Mutex Type	Standard	Key Characteristics	Use Case
`std::mutex`	C++11	**Non-recursive**. Simplest and most performant. Attempting to lock an already-locked `std::mutex` from the same thread is **undefined behavior** (usually a deadlock).	General-purpose, non-recursive critical sections. The default choice.
`std::timed_mutex`	C++11	**Non-recursive** with **timed** locking. Provides `try_lock_for()` and `try_lock_until()` methods.	Situations where a thread shouldn't wait indefinitely for a lock.
`std::recursive_mutex`	C++11	Allows the **same thread** to acquire the lock multiple times (recursively). The lock is released only when the thread calls `unlock()` the same number of times it called `lock()`.	Recursive function calls or complex object internal locking where a thread might re-enter its own critical section.

std::recursive_timed_mutex	C++11	Combines the features of a recursive mutex with **timed** locking.	Recursive functions that also require timeout logic.

Mutex Type	Standard	Access Model	Key Characteristics
std::shared_mutex	C++17	**Reader-Writer Lock**. Provides separate lock_shared() (for readers) and lock() (for the single writer).	Best for data that is read far more often than it is written, improving concurrency.
std::shared_timed_mutex	C++14	**Timed Reader-Writer Lock**. Provides all features of std::shared_mutex plus **timed** shared and exclusive locking methods (try_lock_shared_for, try_lock_for, etc.).	Allows readers and writers to avoid indefinite blocking by attempting locks with a timeout.

Wrapper	C++ Standard	Locks	Movable	Manual Lock/Unlock	Deadlock Avoidance

std::lock_guard	C++11	Single	No	No	No
std::unique_lock	C++11	Single	Yes	Yes	Optional*
std::shared_lock	C++14	Single (Shared)	Yes	Yes	No
std::scoped_lock	C++17	One or more	No	No	Yes

Semaphore Type	Primary Purpose	Key Feature	Use Case
std::counting_semaphore	**Resource Counting**	The counter is initialized to $N > 1$. Threads acquire (decrement) and release (increment) the count. Threads block if the count is zero.	Limiting the number of concurrent connections to a database or pool of worker threads to, say, five at a time.

std::binary_semaphore	**Simple Signaling (Mutex Equivalent)**	A specialized $\text{std::counting\_semaphore}$ initialized to $N=1$. It behaves similarly to a std::mutex, but is primarily used for cross-thread signaling rather than protecting shared data.	Simple synchronization problems like producer-consumer where a resource is simply "available" or "unavailable."