Workshop C++ 17, 20, 23

make it **clever**

Oxiane Institut

Workshop Structure and Program Outline

Educational Prerequisites:

- Basic proficiency in object-oriented programming and understanding of fundamental C++ concepts.

Technical Requirements:

- Laptop with development tools installed (C++17/C++20/C++23 compatible compiler, IDE).
- Internet access for real-time references to documentation and examples.

Program Objectives

1. Introduce and utilize new C++ features from versions 17 to 23.
2. Enhance code clarity and boost performance with modern C++ techniques.
3. Explore practical implementations of pattern matching, coroutines, and parallel algorithms.

Workshop 1: Diving into `if constexpr` and Handling Complex Types

Scenario: This workshop explores using `if constexpr` in C++ to adapt code behavior based on different data types. We'll also work with `std::variant` to handle multiple types within a single variable. The objective is to create specific structures to represent different temperature units (e.g., Celsius and Fahrenheit) without embedding conversion methods directly in these structures.

Step 1: Create a Function that Accepts Multiple Data Types and Displays Specific Messages

- **Objective**: Design a generic function that can accept multiple data types (such as integers, floats, and strings) and display a specific message based on the type detected.
- **Details**:
  - Use `if constexpr` to check the type at compile time and adjust the behavior accordingly.
  - For example, the function could display "Integer detected" for an `int`, "Float detected" for a `float`, and "String detected" for a `std::string`.
  - The goal is to keep the code flexible and readable.

Workshop 1: Diving into `if constexpr` and Handling Complex Types

Exercise 2: Use `std::variant` to Handle Different Data Types with Specific Structures

- **Objective**: Create separate structures to represent temperature units, such as Celsius and Fahrenheit, without including conversion methods in the structures. Use `std::variant` to encapsulate these structures and manage conversions with an external function.
- **Details**:
  - Define `Celsius` and `Fahrenheit` structures, each containing a temperature value but no conversion methods.
  - Create an external conversion function that accepts a `std::variant<Celsius, Fahrenheit>` and performs conversions based on the unit type.
  - Use `std::visit` with `if constexpr` to identify the type stored in the `std::variant` and apply the appropriate conversion.
- **Example Scenario**:
  - Given a `Celsius` structure with a temperature value, the external function could convert it to Fahrenheit.
  - Conversely, if a `Fahrenheit` structure is provided, the function could convert it to Celsius.

Workshop 2: Parallel Algorithm Optimization with `std::execution`

Objective: Apply parallel algorithms to boost performance with heavy data processing using the `std::execution` library.

- **Context Scenario:**
  You are working with large datasets and need to accelerate operations like sorting and transformations to improve performance.

Step 1: Parallel Processing with `std::for_each`

- **Objective:** Implement parallel processing with `std::for_each`.
- **Instructions:**
  1. Define a large array or vector of integers.
  2. Use `std::for_each` with `std::execution::par` to perform operations on each element (e.g., increment each element by a certain value).
  3. Measure and compare the time taken for parallel vs. sequential processing.

Workshop 2: Parallel Algorithm Optimization with `std::execution`

Step 2: Parallel Sorting Comparison

- **Objective:** Use parallel sorting to improve performance in heavy workloads.
- **Instructions:**
  1. Generate a large array with random integers.
  2. Sort the array sequentially using `std::sort` and record the time taken.
  3. Sort it again using `std::sort` with `std::execution::par` and record the time.
  4. Document and discuss the performance improvements observed with parallel execution.

Workshop 3: Asynchronous Programming with Coroutines

Objective: Utilize coroutines (co_await, co_yield) to improve asynchronous task management while keeping the program responsive.

Scenario: Your application needs to manage multiple network connections without blocking the main execution thread. You must integrate coroutines to ensure maximum responsiveness.

Step 1: Setting Up a Basic Coroutine

Objective: Understand the syntax and structure of a coroutine for asynchronous operations.

Instructions:

1. Create a coroutine function fetch_data using co_await to simulate waiting for data from a network connection.
2. Within fetch_data, simulate a delay with co_await (using a sleep operation or a mock asynchronous network call).
3. Display the fetched data in the main program once the coroutine completes.

Workshop 3: Asynchronous Programming with Coroutines

Step 2: Implementing co_yield for Progressive Data Fetching

Objective: Use co_yield to manage data retrieval incrementally, allowing for real-time processing and improved memory usage.

Instructions:

1. Modify fetch_data to use co_yield and return chunks of data incrementally.
2. Set up a loop in the main program to process each data chunk as it arrives.
3. Observe how co_yield allows the function to yield control back to the caller, freeing resources between data chunks.
4. Confirm that the main execution remains responsive while data is processed in smaller parts.

Workshop 3: Asynchronous Programming with Coroutines

Step 3: Creating a Coroutine to Handle Multiple Network Requests

Objective: Manage multiple asynchronous network connections using coroutines without blocking the main thread.

Instructions:

1. Implement a new coroutine manage_connections to simulate handling multiple network requests.
2. In manage_connections, call fetch_data in parallel for several simulated network sources (e.g., sensor data, user messages).
3. Use co_await to wait for each data source to complete, processing the results as they arrive.
4. Verify that the application can handle multiple connections without blocking or delaying the main program

Workshop 4: Pattern Matching and Error Handling with `std::expected`

Objective: Simplify handling of structured data with pattern matching and manage errors without exceptions, using `std::expected` to handle potential issues gracefully.

- **Context Scenario:**
  In a multi-format data processing project, you need to handle different input structures (JSON, XML, etc.) while ensuring proper error handling without using exceptions.

Step 1: Pattern Matching for Data Formats

- **Objective:** Apply pattern matching to handle diverse data formats effectively.
- **Instructions:**
  1. Define a function `process_data` that accepts different data formats (e.g., JSON, XML).
  2. Use `std::visit` to extract and print data according to the type.
  3. Implement specific cases for each format, such as parsing JSON keys and handling XML tags.
  4. Test with multiple formats to validate that each is correctly processed.

Workshop 4: Error Handling with `std::expected`

Step 2: Managing Errors with `std::expected`

- **Objective:** Handle errors gracefully without exceptions, ensuring robust data handling.
- **Instructions:**
  1. Define a function `load_file` using `std::expected` to manage success or failure states.
  2. If the file loads successfully, return its contents; otherwise, return an error message.
  3. Test the function with a missing file to observe how errors are handled without disrupting the program.
  4. Log successful and unsuccessful attempts, ensuring errors are traceable.

Workshop 5: Building a Real-Time Data Monitoring System

Objective: Create a comprehensive real-time data monitoring system that processes sensor data from various sources, uses asynchronous data handling, leverages parallel processing, and implements modern error-handling techniques.

- **Context Scenario:**
  You are tasked with developing a real-time data monitoring system for a smart factory. The system collects and processes data from various sensors (e.g., temperature, pressure, and operational status) attached to machines. The application must handle asynchronous data fetching, process large datasets efficiently, and ensure proper error handling to maintain robust operation.

Step 1: Multi-Type Data Handling with `std::variant`

- **Objective:** Handle different types of sensor data (e.g., integers for pressure, floats for temperature, and strings for status).
- **Instructions:**
  1. Define a `std::variant` type that can store multiple sensor data types.
  2. Create a function to process each type using pattern matching.
  3. Display the sensor data with appropriate formatting.

Workshop 5: Building a Real-Time Data Monitoring System

Step 2: Asynchronous Data Fetching with Coroutines

- **Objective:** Fetch data asynchronously from multiple sensors.
- **Instructions:**
  1. Create coroutines to receive data from each sensor asynchronously.
  2. Display each data point as it is received in real-time.
  3. Verify that the main thread remains responsive, allowing other operations to continue without interruption.

Step 3: Large Dataset Processing with Parallel Algorithms

- **Objective:** Use parallel algorithms to handle high data volumes and increase efficiency.
- **Instructions:**
  1. Collect and organize a large set of data from the sensors.
  2. Apply `std::for_each` with `std::execution::par` to process the data efficiently.
  3. Measure the performance improvements and discuss findings.

Workshop 5: Building a Real-Time Data Monitoring System

Step 4: Implementing Error Handling with `std::expected`

- **Objective:** Ensure robust error handling, allowing the system to function smoothly despite occasional data issues.
- **Instructions:**
  1. Use `std::expected` to handle errors such as invalid data formats or failed data retrieval.
  2. Log error messages when issues arise, ensuring operators are notified of errors without disrupting overall functionality.
  3. Validate that the system can continue processing other data when errors occur.