http://www.ksi.mff.cuni.cz/~svoboda/courses/231-NPRG041/

Practical Class

NPRG041: Programming in C++

2023/24 Winter

Martin Svoboda
martin.svoboda@matfyz.cuni.cz

Charles University, Faculty of Mathematics and Physics

Class 1: **Introduction**

Function main
Standard output
Decomposition
Array

Required Tools

Visual Studio Community / Enterprise 2022

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

Gitlab

- https://gitlab.mff.cuni.cz/
  - .../teaching/nprg041/2023-24/svoboda-1540/

TortoiseGit

- https://tortoisegit.org/

Required Tools

Mattermost

- https://ulita.ms.mff.cuni.cz/mattermost/
  - .../ar2324zs/channels/nprg041-cpp-english

ReCodEx

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

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 local variable for now
  - `const char items[] = { 'A', 'B', 'C', 'D' };`
  - `const size_t count =`
    `sizeof(items) / sizeof(items[0]);`
- 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;`

Class 2: **Options I**

Header files
Program arguments
Strings std::string
Container std::vector
Type aliases
Passing parameters
Iteration
Named constants

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…

- Separate definitions from declarations in header files
  - `#ifndef`, `#define`, `#endif`
  - `#include "..."`
- Setting input arguments in VS
  - (Project) *Properties* $\rightarrow$ *Debugging* $\rightarrow$ *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>

Class 3: **Options II**

Parsing of numbers
Functions std::stoi and std::stof
Handling of exceptions
Structure struct

E1: Parsing of Numbers

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

- Recognition of integer / floating point numbers
  - `int stoi(const std::string& s, size_t* p)`
    - Library `<string>`
    - `std::invalid_argument`, `std::out_of_range`
  - `float stof(const std::string& s, size_t* p)`
- Expect the following behavior
  - Integers: `-r`, `--red`, `-g`, `--green`, `-b`, `--blue`
  - Floats: `-a`, `--alpha`
  - Strings: `-o`, `--output`
- Print suitable error messages to the standard output
  - Value <text> is not a valid integer number!
  - Value <text> is not a valid floating point number!

E2: Storing Options

Store detected options and their values in a suitable **structure**

- Define this structure in our header file
  - ```
    struct options_t {
      bool flag_x = false;
      bool flag_r = false; int value_r;
      ...
    }
    ```
- Store standalone values within a vector of strings
  - `values.push_back(…)`
- Print the stored information at the end of our program
  - Flag option <x> is <disabled>
  - Value option <r|red> is <enabled> and associated with value <255>
  - Standalone value <something>
    ```

Class 4: **Counter**

Streams std::istream, std::ostream
File streams std::ifstream, std::ofstream
Function std::getline
Classes with static methods
Throwing exceptions
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& is,`
    `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* letters);`
  - `void process(std::istream& is, size_t* letters);`
  - `void print(const std::string& filename, const size_t* letters);`
  - `void print(std::ostream& os, const size_t* letters);`
- Variable `letters` 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) { ... }`

E3: Extended Counter

Extend the previous code to **calculate selected statistics**

- Let us have the following assumptions about the text input
  - It 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 only digits `0` to `9` or dot `.`
- Use a class to detect and store these records across all inputs
  - Overall number of lines, sentences, words, and numbers
  - Overall number of letters, digits, spaces, symbols
  - Sum of all integer and separately decimal numbers
- Use the following functions
  - `int isdigit(int c); int isalpha(int c);`
- Enable printing of the calculated statistics again

Class 5: **Database I**

Streams std::stringstream
Function std::getline (with separator)
Class with data members
Constructors and initializers
Inline functions
Function std::move and rvalue references
Emplace mechanism
Container std::set

E1: Movie Representation

Propose a class for a movie database record representation

- Each movie has the following private data items
  - Name (`std::string`)
  - Filming year (`unsigned short`)
  - Genre (`std::string`)
  - Rating (`unsigned short`)
  - Set of actor names (`std::set<std::string>`)
- Implement the following functions first
  - Parameterized constructor
  - 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
  - We first create a container for movie instances
    - `std::vector<Movie> db;`
  - We then 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`
  - Mechanism `emplace_back`

E3: Importing Movies

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

- Assume class `Database` and its static member functions
  - `void import(const std::string& filename, std::vector<Movie>& db);`
  - `void import(std::istream& stream, std::vector<Movie>& db);`
- Use the following constructs to parse the CSV records
  - `std::istringstream` (library `<sstream>`)
  - `istream& std::getline(istream& stream, string& line, char delimiter);`
- Specifically, the following delimiters are assumed
  - Semicolon `;` for records and comma `,` for actors

E3: Importing Movies

Cont'd…

- Extreme situations will be treated using **structured exceptions**
  - `struct Exception { int code; std::string text; }`
- Code `1`
  - Unable to open input file <filename>
- Code `2` (fields `name`, `year`, `genre`, `rating`, and `actors`)
  - Missing field <name> on line <line>
  - Empty string in field <name> on line <line>
  - Invalid integer in field <name> on line <line>
  - Overflow integer in field <name> on line <line>
  - Malformed integer in field <name> on line <line>
  - Integer 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**

- **Q1**: all movies
  - `void db_query_1(const std::vector<Movie>& db, 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 std::vector<Movie>& db, std::ostream& stream = std::cout);`
  - Print names of the found movies only

Class 6: **Expressions I**

Classes with inheritance
Constructors and destructors
Virtual and pure virtual functions
Enumeration classes
Dynamic allocation (non-trivial life cycle)

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;`
    - Avoid usage 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, too

E1: Arithmetic Expressions

Cont'd…

- **Dynamic allocation** mechanism is assumed to be used
  - `Node* node_ptr = new NumberNode(2);`
  - `delete node_ptr;`
- Do not forget **virtual destructor**
  - `~Node();`
- Finally, 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;`

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
  - Only print absolutely necessary parentheses
    - 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)`

Class 7: **Expressions II**

Polymorphic container
Container std::stack
Shunting-yard algorithm
Hierarchy of exceptions

E1: Custom Exceptions

Propose your own hierarchy of classes for exceptions

- Common ancestor `Exception`
  - Constructors
    - `inline Exception(const std::string& message);`
    - `inline Exception(std::string&& message);`
  - Method `inline const std::string& what() const;`
- Derived classes
  - `EvaluationException`
  - `ParsingException`
  - `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

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()`

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
```

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

Non-standard situations will be handled using exceptions

- `ParsingException`
  - Unknown token (e.g., `a`, `3a`, …)
    - `Unknown token`
  - Unsuccessful number recognition including overflows
    - `Malformed number token`
  - 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; }
      ```
- Finally, add a new constructor to the `Expression` class
  - `Expression(const std::string& input);`
      ```

Class 8: **Database II**

Container std::set (custom class members)
Custom comparison operators
Custom stream insertion / extraction operators
Friend mechanism
Smart pointers std::shared_ptr
Dynamic casting

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 (`unsigned short`)
- 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 }`
- **Importing actors** will also be solved with our own operator >>
  - `std::istream& operator>>(std::istream& stream, Actor& actor);`
  - Individual data items are separated by spaces
    - `Ivan Trojan 1964`
  - Entirely empty actors will be skipped

E1: Structured Actors

Cont'd…

- **Actor import errors** will again be handled via exceptions
  - We will use conversion of the stream to a logical value
  - Final text messages will be constructed in two stages
- Code 2 (attributes `name`, `surname`, and `year`)
  - Missing attribute <name> in actor <actor> on line <line>
  - Missing, invalid or overflow value in attribute <year> in actor <actor> on line <line>
  - Integer out of range <min, max> in attribute <year> in actor <actor> on line <line>
    - In particular, interval $[1850, 2100]$ is assumed for the years
- Refactor the remaining parts of the current code as well
  - I.e., at least the 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>`)
    - `std::vector<std::shared_ptr<Title>> db;`
    - Function `std::make_shared<Title>(...);`
- Next, propose a new hierarchy of titles
  - Class `Title` will become abstract
  - Derived class `Movie` with an additional item
    - Length in minutes (`unsigned short`) with values $[0, 300]$
  - Derived class `Series` with additional items
    - Number of seasons (`unsigned short`) with values $[0, 100]$
    - Number of episodes (`unsigned short`) 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
  - 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 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 Conversion

Prepare the following two simple **database queries**

- **Q3**: series with at least `seasons` number of seasons or at least `episodes` number of episodes
  - ```
    void db_query_3(
      const std::vector<std::shared_ptr<Title>>& db,
      unsigned short seasons,
      unsigned short episodes,
      std::ostream& stream = std::cout
    );
    ```
  - Dynamic retyping of smart pointers
    - `(Series*)&*title_ptr;`
    - `dynamic_cast<Series*>(&*title_ptr);`
    - `std::dynamic_pointer_cast<Series>(title_ptr);`
  - Print whole JSON objects of the found series
    ```

E3: Type Conversion

Cont'd…

- **Q4**: names of titles with type `type` filmed in years [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,
      std::ostream& stream = std::cout
    );
    ```
  - Interpret the interval of years as open from the right
  - Title type is determined using the class type
    - I.e., not using our enumeration
    - `std::type_info` (library `<typeinfo>`)
    - `typeid(…);`
  - Print names of the found titles only
    ```

Class 9: **Database III**

Containers std::map and std::multimap
Structure std::pair and function std::make_pair
Container std::unordered_multimap
Structures std::less, std::hash, and std::equal_to
Functions std::copy, std::copy_if, std::remove_if, and std::erase
Functions std::sort and std::for_each
Functors
Lambda expressions

E1: Title Names

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

- Use an ordered map container
  - `std::map<std::string, std::shared_ptr<Title>>`
  - Library `<map>`
- Create this index using the following function
  - ```
    void db_index_titles(
      const std::vector<std::shared_ptr<Title>>& db,
      std::map<std::string,
        std::shared_ptr<Title>>& index
    );
    ```
- Inserting 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**

- **Q5**: title with name `name`
  - `void db_query_5(`
    `const std::map<std::string,`
    `std::shared_ptr<Title>>& index,`
    `const std::string& name,`
    `std::ostream& stream = std::cout`
    `);`
  - Finding the intended title
    - Function `index.find(name);`
  - Internal pair `std::pair` (items `first` and `second`)
  - Print the whole JSON object of the found title
    - `"name" -> { ... }`
    - Or `"name" -> Not found!` otherwise

E2: Filming Years

Create an index for **searching titles by years of filming**

- Use an ordered multimap container
  - `std::multimap<`
    `unsigned short,`
    `std::shared_ptr<Title>`
    `>`
  - Default functor for element ordering is assumed
    - `std::less<unsigned short>`
- Create this index using the following function
  - ```
    void db_index_years(
      const std::vector<std::shared_ptr<Title>>& db,
      std::multimap<unsigned short,
        std::shared_ptr<Title>>& index
    );
    ```
    ```

E2: Filming Years

Implement the following **database queries**

- **Q6**: titles filmed in year `year`
  - ```
    void db_query_6(
      const std::multimap<unsigned short,
        std::shared_ptr<Title>>& index,
      unsigned short 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 names of the found titles only
    - `year -> "name"` for each title
    - Or `year -> Not found!` otherwise
    ```

E2: Filming Years

Cont'd…

- **Q7**: titles filmed between years [`begin`, `end`)
  - ```
    void db_query_7(
      const std::multimap<unsigned short,
        std::shared_ptr<Title>>& index,
      unsigned short begin, unsigned short end,
      std::ostream& stream = std::cout
    );
    ```
  - Finding the intended titles
    - Iterator from: `index.lower_bound(begin);`
    - Iterator to: `index.lower_bound(end);`
  - Print names of the found titles only
    - `year -> "name"` for each title
    - Or `[begin, end) -> Not found!` otherwise
    ```

E3: Actors Cast

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

- Use an unordered multimap container
  - `std::unordered_multimap<`
    `Actor,`
    `std::shared_ptr<Title>`
    `>`
  - Library `<unordered_map>`
- Create this index using the following function
  - `void db_index_actors(`
    `const std::vector<std::shared_ptr<Title>>& db,`
    `std::unordered_multimap<Actor,`
    `std::shared_ptr<Title>>& index`
    `);`

E3: Actors Cast

Cont'd…

- Implement a **hash functor specialization**
  - `template<>`
    `struct std::hash<Actor> { ... }`
  - Function `size_t operator()(const Actor& actor) const noexcept;`
  - Use actor last name and `std::hash<std::string>{}(…);`
- And also a **comparison operator for actors**
  - Global function `bool operator==(const Actor& actor1, const Actor& actor2);`

E3: Actors Cast

Implement the following **database query**

- **Q8**: titles where actor `actor` played
  - ```
    std::vector<Title*> db_query_8(
      const std::unordered_multimap<Actor,
        std::shared_ptr<Title>>& index,
      const Actor& actor
    );
    ```
  - Finding the intended titles
    - `for (const auto& [key, value] : index) { ... };`
  - Put the found titles into the output container
    - In the form of C-style pointers, in particular
    ```

E4: Title Sorting

Implement the following **database query**

- **Q9**: titles where actor `actor` played
  - ```
    std::vector<std::shared_ptr<Title>> db_query_9(
      const std::vector<std::shared_ptr<Title>>& db,
      const Actor& actor
    );
    ```
- Use of selected standard algorithms is expected
  - Library `<algorithm>`
- Put the found titles into the output container
  - First, **copy** all the titles to the output container
    - Method `resize(count);`
    - Function `std::copy(begin, end, target);`
    ```

E4: Title Sorting

Cont'd…

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

E4: Title Sorting

Cont'd…

- Finally, **sort** the records of titles
  - Function `std::sort(begin, end, comparator);`
- Implement the **sorting comparator** using a functor, too
  - 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
      );
      ```
    - 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
      ```

E5: Title Genres

Implement the following **database query**

- **Q10**: titles having genre `genre`
  - ```
    std::vector<std::shared_ptr<Title>> db_query_10(
      const std::vector<std::shared_ptr<Title>>& db,
      const std::string& genre
    );
    ```
- Put the found titles into the output container
  - First, **initialize** it to the required size
  - Next, **copy** the appropriate title records
    - `std::copy_if(begin, end, target, predicate);`
  - Finally, **order** the titles in ascending order by their names
- Use **lambda expressions** in both cases
    ```

E6: Title Aggregation

Implement the following **database queries**

- **Q11**: integer average rating of titles having type `type` and genre `genre`
  - ```
    int db_query_11(
      const std::vector<std::shared_ptr<Title>>& db,
      Type type, const std::string& genre
    );
    ```
- Pass the calculated average using the return value
  - `std::for_each(begin, end, function);`
  - Implement everything using a custom functor
    ```

E6: Title Aggregation

Cont'd…

- **Q12**: number of titles having genre `genre`
  - ```
    int db_query_12(
      const std::vector<std::shared_ptr<Title>>& db,
      const std::string& genre
    );
    ```
- Pass the calculated number using the return value again
  - Use `std::for_each` and a lambda expression
    ```

Class 10: **Matrix**

Class and function templates
Inner classes
Container std::array
Custom arithmetic operators
Custom subscript operators
Conversion const_cast

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 to a given value
    - Use method `data_.fill(…);`
    ```

E1: Matrix Core

Cont'd…

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

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 coordinates directly to 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: 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

E4: 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
    );
    ```

Class 11: **Array I**

Custom container
Low-level dynamic allocation
Functions malloc and free
Placement new operator
Structure std::initializer_list
Standard exceptions

E1: Flexible Array

Implement a **custom flexible array container**

- Single template parameter
  - Item type `element`
- **Internal storage organization**
  - **First level**
    - Standard vector of C-style pointers to item blocks
  - **Second level** (one block)
    - 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 items in a block)
  - Internal storage as such
  - Current capacity and current number of items
- **Constructor**
  - `Array(size_t block_size = 10);`
    - Parameter is 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 size_t capacity() const;`
    - Returns the current internal storage capacity
  - `void print(std::ostream& stream = std::cout) const;`
    - Example: `[1, 2, 3, 4, 5]`
    - Each individual item is printed using its << operator
  - `std::ostream& operator<<(`
    `std::ostream& stream,`
    `const Array<element>& array`
    `);`

E2: Items Manipulation

Implement functions for **adding and removing items**

- **Internal block addition**
  - Determining required memory size
    - Operator `sizeof(type)`
  - Block dynamic allocation
    - Function `void* malloc(size_t size);`
    - Library `<cstdlib>`
    - Returns `nullptr` if not successful
  - **Ensuring atomicity** in case of failure
    - Throwing `std::bad_alloc` exception
    - Beware of the `push_back` operation failure at the first level
- **Internal block removal**
  - Block deallocation
    - Function `void free(void* ptr);`

E2: Items Manipulation

Cont'd…

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

E3: Initializer List

Finalize basic functionality of our flexible array

- **Destructor**
  - `~Array() noexcept;`
    - Removes all existing items
- Initializer list **constructor**
  - `Array(std::initializer_list<element> items);`
    - Library `<initializer_list>`
    - `for (auto&& item : items) { ... }`
  - **Ensuring atomicity** again

E4: Access Functions

Extend the functionality of our flexible array

- **Access functions**
  - `element& at(size_t index);`
  - `const element& at(size_t index) const;`
  - `element& operator[](size_t index);`
  - `const element& operator[](size_t index) const;`

E5: Debug Exceptions

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

- Activation using a macro
  - `#define __DEBUG__`
  - `#ifdef __DEBUG__`
  - `#endif`
- In particular, the following **standard exceptions** are assumed
  - Library `<exception>`
  - `std::out_of_range("Invalid index")`
    - For an invalid index in functions `at(…)` and `operator[](…)`
  - `std::invalid_argument("Empty array")`
    - When trying to remove an item from an empty array

Class 12: **Array II**

Copy and move constructors
Copy and move assignment operators
Custom iterators
Nested templates
Conversion operators
Custom namespace
Doxygen documentation

E1: Advanced Constructors

Extend the implementation of our flexible array

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

E1: Assignment Operators

Cont'd…

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

E2: Forward Iterator

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

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

E2: 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 = element;`
  - `using pointer = element*;`
  - `using reference = element&;`
  - `using difference_type = std::ptrdiff_t;`

E2: 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;`

E2: Forward Iterator

Cont'd…

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

E3: Constant Iterator

Extend the functionality of our iterator

- **The goal is to distinguish `iterator` and `const_iterator`**
  - Ideally without code repetition
- First, refactor the current iterator class
  - Declaration

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

  - Definition

```
template<typename element>
template<bool constant>
class Array<element>::iterator_base { ... };
```

- Update definitions of all the other existing methods

E3: 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;`

E3: 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 the following construct for this purpose
  `std::conditional_t<bool B, class T, class F>`
  - Library `<type_traits>`
  - Makes type name `T` or `F` available based on the value of `B`
- Example of use

```
using array_pointer = std::conditional_t<
  constant,
  const Array<element>*, Array<element>*
>;
```

E3: 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
    - Target type name is provided
    - Its new instance is returned
    - Return type as such is omitted

E4: 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);`

E4: 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, `operator-`
  - ```
    difference_type operator-(
      const iterator_base& other
    ) const;
    ```
  - `iterator_base& operator+=(difference_type n);`
  - Analogously, `operator-=`

E4: Iterator Extension

Cont'd…

- Expected methods…
  - `reference operator[](difference_type n) const;`
  - `bool operator<(`
    `const iterator_base& other`
    `) const;`
  - Analogously, `operator<=`, `operator>` a `operator>=`
- Finally, one global function
  - `iterator_base operator+(`
    `difference_type n,`
    `const iterator_base& it`
    `);`
    - Needs to be declared and defined using one flat template
    - `template<typename E, bool C>`

E5: Custom Namespace

Refactor the existing flexible array code

- Put the entire **implementation to namespace `lib`**
  - `namespace lib { ... };`

E6: 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`
  - …

E6: 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 …`

E6: Doxygen Documentation

Cont'd…

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