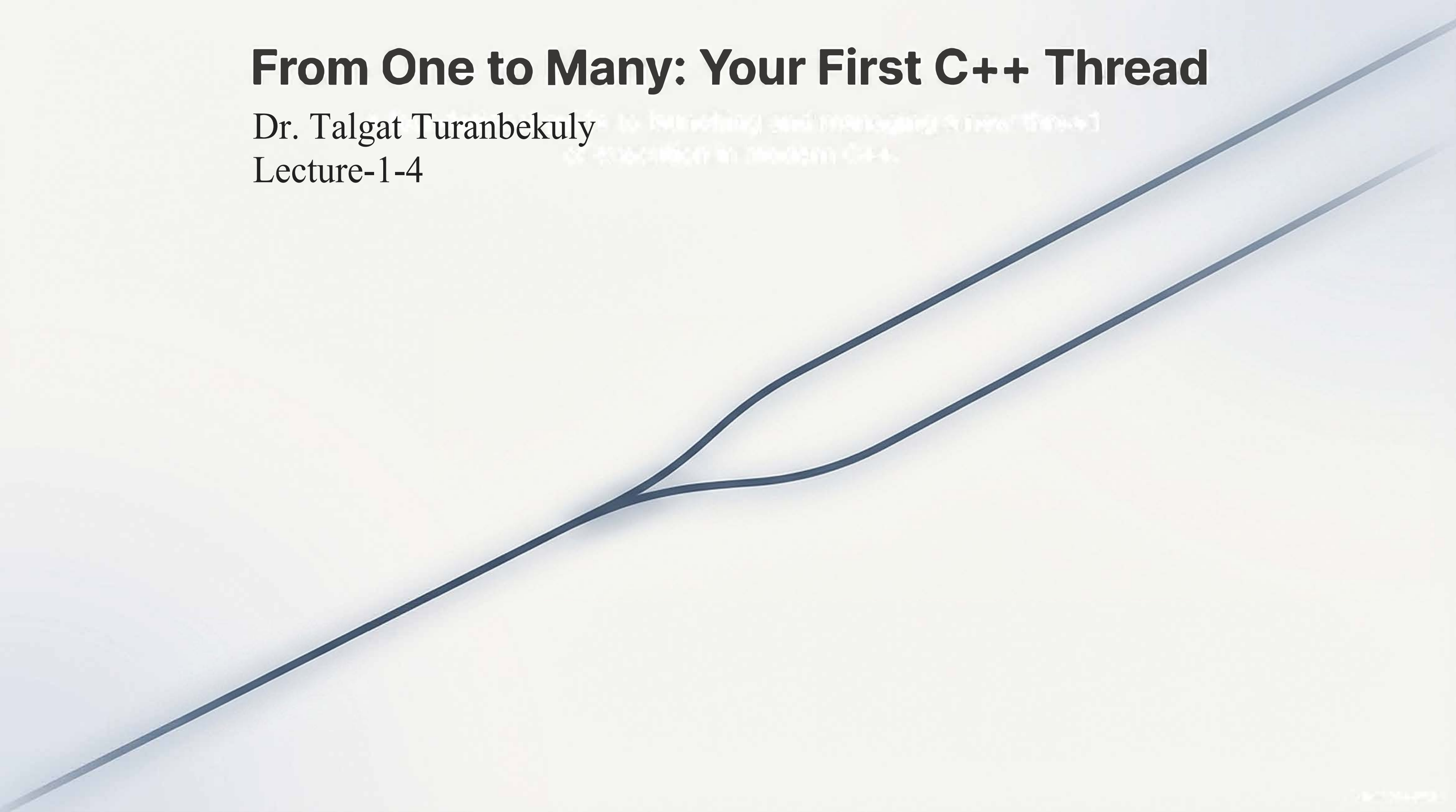
From One to Many: Your First C++ Thread

Dr. Talgat Turanbekuly

Lecture-1-4



Every Program Begins with a Single Thread

Every C++ program looks much like any other, with the usual mix of variables, classes, and functions. By default, it runs on a single thread of execution, starting and ending within `main()`.

```cpp
// The baseline: single-threaded "Hello World"
#include <iostream>


int main()
{
    std::cout << "Hello World\n";
}
```

Introducing a Second Thread of Execution

To run functions concurrently, we use specific objects to manage new threads. Let's compare our baseline to a program that starts a separate thread to display its message.

Single Thread

```
// main()
#include <iostream>

int main() {
    std::cout << "Hello World\n";
}
```

Two Threads

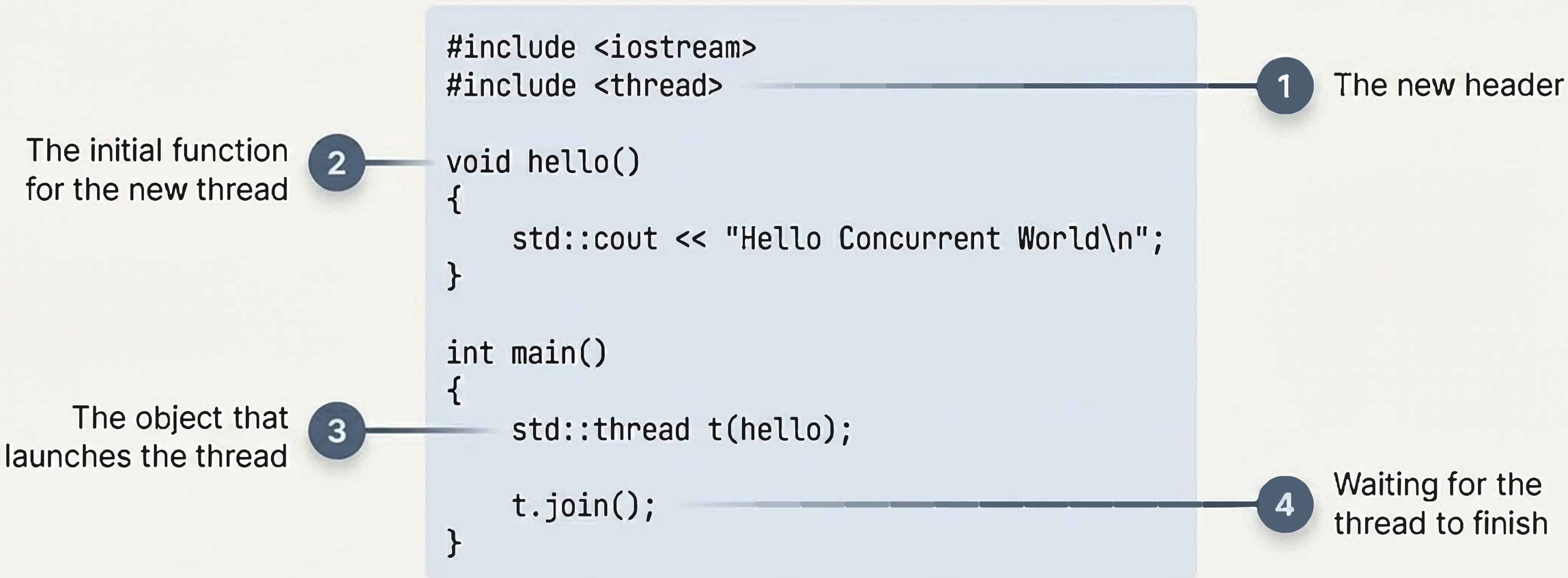
```
// main() with a new thread
#include <iostream>
#include <thread> // New

void hello() { // New
    std::cout << "Hello Concurrent World\n";
}

int main() {
    std::thread t(hello); // New
    t.join();             // New
}
```

Anatomy of a Concurrent Program

The transition to a multi-threaded approach involves four key changes. Let's break them down one by one.



1. The Gateway to Concurrency: <thread>

The declarations for multithreading support in the Standard C++ Library are in new headers. The primary functions and classes for managing threads, such as `std::thread`, are declared in `<thread>`.

```
#include <thread>
```

Other headers, such as <mutex> and <future>, provide tools for protecting shared data and managing asynchronous tasks.

2. Every Thread Needs an Initial Function

Every thread must have an initial function where its execution begins. For the application's initial thread, this is main(). For any new thread we create, we must specify its entry point.

```
// This function serves as the starting point
// for our new thread.

void hello()
{
    std::cout << "Hello Concurrent World\n";
}
```

3. Launching the New Thread

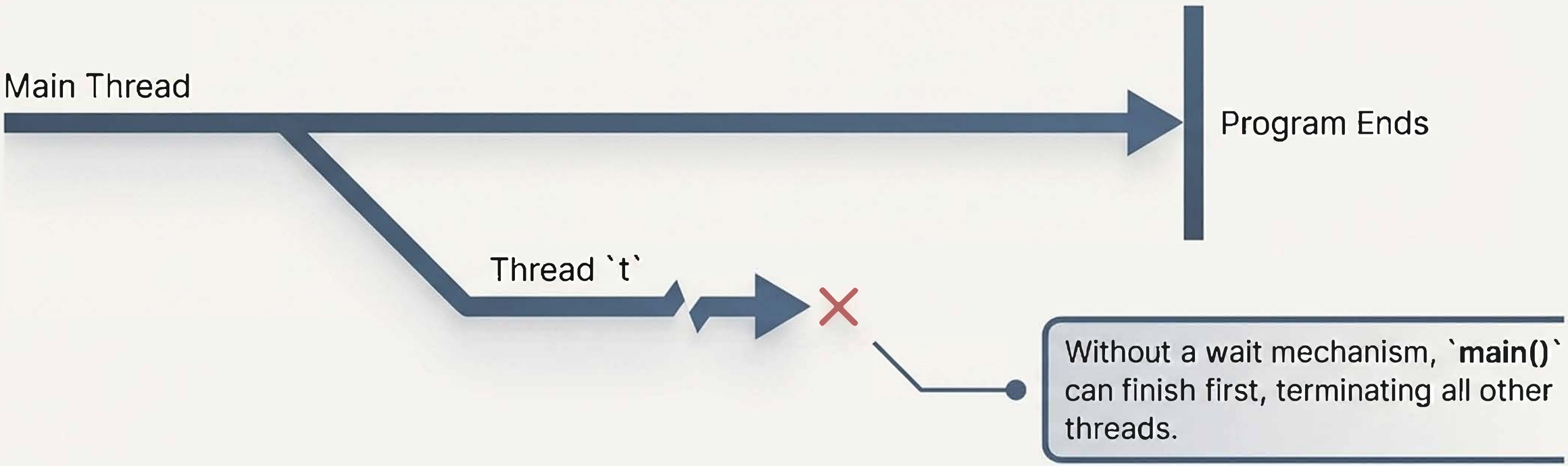
The `std::thread` object is the core of thread management. Its constructor takes the initial function to execute. The moment this object is created, the new thread may begin running concurrently.

```
// The std::thread object 't' is created.
// The 'hello' function is passed to its constructor.
// This launches the new thread.

std::thread t(hello);
```

The Race to the End

After the new thread is launched, the initial thread continues its execution. If it doesn't wait, it could reach the end of `main()` and terminate the entire program—possibly before the new thread even has a chance to run.

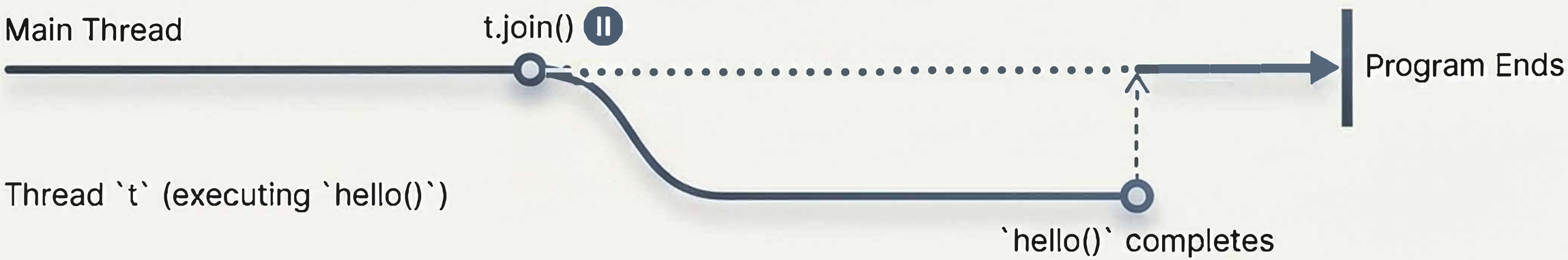


4. Waiting for Completion with `join()`

The call to `join()` solves this problem. It is a crucial synchronization point that blocks the calling thread until the thread it's called on has finished its execution.

> This causes the calling thread (in `main()`) to wait for the thread associated with the `std::thread` object, in this case, `t`.

```
// The main thread pauses here...
t.join();
// ...and only resumes after thread 't' has completed.
```



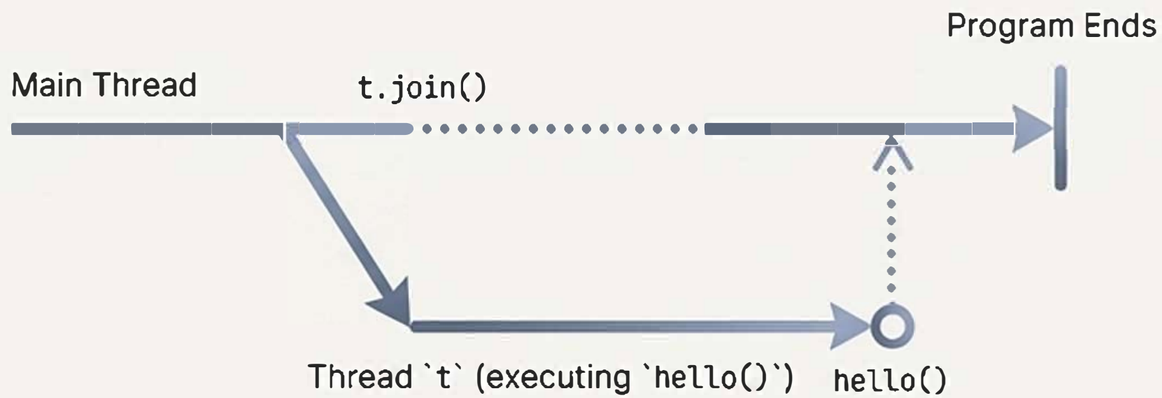
The Full Picture: Two Threads Synchronized

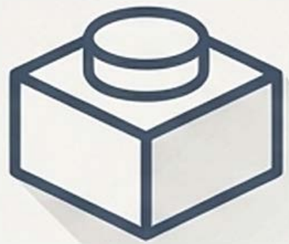
Putting it all together: the `main` thread creates a new thread `t`, then waits at the `join()` call for `t` to finish its work before proceeding to terminate the program.

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

void hello() {
    std::cout << "Hello Concurrent World\n";
}

int main() {
    std::thread t(hello);
    t.join();
}
```





An Illustration, Not a Performance Play

Is this a lot of effort just to write a message to the console? Yes. The goal here is to illustrate the mechanics of thread creation, not to demonstrate a practical performance gain.

> “It’s generally not worth the effort to use multiple threads for such a simple task, especially if the initial thread has nothing to do in the meantime.” `main()`

The Journey Ahead

You now have the fundamental tool for creating concurrent execution paths. The next great challenge is ensuring that when these thread threads need to cooperate, any shared data between them is accessed safely.

This leads to the next topic in concurrency: protecting shared data.