C++ Concurrency Cheat Sheet

Threading Constructs

std::thread:

- Manual thread object
- Must join/detach manually
- No return values

std::async:

- Task-based API
- Returns std::future
- Automatically managed thread

std::jthread (C++20):

- Auto-joining thread (RAII)
- Supports std::stop_token
- No detach

Mutex Wrappers

std::lock_guard:

- Simple RAII lock
- No unlock() support
- Fast and safe

std::unique_lock:

- Flexible: lock/unlock manually
- Can use with timed_mutex
- Slightly more overhead

std::scoped_lock (C++17):

- Deadlock-safe multiple mutex lock
- Fixed scope (RAII)

std::shared_lock (C++17):

- For std::shared_mutex
- Multiple readers allowed
- Manual unlock/lock support

Common Tools

std::atomic<T>:

  - Lock-free thread-safe data

std::condition_variable:

  - For thread signaling/waiting
  - Wait with predicate to avoid spurious wakeups

std::future + std::promise:

  - Future: holds async result
  - Promise: sets value in another thread

Best Practices

- Always join/detach threads
- Use lock_guard or unique_lock (RAII)
- Avoid deadlocks by locking in consistent order
- Use condition_variable instead of busy loops
- Prefer atomic for simple counters/flags

✅ Threading Constructs Comparison

Feature	std::thread	std::async	std::jthread (C++20)
Type	Manual thread object	Task-based (launch + future)	Joining thread (RAII style)
Return values	❌ No (use std::promise)	✅ Yes (std::future)	❌ No (use std::promise if needed)
Auto join	❌ No (must join/detach)	✅ Yes (managed internally)	✅ Yes (joins in destructor)
Stop token	❌ No	❌ No	✅ Yes (std::stop_token)
Detach	✅ Yes	❌ Not applicable	❌ Not supported
Use case	Full control, low-level	Easy task/thread return, background work	Safe RAII-style threads with cancel

💡 When to Use

- **std::thread**: Low-level thread control; when return values not needed.
- **std::async**: Easy concurrency when return value or exception forwarding is required.
- **std::jthread**: Safer, modern alternative to std::thread with cancellation support.

🔐 Mutex Wrapper Comparison

Feature	std::lock_guard	std::unique_lock	std::scoped_lock (C++17)	std::shared_lock (C++17)
Lock control	❌ Fixed (locks on construct)	✅ Can lock/unlock manually	❌ Fixed	✅ Manual lock/unlock (read-only lock)
Multi-mutex locking	❌ No	❌ No	✅ Yes (deadlock-safe)	❌ No
Size/overhead	Small	Slightly more	Similar to lock_guard	Similar to unique_lock
Use with	std::mutex only	std::mutex, std::timed_mutex, etc.	std::mutex, multiple mutexes	std::shared_mutex only
Read-write support	❌ No	❌ No	❌ No	✅ Shared (reader) locking

💡 When to Use

- **std::lock_guard**: Fastest and safest for simple critical sections (RAII).
- **std::unique_lock**: Needed when you require unlock(), lock(), or timed locks.
- **std::scoped_lock**: Deadlock-safe locking of **multiple mutexes** at once.
- **std::shared_lock**: For **reader** threads in **read-write** lock scenarios using std::shared_mutex.

---

the Standard Library provides several types of **mutexes** under the <mutex> and <shared_mutex> headers. These are used for thread synchronization — to **protect shared data** from concurrent access.

🔐 Types of Mutexes in C++

Mutex Type	**Header**	**Description**
std::mutex	<mutex>	Basic, non-recursive mutex. Blocks other threads until unlocked.
std::recursive_mutex	<mutex>	Same thread can lock multiple times (must unlock same number of times).
std::timed_mutex	<mutex>	Supports try_lock_for() and try_lock_until() (time-based locking).
std::recursive_timed_mutex	<mutex>	Combines recursive + timed behavior.
std::shared_mutex	<shared_mutex>	Allows multiple readers OR one writer at a time (C++17).
std::shared_timed_mutex	<shared_mutex>	Like shared_mutex + timeout support (deprecated in C++20).

✅ Summary of Key Characteristics

Feature	**mutex**	**recursive_mutex**	**timed_mutex**	**shared_mutex**
Basic locking	✅	✅	✅	✅
Reentrant (same thread lock multiple times)	❌	✅	❌	❌
Timeout-based lock	❌	❌	✅	✅ (timed only)
Shared read locks	❌	❌	❌	✅
Exclusive write lock	✅	✅	✅	✅

📌 Example: std::shared_mutex Usage

```
#include <shared_mutex>
#include <thread>
#include <iostream>

std::shared_mutex smutex;
int shared_data = 0;

void reader() {
   std::shared_lock lock(smutex); // multiple allowed
   std::cout << "Read: " << shared_data << "\n";
}

void writer() {
   std::unique_lock lock(smutex); // exclusive lock
   shared_data += 1;
}
```

□ Best Practices

- Use std::mutex unless you have a specific reason for others.
- Use std::recursive_mutex **only if truly needed** — better to avoid reentrancy when possible.
- Use std::shared_mutex when you have **many readers, few writers** (like caching).
- Prefer **RAII wrappers** like std::lock_guard, std::unique_lock, or std::shared_lock.

✅ Code Snippets for Each

std::thread

```
std::thread t([] { std::cout << "Running\n"; });
t.join();
```

std::async

```
auto fut = std::async(std::launch::async, [] { return 42; });
int val = fut.get();
```

std::jthread (C++20)

```
std::jthread jt([](std::stop_token st) {
   while (!st.stop_requested()) {
      std::this_thread::sleep_for(1s);
   }
});
```

std::lock_guard

```
std::mutex m;
void safe() {
   std::lock_guard<std::mutex> lock(m);
}
```

std::unique_lock

cpp

CopyEdit

```
std::unique_lock<std::mutex> lock(m);
lock.unlock(); // delayed unlock
```

std::scoped_lock

```
std::mutex m1, m2;
std::scoped_lock lock(m1, m2); // No deadlock
```

std::shared_lock

```
std::shared_mutex sm;
std::shared_lock lock(sm); // multiple readers allowed
```

□ Summary

Feature	Best Choice
Simple thread	std::thread
Safe thread with return	std::async
Thread with RAII/cancel	std::jthread (C++20)
Simple lock	std::lock_guard
Lock with unlock()	std::unique_lock
Multiple mutexes	std::scoped_lock
Reader locks	std::shared_lock + std::shared_mutex

📦 std::promise<T>

Used to **set a value** or exception **from one thread**, which another thread can **retrieve using std::future<T>**.

✅ Key Functions

Function	Purpose
set_value(const T& val)	Sets the value to be retrieved by future
set_value(T&& val)	Sets the value (move version)
set_exception(std::exception_ptr)	Sets an exception instead of a value
get_future()	Returns the std::future<T> associated

🔧 Example

```
std::promise<int> p;
std::future<int> f = p.get_future();

std::thread t([&p] {
   p.set_value(42);
});
t.join();
int result = f.get(); // returns 42
```

🔮 std::future<T>

Used to **receive the result** from a std::promise or std::async.

✅ Key Functions

Function	Purpose
get()	Blocks and retrieves the value (or throws if exception set)
wait()	Blocks until result is ready
wait_for(duration)	Waits for a specific time
wait_until(time_point)	Waits until a specific time
valid()	Returns true if future has a shared state

🔧 Example with wait_for

```
if (f.wait_for(std::chrono::seconds(1)) == std::future_status::ready) {
  std::cout << f.get() << "\n";
} else {
  std::cout << "Timeout\n";
}
```

💣 Exception Propagation

You can transmit exceptions across threads:

```
std::promise<int> p;
std::future<int> f = p.get_future();

std::thread t([&p] {
  try {
    throw std::runtime_error("fail");
  } catch (...) {
    p.set_exception(std::current_exception());
  }
});
t.join();
```

```
try {
    int x = f.get();  // rethrows exception
} catch (const std::exception& e) {
    std::cerr << e.what(); // prints "fail"
}
```

□ Summary

Use Case	Use
Passing result between threads	std::promise + std::future
Getting async result	std::async → returns std::future
Timeout waiting	future.wait_for() / wait_until()
Exception forwarding	promise.set_exception() → future.get() throws