Workshop Program: C++17, C++20, and C++23 Updates

Educational Prerequisites:

- Having watched the video ‘New Features of C++17, C++20, and C++23.’.

Technical Prerequisites:

- A laptop with a development environment set up (C++17/C++20/C++23 compiler and code editor).
- Internet connection for accessing resources and documentation.

Objectives:

- Understand and utilize the new features introduced in recent versions of C++.
- Improve code readability and performance with modern C++ tools.
- Apply concepts such as pattern matching, coroutines, and parallel algorithms to practical use cases.

1. Welcome and Introduction

- Introduction to the objectives and structure of the workshop.
- Overview of participants’ individual goals and how they align with the workshop content.

2. Introduction to the New Features of C++17, C++20, and C++23

- Overview of the major new features and their impact on software development.
- Discussion on improvements in performance and code readability.

Practical Workshops:

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

Objective: Use `if constexpr` to write smarter, compilation-adaptive code while mastering complex types (`std::optional` , `std::variant` , etc.).

Scenario:

Imagine you're working on a simulation project that requires automatically adapting your code based on incoming data types (e.g., sensors sending integers, floats, or strings). Use `if constexpr` to handle these variations without complicating the code.

Exercises:

- **Exercise 1**: Create a function that accepts multiple data types (integers, floats, strings) and displays specific messages according to the type. The goal is to make the code flexible and clear.
- **Exercise 2**: Use `std::variant` to manipulate data of different types (e.g., temperature units like Celsius and Fahrenheit) and simplify type conversions using pattern matching.

Workshop 2: Optimizing Performance with Parallel Algorithms

Objective: Leverage parallel algorithms to enhance performance during heavy workloads, utilizing `std::execution`.

Scenario:

You are tasked with processing large amounts of data (e.g., intensive calculations for simulations or image transformations). You need to speed up these processes using parallel algorithms.

Exercises:

- **Exercise 1**: Use `std::for_each` with parallel execution to handle large datasets (e.g., mathematical calculations or image transformations).
- **Exercise 2**: Compare sequential and parallel execution on sorting algorithms (`std::sort`) to measure performance gains.

Workshop 3: Mastering Coroutines for Efficient Asynchronous Programming

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.

Exercises:

- **Exercise 1**: Create a coroutine to handle multiple non-blocking network requests and display results as soon as they are available, simulating asynchronous data flow management.
- **Exercise 2**: Implement a coroutine that generates data progressively (e.g., real-time statistics) and returns it using `co_yield` to avoid memory overload.

Workshop 4: Pattern Matching and `std::expected` in Action

Objective: Simplify the management of complex structures with pattern matching and improve error handling without exceptions using `std::expected`.

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.

Exercises:

- **Exercise 1**: Use pattern matching to handle data from various formats and extract relevant information based on the file type.
- **Exercise 2**: Create a function using `std::expected` to load files and return either a result in case of success or an error message in case of failure (e.g., corrupted or missing file).

Workshop 5: Introduction to Network API and Task Management with Executors

Objective: Discover the C++ network API and learn to efficiently manage tasks via executors (`std::execution::executor`).

Scenario:

Your application needs to handle multiple network connections in parallel and efficiently distribute tasks across multiple threads without overloading the system.

Exercises:

- **Exercise 1**: Implement a small network application that manages multiple connections using a thread pool via executors.
- **Exercise 2**: Create a task management system where complex calculations are distributed across multiple threads to maximize CPU resource utilization.

Workshop 6: Building a Real-Time Data Monitoring System

Objective:

Create a real-time data monitoring system that processes sensor data from multiple sources, manages asynchronous communication, uses parallel algorithms to handle large-scale data, and implements modern error-handling techniques.

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 attached to machines. These sensors provide different data types such as temperature (float), pressure (int), and operational status (string). The application needs to handle asynchronous data fetching, process large datasets efficiently using parallel algorithms, and ensure proper error handling for robust operation. The data should be displayed in real-time for factory operators to make timely decisions.

Exercises:

1. **Handling Multiple Data Types with `std::variant` and Pattern Matching**
   Task: Use `std::variant` to manage different types of sensor data (e.g., temperature, pressure, and status messages) and use pattern matching to process and display the data accordingly.

2. **Asynchronous Data Fetching with Coroutines**
   Task: Fetch data asynchronously from sensors using coroutines (`co_await`) while keeping the main thread responsive. Ensure that data from multiple sensors is fetched and processed without blocking the main execution flow.

3. **Processing Large Datasets with Parallel Algorithms**
   Task: Use parallel algorithms such as `std::for_each` to process large amounts of sensor data efficiently. Compare sequential and parallel processing to measure performance gains.

4. **Error Handling with `std::expected`**
   Task: Implement a robust error-handling mechanism using `std::expected` to manage potential issues such as failed sensor data retrieval or incorrect data formats.

5. **Combining Executors for Task Management**
   Task: Utilize executors to distribute sensor data processing across multiple threads. Ensure that all tasks are efficiently managed, and the system remains responsive under heavy load.

6. Demonstrations and Use Cases

- Presentation of projects integrating the new C++ features.
- Discussion on the benefits in terms of performance, code readability, and error management.

7. Q&A and Closing

- Q&A session to clarify technical points discussed.
- Conclusion and issuance of participation certificates.

Documentation and Resources:

- Official C++ documentation: cppreference.com
- Tutorials and guides on the new features.
- Code examples provided during the workshop.