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

Practical Class

NPRG041: Programming in C++

2025/26 Winter

Martin Svoboda
martin.svoboda@matfyz.cuni.cz

Charles University, Faculty of Mathematics and Physics

NPRG041: Programming in C++

2025/26 Winter | Martin Svoboda | martin.svoboda@matfyz.cuni.cz

Class 1: **Subsets**

Project structure
Function main
Header files
Passing parameters
Constness qualifier
Control structures
C-style arrays
Standard output
Memory allocation

E1: Hello World

Create a traditional Hello World application

- I.e., print the aforementioned greeting to the standard output
- Creating a new project in Visual Studio
  - Language: *C++*
  - Project type: *Empty Project*
- Useful hints
  - `#include <iostream>`
  - `int main(int argc, char** argv) { /* ... */ }`
  - `int main() { /* ... */ }`
  - `std::cout << "..." << std::endl;`

E2: Finding Subsets

Find and print **all subsets of a given set** on the input

- Simulate the input using a **constant expression**
  - Put it into a **header file** called `Input.h`
  - `#include "..."`
- Add **inclusion guards** to avoid repeated inclusion
  - Directives `#ifndef`, `#define`, `#endif`
- Assume, in particular, the following input
  - `constexpr char ITEMS[] = { 'A', 'B', 'C', 'D' };`
  - `constexpr size_t COUNT = sizeof(ITEMS) / sizeof(ITEMS[0]);`
- Whole program must be **universal**, though
  - I.e., it must work even with different input arrays

E2: Finding Subsets

Cont'd…

- **Decompose** the entire problem into appropriate functions
- Print each found subset to the **standard output**
  - Put exactly one subset on each line
    - Preserve the order of individual elements
    - Presence of an element takes precedence over its absence
  - Output format: `{ A, C, D }`
- **Dynamic allocation** of an array with size unknown in advance
  - `bool* signature = new bool[count];`
  - `delete[] signature;`
  - We will not solve possible allocation failures yet

NPRG041: Programming in C++

2025/26 Winter | Martin Svoboda | martin.svoboda@matfyz.cuni.cz

Class 2: **Options**

Declarations and definitions
Program arguments
Vector container
Strings
Iterators
Named constants
Type aliases
Short-circuit evaluation
Range-based for loops

E1: Printing Arguments

Print all the provided **input arguments** to the standard output

- Use the extended main function interface
  - `int main(int argc, char** argv) { /* ... */ }`
- First, transform the arguments to strings `std::string` and insert them into a container `std::vector`
  - `#include <string>`
  - `#include <vector>`
  - `using args_t = std::vector<std::string>;`
  - `args_t arguments(argv + 1, argv + argc);`
- Wrap the executive code into a separate function
  - Pass the container with arguments using a reference
  - Use the following approach to iterate over its items
    - `for (auto&& item : arguments) { /* ... */ }`

E1: Printing Arguments

Cont'd…

- Accessing namespace names
  - `using namespace std;`
  - Or just selectively `using std::cout, std::endl;`
  - Never use this construct in header files
- Separate definitions from declarations in header files
  - `#ifndef`, `#define`, `#endif`
  - `#include "..."`
- Setting input arguments in VS
  - (Project) *Properties* → *Debugging* → *Command Arguments*

E2: Options Detection

Detect a predefined set of expected **short and long options**

- In particular, expect the following options
  - `-t`, `-x`, `-y`
  - `--grayscale`, `--transparent`
- Introduce names of these options via global named constants
  - `constexpr char OPTION_TRANSPARENT_SHORT = 't';`
  - `constexpr char OPTION_TRANSPARENT_LONG[] = "transparent";`
- Allow grouping of short options, too
  - E.g.: `-xy`
- Print the recognized options to the standard output
  - Flag option <x> detected
  - Unknown option <something> found!

E2: Options Detection

Cont'd…

- Use iterators to iterate over the arguments this time
  - It allows us to control the course of iteration manually
  - ```
    for (
      auto it = arguments.begin();
      it != arguments.end();
      ++it
    ) { /* ... */ }
    ```
    - Iterator data type is `args_t::const_iterator`
    - And so `std::vector<std::string>::const_iterator`
- Iterator dereferencing
  - `const std::string& item = *it;`
    ```

E2: Options Detection

Cont'd…

- Useful methods over strings
  - `std::string substr(size_t pos, size_t len)`
    - Second parameter can be omitted
  - `size_t size()`
- Determine the exit code based on the detection success
  - `0` in the case of success, `1` otherwise

E3: Value Options

Extend our program with **detection of value options**

- In particular, expect the following new value options
  - `-r`, `-g`, `-b`, `-a`
  - `--red`, `--green`, `--blue`, `--alpha`
- Support the following means of passing values
  - `-xy -r 255`, `-xyr255`, `-xyr 255`
  - `-xy --red 255`
- Detect missing values as well as extra standalone values
  - `-r`, `-x something`
- Print everything to the standard output again
  - Value option <r> detected with value <255>
  - Value option <r> detected but its value is missing!
  - Standalone value detected <something>

NPRG041: Programming in C++

2025/26 Winter | Martin Svoboda | martin.svoboda@matfyz.cuni.cz

Class 3: **Counter**

Streams and files
Stream manipulators
Contextual conversions
Parsing numbers
Structures and classes
Static methods
Function overloading
Handling exceptions
C-style pointers

E1: Printing File

Print the contents of an input text file to the standard output

- Use the following constructs
  - Libraries `<iostream>`, `<fstream>`, `<string>`
  - `std::ifstream`
    - `void open(const char* filename);`
    - `bool good();`
    - `void close();`
  - `std::istream& std::getline(`
    `std::istream& input,`
    `std::string& line`
    `);`
- Print the following message after an unsuccessful file opening
  - `Unable to open input file`

E2: Counting Letters

Count and print the overall number of characters

- Place the code into an appropriate class and its static methods
  - `void process(const std::string& filename, size_t* chars);`
  - `void process(std::istream& stream, size_t* chars);`
  - `void print(const std::string& filename, const size_t* chars);`
  - `void print(std::ostream& stream, const size_t* chars);`
- Variable `chars` will be initialized by the caller
  - That allows to accumulate the value across multiple inputs
    - These can be **input files**, but also the **standard input**

E2: Counting Letters

Cont'd…

- Throw a **text exception** after an unsuccessful file opening
  - `throw "...";`
    - `Unable to open input file`
    - `Unable to open output file`
  - ```
    try { /* ... */ }
    catch (const char* e) { /* ... */ }
    ```
- Define the text messages via **named constants**
    ```

E3: Parsing of Numbers

Extend our program with **parsing of numeric values**

- We specifically want to recognize **integer numbers**
  - Use the following standard functions for that
  - `int std::isdigit(int c)`
    `int std::isalpha(int c)`
    - Library `<cctype>`
  - `int std::stoi(const std::string& s, size_t* p)`
    - Library `<string>`
    - Exceptions `std::invalid_argument`, `std::out_of_range`
- Throw text exception in case of invalid inputs
  - `Invalid integer number detected`
- We will temporarily assume a simplified **input format**
  - There is always one word or one number on each line
    - Skip possible empty lines

E3: Parsing of Numbers

Cont'd…

- We also want to extend the **detected statistics**
  - Number of lines, words, and numbers
  - Sum of all numbers
- Encapsulate all these records into a suitable **structure**
  - Define it in our header file
  - ```
    struct Statistics {
      size_t lines = 0;
      /* ... */
    }
    ```
- Alter printing of these statistics, too
  - E.g., one record on each line in the form `Lines: ?`
    ```

E4: Extended Counter

Add comprehensive **input text parsing** and additional statistics

- Considered input format
  - Input now contains an arbitrary number of sentences
  - Sentences are ended by `.` `!` `?` and separated by spaces
  - **Sentence** contains words or numbers separated by spaces
  - **Word** contains only letters
  - **Number** contains only digits `0` to `9` and possibly dot `.`
- Detect and store these records in our structure
  - Overall number of lines, sentences, words, and numbers
  - Overall number of letters, digits, spaces, and symbols
  - Sum of all integer and separately decimal numbers
- Print the calculated statistics again

E4: Extended Counter

Cont'd…

- **Parsing** floating point numbers
  - `float std::stof(const std::string& s, size_t* p)`
- **Printing** floating point numbers
  - Use **stream manipulators**
  - Library `<iomanip>`
  - `std::fixed`
  - `std::setprecision(precision)`
  - `std::defaultfloat`

NPRG041: Programming in C++

2025/26 Winter | Martin Svoboda | martin.svoboda@matfyz.cuni.cz

Class 4: **Movies I**

Classes
Constructors and destructors
Member initializer lists
Inline functions
Move semantics
Container set
Emplace mechanism
Legacy and uniform initialization
String streams

E1: Movie Representation

Propose a class for a movie representation

- Each movie has the following private data items
  - Name (`std::string`)
  - Filming year (`int`)
  - Genre (`std::string`)
  - Rating (`int`)
  - Set of actor names (`std::set<std::string>`)
    - Library `<set>`
- Implement the following functions first
  - Parameterized **constructor**
  - **Getter functions** for accessing individual data items
    - In the form of `inline` functions

E1: Movie Representation

Cont'd…

- Add a function for **printing** the movie as a JSON object
  - `void print_json(std::ostream& stream = std::cout) const;`
    - `{ name: "Bobule", year: 2008, genre: "comedy", rating: 65, actors: [ "Krystof Hadek", "Tereza Voriskova" ] }`
  - Actors field is not listed at all when no actors are provided
- Experimentally test your code directly in the `main` function
  - Create a container for movie instances
    - `std::vector<Movie> database;`
  - Manually add a couple of sample movies
  - And print the container content to the standard output

E2: Movie Construction

Allow for **more efficient creation of movie objects**

- Implement a constructor accepting rvalue references
  - In particular, for name, genre, and set of actors data items
- Try the following means of new movies creation and insertion
  - Standard `push_back`
  - Improved `push_back` combined with function `std::move`
    - Library `<utility>`
  - Mechanism `emplace_back`

E3: Importing Movies

Extend our database by **importing movies from CSV files**

- Add a **type alias** for our database first
  - To simplify the intended changes in the following tasks
  - `using database_t = std::vector<Movie>;`
  - Header file `Storage.h`
- Encapsulate the import functionality in a `Database` class
  - Header file `Database.h`
- Offer the following static member functions in particular
  - ```
    void import(const std::string& filename,
      database_t& database);
    ```
  - ```
    void import(std::istream& stream,
      database_t& database);
    ```

E3: Importing Movies

Cont'd…

- Use the following constructs for **parsing CSV** records
  - `std::istringstream`
    - Library `<sstream>`
  - `istream& std::getline( istream& stream, string& line, char delimiter );`
- The following **delimiters** are specifically assumed
  - Semicolon `;` for records as such
  - Comma `,` for actors
- Edge situations will be treated via **structured exceptions**
  - `struct Exception { int code; std::string text; }`

E3: Importing Movies

Cont'd…

- **Exceptions with code** `1` (inputs)
  - Unable to open input file <filename>
- **Exceptions with code** `2` (parsing)
  - Field names: `name`, `year`, `genre`, `rating`, and `actors`
  - Missing field <name> on line <line>
  - Empty string in field <name> on line <line>
  - Invalid integer <value> in field <name> on line <line>
  - Overflow integer <value> in field <name> on line <line>
  - Malformed integer <value> in field <name> on line <line>
  - Integer <value> out of range <min, max> in field <name> on line <line>
    - Intervals $[1900, 2100]$ for years and $[0, 100]$ for ratings

E4: Retrieving Movies

Prepare the following two simple **database queries**

- Header file `Queries.h`
- **Q1**: all movies
  - `void db_query_1(const database_t& database, std::ostream& stream = std::cout);`
  - Print the whole JSON objects of the found movies
- **Q2**: names of *comedies* filmed before *2010*, in which *Ivan Trojan* or *Tereza Voriskova* played
  - `void db_query_2(const database_t& database, std::ostream& stream = std::cout);`
  - Hard-wire all the query parameter values
  - Print names of the found movies only

NPRG041: Programming in C++

2025/26 Winter | Martin Svoboda | martin.svoboda@matfyz.cuni.cz

Class 5: **Expressions I**

Single inheritance
Virtual functions
Abstract classes
Inheriting constructors
Virtual destructors
Enumeration classes
Dynamic allocation
Operators new and delete
Null pointers

E1: Arithmetic Expressions

Assume simple integer **arithmetic expressions**

- These expressions may only contain…
  - Basic binary operations
    - Addition `+`, subtraction `-`, multiplication `*` and division `/`
  - Natural numbers including zero as simple operands

Propose **classes for inner tree nodes** of such expressions

- Abstract class `Node` as a common ancestor
- Final derived class `NumberNode` for leaf nodes with numbers
- Abstract derived class `OperationNode` for inner nodes
- Final derived classes for individual operations
  - `AdditionNode`, `SubtractionNode`, `MultiplicationNode`, `DivisionNode`

E1: Arithmetic Expressions

Cont'd…

- Basic use of the **inheritance concept**
  - `class NumberNode final : public Node { /* */ }`
- Distribute **data members** appropriately into individual classes
  - Leaf nodes: `private` number
  - Inner nodes: `protected` pointers to left and right subtrees
- Define the following **constructors**
  - `NumberNode(int number);`
  - `OperationNode(Node* left, Node* right);`
    - `using OperationNode::OperationNode;`
- Use enum class to distinguish between these two node types
  - `enum class Type { /* ... */ }`

E1: Arithmetic Expressions

Cont'd…

- Use **virtual member functions** appropriately
  - `virtual Type get_type() const;`
  - `virtual Type get_type() const = 0;`
  - `Type get_type() const override;`
- In particular, implement the following **member functions**
  - `Type get_type() const;`
    - As a `public` function for all nodes
    - Avoid use of data members to store the types of nodes
  - `char get_operator() const;`
    - Only as a `protected` function for operation nodes
    - Define operator symbols via global constants
    - Do without data members for these operators again

E1: Arithmetic Expressions

Cont'd…

- **Dynamic allocation** mechanism is assumed to be used
  - `Node* node_ptr = new NumberNode(2);`
  - `delete node_ptr;`
  - `nullptr` protection
- Do not forget **virtual destructor**
  - `~Node();`
- Add `Expression` class to encapsulate the expression
  - Constructor `Expression(Node* root);`
  - Destructor

E1: Arithmetic Expressions

Cont'd…

- Test all functionality experimentally
  - Implicit input: `(2+3)*4`
  - ```
    Expression e1(
      new MultiplicationNode(
        new AdditionNode(
          new NumberNode(2), new NumberNode(3)
        ),
        new NumberNode(4)
      )
    );
    ```
    ```

E2: Expression Evaluation

Extend our application for arithmetic expressions

- Add a function for **calculating the expression result**
  - `int evaluate() const;`
    - We will not deal with division by zero yet

E3: Expression Printing

Extend our application for arithmetic expressions

- Add a function for **printing the expression in postfix notation**
  - I.e., the so-called reverse Polish notation
    - You just need to perform a postorder depth-first tree traversal
  - ```
    void print_postfix(
      std::ostream& stream = std::cout
    ) const;
    ```
  - Always separate operators and numbers with exactly one space
- Example
  - Implicit input: `1*2+3*(4+5)-6`
  - Output: `1 2 * 3 4 5 + * + 6 -`
    ```

E4: Expression Printing

Extend our application for arithmetic expressions

- Add a function for **printing the expression in infix notation**
  - `void print_infix(`
    `std::ostream& stream = std::cout`
    `) const;`
  - Do not print any **spaces** around operators or parentheses
  - Print **parentheses** only when really necessary
    - Operations `*` and `/` have higher precedence than `+` and `-`
- Example
  - Implicit input: `(7+(9-(3*1))/3)-(5-1)`
  - Output: `7+(9-3*1)/3-(5-1)`

NPRG041: Programming in C++

2025/26 Winter | Martin Svoboda | martin.svoboda@matfyz.cuni.cz

Class 6: **Expressions II**

Polymorphic containers
Container stack
String views
Compiler-generated methods
Compiler attributes
Custom exception hierarchies
Dynamic allocation failures
Memory leaks avoidance

E1: Custom Exceptions

Propose your own hierarchy of classes for exceptions

- Common ancestor `Exception`
  - Constructor `inline Exception(const char* message);`
  - Method `inline const char* message() const;`
  - Destructor `virtual ~Exception() = default;`
- Derived classes
  - `EvaluationException`
  - `ParseException`
  - `MemoryException`
- Deal with **division by zero** when evaluating expressions
  - Exception `EvaluationException`
  - Text message `Division by zero`

E2: Expression Parsing

Create a simple **parser for infix arithmetic expressions**

- Only syntactically well-formed expressions are considered
  - We continue to work only with natural **numbers** and zero
    - I.e., numbers cannot be preceded by a unary minus `-`
  - They may also contain auxiliary round **parentheses** `()`
- Convert the input expression to **postfix notation**
  - I.e., print the expression in postfix notation to the output
    - Input: `10*2+3*((1+14)-18)-10`
    - Output: `10 2 * 3 1 14 + 18 - * + 10 -`
  - Separate operators and numbers with exactly one space
- Use the **shunting-yard algorithm** for the transformation

E2: Expression Parsing

```
1  foreach token t in the input infix expression do
2      if t is a number then  print t to the standard output
3      else if t is an opening parenthesis ( then  put ( onto the stack
4      else if t is a closing parenthesis ) then
5          while there is an operator o on top of the stack do
6              remove o from the stack and print it to standard output
7          remove ( from the stack
8      else t is an operator n
9          while there is an operator o with precedence higher than n,
10         or the same, but only if n is left-associative do
11             remove o from the stack and print it to standard output
12         add n onto the stack
13 while the stack is non-empty do
14     remove o from the stack and print it to standard output
```

E2: Expression Parsing

Cont'd…

- We assume the following **properties of operations**
  - They are all left-associative
  - Operations `*` and `/` have higher precedence than `+` and `-`
- Use the standard **stack** container
  - `std::stack<char>` (library `<stack>`)
  - Methods `push(…)`, `top()`, `pop()`, `size()`, `empty()`
- **Expected implementation**
  - Class `Parser`
  - Static method `void parse(std::string_view input);`
- **String views**
  - Class `std::string_view` (library `<string_view>`)

E3: Syntactic Tree

Extend our parser for arithmetic expressions

- **Construct a syntactic tree representing the input expression**
- Use a modified shunting-yard algorithm
  - We will now also need a second stack for operands
    - `std::stack<Node*>`
  - Creation of leaf nodes for **numbers**…
    - Create a new node and put it onto this stack
  - Creation of internal nodes for **operations**…
    - Remove the right and then left operand from this stack
    - Create a new node and insert it onto this stack
  - We will find the **root node** on this stack at the very end
    - It will be its only element

E3: Syntactic Tree

```
1  foreach token t in the input infix expression do
2      if t is a number then  create a new leaf node for t…
3      else if t is an opening parenthesis ( then  put ( onto the operator st.
4      else if t is a closing parenthesis ) then
5          while there is an operator o on top of the stack of operators do
6              remove o from the stack and create a new inner node for o…
7          remove ( from the stack of operators
8      else t is an operator n
9          while there is an operator o with precedence higher than n,
10         or the same, but only if n is left-associative do
11             remove o from the stack and create a new inner node for o…
12         add n onto the stack of operators
13 while the stack of operators is non-empty do
14     remove o from the stack and create a new inner node for o…
```

E3: Syntactic Tree

Cont'd…

- **Expected implementation**
  - `Parser` class extension
    - Data members for both the stacks
  - `[[nodiscard]] static Node* parse(std::string_view input);`
    - Creation of a **parser instance** using a private constructor
    - **Attribute** `[[nodiscard]]` enforces handling of return values
  - Other **helper functions** will be normal (non-static)
    - Decomposition at least for individual tokens
    - No need to pass references to the stacks
- Finally, add a new `Expression` **class constructor**
  - `Expression(std::string_view input);`

E3: Syntactic Tree

Non-standard situations will be handled using exceptions

- `ParseException`
  - Unknown token (e.g., `a`, `3a`, `a3`, …)
    - `Unknown token`
  - Number value overflow
    - `Overflow number`
  - Lack of operands when creating an operation node
    - `Missing operands`
  - Unpaired opening / closing round parentheses
    - `Unmatched opening parenthesis`
    - `Unmatched closing parenthesis`
  - Incorrect number of operand nodes at the algorithm end
    - `Unused operands`
    - `Empty expression`

E3: Syntactic Tree

Cont'd…

- `MemoryException`
  - Out of memory for dynamically allocated operands
    - `Unavailable memory`
    - Response to the exception `std::bad_alloc`
- Pay attention to ensuring **atomic behavior**
  - I.e., we must **empty the operand stack** in the event of errors
    - This means we need to deallocate all the prepared nodes
    - We would otherwise uncontrollably lose our memory
  - **Exception rethrowing**
    - ```
      try { /* ... */ }
      catch (const Exception& e) { /* ... */ throw; }
      ```
      ```

NPRG041: Programming in C++

2025/26 Winter | Martin Svoboda | martin.svoboda@matfyz.cuni.cz

Class 7: **Movies II**

Code refactoring
Container set
Custom operators
Shared smart pointers
Declaration friend
Optional values
Formatted strings
Static retyping
Dynamic retyping

E1: Structured Actors

Modify and extend our movie database application

- Actor will no longer be just an atomic string with a name, but a structured record with the following items
  - First name (`std::string`), last name (`std::string`)
  - Year of birth (`int`)
- Propose a class to represent such an actor
  - Prepare default and parameterized constructors
    - `Actor() = default;`
  - Add access functions for individual items, too
- Implement a **custom comparison operator** for actors
  - Global function `bool operator<(const Actor& actor1, const Actor& actor2);`
    - Order is defined by a triple of surname, first name, and year

E1: Structured Actors

Cont'd…

- Allow for **printing of actors** via a custom operator <<
  - `std::ostream& operator<<(std::ostream& stream, const Actor& actor);`
  - We will again utilize a JSON object
    - `{ name: "Ivan", surname: "Trojan", year: 1964 }`
- **Import of actors** will also be solved with our own operator >>
  - Entirely empty actors will again be skipped
  - `std::istream& operator>>(std::istream& stream, Actor& actor);`
  - Individual attributes are separated by spaces
    - `Ivan Trojan 1964`

E1: Structured Actors

Cont'd…

- **Actor import errors** will again be handled via **exceptions**
  - Use stream contextual conversion
  - Final text messages will be constructed in **two stages**
    - Actor stream **extraction operator** generates the first part …
    - … so that it can then be finalized in the **import process**
- **Exceptions with code** 2
  - Attribute names: `name`, `surname`, and `year`
  - Missing attribute <name>
  - Missing, invalid, or overflow value in attribute <name>
  - Integer <value> out of range <min, max> in attribute <name>
    - In particular, interval $[1850, 2100]$ is assumed for the years
  - … in actor <actor> on line <line>

E1: Structured Actors

Cont'd…

- Use **formatted strings** for exception messages
  - `std::format(format, /* ... */)`
    - Library `<format>`
- Refactor the remaining parts of the current code as well
  - Import process, database queries, …

E2: Titles Hierarchy

Extend our application to support different types of titles

- First, refactor the current code
  - Rename class `Movie` to `Title`
  - Database container will now contain smart pointers
    - `std::shared_ptr<...>` (library `<memory>`)
    - Function `std::make_shared<Title>(...);`
- Next, propose a new hierarchy of titles
  - Class `Title` will become abstract
  - Derived class `Movie` with an additional item
    - Optional length in minutes (`int`) with values $[0, 300]$
    - Class `std::optional<...>` (library `<optional>`)
  - Derived class `Series` with additional items
    - Number of seasons (`int`) with values $[0, 100]$
    - Number of episodes (`int`) with values $[0, 10000]$

E2: Titles Hierarchy

Cont'd…

- Add also the following functions
  - Constructors and functions for accessing new items
  - Enumeration to distinguish types of titles
    - `enum class Type { MOVIE, SERIES };`
  - Function for returning such a type
    - `Type type() const;`
- Modify the function for **printing titles**
  - Add a field describing the title type to the beginning
    - Movies: `{ type: "MOVIE", ... }`
    - Series: `{ type: "SERIES", ... }`
  - Add new specific items to the end, on the contrary
    - Movies: `{ ..., length: 112 }`
    - Series: `{ ..., seasons: 8, episodes: 73 }`

E2: Titles Hierarchy

Cont'd…

- Modify the function for **importing titles**
  - Expect a string distinguishing the title type at the beginning
    - Movies: `MOVIE;...`
    - Series: `SERIES;...`
  - Expect newly added specific items at the end, on the contrary
    - Movies: `...;112`
    - Series: `...;8;73`
- We continue to use **exceptions** to treat extreme situations
  - Code `2` (also for fields `type`, `length`, `seasons`, and `episodes`)
    - Invalid type selector <selector> in field <name> on line <line>
- Refactor the remaining parts of the current code as well
  - I.e., at least the database queries

E3: Type Conversions

Prepare the following three **database queries**

- **Q3**: titles having a type `type` filmed in years [begin, end)
  - ```
    void db_query_3(
      const database_t& database,
      Type type, int begin, int end,
      std::ostream& stream = std::cout
    );
    ```
  - Interpret the interval of years as open from the right
  - Return the found titles as strings with JSON objects
    - Only types, names, and years of these titles will be included
    - `{ type: "…", name: "…", year: … }`
    ```

E3: Type Conversions

Cont'd…

- **Q4**: series with at least `seasons` number of seasons or at least `episodes` number of episodes
  - ```
    void db_query_4(
        const database_t& database,
        int seasons, int episodes,
        std::ostream& stream = std::cout
    );
    ```
  - Use **static retyping**
    - `(Series*)&*title_ptr;`
    - `static_cast<Series*>(&*title_ptr);`
    - `std::static_pointer_cast<Series>(title_ptr);`
  - Return the found titles as strings with JSON objects again
    - `{ name: "…", seasons: …, episodes: … }`
    ```

E3: Type Conversions

Cont'd…

- **Q5**: movies with a length at least `length` minutes
  - ```
    void db_query_5(
      const database_t& database,
      int length,
      std::ostream& stream = std::cout
    );
    ```
  - Use **dynamic retyping**
    - `dynamic_cast<Movie*>(&*title_ptr);`
    - `std::dynamic_pointer_cast<Movie>(title_ptr);`
  - Return the found titles as strings with JSON objects again
    - `{ name: "…", length: … }`
    ```

NPRG041: Programming in C++

2025/26 Winter | Martin Svoboda | martin.svoboda@matfyz.cuni.cz

Class 8: **Matrix**

Class and function templates
Template instantiation
Dependent names
Array container
Inner classes
Two-level indexing
Implementation of operators
Constness conversion
Copy-on-write mechanism

E1: Matrix Core

Create a template class for a two-dimensional numeric matrix

- Template parameters: element type, matrix height and width
  - ```
    template<typename Element, size_t Height,
      size_t Width>
    class Matrix { /* ... */ }
    ```
- Use `std::array` container for the **inner storage**
  - However, only one flat, not an array with embedded arrays
    - We will therefore use the following index arithmetic
    - `data_[row * Width + column]`
  - Two template parameters: element type, number of elements
- Define the following constructor
  - `Matrix(const Element& value = 0);`
    - Initialize all matrix elements using `data_.fill(…);`
    ```

E1: Matrix Core

Cont'd…

- Implement the following **member functions**
  - `Element& get(size_t row, size_t column);`
    - Returns a modifiable reference to the element at a given logical position
  - `const Element& get(size_t row, size_t column) const;`
    - Analogously returns a constant reference
  - `void set(size_t row, size_t column, const Element& value);`
    - Sets a new value of the element at a specified position

E1: Matrix Core

Cont'd…

- Implement the following print function
  - `void print(std::ostream& os = std::cout) const;`
    - Prints the matrix to a given output stream
    - Use the following format: `[[1, 2], [3, 4], [5, 6]]`
- Finally, implement the **stream insertion operator** as well
  - ```
    std::ostream& operator<<(
      std::ostream& stream,
      const Matrix<Element, Height, Width>& matrix
    );
    ```
- Experimentally try them all
    ```

E2: Increment Operators

Extend our matrix by adding the following operators

- **Pre-increment operator**
  - `Matrix& operator++();`
- **Post-increment operator**
  - `Matrix operator++(int);`
- Implement both the operators as member functions
  - Global functions could alternatively be used as well

E3: Subscript Operators

Extend our matrix by adding the following **subscript operators**

- We start with a solution that is easier to implement…
- **Single-level indexing** (e.g., `matrix[5]`)
  - Physical positions within the internal storage will be used
- Required operators
  - `Element& operator[](size_t index);`
  - `const Element& operator[](size_t index) const;`
- We then replace this code with a better solution…

E3: Subscript Operators

Cont'd…

- **Two-level indexing** (e.g., `matrix[1][2]`)
  - Particular **row** is specified first, **column** subsequently
  - Auxiliary class `Request` will be needed
    - Requested row and matrix reference will be stored within it
- **First level of operators** over the `Matrix` class
  - `Request operator[](size_t row);`
  - `const Request operator[](size_t row) const;`
- **Second level of operators** over the `Request` class
  - `Element& operator[](size_t column);`
    - Concealing constancy with conversion `const_cast<…>(…);`
  - `const Element& operator[](size_t column) const;`
- Use of member functions is necessary in all cases this time

E4: Deferred Copying

Add support for the **copy-on-write** mechanism to our matrix

- In order to ensure **content sharing** across matrix instances
  - And their separation (and thus copying) only when necessary
- We adjust the **internal storage** first
  - Use a **shared pointer** to detach it outside of the matrix
  - `std::shared_ptr<std::array<…, …>> data_;`
- Prepare an internal method for **data separation**
  - `void ensure_ownership();`
  - Ensures the need for separation and its execution
    - Smart pointer method `long use_count();`
  - Call the method in every **modifying operation** on the matrix
    - Including modifying variants of `get` and `[]`
- Make necessary adjustments to the current code

E5: Arithmetic Operators

Extend our matrix by adding the following operators

- **Adding a constant to a matrix**
  - ```
    Matrix<Element, Height, Width> operator+(
      const Matrix<Element, Height, Width>& matrix,
      const Element& increment
    );
    ```
- **Multiplying a matrix by a constant**
  - ```
    Matrix<Element, Height, Width> operator*(
      const Matrix<Element, Height, Width>& matrix,
      const Element& factor
    );
    ```
- Solve all these operators as global functions
  - Member functions could alternatively be used as well

E5: Arithmetic Operators

Cont'd…

- **Addition of two matrices**
  - ```
    Matrix<Element, Height, Width> operator+(
      const Matrix<Element, Height, Width>& matrix1,
      const Matrix<Element, Height, Width>& matrix2
    );
    ```
- **Multiplication of two matrices**
  - ```
    Matrix<Element, Height, Width> operator*(
      const Matrix<Element, Height, Depth>& matrix1,
      const Matrix<Element, Depth, Width>& matrix2
    );
    ```

NPRG041: Programming in C++

2025/26 Winter | Martin Svoboda | martin.svoboda@matfyz.cuni.cz

Class 9: **Movies III**

Associative containers
Helper structures
Structured binding
Functors
Standard functors
Passing callable objects
Template specialization
Observer pointers
Copy elision mechanism

E1: Title Names

Create an index for **finding titles by their names**

- Use an **ordered map container**
  - `using index_titles_by_names_t = std::map<std::string, std::shared_ptr<Title>>`
  - Library `<map>`
- Create this **index** using the following function
  - `void db_index_1( const database_t& database, index_titles_by_names_t& index );`
- **Insertion of entries** into the index
  - `std::pair<…, …> item;` or `std::make_pair(…, …);`
  - Methods `index.insert(…);` or `index.emplace(…);` resp.

E1: Title Names

Implement the following **database query**

- **Q6**: title with a name `name`
  - ```
    void db_query_6(
      const index_titles_by_names_t& index,
      std::string_view name,
      std::ostream& stream = std::cout
    );
    ```
  - Finding the intended title
    - Function `index.find(name);`
  - Internal **pair** `std::pair`
    - Data members `first` and `second`
  - Print the whole JSON object of the found title
    - Or `Not found!` otherwise
    ```

E2: Numbers of Actors

Create an index for **finding actors by their years of birth**

- Use an ordered map container again
  - `using index_actors_by_years_t = std::map<int, std::set<Actor>>`
- Create this index using the following function
  - `void db_index_2( const database_t& database, index_actors_by_years_t& index );`
- Insertion of entries into the index
  - Use the `[]` **operator** at the level of the outer map

E2: Numbers of Actors

Cont'd…

- **Q7**: overall number of actors born during years [`begin`, `end`)
  - ```
    void db_query_7(
      const index_actors_by_years_t& index,
      int begin, int end,
      std::ostream& stream = std::cout
    );
    ```
  - Finding the intended years
    - Iterator from: `index.lower_bound(begin);`
    - Iterator to: `index.lower_bound(end);`
  - Expected output
    - `count actors` or `actor` accordingly
    ```

E3: Filming of Movies

Create an index for **finding titles by years of their filming**

- Use an **ordered multimap container** this time
  - `using index_titles_by_years_t = std::multimap<int, std::shared_ptr<Title>>`
  - **Ordering of elements**
    - **Default functor** is assumed `std::less<int>`
    - Third template parameter
- Create this index using the following function
  - `void db_index_3( const database_t& database, index_titles_by_years_t& index );`

E3: Filming of Movies

Implement the following **database queries**

- **Q8**: titles filmed in a year `year`
  - ```
    void db_query_8(
      const index_titles_by_years_t& index,
      int year,
      std::ostream& stream = std::cout
    );
    ```
  - Finding the intended titles
    - Function `index.equal_range(year)`;
    - Returns a pair (`std::pair`) of iterators [from, to)
  - Print strings with the following JSON object
    - `{ name: "…", year: … }` for each title
    - Or `Not found!` otherwise
    ```

E3: Filming of Movies

Cont'd…

- **Q9**: titles filmed between years [`begin`, `end`)
  - ```
    void db_query_9(
      const index_titles_by_years_t& index,
      int begin, int end,
      std::ostream& stream = std::cout
    );
    ```
  - Finding the intended titles
    - Iterator from: `index.lower_bound(begin);`
    - Iterator to: `index.lower_bound(end);`
  - Print strings with the following JSON object again
    - `{ name: "…", year: … }` for each title
    - Or `Not found!` otherwise
    ```

E4: Finding Actors

Implement the following **database query**

- **Q10**: casting of actors born between years [`begin`, `end`)
  - ```
    void db_query_10(
      const database_t& database,
      int begin, int end,
      std::ostream& stream = std::cout
    );
    ```
  - We are looking for every single casting of matching actors
    - We only want the actors themselves, duplicates are preserved
  - Use a **multiset container** internally
    - `std::multiset<Actor, Comparator_Q10_Actors>`
    - **Comparison** will be achieved via a **custom functor** …
  - Print the whole JSON object for each actor
    - Or `Not found!` otherwise
    ```

E4: Finding Actors

Cont'd…

- **Comparison functor**
  - Functor is a regular class implementing the `()` operator
  - Specifically in our case
    - ```
      bool operator()(
          const Actor& actor1, const Actor& actor2
      ) const;
      ```
- **Comparing actors**
  - Ascending order by years, first names, and surnames
    - We will apply the following trick
  - **Tuple class** `std::tuple<...>` (library `<tuple>`)
  - Function `std::make_tuple(...)`;
  - Available operator `<`
    - I.e., create auxiliary tuples and then mutually compare them
      ```

E5: Finding Titles

Implement the following **database query**

- **Q11**: titles satisfying a search condition `predicate`
  - ```
    void db_query_11(
      const database_t& database,
      const std::function<bool(const Title*)>&
        predicate,
      std::ostream& stream = std::cout
    );
    ```
  - Purpose of the **predicate function**
    - Expects a title passed via the so-called **observer** pointer
    - Returns `true` for titles we want to include in the result
    - Structure `std::function` (library `<functional>`)
  - Print full JSON objects of the found titles
    - Or `Not found!` otherwise
    ```

E5: Finding Titles

Cont'd…

- We then prepare two particular predicates
- Implement the first one using an ordinary **global function**
  - `bool predicate_Q11_movies(const Title* title);`
    - Find movies (not series) with at least three actors that are not comedies
- Implement the second one as a **functor**
  - ```
    class Predicate_Q11_Titles {
    public:
      bool operator()(const Title* title) const;
    };
    ```
    - Find titles with a rating of at least *80* in which *Tatiana Vilhelmova 1978* played
    ```

E6: Actors Casting

Create an index for **finding titles by their actors**

- Use an **unordered multimap container**
  - `using index_titles_by_actors_t = std::unordered_multimap< Actor, std::shared_ptr<Title> >`
  - Default functors for **hashing** and **ordering** are assumed
    - `std::hash<Actor>` and `std::equal_to<Actor>`
  - Library `<unordered_map>`
- Create this index using the following function
  - `void db_index_4( const database_t& database, index_titles_by_actors_t& index );`

E6: Actors Casting

Cont'd…

- **Hash functor specialization**
  - `template<>`
    `struct std::hash<Actor> { /* ... */ }`
  - Method `size_t operator()(`
    `const Actor& actor`
    `) const noexcept;`
  - Use actor last name
    - Specifically via `std::hash<std::string>{}(…);`
- **Comparison operator**
  - Global function `bool operator==(`
    `const Actor& actor1, const Actor& actor2`
    `);`
    - Use function `std::tie(...)` for auxiliary tuples

E6: Actors Casting

Implement the following **database query**

- **Q12**: titles where an actor with a surname `surname` played
  - `std::vector<Title*> db_query_12(`
    `const index_titles_by_actors_t& index,`
    `string_view surname`
    `);`
  - Finding the intended titles
    - Unfolding of a pair using **structured binding**
    - `for (auto&& [key, value] : index) { /* ... */ };`
  - Put the found titles into the output container
    - In the form of C-style observer pointers

E7: Title Genres

Create an index for **finding actors and titles by genres and years**

- Use an ordered multimap container and tuples
  - `using index_cast_by_genres_t = std::multimap<`
    `std::tuple<std::string, int>,`
    `std::tuple<std::string, std::string,`
    `std::shared_ptr<Title>>`
    `>`
  - Meaning of pairs and triples in the map
    - **Key**: genre and year of title filming
    - **Value**: first and last actor name, pointer to title
- Create this index using the following function
  - `void db_index_5(…, …);`

E7: Title Genres

Implement the following **database query**

- **Q13**: names of actors and names of titles in titles having a genre `genre` filmed in a year `year`
  - ```
    std::vector<std::string> db_query_13(
      const index_cast_by_genres_t& index,
      std::string_view genre,
      int year
    );
    ```
  - **Accessing members of tuples**
    - Function `std::get<position>(tuple)`;
    - Or via structured binding
  - Return found records in the form of strings with JSON objects
    - `{ name: "…", surname: "…", title: "…" }`
    ```

NPRG041: Programming in C++

2025/26 Winter | Martin Svoboda | martin.svoboda@matfyz.cuni.cz

Class 10: **Array I**

Low-level dynamic allocation
Constructors and destructors invocation
Special member functions
Exception guarantees
Resource stealing
Rule of Five / Three / Zero
Implicitly generated functions
Conditional compilation
Standard exceptions

E1: Flexible Array

Implement a **custom flexible array container**

- Single template parameter: element type `T`
- **Internal storage**
  - **First level**
    - Standard vector of C-style pointers to item blocks
  - **Second level** (individual blocks)
    - C-style array for individual items
    - Low-level dynamic allocation will be used
- Assumptions
  - Items will only be added / removed at the end
  - Index arithmetic for accessing items
    - `data_[i / block_size_][i % block_size_];`
  - Maintaining necessary capacity only

E1: Flexible Array

Cont'd…

- **Data members**
  - Selected fixed block size (number of available slots in a block)
  - Internal storage as such
  - Current capacity and current number of items
- **Constructor**
  - `Array(size_t block_size = 10);`
    - Optional parameter determines the selected block size
  - We will add more constructors later on…
- **Destructor**
  - `~Array() noexcept;`
    - We will postpone its implementation for now…

E1: Flexible Array

Cont'd…

- **Basic functions**
  - `inline size_t size() const;`
    - Returns the current number of items stored
  - `inline bool empty() const;`
    - Returns `true` if and only if the array is empty
  - `inline size_t capacity() const;`
    - Returns the current internal storage capacity
- **Access functions**
  - `T& at(size_t index);`
  - `const T& at(size_t index) const;`
  - `T& operator[](size_t index);`
  - `const T& operator[](size_t index) const;`

E2: Element Insertion

Implement functions for **adding and removing items**

- **Internal block addition**
  - Determining required memory size
    - Operator `sizeof(type)`
  - Block dynamic allocation
    - Function `void* ::operator new(size_t size);`
    - Library `<new>`
    - Throws `std::bad_alloc` if not successful
  - **Ensuring atomicity**
    - Notice that the `push_back` call at the first level may fail
    - Rollback will then be needed
- **Internal block removal**
  - Block deallocation
    - Function `void ::operator delete(void* ptr);`

E2: Element Insertion

Cont'd…

- **Item addition**
  - `void push_back(const T& item);`
    `void push_back(T&& item);`
    - Inserts a new item into the flexible array
  - Explicit invocation of item copy / move constructors
    - `new (target) T(item);`
    - `new (target) T(std::move(item));`
  - **Ensuring atomicity**
    - Beware of failed item construction
- **Item removal**
  - `void pop_back();`
    - Removes the last item
  - Explicit destructor call `~T();`

E2: Element Insertion

Cont'd…

- **Destructor** and container emptying
  - `void clear();`
    - Removes all existing items
  - `~Array() noexcept;`

E3: Special Members

Extend the implementation of our flexible array

- **Special member functions**
  - Copy / move constructor / assignment operator
  - But first…
- Global **swap function**
  - ```
    void swap(
      Array<T>& array_1, Array<T>& array_2
    ) noexcept;
    ```
    - Use `std::swap(o1, o2);` on all members
    ```

E3: Special Members

Cont'd…

- **Copy constructor**
  - `Array(const Array& other);`
    - Testing: `Array<int> a; auto b = a;`
- **Copy assignment**
  - `Array& operator=(const Array& other);`
    - Validity check (`this != &other`)
    - Testing: `Array<int> a, b; b = a;`
- Ensuring **atomicity** in both the cases
  - I.e., **strong exception guarantee**

E3: Special Members

Cont'd…

- **Move constructor**
  - `Array(Array&& other) noexcept;`
    - Testing: `Array<int> a; auto b = std::move(a);`
- **Move assignment**
  - `Array& operator=(Array&& other) noexcept;`
    - Validity check (`this != &other`)
    - Testing: `Array<int> a, b; b = std::move(a);`

E4: Debug Exceptions

Add the support for flexible array **user debugging**

- Activation using a macro
  - `#define ARRAY_DEBUG_MODE`
  - `#ifdef ARRAY_DEBUG_MODE`
  - `#endif`
- In particular, the following **standard exceptions** are assumed
  - Library `<stdexcept>`
  - `std::out_of_range("Invalid index")`
    - For an invalid index in access functions
    - Always in `at(…)`
    - Conditionally in `operator[](…)`
  - `std::invalid_argument("Empty array")`
    - Conditionally when removing an item from an empty array

NPRG041: Programming in C++

2025/26 Winter | Martin Svoboda | martin.svoboda@matfyz.cuni.cz

Class 11: **Array II**

Initializer list constructors
Concepts
Requires clauses and expressions
Static assertions
Perfect forwarding
Variadic templates
Custom iterators
Iterator categories
Conversion operators

E1: Array Modifications

Modify the implementation of our flexible array

- Put the entire array code into a **custom namespace** `lib`
  - `namespace lib { /* ... */ };`
- Add public **type aliases** into the array class
  - `using value_type = T;`
  - `using size_type = std::size_t;`
  - `using difference_type = std::ptrdiff_t;`
  - `using reference = T&;`
  - `using const_reference = const T&;`
  - `using pointer = T*;`
  - `using const_pointer = const T*;`
  - Two more will be added later on…

E1: Array Modifications

Cont'd…

- Create a **custom concept** for the array elements
  - ```
    template<typename T>
    concept ArrayElement =
      (
        std::copy_constructible<T> ||
        std::move_constructible<T> ||
        std::default_initializable<T>
      ) && std::destructible<T>;
    ```
  - Library `<concepts>`
- Use this concept within our array
  - ```
    template<ArrayElement T>
    class Array;
    ```

E1: Array Modifications

Cont'd…

- Add compile-time checks using **assert declarations**
  - `static_assert(condition, message);`
    - Violation of the condition triggers a compilation error
  - Copy constructor / assignment and `push_back` methods
    - `std::copy_constructible<T>` or `std::move_constructible<T>` respectively
  - Use meaningful text messages

E2: Advanced Constructors

Add additional constructors to the flexible array class

- **Repeating constructor**
  - `Array(size_t count, const T& item);`
- **Initializer constructor**
  - `Array(std::initializer_list<T> items);`
    - Library `<initializer_list>`
    - `for (auto&& item : items) { /* ... */ }`
- Ensure atomicity
  - And also add relevant **concept checks**

E2: Advanced Constructors

Cont'd…

- **Range constructor**
  - `template<std::input_iterator InputIterator>`
    `Array(InputIterator first, InputIterator last);`
    - Library `<iterator>`
    - Concept `std::input_iterator`
    - Allows both **iterators** and **C-style pointers**
  - Concept check
    - ```
      std::constructible_from<
        T,
        typename std::iterator_traits<InputIterator>
          ::reference
      >
      ```
      ```

E3: Element Insertion

Extend the flexible array insertion capabilities

- Merge both the existing `push_back` functions
  - ```
    template<typename Source>
    requires std::constructible_from<T, Source&&>
    void push_back(Source&& item);
    ```
    - `std::forward<Source>(item);`
    - Library `<utility>`
- Also add the `emplace_back` function
  - We use the following variadic template
  - ```
    template<typename... Args>
    requires std::constructible_from<T, Args&&...>
    T& emplace_back(Args&&... args);
    ```
    - `std::forward<Args>(args)...;`

E4: Forward Iterator

Implement a custom **forward iterator** in our container

- **Inner class**
  - `class iterator;`
  - `template<typename T>`
    `class Array<T>::iterator { /* ... */ };`
- Private **data members**
  - Flexible array pointer
  - Position number
- Public **constructors**
  - `iterator(Array<T>* array, size_t position);`
  - `iterator();`

E4: Forward Iterator

Cont'd…

- Flexible array methods
  - `iterator begin();`
  - `iterator end();`
- Public **type aliases** inside the iterator class
  - `Library <iterator>`
  - `using iterator_category = std::forward_iterator_tag;`
  - `using value_type = T;`
  - `using pointer = T*;`
  - `using reference = T&;`
  - `using difference_type = std::ptrdiff_t;`

E4: Forward Iterator

Cont'd…

- Expected **basic functions**
  - `bool operator==(const iterator& other) const;`
  - `bool operator!=(const iterator& other) const;`
  - `iterator& operator++();`
  - `iterator operator++(int);`
  - `reference operator*() const;`
  - `pointer operator->() const;`

E4: Forward Iterator

Cont'd…

- Experimental testing
  - ```
    for (
      auto it = array.begin();
      it != array.end();
      ++it
    ) { /* ... */ }
    ```
  - `for (auto&& item : array) { /* ... */ }`
    ```

E5: Constant Iterator

Extend the functionality of our iterator

- We want to **distinguish** `iterator` **and** `const_iterator`
  - Ideally without code repetition
- Refactor the current iterator class first
  - Declaration

```
template<bool Constant>
class iterator_base;
```

  - Definition

```
template<typename T>
template<bool Constant>
class Array<T>::iterator_base { /* ... */ };
```

- Update definitions of all the other existing methods

E5: Constant Iterator

Cont'd…

- Add the following **type aliases** into the flexible array class
  - `using iterator = iterator_base<false>;`
  - `using const_iterator = iterator_base<true>;`
- We will now have the following **access functions**
  - `iterator begin();`
  - `iterator end();`
  - `const_iterator begin() const;`
  - `const_iterator end() const;`
  - `const_iterator cbegin() const;`
  - `const_iterator cend() const;`

E5: Constant Iterator

Cont'd…

- Modify the **used types** in the base iterator class
  - In particular, aliases `value_type`, `pointer`, and `reference`
  - And also a pointer to the flexible array as such
- We will use type traits for this purpose
  - `std::conditional_t<bool B, class T, class F>`
  - Library `<type_traits>`
  - Unfolds to type name `T` or `F` based on the value of `B`
- Example of use

```
using array_pointer = std::conditional_t<
  Constant,
  const Array<Element>*, Array<Element>*
>;
```

E5: Constant Iterator

Cont'd…

- Finally, we also add the following **conversion operator**
  - So that we can change `iterator` to `const_iterator`
    - And really only in this direction
  - `operator iterator_base<true>() const;`
    - Base iterator member function
    - New instance of the specified target type is returned

E6: Iterator Extension

Extend the functionality of our iterator

- Extension to a **bidirectional iterator**
  - Tag `std::bidirectional_iterator_tag`
- Expected methods
  - `iterator_base& operator--();`
  - `iterator_base operator--(int);`

E6: Iterator Extension

Cont'd…

- Extension to a **random access iterator**
  - Tag `std::random_access_iterator_tag`
- Expected methods
  - ```
    iterator_base operator+(
      difference_type n
    ) const;
    ```
    - Analogously also `operator-`
  - ```
    difference_type operator-(
      const iterator_base& other
    ) const;
    ```
  - `iterator_base& operator+=(difference_type n);`
    - Analogously also `operator-=`

E6: Iterator Extension

Cont'd…

- Expected methods…
  - `reference operator[](difference_type n) const;`
  - ```
    bool operator<(
      const iterator_base& other
    ) const;
    ```
    - Analogously also `operator<=`, `operator>`, and `operator>=`
- Finally, one global function
  - ```
    friend iterator_base operator+(
      difference_type n,
      const iterator_base& it
    ) { /* ... */ }
    ```
    - Must be defined directly within the friend declaration

E7: Array Modifications

Finally, implement two methods that modify our array

- **Resizing the array**
  - `void resize(size_t size);`
    - Check concept `std::default_initializable<T>`
  - Both array shrinking and expanding is allowed
- **Erasing a range of elements**
  - ```
    iterator erase(
      const_iterator first, const_iterator last
    );
    ```
    - Check concept `std::assignable_from<T&, T&&>`
    - Check condition
      ```
      requires(T& t, T&& s) {
        { t = std::move(s) } noexcept;
      };
      ```
  - Return iterator to an element following the last erased one
      ```

NPRG041: Programming in C++

2025/26 Winter | Martin Svoboda | martin.svoboda@matfyz.cuni.cz

Class 12: **Movies IV**

Standard algorithms
Fake iterators
Lambda expressions
Ranges and views
Passing callable objects
Regular expressions
Raw string literals
Variant type
Doxygen documentation

E1: Storage Change

Change the **movie database storage** to our **flexible array**

- I.e., replace `std::vector` with `lib::Array`
  - At least if you dare to …

E2: Title Sorting

Implement the following **database query**

- **Q14**: titles where an actor `actor` played
  - ```
    database_t db_query_14(
      const database_t& database,
      const Actor& actor
    );
    ```
- Use of particular **standard algorithms** is expected
  - Library `<algorithm>`
- **Copy** all titles into the output container first
  - Method `resize(count);`
  - Function `std::copy(begin, end, target);`
    ```

E2: Title Sorting

Cont'd…

- **Remove** all non-matching title records
  - Function `std::remove_if(begin, end, predicate);`
  - Method `erase(begin, end);`
- Implement the **filtering predicate** using a functor
  - `class Predicate_Q14_Actor { … };`
  - Its parameter will be a specific actor `actor`
  - Add the round parentheses operator then
    - ```
      bool operator()(
        const std::shared_ptr<Title>& title_ptr
      ) const;
      ```
    - Return `true` if a given title is to be removed
      ```

E2: Title Sorting

Cont'd…

- Finally, **sort** the records of titles
  - Function `std::sort(begin, end, comparator);`
- Implement the **sort comparator** using a functor, too
  - `class Comparator_Q14_Years { /* ... */ };`
  - Add the round parentheses operator within it again
    - ```
      bool operator()(
        const std::shared_ptr<Title>& title_ptr_1,
        const std::shared_ptr<Title>& title_ptr_2
      ) const;
      ```
    - Return `true` if the first object precedes the second
    - I.e., simulate the behavior of a common < operator
  - Specifically, we want to sort the titles in descending order by years of filming and in ascending order by their names
      ```

E3: Years of Filming

Implement the following **database query**

- **Q15**: movies (not series) filmed in a year `year`
  - ```
    void db_query_15(
      const database_t& database,
      int year,
      std::ostream& stream = std::cout
    );
    ```
- Put suitable titles into an auxiliary container first
  - Choose our `database_t` type
  - **Initialize** it as an empty container
  - `std::copy_if(begin, end, target, predicate);`
  - Use a **fake iterator** for the target (library `<iterator>`)
    - `std::back_inserter(container);`
    - Creates an `std::back_insert_iterator` instance
    ```

E3: Years of Filming

Cont'd…

- Define the filtering predicate via a **lambda expression**
  - `[year](const std::shared_ptr<Title>& title_ptr) -> bool { /* ... */ }`
- **Sort** the titles in ascending order by their names
  - Use a lambda expression again
- Finally, **print** the titles into the provided stream
  - `std::transform(begin, end, target, action);`
  - Use a **fake iterator** for the target, terminate movies via `"\n"`
    - `std::ostream_iterator<std::string>(stream, delimiter);`
  - Use a lambda expression for the transformation action again
    - `{ name: "title", year: year }`

E4: Titles Aggregation

Implement the following **database queries**

- **Q16**: integer average rating of titles having a type `type` and a genre `genre`
  - ```
    std::optional<int> db_query_16(
      const database_t& database,
      Type type, std::string_view genre
    );
    ```
- Pass the calculated average via the return value
  - `std::for_each(begin, end, action);`
  - Implement everything using a custom **functor**
    - `class Visitor_Q16_Rating { /* ... */ };`
    - Return `std::nullopt` if there are no titles found
  - Function `for_each` creates a copy from the passed functor
    - This used instance is then returned via the return value
    ```

E4: Titles Aggregation

Cont'd…

- **Q17**: overall sum of the numbers of actors playing in titles with a rating of at least `rating`
  - ```
    size_t db_query_17(
      const database_t& database,
      int rating
    );
    ```
- Use `std::for_each` and a **lambda expression**
    ```

E5: Complex Query

Implement the following **database query**

- **Q18**: titles satisfying all the following conditions …
  - Genre of a given title is `genre`
  - Its name matches a regular expression `pattern`
  - Rating is equal to at least `rating`
  - Title has an above-average number of actors
  - All actors of a given title were born before a year `year`
  - ```
    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
    );
    ```
    ```

E5: Complex Query

Cont'd…

- **Average number of actors**
  - `std::accumulate(begin, end, initial, operation);`
    - For calculating the sum of numbers of actors over all titles
    - Library `<numeric>`
  - Calculate the average as a decimal number (`double`)
- **Years of birth of actors**
  - Function `std::all_of(begin, end, predicate);`
    - Simulates the behavior of the **universal quantifier**

E5: Complex Query

Cont'd…

- **Title name**
  - Class for a **regular expression** representation `std::regex`
    - Library `<regex>`
  - Constructor `std::regex(pattern, flags);`
    - **Case-insensitivity** flag – `std::regex_constants::icase`
  - Function `std::regex_match(string, regex);`
- **Raw string literals**
  - `R"delim(text)delim"`
    - Without standard escape sequences
    - Selectable delimiter (including an empty one)
  - E.g., `R"(\b\w*bobule\b)"`

E5: Complex Query

Cont'd…

- **Rating of titles**
  - We also allow **ratings in the form of stars**
    - There can be 0 to 5 stars, each worth 20 points
    - E.g., `***` = 60
  - We will use type `std::variant<...>`
    - Library `<variant>`
    - Specifically `std::variant<int, std::string>`
  - Useful functions
    - `std::holds_alternative<type>(variant);`
    - `std::get<type>(variant);`

E5: Complex Query

Cont'd…

- Use solely **lambda expressions**
  - For all conditions, actions, and operations
- Titles processing
  - Again use `std::for_each`
- Return a string with a JSON object for each matching title
  - `{ name: "title", year: year }`

E6: Actor Counts

Implement the following **database query**

- **Q19**: titles filmed between years [`begin`, `end`)
  - ```
    void db_query_19(
      const database_t& database,
      int begin, int end,
      std::ostream& stream = std::cout
    );
    ```
- **Find matching titles** and transform them first
  - Library <ranges>
  - Adapter `std::views::filter(predicate);`
  - Adapter `std::views::transform(action);`
    - Generate triples: title name, filming year, number of actors
    - `std::tuple<std::string, int, size_t>;`
  - Use **lambda expressions** for both filtering and transformation
    ```

E6: Actor Counts

Cont'd…

- Create the resulting **view** by chaining the `|` operator
  - Insert the found records into an auxiliary container
  - `std::vector<…> records(begin, end);`
- **Sort all records**
  - By years and title names, both in ascending order
  - Function `std::ranges::sort(range, comparator);`
  - Use a lambda expression again
- **Serialize and output records** to the provided stream
  - Use the `transform` view and a lambda expression
    - `{ name: "title", year: year, actors: actors }`
  - Function `std::ranges::copy(range, target);`
  - Use a **fake iterator** over the stream for the target again

E7: General Query

Implement the following **database query**

- **Q20**: titles satisfying a general search condition
  - ```
    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
    );
    ```
    ```

E7: General Query

Cont'd…

- **Find** titles satisfying a condition `selector`
  - ```
    bool operator()(
      const std::shared_ptr<Title>& title_ptr
    );
    ```
- **Sort** them using a comparison `comparator`
  - ```
    bool operator()(
      const std::shared_ptr<Title>& title_1,
      const std::shared_ptr<Title>& title_2
    );
    ```
- **Print** them serialized via a `serializer` to a given stream
  - ```
    std::string operator()(
      const std::shared_ptr<Title>& title_ptr
    );
    ```
    ```

E8: Doxygen Documentation

Get acquainted with the **Doxygen** documentation tool

- Download link
  - https://www.doxygen.nl/download.html
- Installation
  - Add path to the `bin` directory to the `PATH` system variable
- Generate a configuration file
  - `doxygen -g config.ini`
- Configure the following directives
  - `PROJECT_NAME = "..."`
  - `EXTRACT_PRIVATE = YES`
  - `EXTRACT_STATIC = YES`
  - …

E8: Doxygen Documentation

Learn how to document selected code fragments

- Files
  - `/// @file filename`
- Classes and template parameters
  - `/// …`
    `/// @tparam argname …`
- Class members
  - `/// …`
- Global and member functions
  - `/// …`
    `/// @param argname …`
    `/// @return …`
    `/// @exception typename …`

E8: Doxygen Documentation

Cont'd…

- Generate and browse the exported documentation
  - `doxygen config.ini`