuly jako jednoduché operandy

Navrhněte **třídy pro reprezentaci uzlů syntaktického stromu** (vnitřní stromové struktury) takových výrazů

- Abstraktní třída `Node` jako společný předek
- Finální odvozená třída `NumberNode` pro listové uzly s čísly
- Abstraktní odvozená třída `OperationNode` pro vnitřní uzly
- Finální odvozené třídy pro jednotlivé operace
  - `AdditionNode`, `SubtractionNode`, `MultiplicationNode`, `DivisionNode`

P1: Aritmetické výrazy

Pokračování…

- Základní **použití konceptu dědičnosti**
  - `class NumberNode final : public Node { ... }`
- **Datové položky** vhodně rozmístěte do jednotlivých tříd
  - Listové uzly: `private` číslo
  - Vnitřní uzly: `protected` ukazatel na levý a pravý podstrom
- Definujte následující **konstruktory**
  - `NumberNode(int number);`
  - `OperationNode(Node* left, Node* right);`
    - `using OperationNode::OperationNode;`
- Pro rozlišení těchto dvou typů uzlů použijte enumerační třídu
  - `enum class Type { ... }`

P1: Aritmetické výrazy

Pokračování…

- Vhodně používejte **virtuální členské funkce**
  - `virtual Type get_type() const;`
  - `virtual Type get_type() const = 0;`
  - `Type get_type() const override;`
- Konkrétně implementujte následující členské funkce
  - `Type get_type() const;`
    - Obejděte se bez uložení typu uzlu pomocí datové položky
  - `char get_operator() const;`
    - Jen jako `protected` funkce pro uzly operací
    - Symboly operátorů zadefinujte pomocí globálních konstant
    - I zde se obejděte bez datové položky pro tyto operátory
- Nezapomeňte na **virtuální destruktor**
  - `~Node();`

P1: Aritmetické výrazy

Pokračování…

- Implementujte také funkci na **vypsání výrazu**
  - A to konkrétně v postfixové (reverzní polské) notaci
    - Stačí jen realizovat postorder průchod doleva do hloubky
  - `void print_postfix(std::ostream& os = std::cout) const;`
    - Pomyslný vstup: `1*2+3*(4+5)-6`
    - Výstup: `1 2 * 3 4 5 + * + 6 -`
    - Operátory a čísla oddělte vždy právě jednou mezerou
- Nakonec navrhněte třídu `Expression` zapouzdřující celý výraz
  - Konstruktor `Expression(Node* root);`
  - Destruktor
  - Funkce `void print_postfix(std::ostream& os = std::cout) const;`

P1: Aritmetické výrazy

Pokračování…

- Předpokládáme použití mechanismu **dynamické alokace**
  - `Node* node_ptr = new NumberNode(2);`
  - `delete node_ptr;`
- Veškerou funkcionalitu dostatečně experimentálně otestujte
  - Pomyslný vstup: `(2+3)*4`
  -
```
Expression e1(
  new MultiplicationNode(
    new AdditionNode(
      new NumberNode(2), new NumberNode(3)
    ),
    new NumberNode(4)
  )
);
```

P2: Vyhodnocení výrazu

Rozšiřte naši aplikaci pro aritmetické výrazy

- Přidejte funkci na **spočítání výsledku výrazu**
  - `int evaluate() const;`

P3: Vypsání výrazu

Rozšiřte naši aplikaci pro aritmetické výrazy

- Přidejte funkci na **vypsání výrazu v infixové notaci**
  - `void print_infix(std::ostream& os = std::cout) const;`
  - Kolem operátorů ani závorek žádné mezery nevypisujte
  - Vypisujte však jen nezbytně nutné závorky
    - Operace `*` a `/` mají vyšší precedenci než operace `+` a `-`
- Příklad
  - Pomyslný vstup: `(7+(9-(3*1))/3)-(5-1)`
  - Výstup: `7+(9-3*1)/3-(5-1)`

Cvičení 7: **Výrazy II**

Polymorfní kontejner
Kontejner std::stack
Algoritmus shunting-yard

P1: Parsování výrazů

Vytvořte jednoduchý **parser pro infixové aritmetické výrazy**

- Uvažujeme jen syntakticky dobře formované vstupní výrazy
  - Ty mohou obsahovat i pomocné kulaté závorky `()`
  - Nadále pracujeme jen s přirozenými čísly včetně nuly
    - Jinými slovy před číslem nemůže být unární mínus `-`
- Zatím jen vstupní výraz převeďte do postfixové notace
  - Aneb výsledný výraz v postfixové notaci vypište na výstup
    - Vstup: `10*2+3*((1+14)-18)-10`
    - Výstup: `10 2 * 3 1 14 + 18 - * + 10 -`
  - Operátory a čísla opět oddělte vždy právě jednou mezerou
- K transformaci použijte **algoritmus shunting-yard**

P1: Parsování výrazů

Pokračování…

- Předpokládáme následující **vlastnosti operací**
  - Všechny uvedené operace jsou zleva asociativní
  - Operace `*` a `/` mají vyšší precedenci než operace `+` a `-`
- Využijte standardní **zásobník**
  - `std::stack<char>` (knihovna `<stack>`)
  - Metody `push(…)`, `top()`, `pop()`, `empty()`

P1: Parsování výrazů

```
1  foreach token t ve vstupním infixovém výrazu do
2      if t je číslo then  vypiš t na standardní výstup
3      else if t je otevírací závorka ( then  dej ( na zásobník
4      else if t je zavírací závorka ) then
5          while na vrcholu zásobníku je nějaký operátor o do
6              odeber o ze zásobníku a vypiš jej na výstup
7          odeber ( ze zásobníku
8      else t je operátor n
9          while na zásobníku je operátor o s precedencí vyšší než n nebo
             také stejnou, je-li ovšem n zleva asociativní do
10             odeber o ze zásobníku a vypiš jej na výstup
11         přidej n na zásobník
12 while zásobník je neprázdný do
13     odeber o ze zásobníku a vypiš jej na výstup
```

P2: Syntaktický strom

Rozšiřte náš parser aritmetických výrazů

- **Zkonstruujte syntaktický strom reprezentující vstupní výraz**
- Použijeme upravený algoritmus shunting-yard
  - Nově budeme potřebovat ještě druhý zásobník pro operandy
    - `std::stack<Node*>`
  - Vytváření listových uzlů pro čísla…
    - Vytvoříme nový uzel a vložíme jej do tohoto zásobníku
  - Vytváření vnitřních uzlů pro operace…
    - Nejprve ze zásobníku vyjmeme pravý a následně levý operand
    - Následně vytvoříme nový uzel a vložíme jej do zásobníku
  - Kořenový uzel najdeme na úplném konci v zásobníku operandů
    - Půjde o jeho jediný prvek

P2: Syntaktický strom

```
1  foreach token t ve vstupním infixovém výrazu do
2      if t je číslo then  vytvoř pro t nový listový uzel…
3      else if t je otevírací závorka ( then  dej ( na zásobník operátorů
4      else if t je zavírací závorka ) then
5          while na vrcholu zásobníku operátorů je nějaký operátor o do
6              odeber o ze zásobníku a vytvoř pro o nový vnitřní uzel…
7          odeber ( ze zásobníku operátorů
8      else t je operátor n
9          while na zásobníku operátorů je operátor o s precedencí vyšší
             než n nebo také stejnou, je-li ovšem n zleva asociativní do
10             odeber o ze zásobníku a vytvoř pro o nový vnitřní uzel…
11         přidej n na zásobník operátorů
12 while zásobník operátorů je neprázdný do
13     odeber o ze zásobníku a vytvoř pro o nový vnitřní uzel…
```

Cvičení 8: **Databáze II**

Kontejner std::set nad vlastní třídou
Vlastní porovnávací operátory
Vlastní operátory zápisu do / čtení ze streamu
Konstrukce friend
Chytré ukazatele std::shared_ptr
Dynamické přetypování

P1: Strukturovaní herci

Upravte naši aplikaci databáze filmů

- Nově už herec nebude jen atomickým řetězcem se jménem, ale strukturovaným záznamem s následujícími položkami
  - Křestní jméno (`std::string`)
  - Příjmení (`std::string`)
  - Rok narození (`unsigned short`)
- Navhrněte třídu pro reprezentaci takového herce
  - Připravte defaultní a parametrické konstruktory
    - `Actor() = default;`
  - Přidejte také přístupové funkce k jednotlivým položkám
- Dále implementujte **vlastní porovnávací operátor** pro herce
  - Globální funkce `bool operator<(const Actor& actor1, const Actor& actor2);`
    - Uspořádání je definováno trojicí příjmení, jméno a rok

P1: Strukturovaní herci

Pokračování…

- **Vypisování herců** vyřešte pomocí vlastního operátoru <<
  - `std::ostream& operator<<(std::ostream& os, const Actor& actor);`
  - Z hlediska formátu výstupu opět využijeme JSON objekt
    - `{ name: "Ivan", surname: "Trojan", year: 1964 }`
- **Importování herců** vyřešíme také vlastním operátorem >>
  - `std::istream& operator>>(std::istream& is, Actor& actor);`
  - Jednotlivé údaje jsou odděleny mezerou
    - `Ivan Trojan 1964`
- Refaktorujte i všechny ostatní zbývající části současného kódu
  - Konkrétně tedy minimálně dotazy

P2: Hierarchie titulů

Rozšiřte naši aplikaci o podporu různých typů titulů

- Nejprve refaktorujte současný kód
  - Změňte třídu `Movie` na `Title`
  - Kontejner databáze nově bude obsahovat chytré ukazatele
    - `std::shared_ptr<…>` (knihovna `<memory>`)
    - `std::vector<std::shared_ptr<Title>> db;`
    - Funkce `std::make_shared<Title>(...);`
- Následně navrhněte novou hierarchii titulů
  - Třída `Title` se stane abstraktní
  - Odvozená třída `Movie` bude navíc obsahovat položku
    - Délka v minutách (`unsigned short`)
  - Odvozená třída `Series` bude navíc obsahovat položky
    - Počet sezón (`unsigned short`)
    - Počet epizod (`unsigned short`)

P2: Hierarchie titulů

Pokračování…

- Dále přidejte následující funkce
  - Konstruktory a funkce pro přístup k novým položkám
  - Enumeraci pro rozlišení typů titulů a funkci vracející tento typ
    - `Type type() const;`
- Upravte funkci na **vypisování titulů**
  - Na začátek JSON objektu přidejte položku popisující typ titulu
    - Filmy: `{ type: "MOVIE", ... }`
    - Seriály: `{ type: "SERIES", ... }`
  - Na konec naopak přidejte nové specifické položky
    - Filmy: `{ ..., length: 112 }`
    - Seriály: `{ ..., seasons: 8, episodes: 73 }`

P2: Hierarchie titulů

Pokračování…

- Upravte funkci na **importování titulů**
  - Na začátku nově očekávejte řetězec rozlišující typ titulu
    - Filmy: `MOVIE;...`
    - Seriály: `SERIES;...`
  - Nově přidané specifické položky očekávejte naopak na konci
    - Filmy: `...;112`
    - Seriály: `...;8;73`
- Upravte i všechny ostatní zbývající části současného kódu
  - Konkrétně tedy minimálně dotazy

P3: Typová konverze

Připravte následující dva jednoduché **databázové dotazy**

- **Q3**: seriály mající alespoň `seasons` sezón nebo `episodes` epizod
  - ```
    void db_query_3(
      const std::vector<std::shared_ptr<Title>>& db,
      unsigned short seasons,
      unsigned short episodes
    );
    ```
  - Dynamické přetypování chytrých ukazatelů
    - `(Series*)&*title_ptr;`
    - `dynamic_cast<Series*>(&*title_ptr);`
    - `std::dynamic_pointer_cast<Series>(title_ptr);`
  - Vypište celé JSON objekty nalezených seriálů
    ```

P3: Typová konverze

Pokračování…

- **Q4**: názvy titulů typu `type` natočených mezi roky [begin, end)
  - ```
    void db_query_4(
      const std::vector<std::shared_ptr<Title>>& db,
      const std::type_info& type,
      unsigned short begin, unsigned short end
    );
    ```
  - Interval roků vnímejte jako zprava otevřený
  - Typ titulu je určen pomocí typu třídy
    - Tedy nikoli pomocí naší enumerace
    - `std::type_info` (knihovna `<typeinfo>`)
    - `typeid(…);`
  - Vypište jen názvy nalezených titulů
    ```

Cvičení 9: **Databáze III**

Kontejnery std::map a std::multimap
Struktura std::pair a funkce std::make_pair
Kontejner std::unordered_multimap
Struktury std::less, std::hash a std::equal_to
Funkce std::copy, std::copy_if, std::remove_if a std::erase
Funkce std::sort a std::for_each
Funktory
Lambda výrazy

P1: Názvy titulů

Vytvořte index pro **hledání titulů podle jejich názvů**

- Použijte kontejner uspořádané mapy
  - `std::map<std::string, std::shared_ptr<Title>>`
  - Knihovna `<map>`
- Index vytvořte pomocí následující funkce
  - ```
    void db_index_titles(
      const std::vector<std::shared_ptr<Title>>& db,
      std::map<std::string,
        std::shared_ptr<Title>>& index
    );
    ```
- Vkládání položek do indexu
  - `std::pair<…, …> item;` nebo `std::make_pair(…, …);`
  - Metody `index.insert(…);` resp. `index.emplace(…);`
    ```

P1: Názvy titulů

Implementujte následující **databázový dotaz**

- **Q5**: titul mající název `name`
  - `void db_query_5(`
    `const std::map<std::string,`
    `std::shared_ptr<Title>>& index,`
    `const std::string& name`
    `);`
  - Nalezení hledaného titulu
    - Metoda `index.find(name);`
    - Vrací iterátor na nalezený prvek, jinak na konec
  - Vnitřní struktura dvojice `std::pair`
    - Položky `first` a `second`
  - Vypište celý JSON objekt nalezeného titulu
    - `"name" -> { ... }`
    - Nebo `"name" -> Not found!` v opačném případě

P2: Roky natočení

Vytvořte index pro **hledání titulů podle roků natočení**

- Použijte kontejner uspořádané multimapy
  - `std::multimap<`
    `unsigned short,`
    `std::shared_ptr<Title>`
    `>`
  - Pro řazení prvků předpokládáme výchozí funktor
    - `std::less<unsigned short>`
- Index vytvořte pomocí následující funkce
  - ```
    void db_index_years(
      const std::vector<std::shared_ptr<Title>>& db,
      std::multimap<unsigned short,
        std::shared_ptr<Title>>& index
    );
    ```
    ```

P2: Roky natočení

Implementujte následující **databázové dotazy**

- **Q6**: tituly natočené v roce `year`
  - ```
    void db_query_6(
      const std::multimap<unsigned short,
        std::shared_ptr<Title>>& index,
      unsigned short year
    );
    ```
  - Nalezení hledaných titulů
    - Funkce `index.equal_range(year);`
    - Vrací dvojici (`std::pair`) iterátorů [od, do)
  - Vypište jen názvy nalezených titulů
    - `year -> "name"` pro každý titul
    - Nebo `year -> Not found!` v opačném případě
    ```

P2: Roky natočení

Pokračování…

- **Q7**: tituly natočené mezi roky [`begin`, `end`)
  - ```
    void db_query_7(
      const std::multimap<unsigned short,
        std::shared_ptr<Title>>& index,
      unsigned short begin, unsigned short end
    );
    ```
  - Nalezení hledaných titulů
    - Iterátor od: `index.lower_bound(begin);`
    - Iterátor do: `index.lower_bound(end);`
  - Vypište jen názvy nalezených titulů
    - `year -> "name"` pro každý titul
    - Nebo `[begin, end) -> Not found!` v opačném případě
    ```

P3: Herecké obsazení

Vytvořte index pro **hledání titulů podle jejich herců**

- Použijte kontejner neuspořádané multimapy
  - `std::unordered_multimap<`
    `Actor,`
    `std::shared_ptr<Title>`
    `>`
  - Knihovna `<unordered_map>`
- Index vytvořte pomocí následující funkce
  - ```
    void db_index_actors(
      const std::vector<std::shared_ptr<Title>>& db,
      std::unordered_multimap<Actor,
        std::shared_ptr<Title>>& index
    );
    ```
    ```

P3: Herecké obsazení

Pokračování…

- **Implementujte specializaci hašovacího funktoru**
  - `template<>`
    `struct std::hash<Actor> { ... }`
  - Funkce `size_t operator()(const Actor& actor) const noexcept;`
  - Použijte příjmení herce a `std::hash<std::string>{}(…);`
- A současně také **porovnávací operátor pro herce**
  - Globální funkce `bool operator==(const Actor& actor1, const Actor& actor2);`

P3: Herecké obsazení

Implementujte následující **databázový dotaz**

- **Q8**: tituly, ve kterých hrál herec `actor`
  - ```
    void db_query_8(
      const std::unordered_multimap<Actor,
        std::shared_ptr<Title>>& index,
      const Actor& actor
    );
    ```
  - Nalezení hledaných titulů
    - Funkce `index.equal_range(actor);`
  - Vypište jen názvy nalezených titulů
    - `{ ... } -> "name"` pro každý titul
    - Nebo `{ ... } -> Not found!` v opačném případě
    - Herce vypište v podobě JSON objektu
    ```

P4: Řazení titulů

Implementujte následující **databázový dotaz**

- **Q9**: tituly, ve kterých hrál herec `actor`
  - ```
    void db_query_9(
      const std::vector<std::shared_ptr<Title>>& db,
      std::vector<std::shared_ptr<Title>>& result,
      const Actor& actor
    );
    ```
- Cílem je naučit se používat vybrané standardní algoritmy
  - Knihovna `<algorithm>`
- Nalezené tituly uložte do připraveného výstupního kontejneru
  - Nejprve všechny tituly do výstupního kontejneru **nakopírujte**
    - Metoda `resize(count);`
    - Funkce `std::copy(begin, end, target);`
    ```

P4: Řazení titulů

Pokračování…

- Následně **odeberte** nevyhovující záznamy titulů
  - Funkce `std::remove_if(begin, end, predicate);`
  - Metoda `erase(begin, end);`
- **Filtrovací predikát** naimplementujte pomocí funktoru
  - Jeho parametrem bude konkrétní herec `actor`
  - Následně naprogramujte operátor kulatých závorek
    - ```
      bool operator()(
          const std::shared_ptr<Title>& title_ptr
      );
      ```
    - Cílem je vrátit `true`, pokud daný objekt má být odebrán
      ```

P4: Řazení titulů

Pokračování…

- Záznamy nakonec specifickým způsobem **seřaďte**
  - Funkce `std::sort(begin, end, comparator);`
- **Řadící komparátor** naimplementujte také pomocí funktoru
  - V rámci něj opět naprogramujte operátor kulatých závorek
    - ```
      bool operator()(
          const std::shared_ptr<Title>& title_ptr_1,
          const std::shared_ptr<Title>& title_ptr_2
      );
      ```
    - Cílem je vrátit `true`, pokud první objekt předchází druhý
    - Simulujeme tedy chování klasického operátoru <
  - Konkrétně chceme tituly seřadit sestupně podle roku natočení a vzestupně podle jména
      ```

P5: Žánry titulů

Implementujte následující **databázový dotaz**

- **Q10**: tituly mající žánr `genre`
  - ```
    void db_query_10(
      const std::vector<std::shared_ptr<Title>>& db,
      std::vector<std::shared_ptr<Title>>& result,
      const std::string& genre
    );
    ```
- Nalezené tituly uložte do připraveného výstupního kontejneru
  - Nejprve výstupní kontejner **inicializujte** na potřebnou velikost
  - Následně **překopírujte** vyhovující záznamy titulů
    - `std::copy_if(begin, end, target, predicate);`
  - Nakonec tituly **seřaďte** vzestupně podle jejich jmen
- V obou případech použijte **lambda výrazy**
    ```

P6: Agregace titulů

Implementujte následující **databázové dotazy**

- **Q11**: celočíselné průměrné hodnocení titulů majících typ `type` a žánr `genre`
  - ```
    void db_query_11(
        const std::vector<std::shared_ptr<Title>>& db,
        Type type, const std::string& genre,
        int& result
    );
    ```
- Spočítaný průměr uložte do výstupního parametru
  - `std::for_each(begin, end, function);`
  - Vše implementujte pomocí vlastního funktoru
    ```

P6: Agregace titulů

Prokračování…

- **Q12**: počet titulů majících žánr `genre`
  - ```
    void db_query_12(
      const std::vector<std::shared_ptr<Title>>& db,
      const std::string& genre, int& result
    );
    ```
- Spočítaný počet opět uložte do výstupního parametru
  - Použijte `std::for_each` a lambda výraz
    ```

Cvičení 10: **Matice**

Šablony tříd a funkcí
Závislá jména
Vnitřní třídy
Kontejner std::array
Vlastní aritmetické operátory
Vlastní operátory indexace
Konstrukce const_cast

P1: Jádro matice

Vytvořte šablonovanou třídu pro dvourozměrnou číselnou matici

- Parametry šablony budou typ prvku a výška a šířka matice
  - `template<typename element, size_t height, size_t width>`
    `class Matrix { ... }`
- Pro **vnitřní úložiště** použijte kontejner `std::array`
  - Ovšem jen jeden plochý, nikoli pole se zanořenými poli
  - Použijeme následující indexovou aritmetiku
    - `data_[row * width + column]`
- Definujte následující konstruktor
  - `Matrix(element value = 0);`
    - Inicializuje všechny prvky matice na danou hodnotu
    - Použijte funkci `data_.fill(…);`

P1: Jádro matice

Pokračování…

- Naprogramujte i následující **členské funkce**
  - `element get(size_t row, size_t column) const;`
    - Vrátí prvek na dané pozici
  - `void set(size_t row, size_t column, element value);`
    - Nastaví hodnotu prvku na určené pozici
  - `void print(std::ostream& os = std::cout) const;`
    - Vypíše matici do uvedeného výstupního streamu
    - Použijte následující formát: `[[1, 2], [3, 4], [5, 6]]`
- Nakonec naprogramujte i **operátor zápisu do streamu**
  - `std::ostream& operator<<(std::ostream& os, const Matrix<element, height, width>& matrix);`
- Všechny experimentálně vyzkoušejte

P2: Operátory inkrementace

Rozšiřte naši matici přidáním následujících operátorů

- Nejprve přidejte **operátor preinkrementace**
  - `Matrix& operator++();`
- Následně přidejte i **operátor postinkrementace**
  - `Matrix operator++(int);`
- Oba operátory implementujte jako členské funkce
  - Alternativně by šlo použít i variantu globálních funkcí
- Nové funkce opět experimentálně vyzkoušejte

P3: Operátor indexace

Rozšiřte naši matici přidáním následujících operátorů

- Cílem je umožnit psaní výrazů ve tvaru `matrix[1][2]`
  - V první úrovni specifikujeme **řádek**, ve druhé následně **sloupec**
  - Pomůžeme si prostřednictvím vnitřní pomocné třídy `Request`
    - Pomocí ní si uložíme referenci na matici a požadovaný řádek
- **První úroveň operátorů** nad třídou `Matrix`
  - `Request operator[](size_t row);`
  - `const Request operator[](size_t row) const;`
- **Druhá úroveň operátorů** nad pomocnou třídou `Request`
  - `element& operator[](size_t column);`
    - Zakrytí konstantnosti pomocí konverze `const_cast<…>(…);`
  - `const element& operator[](size_t column) const;`
- Ve všech případech je tentokrát nutné použít členské funkce

P4: Aritmetické operátory

Rozšiřte naši matici přidáním následujících operátorů

- **Přičtení konstanty k matici / vynásobení matice konstantou**
  - ```
    Matrix<element, height, width> operator+(
      const Matrix<element, height, width>& matrix,
      element increment
    );
    ```
  - ```
    Matrix<element, height, width> operator*(
      const Matrix<element, height, width>& matrix,
      element factor
    );
    ```
- Všechny operátory vyřešte jako globální funkce
  - Alternativně by šlo použít i variantu členských funkcí

P4: Aritmetické operátory

Pokračování…

- **Sečtení dvou matic / vynásobení dvou matic**
  - ```
    Matrix<element, height, width> operator+(
      const Matrix<element, height, width>& matrix1,
      const Matrix<element, height, width>& matrix2
    );
    ```
  - ```
    Matrix<element, height, width> operator*(
      const Matrix<element, height, depth>& matrix1,
      const Matrix<element, depth, width>& matrix2
    );
    ```

Cvičení 11: **Pole I**

Vlastní kontejner
Chytré ukazatele std::unique_ptr
Placement new operátor
Výjimky

P1: Gumové pole

Naprogramujte **vlastní kontejner gumového pole**

- Pole bude šablonované
  - Jediný parametr `element` bude určovat typ prvku
- Vnitřně data organizujte následujícím způsobem
  - Použijte vektor chytrých unikátních ukazatelů na bloky prvků
    - `std::unique_ptr<element[]>`
    - `std::make_unique<element[]>(block_size_);`
  - Každý blok reprezentujte jako tradiční pole fixní velikosti
    - Tuto velikost specifikujte pomocí parametrického konstruktoru
  - Pamatujte si také počet aktuálně vložených prvků
  - Pro přístup k prvkům použijeme indexovou aritmetiku
    - `data_[i / block_size_][i % block_size_];`

P1: Gumové pole

Pokračování…

- **Implementujte následující funkce**
  - `void push_back(const element& item);`
    `void push_back(element&& item);`
    - Vloží nový prvek do gumového pole
  - `void pop_back();`
    - Odebere poslední prvek (existuje-li)
  - Nápověda…
    - Explicitní volání destruktoru `~element();`
    - Explicitní volání výchozího / copy / move konstruktoru pomocí placement new operátoru

```
new (target) element();
new (target) element(item);
new (target) element(std::move(item));
```

P1: Gumové pole

Pokračování…

- **Implementujte následující funkce**
  - `inline size_t size() const;`
    - Vrátí aktuální počet uložených prvků
  - `void print(std::ostream& os = std::cout) const;`
    - Příklad: `[1, 2, 3, 4, 5]`
    - Každý jednotlivý prvek se vypíše pomocí jeho operátoru <<
  - ```
    std::ostream& operator<<(
      std::ostream& os,
      const Array<element>& array
    );
    ```
- Všechny funkce experimentálně vyzkoušejte
    ```

P2: Přístupové funkce

Rozšiřte funkcionalitu našeho gumového pole

- **Implementujte následující funkce**
  - `element& at(size_t index);`
  - `const element& at(size_t index) const;`
  - `element& operator[](size_t index);`
  - `const element& operator[](size_t index) const;`
- Všechny funkce opět experimentálně vyzkoušejte

P3: Vlastní výjimky

Navrhněte vlastní hierarchii tříd pro výjimky

- Společný předek `Exception`
  - Konstruktory
    - `Exception(const std::string& text);`
    - `Exception(std::string&& text);`
  - Metoda `inline const std::string& what() const;`
- Odvozené třídy
  - `OutOfRangeException`
    - Pro neplatný index ve funkcích `at(…)` a `operator[](…)`
  - `EmptyArrayException`
    - Při pokusu odebrat prvek z prázdného pole
  - `UnavailableMemoryException`
    - Pro případ neúspěšné alokace paměti u nového vnitřního bloku
    - Aneb jakožto reakce na výjimku `std::bad_alloc`

P3: Vlastní výjimky

Pokračování…

- **Chybové situace detekujte a ošetřete**
  - `throw …Exception(…);`
  - `try { ... } catch (Exception& e) { ... }`

Cvičení 12: **Pole II**

Copy a move konstruktory
Copy a move operátory přiřazení
Vlastní iterátory
Vlastní jmenný prostor

P1: Pokročilé konstruktory

Rozšiřte implementaci našeho gumového pole

- **Copy constructor**
  - `Array(`
    `const Array<element>& other`
    `);`
    - Otestování: `Array<int> a; auto b = a;`
- **Move constructor**
  - `Array(`
    `Array<element>&& other`
    `) noexcept;`
    - `std::swap(o1, o2);` nebo `std::move(…);`
    - Otestování: `Array<int> a; auto b = std::move(a);`

P1: Operátory přiřazení

Pokračování…

- **Copy assignment**
  - `Array<element>& operator=( const Array<element>& other );`
    - Otestování: `Array<int> a, b; b = a;`
- **Move assignment**
  - `Array<element>& operator=( Array<element>&& other ) noexcept;`
    - Kontrola smysluplnosti (`this != &other`)
    - Otestování: `Array<int> a, b; b = std::move(a);`

P2: Dopředný iterátor

Implementujte v našem kontejneru **dopředný iterátor**

- Realizujte jej pomocí vnitřní třídy `iterator`
  - `Knihovna <iterator>`
  - `template<typename element>`
    `class Array<element>::iterator { ... }`
  - Vnitřně uchovejte ukazatel na gumové pole a číslo pozice
- Přidejte následující tagy
  - `using iterator_category =`
    `std::forward_iterator_tag;`
  - `using difference_type = std::ptrdiff_t;`
  - `using value_type = element;`
  - `using pointer = element*;`
  - `using reference = element&;`

P2: Dopředný iterátor

Pokračování…

- Definujte následující **konstruktor**
  - ```
    iterator(
      Array<element>* array,
      size_t position
    );
    ```
- Implementujte následující **základní funkce**
  - `bool operator==(const iterator& other) const;`
  - `bool operator!=(const iterator& other) const;`
  - `iterator& operator++();`
  - `iterator operator++(int);`
  - `reference operator*() const;`
  - `pointer operator->() const;`
    ```

P2: Dopředný iterátor

Pokračování…

- Do třídy pole přidejte následující funkce
  - `iterator begin();`
  - `iterator end();`
- Nové funkce opět experimentálně vyzkoušejte
  - `for (auto&& item : array) { ... }`
  - `for (auto it = array.begin(); it != array.end(); ++it) { ... }`

P3: Rozšíření iterátoru

Rozšiřte funkcionalitu našeho současného iterátoru

- Implementujte následující operátory
  - `iterator& operator--();`
  - `iterator operator--(int);`
  - ```
    iterator operator+(
      const difference_type& movement
    ) const;
    ```
  - ```
    iterator operator-(
      const difference_type& movement
    ) const;
    ```

P4: Jmenné prostory

Refaktorujte současný kód našeho gumového pole

- Veškerou **implementaci vložte do jmenného prostoru** `lib`
  - `namespace lib { ... };`

Cvičení 13: **Pole III**

Dokumentace Doxygen

P1: Doxygen dokumentace

Seznamte se s dokumentačním nástrojem **Doxygen**

- Odkaz ke stažení
  - https://www.doxygen.nl/download.html
- Instalace
  - Přidejte cestu k adresáři `bin` do systémové proměnné `PATH`
- Vygenerujte konfigurační soubor
  - `doxygen -g config.ini`
- Nastavte následující direktivy
  - `PROJECT_NAME = "…"`
  - `EXTRACT_PRIVATE = YES`
  - `EXTRACT_STATIC = YES`
  - …

P1: Doxygen dokumentace

Zdokumentujte zdrojový kód našeho gumového pole

- Soubory
  - `/// @file filename`
- Třídy a parametry šablon
  - `/// …`
    `/// @tparam argname …`
- Položky tříd
  - `/// …`
- Globální a členské funkce
  - `/// …`
    `/// @param argname …`
    `/// @return …`
    `/// @exception typename …`

P1: Doxygen dokumentace

Pokračování…

- Dokumentaci vygenerujte a prostudujte
  - `doxygen config.ini`