WAYS TO CREATE THREADS IN C++11

1. Function Pointers
2. Lambda Functions
3. Functors
4. Member Functions
5. Static Member functions

Use Of join(), detach() and joinable() In Thread In C++ (C++11)

JOIN NOTES:

1. Double join will result into program termination.
2. If needed we should check thread is joinable before joining. (using joinable() function)

DETACH NOTES:

1. Always check before detaching a thread that it is joinable otherwise we may end up double detaching and
2. double detach() will result into program termination.

3. If we have detached thread and main function is returning then the detached thread execution is suspended.

NOTES:

Either join() or detach() should be called on thread object, otherwise during thread object's destructor it will terminate the program. Because inside destructor it checks if thread is still joinable? if yes then it terminates the program.

std::mutex::try_lock() On Mutex In C++11 Threading:

0. try_lock() Tries to lock the mutex. Returns immediately. On successful lock acquisition returns true otherwise returns false.

1. If try_lock() is not able to lock mutex, then it doesn't get blocked that's why it is called non-blocking.

2. If try_lock is called again by the same thread which owns the mutex, the behavior is undefined.

It is a dead lock situation with undefined behaviour. (if you want to be able to lock the same mutex by same thread more than one time the go for recursive_mutex)

std::try_lock() On Mutex:

0. std::try_lock() tries to lock all the mutexes passed in it one by one in given order.

1. On success this function returns -1 otherwise it will return 0-based mutex index number which it could not lock.

2. If it fails to lock any of the mutex then it will release all the mutex it locked before.

3. If a call to try_lock results in an exception, unlock is called for any locked objects before rethrowing.

Timed Mutex In C++ Threading (std::timed_mutex):

0. std::timed_mutex is blocked till timeout_time or the lock is acquired and returns true if success otherwise false.

1. Member Function:
   a. lock
   b. try_lock
   c. try_lock_for   ---\ These two functions makes it different from mutex.
   d. try_lock_until ---/
   e. unlock

try_lock_for():

Waits until specified timeout_duration has elapsed or the lock is acquired, whichever comes first.

On successful lock acquisition returns true, otherwise returns false.

try_lock_until():

Waits until specified timeout_time has been reached or the lock is acquired, whichever comes first.

On successful lock acquisition returns true, otherwise returns false.

Recursive Mutex In C++ (std::recursive_mutex):

0. It is same as mutex but, Same thread can lock one mutex multiple times using recursive_mutex.

1. If thread T1 first call lock/try_lock on recursive mutex m1, then m1 is locked by T1, now as T1 is running in recursion T1 can call lock/try_lock any number of times there is no issue.

2. But if T1 have aquired 10 times lock/try_lock on mutex m1 then thread T1 will have to unlock it 10 times otherwise no other thread will be able to lock mutex m1.
It means recursive_mutex keeps count how many times it was locked so that many times it should be unlocked.

3. How many time we can lock recursive_mutex is not defined but when that number reaches and if we were calling lock() it will return std::system_error OR if we were calling try_lock() then it will return false.

lock_guard In C++ (std::lock_guard<mutex> lock(m1)):

0. It is very light weight wrapper for owning mutex on scoped basis.

1. It aquires mutex lock the moment you create the object of lock_guard.

2. It automatically removes the lock while goes out of scope.

3. You can not explicitly unlock the lock_guard.

4. You can not copy lock_guard.

unique_lock In C++ (std::unique_lock<mutex> lock(m1)):

1. The class unique_lock is a mutex ownership wrapper.

2. It Allows:
    a. Can Have Different Locking Strategies
    b. time-constrained attempts at locking (try_lock_for, try_lock_until)
    c. recursive locking
    d. transfer of lock ownership (move not copy)

e. condition variables.

Locking Strategies:

TYPE	EFFECTS(S)
1. defer_lock	do not acquire ownership of the mutex.
2. try_to_lock	try to acquire ownership of the mutex without blocking.
3. adopt_lock	assume the calling thread already has ownership of the mutex.

Unique_lock vs lock_guard():

VS Comparison Table

Feature	`std::lock_guard`	`std::unique_lock`
Locks on construction	✓ Always	✓ (unless deferred)
Unlocks on destruction	✓ Always	✓ Always
Can unlock early	✗ No	✓ Yes
Can re-lock	✗ No	✓ Yes
Can defer locking	✗ No	✓ Yes (`std::defer_lock`)
Needed for `condition_variable`	✗ No	✓ Yes
Movable	✗ No	✓ Yes
Overhead	Very low	Slightly higher

`std::lock_guard`

A **simple, lightweight RAII lock** that locks a mutex on construction and unlocks it on destruction.

✔ Key characteristics

- **Always locks immediately** when created
- **Always unlocks automatically** when destroyed
- **Cannot be unlocked early**
- **Cannot be moved or transferred**
- Very small & efficient

`std::unique_lock`

A **more flexible RAII lock** that provides advanced functionality.

✔ Key features (over `lock_guard`)

- Can **lock later** (`defer_lock`)
- Can **unlock manually** **if you want to otherwise it will be automatically unlocked.**
- Can **re-lock again**
- Supports **`std::condition_variable`**
- Movable (lock ownership can be transferred)
- Slightly more overhead than `lock_guard.`

`std::thread` copies all the arguments internally:

It stores them somewhere so it can call the function later in the new thread.

This means:

- It tries to **copy `x`**
- Then pass that copy to `foo`
- That fails because `foo` expects `int&`, a reference, not a temporary copy

Normal function calls bind references directly.

`std::thread` stores arguments, so it must copy them → references cannot be copied → we use `std::ref` to wrap them.

A `std::condition_variable` requires a `std::unique_lock<std::mutex>` because:

1. **It must be able to unlock the mutex when waiting**
2. **It must be able to re-lock the mutex when waking**
3. **Only `std::unique_lock` provides unlock/relock control**

`std::lock_guard` cannot do this cv.wait(lock);

Key rule to remember

`cv.wait(lock, predicate)` means:

Wait while `predicate` is FALSE
Continue when `predicate` becomes TRUE

```
while (!predicate()) {
    cv.wait(lock);
}
```

What happens when you call cv.wait(lck)?

In C++ (std::condition_variable), this call does three atomic steps:

1.)Releases the mutex associated with lck

2.)Puts the current thread to sleep

3.)Re-acquires the mutex before returning from wait()

So during the time the thread is waiting:

→ The mutex is NOT locked by that thread

What happens inside wait()?

◈ Step 1: WAIT begins → mutex must be unlocked

Because you want other threads to modify shared data while you're waiting.

◈ Step 2: The thread goes to sleep (blocked)

◈ Step 3: When notified → mutex must be locked again

So you can safely check your condition.

This requires the ability to:

unlock the mutex

lock it again later

1. Condition variables allow us to synchronize threads via notifications.

   a. notify_one();

   b. notify_all();

2. You need mutex to use condition variable

3. Condition variable is used to synchronize two or more threads.

4. Best use case of condition variable is Producer/Consumer problem.

5. Condition variables can be used for two purposes:

   a. Notify other threads

   b. Wait for some condition

std::lock() In C++11

It is used to lock multiple mutex at the same time.

IMPORTANT:

syntax---> std::lock(m1, m2, m3, m4);

1. All arguments are locked via a sequence of calls to lock(), try_lock(), or unlock() on each argument.

2. Order of locking is not defined (it will try to lock provided mutex in any order and ensure that there is no dead lock).

3. It is a blocking call.

[Example:0] --> No deadlock.

Thread 1	Thread 2
std::lock(m1,m2);	std::lock(m1,m2);

[Example:1] --> No deadlock.

Thread 1	Thread 2
std::lock(m1, m2);	std::lock(m2, m1);

[Example:2] --> No deadlock.

Thread 1	Thread 2
std::lock(m1, m2, m3, m4);	std::lock(m3, m4);
	std::lock(m1, m2);

// [Example:3] --> Yes, the below can deadlock.

Thread 1	Thread 2
std::lock(m1,m2);	std::lock(m3,m4);
std::lock(m3,m4);	std::lock(m1,m2);

std::future(consumes) and std::promise(produces) in threading.

1. std::promise

    a. Used to set values or exceptions.

2. std::future

    a. Used to get values from promise.

    b. Ask promise if the value is available.

    c. Wait for the promise.

```
std::promise<int> p;

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

You pass the future to a thread needing the result and keep the promise in the thread producing the result.

Std::async:

std::async launches a function **asynchronously** (usually in a new thread or deferred) and returns a **std::future** that represents the result of that function.

1. It runs a function asynchronously (potentially in a new thread) and returns a std::future that will hold the result.
2. There are three launch policies for creating task:
    a. std::launch::async
    b. std::launch::deffered
    c. std::launch::async | std::launch::deffered

HOW IT WORKS:
1. It automatically creates a thread (Or picks from internal thread pool) and a promise object for us.

2. Then passes the std::promise object to thread function and returns the associated std::future object.

9️⃣ Thread Pool vs `std::async`

Aspect	`std::async`	Thread Pool
Thread creation	May create new threads	Reuses threads
Control	Limited	Full
Overhead	Higher	Lower
Production systems	Rare	Preferred

3. When our passed argument function exits then its value will be set

Static Variable Is Thread Safe OR Not:

⭐ Summary Table

Use Case	Thread-Safe?	Why
Static variable initialization	✔ Yes	Guaranteed by C++11
Static variable read-only	✔ Yes	No data race
Static variable read/write from multiple threads	❌ No	Data race
Static variable protected by mutex	✔ Yes	Mutual exclusion
Static variable is atomic	✔ Yes	No data race

A static variable is *NOT* automatically thread-safe.
↪ It is only thread-safe under specific conditions.

Static variables are NOT automatically thread-safe.
They must be protected using:

- mutex

- atomic
- or read-only access

Otherwise, you get unpredictable results.

Std::scoped_lock:

It is lock guard that can lock **one or multiple mutexes at once**, safely and without risking **deadlock.** It replaces `std::lock_guard` when you need to lock more than one mutex, or simply want a more flexible/modern lock wrapper

✔ What `std::scoped_lock` Does

- Locks one or more mutexes *in a deadlock-free manner* (using `std::lock()` **internally**).
- Unlocks them automatically when it goes out of scope.
- Cannot be copied (like most lock guards).
- Very lightweight and efficient.

Oversubscription in multithreading:

It occurs when a program creates **more runnable threads than the number of available CPU hardware threads (cores × SMT/Hyper-Threading)**.

In simpler terms:

> **Too many threads chasing too few CPU cores.**

✔ How to Avoid Oversubscription

Strategy	Explanation
Use a thread pool	**e.g., `std::jthread`, `std::async`, or custom pool.**
Match threads ≈ hardware concurrency	`std::thread::hardware_concurrency()`
Use task-based parallelism	**Use libraries like TBB, OpenMP, std::execution.**
Avoid thread-per-task design	**Threads are heavy. Prefer tasks/futures.**

AtomicVariable:

An **atomic variable** is a special type of variable that supports **atomic (indivisible) operations**, meaning each operation on it happens **as a single, uninterruptible step** — no other thread can observe it halfway done.

In C++, atomic variables are provided by `std::atomic<T>`.

An atomic variable guarantees that read, write, and update operations happen completely or not at all, preventing data races without needing a mutex.

Std::jthread:

`std::jthread` is a **modern thread type introduced in C++20** that improves on `std::thread` by:

- **Automatically joining** when it goes out of scope
- Supporting **cooperative cancellation** via `std::stop_token`

std::jthread::request_stop:

`It simply` **`signals`** `the thread that it` *`should stop`*`, and the thread must` **`cooperate`** `by checking a stop condition.`

🔷 Basic Idea

- `request_stop() sets a stop flag`
- `The thread receives a std::stop_token`
- `The thread checks stop_requested()`
- `The thread exits cleanly`

Concurrency vs Parallelism:

Concurrency vs Parallelism (SDE-2 Master Explanation)

Concept	Concurrency	Parallelism
Meaning	**Dealing with many things at once**	**Doing many things at once**
Core Idea	Structure of the program	Actual execution on hardware
Concern	Correctness, coordination, safety	Performance, speedup
Happens on	Single core **or** multiple cores	**Requires multiple cores**
Achieved by	Context switching, async, multitasking	Multiple CPUs/cores
Controlled by	OS Scheduler + runtime	Hardware + OS