

C++ Concurrency Practice Questions

Futures, Promises, Packaged Tasks, and std::async

LEVEL 1: BEGINNER (Understanding Basics)

std::async - Easy Questions

Q1.1 Write a function that uses `std::async` to calculate the square of a number in a separate thread.

```
cpp

// Expected signature:
std::future<int> async_square(int n);
```

Q1.2 Use `std::async` to run two functions in parallel: `add(3, 4)` and `multiply(5, 6)`. Print both results.

Q1.3 Create an async task that sleeps for 2 seconds then returns the string "Hello". Use both `std::launch::async` and `std::launch::deferred` and observe the difference.

Q1.4 Write code that starts 3 async tasks calculating factorials of 5, 6, and 7. Collect all results.

std::future - Easy Questions

Q1.5 Given a `std::future<int>`, write code that checks if the result is ready without blocking.

Q1.6 Create a future that will timeout after 1 second if the result isn't ready. Handle the timeout case.

Q1.7 Write a function that returns a future, then demonstrate calling `.get()` multiple times (and handle the error).

std::promise - Easy Questions

Q1.8 Create a promise/future pair where the promise sets the value 42 after 1 second delay.

Q1.9 Write code where a promise sets an exception instead of a value.

Q1.10 Create a promise in one thread and set its value from another thread.

LEVEL 2: INTERMEDIATE (Practical Usage)

std::packaged_task - Intermediate Questions

Q2.1 Create a `std::packaged_task` that wraps a lambda function, then execute it manually.

```
cpp

// Wrap this lambda: [](int x, int y) { return x + y; }
```

Q2.2 Store multiple `std::packaged_task` objects in a vector, then execute them one by one.

Q2.3 Create a `packaged_task`, get its future, move the task to a thread, and retrieve the result.

Q2.4 Implement a simple function that takes a `packaged_task` and executes it after a delay.

Multiple Approaches - Intermediate Questions

Q2.5 Implement the same task (computing sum of vector elements) using:

- a) `std::async`
- b) `std::packaged_task`
- c) Manual `std::promise`

Q2.6 Create a "result aggregator" that waits for 3 different async computations to complete and returns their sum.

Q2.7 Implement a timeout mechanism: if a computation doesn't complete within 3 seconds, return a default value.

Q2.8 Write a function that starts async tasks for processing each element of a vector in parallel.

shared_future - Intermediate Questions

Q2.9 Create a `shared_future` and have 3 different threads all wait for the same result.

Q2.10 Implement a "broadcast" mechanism where one thread sets a value and multiple worker threads receive it.

LEVEL 3: ADVANCED (Complex Patterns)

Error Handling and Edge Cases

Q3.1 Create a promise that might set either a value or an exception based on input validation.

Q3.2 Implement exception propagation through a chain of async operations.

Q3.3 Handle the "broken promise" scenario gracefully in your code.

Q3.4 Create a robust async operation that can handle timeouts, exceptions, and cancellation.

Producer-Consumer Patterns

Q3.5 Implement a producer-consumer pattern using promises where:

- Producer generates data over time
- Multiple consumers wait for data
- Use `shared_future` for broadcasting

Q3.6 Create a "pipeline" where output of one async operation becomes input to the next.

Q3.7 Implement a "fan-out, fan-in" pattern: distribute work to multiple threads, then collect results.

Advanced Task Management

Q3.8 Build a simple task scheduler that can:

- Queue packaged_tasks for later execution
- Execute tasks in priority order
- Return futures for results

Q3.9 Implement a "parallel reduce" operation using any combination of these primitives.

Q3.10 Create a system where tasks can depend on other tasks (dependency graph execution).

LEVEL 4: EXPERT (System Design)

Thread Pool Implementation

Q4.1 Build a basic thread pool using `std::packaged_task` that can:

- Accept any callable with any return type
- Return futures for results
- Handle exceptions properly

Q4.2 Extend your thread pool to support task priorities.

Q4.3 Add graceful shutdown to your thread pool (complete pending tasks, reject new ones).

Complex Synchronization Patterns

Q4.4 Implement a "barrier" using promises where N threads wait until all reach a checkpoint.

Q4.5 Create a "future-based semaphore" that limits concurrent access to a resource.

Q4.6 Build a "async cache" where multiple threads can request the same expensive computation, but it's only calculated once.

Performance and Scalability

Q4.7 Implement parallel quicksort (like in our earlier example) but make it:

- Configurable (max depth before switching to sequential)
- Memory efficient (avoid unnecessary copies)
- Exception safe

Q4.8 Create a "worker pool" that can dynamically scale up/down based on workload.

Q4.9 Implement a "map-reduce" framework using these primitives.

Q4.10 Build a system for computing large mathematical operations (like matrix multiplication) in parallel chunks.

BONUS CHALLENGES

Integration and Mixed Patterns

B1 Create a web-like request handler that can:

- Handle multiple requests concurrently
- Each request might spawn sub-requests
- Aggregate results from multiple async operations
- Handle timeouts and retries

B2 Implement a "async iterator" that yields values as they become available from parallel computations.

B3 Build a "reactive stream" where events trigger async operations and results flow to subscribers.

Solution Templates

For each question, try to implement:

```
cpp






#include <future>
#include <thread>
#include <chrono>
#include <iostream>

// Your solution here

int main() {
    // Test your implementation
    return 0;
}
```

Self-Check Guidelines

For each solution, verify:

-  **Correctness:** Does it produce the right result?
-  **Thread Safety:** No race conditions?
-  **Exception Safety:** Handles errors gracefully?
-  **Resource Management:** No leaks, proper cleanup?
-  **Performance:** Efficient use of resources?

Learning Path Recommendation

1. **Start with Level 1** - Master the basics
 2. **Do Level 2** - Build practical skills
 3. **Pick interesting Level 3** problems based on your needs
 4. **Attempt Level 4** only after solid understanding
 5. **Try Bonus** challenges for real-world complexity
-

Additional Resources to Explore

- **Books:** "C++ Concurrency in Action" by Anthony Williams
 - **Online:** cppreference.com sections on `<future>`
 - **Practice:** Implement variations of these patterns in your own projects
 - **Community:** Share solutions on Stack Overflow or Reddit r/cpp
-

Happy Coding! 🚀