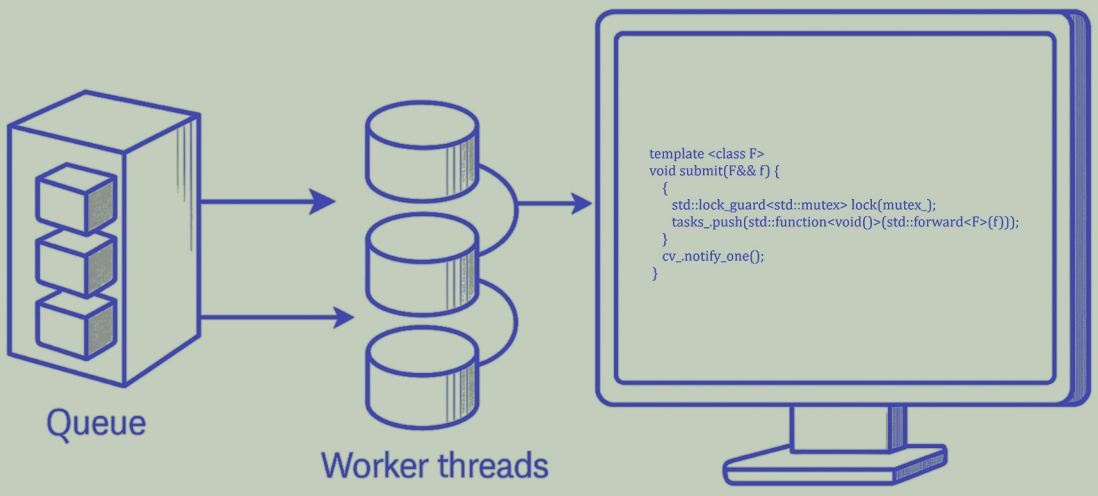
https://simplifycpp.org

Thread Pools

Writing a Simple Thread Pool in Modern C++



Prepared By Ayman Alheraki

Thread Pools

Writing a Simple Thread Pool in Modern C++

Prepared by Ayman Alheraki

simplifycpp.org

November 2025

Contents

Author's Introduction

Modern processors are no longer about a single fast core; instead, they are built around multiple cores, hardware threads, deep caches, and complex memory hierarchies. If our programs use only one core, we are wasting most of the available performance.
For C++ programmers, this raises an immediate question: how do we scale our programs to fully use modern CPUs without drowning in low-level synchronization bugs and complex thread management?
Since C++11, the language has provided a standard threading model: `std::thread`, `std::mutex`, `std::condition_variable`, futures, atomic types, and more.
Later standards (C++17, C++20, C++23) added tools such as `std::shared_mutex`, `std::scoped_lock`, `std::jthread`, and `std::stop_token`, making concurrency more expressive and safer.
However, if you start many short-lived `std::thread` objects directly, you will quickly discover that thread creation and destruction are expensive. Managing dozens or hundreds of threads is inefficient and error-prone. This is where *thread pools* become essential.
A thread pool is a reusable set of worker threads that continuously take tasks from a queue and execute them. Instead of creating a new thread for each small task, we submit tasks to the pool, which schedules them onto a fixed number of threads. This design:

- reduces overhead from thread creation and destruction;
- limits the number of concurrent threads and therefore contention;

- provides a central place to manage shutdown, exceptions, and monitoring;
- makes it easier to build higher-level parallel patterns.

In this mini-booklet, we will:

- build up the concepts of thread pools from first principles;
- use only Modern C++ (C++11 and later) facilities, without platform-specific APIs;
- design and implement a reusable simple thread pool that fits into real projects;
- discuss correctness, exception safety, shutdown protocols, and performance;
- connect thread pools to futures, tasks, and typical parallel patterns;
- point to ongoing standardization efforts (executors, senders/receivers) so you can align your own designs with the direction of ISO C++.

Each chapter ends with exercises followed by solutions. The goal is not only to show a working implementation, but also to train your reasoning about safety, race conditions, and clean Modern C++ design. By the end, you should feel confident writing, adapting, and reviewing thread pool code for production systems.

Chapter 1

Motivation and Fundamentals of Thread Pools

1.1 Why Not Just Use std::thread Everywhere?

The simplest way to run something in parallel in Modern C++ is:

```cpp
#include <thread>

void do_work(int id) {
    // ... some CPU-bound or I/O-bound work ...
}

int main() {
    std::thread t1(do_work, 1);
    std::thread t2(do_work, 2);

    t1.join();
```

```
    t2.join();
}
```

This approach is fine for a handful of threads, but it does not scale when:

- You have thousands of small tasks.
- Tasks arrive dynamically while the program runs.
- You need to limit concurrency to (for example) the number of hardware cores.

Creating and destroying a thread is relatively expensive:

- It requires kernel involvement (creating kernel thread structures).
- It may involve stack allocation, scheduler registration, and OS bookkeeping.
- Exiting a thread triggers cleanup and scheduling overhead again.

If each task is short-lived (milliseconds or microseconds), the overhead of creating a thread per task becomes dominating. Moreover, if you create too many threads, you increase context switching and cache thrashing, harming performance.

1.2 The Thread Pool Idea

A thread pool reuses a fixed number of worker threads:

- At startup, you create N worker threads, where N is typically equal to or slightly larger than the number of hardware cores (for CPU-bound work).
- You maintain a concurrent queue of tasks waiting to be executed.
- When a task is submitted, it is pushed into the queue.

- Each worker waits for tasks; when one arrives, it pops the task and runs it.
- At shutdown, you signal workers to stop, then join them.

This changes the cost structure:

- Thread creation cost is paid only once at startup.
- Per-task overhead is reduced to queue push/pop and a few synchronizations.
- Concurrency is bounded by the number of worker threads in the pool.

1.3 Basic Design Goals

Even for a "simple" thread pool, we want:

1) **Safety**: No data races, no undefined behavior, clean shutdown.
2) **Exception safety**: Exceptions in tasks should not terminate the process unexpectedly.
3) **RAII management**: Resources (threads, queues) are owned and released deterministically.
4) **Clear API**: Submitting work should be easy and natural.
5) **Modern C++ style**: Use `std::thread`, `std::mutex`, `std::condition_variable`, `std::future`, and (optionally) newer features such as `std::jthread` or `std::stop_token`.

1.4 Tasks as Callables

The natural abstraction for "work" in C++ is a *callable*: a function, lambda, functor, or `std::function`. A thread pool typically accepts a generic callable and optional arguments. A common API sketch is:

```
class thread_pool {
public:
    explicit thread_pool(std::size_t thread_count);

    template <class F, class... Args>
    auto submit(F&& f, Args&&... args)
        -> std::future<std::invoke_result_t<F, Args...>>;

    ~thread_pool();

    // Non-copyable, movable or not movable depending on design
};
```

The `submit` member returns a `std::future<R>` so the caller can retrieve the result (or exception) produced by the task.

1.5 Exercises

1) Explain in your own words why creating a `std::thread` per task is usually a bad idea for many small tasks.

2) Describe the main responsibilities of a thread pool in a multithreaded application.

3) Suppose you have a CPU-bound task that takes 1 ms. Roughly estimate why launching a new thread per task can be inefficient.

4) In what scenarios could creating a thread per task still be acceptable?

Solutions to Exercises (Chapter 1)

1) Thread creation/destruction has non-trivial overhead in the operating system. For many small tasks, the time spent creating and destroying threads can exceed the actual work time. It also increases context switching and memory usage.

2) A thread pool manages a fixed set of worker threads, maintains a queue of tasks, schedules tasks onto workers, and provides mechanisms for shutdown and possibly result handling and error propagation.

3) If a task is only 1 ms but thread creation takes, for example, 0.1–0.5 ms (or more depending on the system), then a significant fraction of total time is spent in thread management. For very small tasks, overhead can completely dominate.

4) For long-running tasks or for applications with only a few tasks, thread-per-task may be acceptable and simpler to implement. It is also sometimes used in quick prototypes or scripts where simplicity matters more than efficiency.

Chapter 2

Modern C++ Concurrency Building Blocks

2.1 Threads and Joinability

The core abstraction is `std::thread`:

```cpp
#include <thread>
#include <iostream>

void worker() {
    std::cout << "Hello from worker thread\n";
}

int main() {
    std::thread t(worker);
    if (t.joinable()) {
        t.join();
```

```
    }
}
```

2.1.1 `std::jthread` and Stop Tokens (C++20)

C++20 introduced `std::jthread`, which automatically joins in its destructor and can be cooperatively stopped via `std::stop_token`:

```cpp
#include <thread>
#include <iostream>
#include <chrono>

void worker(std::stop_token st) {
    while (!st.stop_requested()) {
        // do some work
        std::this_thread::sleep_for(std::chrono::milliseconds(10));
    }
    std::cout << "worker stopped\n";
}

int main() {
    std::jthread t(worker);              // automatically joins
    std::this_thread::sleep_for(std::chrono::seconds(1));
    t.request_stop();                    // cooperative cancellation
}
```

For a basic thread pool we can use `std::thread`. With C++20 and later, we can refine the design using `std::jthread` and stop tokens to improve cancellation and shutdown.

2.2 Mutexes, Locks, and Condition Variables

We need safe synchronization for our task queue. The core tools are:

- `std::mutex` – a basic mutual exclusion lock.
- `std::unique_lock` – RAII wrapper, flexible locking.
- `std::lock_guard` – RAII wrapper, simpler but less flexible.
- `std::condition_variable` – wait/notify mechanism for threads.

A typical pattern for a producer-consumer queue:

```
#include <mutex>
#include <condition_variable>
#include <queue>
#include <optional>

template <class T>
class blocking_queue {
public:
    void push(T value) {
        {
            std::lock_guard<std::mutex> lock(mutex_);
            queue_.push(std::move(value));
        }
        cv_.notify_one();
    }

    T pop() {
```

```
        std::unique_lock<std::mutex> lock(mutex_);
        cv_.wait(lock, [this] { return !queue_.empty() || stopped_;
        ↪  });
        if (stopped_ && queue_.empty()) {
            throw std::runtime_error("queue stopped");
        }
        T value = std::move(queue_.front());
        queue_.pop();
        return value;
    }

    void stop() {
        {
            std::lock_guard<std::mutex> lock(mutex_);
            stopped_ = true;
        }
        cv_.notify_all();
    }

private:
    std::mutex mutex_;
    std::condition_variable cv_;
    std::queue<T> queue_;
    bool stopped_{false};
};
```

This pattern is the heart of a thread pool: workers repeatedly call `pop()` to get a task.

2.3 Futures, Promises, and Packaged Tasks

To return values from tasks we will use `std::future` and `std::packaged_task`.

- `std::packaged_task<F>` wraps a callable `F` and connects it to a `std::future`.
- When the task runs, it stores the result (or exception) into the shared state.
- The caller holds the `std::future` and calls `get()` to retrieve the result.

Example:

```
#include <future>
#include <iostream>

int compute_value(int x) {
    if (x < 0) throw std::runtime_error("negative!");
    return x * 2;
}

int main() {
    std::packaged_task<int(int)> task(compute_value);
    std::future<int> f = task.get_future();

    // run packaged_task synchronously for demo
    task(21);

    try {
        std::cout << "result = " << f.get() << "\n";
    } catch (const std::exception& e) {
        std::cerr << "error: " << e.what() << "\n";
```

```
20      }
21  }
```

In our thread pool, we will place `std::packaged_task` objects into the queue so workers can execute them and callers can wait on the associated `std::future`.

2.4 Exercises

1) What is the difference between `std::lock_guard` and `std::unique_lock`? When would you prefer each?

2) Explain how `std::condition_variable::wait` avoids busy waiting.

3) Why is `std::packaged_task` useful in a thread pool implementation?

4) How does `std::jthread` improve safety compared to `std::thread`?

Solutions to Exercises (Chapter 2)

1) `std::lock_guard` is a simple RAII wrapper that locks the mutex on construction and unlocks it on destruction; it cannot be unlocked manually. `std::unique_lock` is more flexible; you can lock/unlock explicitly, move it, and use it with `std::condition_variable`. Use `lock_guard` for simple scopes; use `unique_lock` when you need this extra flexibility.

2) `wait` atomically releases the mutex and suspends the thread until a notification occurs and the predicate becomes true. The thread does not continuously spin; instead, it sleeps and is woken up by `notify_one` or `notify_all`.

3) `std::packaged_task` connects a callable to a `std::future`. This allows us to execute work in the thread pool and still give the caller a future to retrieve the result (or exception) once the task completes.

4) `std::jthread` automatically joins in its destructor, which prevents accidental `std::terminate()` caused by a destructed joinable thread. It also supports cooperative stop requests via `std::stop_token`, which is very convenient for clean shutdown and cancellation.

Chapter 3

Designing a Simple Thread Pool Interface

3.1 High-Level API

We aim for a minimal but powerful interface:

```
#include <vector>
#include <thread>
#include <future>
#include <queue>
#include <mutex>
#include <condition_variable>
#include <functional>
#include <atomic>

class thread_pool {
public:
    explicit thread_pool(std::size_t thread_count =
        std::thread::hardware_concurrency());
```

```
14
15      template <class F, class... Args>
16      auto submit(F&& f, Args&&... args)
17          -> std::future<std::invoke_result_t<F, Args...>>;
18
19      ~thread_pool();
20
21      thread_pool(const thread_pool&) = delete;
22      thread_pool& operator=(const thread_pool&) = delete;
23
24  private:
25      void worker_loop();
26
27      std::vector<std::thread> workers_;
28      std::queue<std::function<void()>> tasks_;
29
30      std::mutex mutex_;
31      std::condition_variable cv_;
32      bool stop_{false};
33  };
```

Key points:

- The constructor accepts a thread count (defaulting to hardware concurrency).
- `submit` is a function template exposing a generic task API.
- We store tasks as `std::function<void()>` inside the queue.
- We use a `bool stop_` flag to signal workers when to exit.

3.2 Choosing the Thread Count

`std::thread::hardware_concurrency()` returns a hint about how many hardware threads are available. For CPU-bound workloads, using that number (or slightly less) is usually a reasonable default. For I/O-bound workloads, sometimes a larger number of threads makes sense, but it can still be beneficial to keep it bounded.

3.3 Task Representation

We will adapt user-provided callables into `std::packaged_task<R()>` and then move them into our queue via a generic `std::function<void()>` wrapper.
Sketch:

```
template <class F, class... Args>
auto thread_pool::submit(F&& f, Args&&... args)
    -> std::future<std::invoke_result_t<F, Args...>>
{
    using result_type = std::invoke_result_t<F, Args...>;

    auto bound_task = std::bind(std::forward<F>(f),
                                std::forward<Args>(args)...);

    auto packaged =

        ↪  std::make_shared<std::packaged_task<result_type()>>(std::move(b

    std::future<result_type> result = packaged->get_future();

    {
```

```
        std::lock_guard<std::mutex> lock(mutex_);
        if (stop_) {
            throw std::runtime_error("thread_pool is stopping");
        }
        tasks_.emplace([packaged]() { (*packaged)(); });
    }

    cv_.notify_one();
    return result;
}
```

We will refine and fully implement this in the next chapter.

3.4 Shutdown Strategy

A simple and reliable shutdown strategy for our basic pool:

1) In the destructor, acquire the mutex and set `stop_ = true`.

2) Call `cv_.notify_all()` to wake all workers.

3) Join all worker threads.

4) Workers check `stop_` and exit their loops once no more tasks are available.

This ensures we join all threads before destroying the pool, avoiding undefined behavior.

3.5 Exercises

1) Why should a thread pool typically be non-copyable?

2) What happens if a task submitted to the pool throws an exception and you do not use futures?

3) Why do we need a condition variable to implement the worker loop efficiently?

4) In what situations would you want to provide an explicit thread count instead of using `std::thread::hardware_concurrency()`?

Solutions to Exercises (Chapter 3)

1) A thread pool manages threads and a shared queue. Copying it would require duplicating threads, queues, and ownership of resources, which is complex and usually not well-defined. Non-copyability prevents accidental shallow copies that would break invariants.

2) If a task throws and there is no exception handling inside the thread function, the exception will call `std::terminate()`, aborting the program. Using futures allows exceptions to be captured and rethrown in the context of the caller when `future::get()` is called.

3) Without a condition variable, worker threads would need to poll the queue, repeatedly acquiring a mutex to check if tasks are available. This wastes CPU cycles. A condition variable lets workers sleep and wake only when there is new work or shutdown.

4) You might want an explicit thread count to limit resource usage on heavily loaded systems, or when the pool is dedicated to a specific subsystem whose workload is known. In virtualized or containerized environments, hardware concurrency hints can also be inaccurate.

Chapter 4

Implementing the Basic Thread Pool

4.1 Complete Implementation

Here is a compact, fully working implementation using Modern C++:

```
#ifndef SIMPLE_THREAD_POOL_HPP
#define SIMPLE_THREAD_POOL_HPP

#include <vector>
#include <thread>
#include <future>
#include <queue>
#include <mutex>
#include <condition_variable>
#include <functional>
#include <atomic>
#include <stdexcept>
#include <type_traits>
```

```
14 #include <utility>
15
16 class thread_pool {
17 public:
18     explicit thread_pool(std::size_t thread_count =
19         std::thread::hardware_concurrency())
20     {
21         if (thread_count == 0) {
22             thread_count = 1;
23         }
24
25         try {
26             for (std::size_t i = 0; i < thread_count; ++i) {
27                 workers_.emplace_back(&thread_pool::worker_loop,
                    ↪ this);
28             }
29         } catch (...) {
30             stop();
31             throw;
32         }
33     }
34
35     thread_pool(const thread_pool&) = delete;
36     thread_pool& operator=(const thread_pool&) = delete;
37
38     thread_pool(thread_pool&&) = delete;
39     thread_pool& operator=(thread_pool&&) = delete;
40
41     template <class F, class... Args>
```

```
42      auto submit(F&& f, Args&&... args)
43          -> std::future<std::invoke_result_t<F, Args...>>
44      {
45          using result_type = std::invoke_result_t<F, Args...>;
46
47          auto task =
            ↪  std::make_shared<std::packaged_task<result_type()>>(
48              std::bind(std::forward<F>(f),
                ↪  std::forward<Args>(args)...)
49          );
50
51          std::future<result_type> result = task->get_future();
52
53          {
54              std::lock_guard<std::mutex> lock(mutex_);
55              if (stop_) {
56                  throw std::runtime_error("thread_pool is stopping");
57              }
58              tasks_.emplace([task]() {
59                  try {
60                      (*task)();
61                  } catch (...) {
62                      // Exceptions are already stored in the future
63                      // by packaged_task, so nothing else to do here.
64                  }
65              });
66          }
67
68          cv_.notify_one();
```

```
69          return result;
70      }
71
72      ~thread_pool() {
73          stop();
74          for (auto& worker : workers_) {
75              if (worker.joinable()) {
76                  worker.join();
77              }
78          }
79      }
80
81  private:
82      void worker_loop() {
83          for (;;) {
84              std::function<void()> task;
85
86              {
87                  std::unique_lock<std::mutex> lock(mutex_);
88                  cv_.wait(lock, [this] {
89                      return stop_ || !tasks_.empty();
90                  });
91
92                  if (stop_ && tasks_.empty()) {
93                      return;
94                  }
95
96                  task = std::move(tasks_.front());
97                  tasks_.pop();
```

```
            }

            // Run the task outside the lock
            task();
        }
    }

    void stop() {
        {
            std::lock_guard<std::mutex> lock(mutex_);
            stop_ = true;
        }
        cv_.notify_all();
    }

    std::vector<std::thread> workers_;
    std::queue<std::function<void()>> tasks_;

    std::mutex mutex_;
    std::condition_variable cv_;
    bool stop_{false};
};

#endif // SIMPLE_THREAD_POOL_HPP
```

4.2 Discussion

4.2.1 RAII and Exception Safety

- The constructor creates worker threads and, in case of failure, calls `stop()` and rethrows. This avoids leaving a partially initialized pool running.
- The destructor calls `stop()`, then joins all workers. This ensures no thread is running after the pool is destroyed.
- `submit` throws an exception if the pool is stopping, preventing tasks being enqueued into a dying pool.

4.2.2 Worker Loop Logic

The worker loop:

1) Acquires the lock.
2) Waits until there is either a task available or `stop_` is true.
3) If `stop_` is true and the queue is empty, it returns.
4) Otherwise, it pops one task and releases the lock.
5) Executes the task outside of the lock.

Executing tasks outside the lock is critical to avoid serialized execution.

4.2.3 Exception Handling in Tasks

We wrap the call to `(*task)()` in a `try` block. In this design, `std::packaged_task` automatically stores any thrown exception into the future. We do not need to catch and rethrow; the empty catch block simply prevents the exception from escaping and calling `std::terminate`. This is sufficient and common in simple thread pool designs.

4.3 Using the Thread Pool

Example usage:

```cpp
#include "simple_thread_pool.hpp"
#include <iostream>
#include <vector>

int square(int x) {
    return x * x;
}

int main() {
    thread_pool pool(4);

    std::vector<std::future<int>> results;
    for (int i = 0; i < 10; ++i) {
        results.push_back(pool.submit(square, i));
    }

    for (auto& f : results) {
        std::cout << f.get() << " ";
    }
    std::cout << "\n";
}
```

4.4 Exercises

1) In the implementation above, why do we store tasks as `std::function<void()>` instead of storing `std::packaged_task<R()>` directly?

2) What would happen if we executed the task while still holding the mutex?

3) Modify the design (conceptually) to allow the thread pool to be stopped early by an explicit `shutdown()` call, separate from the destructor.

4) Why do we use `std::shared_ptr` to store the `std::packaged_task` in the queue?

Solutions to Exercises (Chapter 4)

1) `std::function<void()>` provides a uniform, type-erased representation for arbitrary tasks, regardless of return type or arguments. We can enqueue many different tasks into the same queue. The packaged task is wrapped as a callable that fits this signature.

2) If we executed the task while holding the mutex, all other workers would be blocked from accessing the queue during task execution. This could serialize task execution and even cause deadlocks if tasks try to enqueue more work or interact with the pool.

3) A `shutdown()` function would set the `stop_` flag under the mutex and call `cv_.notify_all()`. It would then join all workers, similar to the destructor. The destructor could simply call `shutdown()` if it has not been called already.

4) We use `std::shared_ptr` because we need to copy the callable into the `std::function<void()>` stored in the queue. `std::packaged_task` is move-

only, and wrapping it in `shared_ptr` allows us to have a copyable small object (the pointer) that still refers to the same underlying packaged task.

Chapter 5

Enhancing the Thread Pool

5.1 Adding a Graceful Shutdown API

We can separate explicit `shutdown()` from the destructor and allow multiple calls:

```
class thread_pool {
public:
    explicit thread_pool(std::size_t thread_count =
        std::thread::hardware_concurrency());

    template <class F, class... Args>
    auto submit(F&& f, Args&&... args)
        -> std::future<std::invoke_result_t<F, Args...>>;

    void shutdown();  // new

    ~thread_pool();

```

```
private:
    // ... same as before ...

    std::atomic<bool> stopping_{false};
};
```

Implementation sketch:

```
void thread_pool::shutdown() {
    bool expected = false;
    if (!stopping_.compare_exchange_strong(expected, true)) {
        // already stopping
        return;
    }

    {
        std::lock_guard<std::mutex> lock(mutex_);
        stop_ = true;
    }
    cv_.notify_all();

    for (auto& worker : workers_) {
        if (worker.joinable()) {
            worker.join();
        }
    }
}

thread_pool::~thread_pool() {
    shutdown();
```

```
}
```

Now the pool can be shut down explicitly; additional calls are harmless.

5.2 Supporting `std::jthread` and Stop Tokens

With C++20, we can redesign the worker threads to use `std::jthread` and `std::stop_token`. This improves cancellation semantics. The full integration requires careful reworking of the worker loop predicate to react both to queued tasks and to stop requests.

5.3 Task Priority and Scheduling

A more advanced pool may need:

- Priority queues instead of FIFO.
- Work stealing (each worker has its own deque; idle workers steal tasks).
- Separate queues for different subsystems.

These are beyond the scope of a “simple” pool but follow the same principles.

5.4 Avoiding Unbounded Queues

The current design has an unbounded queue. In high-load scenarios, this can lead to unbounded memory growth. A more robust design may:

- Use a bounded queue and block submitters when full.
- Reject new tasks with a specific error or exception.
- Implement back-pressure mechanisms.

5.5 Exercises

1) Explain the difference between the `stop_` flag and the `stopping_` atomic flag in the enhanced design.

2) Discuss one strategy for bounding the task queue to avoid unbounded memory usage.

3) Conceptually, how would you support task priorities in the queue?

4) Why might it be dangerous to allow new submissions while the pool is shutting down?

Solutions to Exercises (Chapter 5)

1) `stop_` is protected by the mutex and used inside the worker loop as part of the condition variable predicate. `stopping_` is an atomic flag used to ensure that `shutdown()` is executed only once even if called from multiple threads. They serve different purposes.

2) One strategy is to use a bounded blocking queue: if the queue has reached a maximum size, `submit()` blocks until space becomes available. Alternatively, `submit()` can throw an exception or return an error status when the queue is full, allowing callers to handle overload.

3) Instead of a single FIFO queue, use a priority queue where each task has a priority value. Workers always pick the highest priority task. Internally, you can store a struct with priority and callable and use `std::priority_queue`.

4) If new tasks are accepted while shutdown is in progress, some tasks might be enqueued but never executed, or they might execute after parts of the system have already been destructed. This can easily lead to use-after-free bugs and undefined behavior.

Chapter 6

Using the Thread Pool Effectively

6.1 CPU-Bound Workloads

For CPU-bound computation (e.g., numerical algorithms), a thread pool can be used to parallelize loops. Example: a simple parallel `for`:

```
template <class Index, class F>
void parallel_for(Index first, Index last, F&& f, thread_pool& pool)
 ↪  {
    const Index length = last - first;
    if (length <= 0) return;

    const std::size_t hardware_threads =
        std::max<std::size_t>(1,
         ↪  std::thread::hardware_concurrency());
    const std::size_t num_tasks =
        std::min<std::size_t>(hardware_threads,
         ↪  static_cast<std::size_t>(length));
```

```
10
11      const Index block_size = length / static_cast<Index>(num_tasks);
12
13      std::vector<std::future<void>> futures;
14      Index block_start = first;
15
16      for (std::size_t i = 0; i < num_tasks; ++i) {
17          Index block_end =
18              (i == num_tasks - 1) ? last : block_start + block_size;
19
20          futures.push_back(pool.submit([block_start, block_end, &f] {
21              for (Index idx = block_start; idx < block_end; ++idx) {
22                  f(idx);
23              }
24          }));
25
26          block_start = block_end;
27      }
28
29      for (auto& fut : futures) {
30          fut.get();
31      }
32  }
```

6.2 I/O-Bound Workloads

For workloads dominated by I/O (network requests, disk access), a thread pool can:

- Offload blocking operations from the main thread.

- Allow more threads than cores (since many threads will be blocked).
- Still benefit from centralized management and back-pressure.

However, for high-scale networking, asynchronous I/O or event-driven designs are often preferable; thread pools can be part of the implementation but are not the only option.

6.3 Common Patterns

- **Fire-and-forget**: submit tasks without using futures (e.g., tasks that log or send metrics).
- **Map/Reduce**: submit a task per chunk of data, then combine the results.
- **Pipelines**: multiple thread pools processing stages of a pipeline (parse – process – write).

6.4 Exercises

1) In the `parallel_for` implementation, why do we limit the number of tasks to the minimum of hardware threads and the number of iterations?
2) Describe a scenario where using a thread pool for I/O-bound tasks is beneficial.
3) How would you implement a simple map-reduce using the thread pool?
4) Why is it important to call `future::get()` after submitting many tasks that return results?

Solutions to Exercises (Chapter 6)

1) Creating more tasks than iterations provides no benefit and just adds scheduling overhead. Limiting tasks also avoids creating more futures than necessary and reduces memory usage.

2) For example, in a web server that needs to handle a moderate number of concurrent HTTP requests, a thread pool can process each request in a worker thread while the main thread listens for new connections. The pool hides the details of thread management and avoids creating/destroying a thread per request.

3) Conceptually, map-reduce can be implemented by submitting a task per input chunk (map stage), each returning a partial result; then after all futures complete, you aggregate these results in a single-threaded reduce step.

4) `future::get()` is needed to retrieve results and to rethrow any exceptions that occurred in the tasks. If you never call `get()`, exceptions remain unnoticed and tasks may still hold resources until the future is destroyed.

Chapter 7

Testing, Debugging, and Pitfalls

7.1 Testing Thread Pools

Testing concurrent code is inherently harder than testing sequential code. For thread pools:

- Test with many small tasks and ensure all of them complete.
- Test shutdown in the presence of pending tasks.
- Test exception propagation: tasks that throw must have their exceptions visible in futures.
- Test under high load: thousands of tasks, many threads.

7.2 Typical Pitfalls

7.2.1 Data Races Inside Tasks

The thread pool itself is thread-safe, but your tasks may not be. You must still protect shared data in task bodies with appropriate synchronization or use lock-free structures where appropriate.

7.2.2 Deadlocks and Re-entrancy

Tasks that wait on other tasks in the same pool can create deadlocks if the pool size is too small. For example, a task that waits for the results of other tasks via futures while all workers are occupied can deadlock.

7.2.3 Starvation

If tasks are scheduled in strict FIFO and some tasks are extremely long-running, shorter tasks may wait too long. Advanced designs use work stealing or multiple queues to reduce starvation.

7.3 Exercises

1) Describe a test scenario that would reveal a bug in the shutdown logic of the thread pool.

2) Explain how a deadlock could occur if a task submitted to the pool waits for another task in the same pool.

3) How can logging help debug thread pool problems, and what are the dangers of naive logging in multithreaded programs?

4) Provide one strategy to reduce starvation in a simple FIFO-based thread pool.

Solutions to Exercises (Chapter 7)

1) A scenario: submit a large number of tasks, then immediately destroy the pool while some tasks are still running. If shutdown logic is incorrect, some tasks may be left unexecuted, or the destructor might return before all worker threads are joined, leading to data races or crashes.

2) If all worker threads are blocked waiting for futures of other tasks that have not started yet (because the workers are already busy), then no progress can be made, causing a deadlock. This is a typical “thread pool exhaustion” scenario.

3) Logging can show the order of task submission, execution, and shutdown, helping to see race patterns. However, naive logging (e.g., logging from every task without synchronization) can itself introduce races or dramatically slow down the program, changing timing and hiding bugs.

4) One simple strategy is to limit the maximum time each task can run or to break large tasks into smaller ones. More sophisticated strategies involve priority queues or work stealing between threads.

Chapter 8

Future Directions: Executors and Modern C++

8.1 Executors and Senders/Receivers (Overview Only)

The C++ standardization committee has been working on higher-level concurrency abstractions, often called *executors* and *senders/receivers*. The goal is to provide:

- A common abstraction for where and how work is executed.
- Composable asynchronous operations (pipelines, continuations).
- Better integration with coroutines and asynchronous I/O.

A thread pool, conceptually, is a kind of executor: it takes submitted work and runs it on a specific execution context (the worker threads).
Although these proposals are still evolving, designing your own thread pool with clear, executor-like semantics can make it easier to integrate with future standard facilities.

8.2 Coroutines and Thread Pools

C++20 coroutines provide a language-level foundation for asynchronous programming. While coroutines themselves do not create threads, they can be resumed on various execution contexts, including thread pools.
An important design pattern is to use a thread pool as an execution context for coroutine continuations, especially when performing CPU-bound work as part of a larger asynchronous pipeline.

8.3 Exercises

1) In your own words, what is an executor, and how is a thread pool similar to one?

2) How can separating the concept of “what to do” (task) from “where to do it” (executor) improve software design?

3) Why are coroutines not a replacement for thread pools?

Solutions to Exercises (Chapter 8)

1) An executor is an abstraction that accepts units of work and determines where, when, and how they are executed. A thread pool is similar because it accepts tasks and runs them on its worker threads; it can be viewed as a concrete executor.

2) Separating tasks from executors decouples the description of computation from the execution strategy. This improves flexibility: you can run the same tasks on different executors (thread pools, single-threaded executors, GPU executors) without changing task code.

3) Coroutines are a language mechanism for suspending and resuming functions, but they do not provide parallelism by themselves. You still need threads or other execution contexts underneath to run different coroutines concurrently. Thread pools provide these contexts.

Final Conclusion

Thread pools are a fundamental building block in Modern C++ concurrency. They provide a practical compromise between raw threads and very high-level abstractions: you keep control over scheduling, resources, and data layout, yet avoid the heavy overhead and complexity of creating a thread per task.
In this mini-booklet, we:

- Motivated the need for thread pools beyond naive thread-per-task designs.
- Reviewed the core concurrency primitives of Modern C++: threads, mutexes, condition variables, futures, and packaged tasks.
- Designed and implemented a compact, reusable thread pool using only standard C++ facilities.
- Discussed shutdown protocols, exception handling, queue management, and safe task submission.
- Looked at common usage patterns, pitfalls, and strategies for robust testing.
- Connected the thread pool idea to future directions in the C++ standard, including executors and coroutines.

The implementation presented here is intentionally simple, yet it is already sufficient for many real-world applications. You can extend it with:

- bounded or prioritized queues;
- work-stealing architectures;
- metrics and monitoring;
- tight integration with I/O frameworks or coroutine libraries.

By understanding the internal mechanics of a thread pool, you will be better prepared to:

- judge the quality of existing concurrency libraries;
- debug performance and correctness issues in multithreaded systems;
- follow and adopt upcoming C++ concurrency features with confidence.

Modern C++ gives you the tools. A carefully designed thread pool allows you to use them efficiently, safely, and in a way that scales to real-world workloads.

References

Bibliography

[1] **cppreference.com**.
C++ Reference: Concurrency library.
Detailed documentation of `std::thread`, `std::mutex`, `std::condition_variable`, `std::future`, `std::packaged_task`, and related facilities.

[2] Anthony Williams.
C++ Concurrency in Action: Practical Multithreading.
Manning Publications, Second Edition, 2019.
Comprehensive coverage of the C++ memory model and concurrency primitives, including patterns and thread pool designs.

[3] Bjarne Stroustrup.
The C++ Programming Language (Fourth Edition).
Addison–Wesley, 2013.
Chapters on concurrency provide foundational understanding of threads and synchronization in C++11 and later.

[4] ISO C++ Core Guidelines.
Concurrency Rules.

A set of best practices for writing safe and maintainable concurrent C++ code, including guidance on avoiding data races and deadlocks.

[5] ISO/IEC 14882:2020 and later drafts.
Programming Languages – C++.
Sections on the standard concurrency library (threads, mutexes, futures) and newer facilities such as `std::jthread` and stop tokens.

[6] ISO C++ Committee Papers.
P2300 – Sender/Receiver.
Design proposals for high-level asynchronous execution and executors in upcoming C++ standards, motivating future directions beyond simple thread pools.

[7] Herb Sutter.
Concurrency and Parallelism in C++.
Talks and papers discussing best practices and patterns for scalable concurrency in Modern C++.