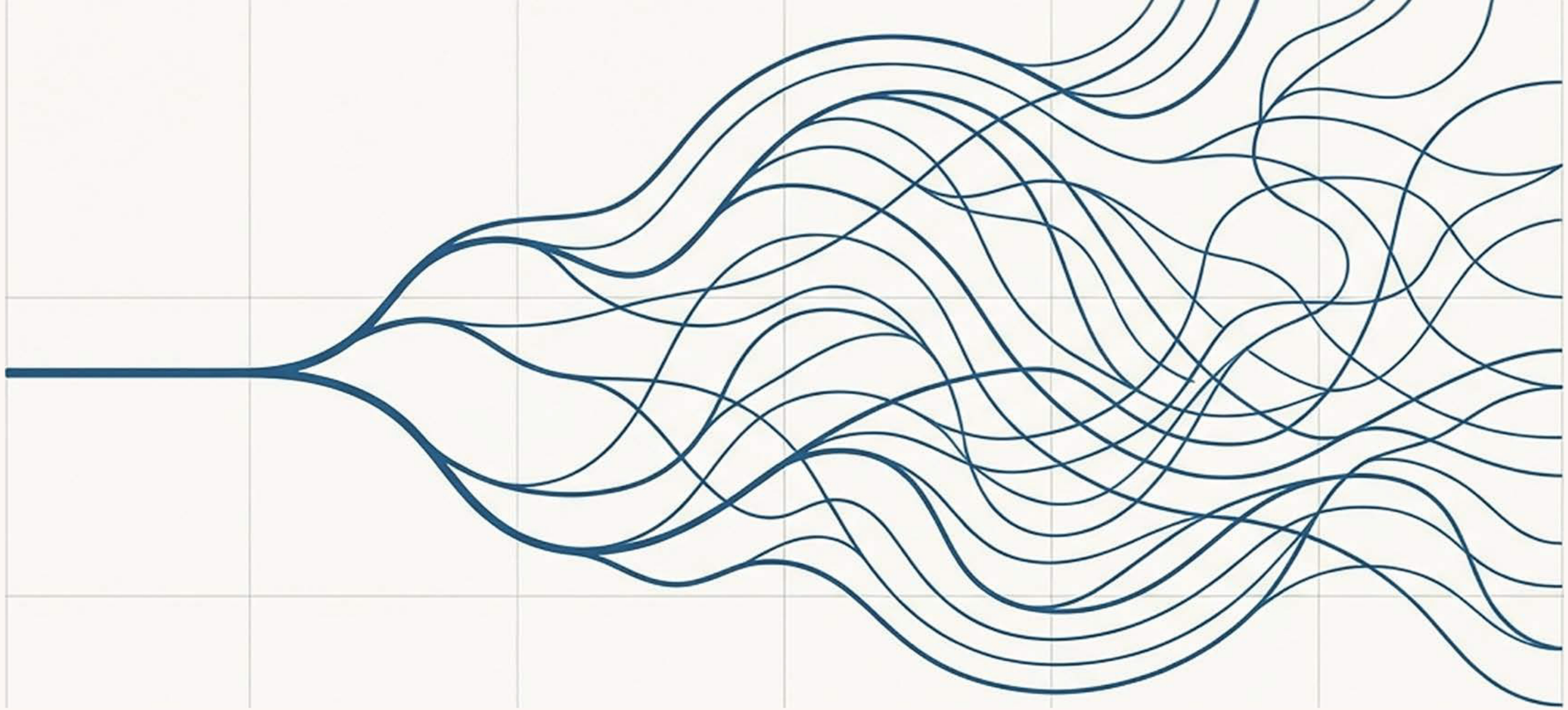
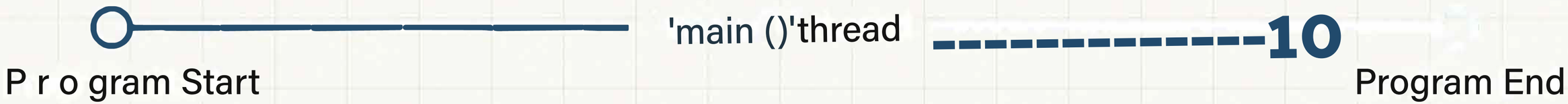


The Thread's Journey

Dr. Talgat Turanbekuly

Lecture-2-1

Every Program Begins with a Single Path



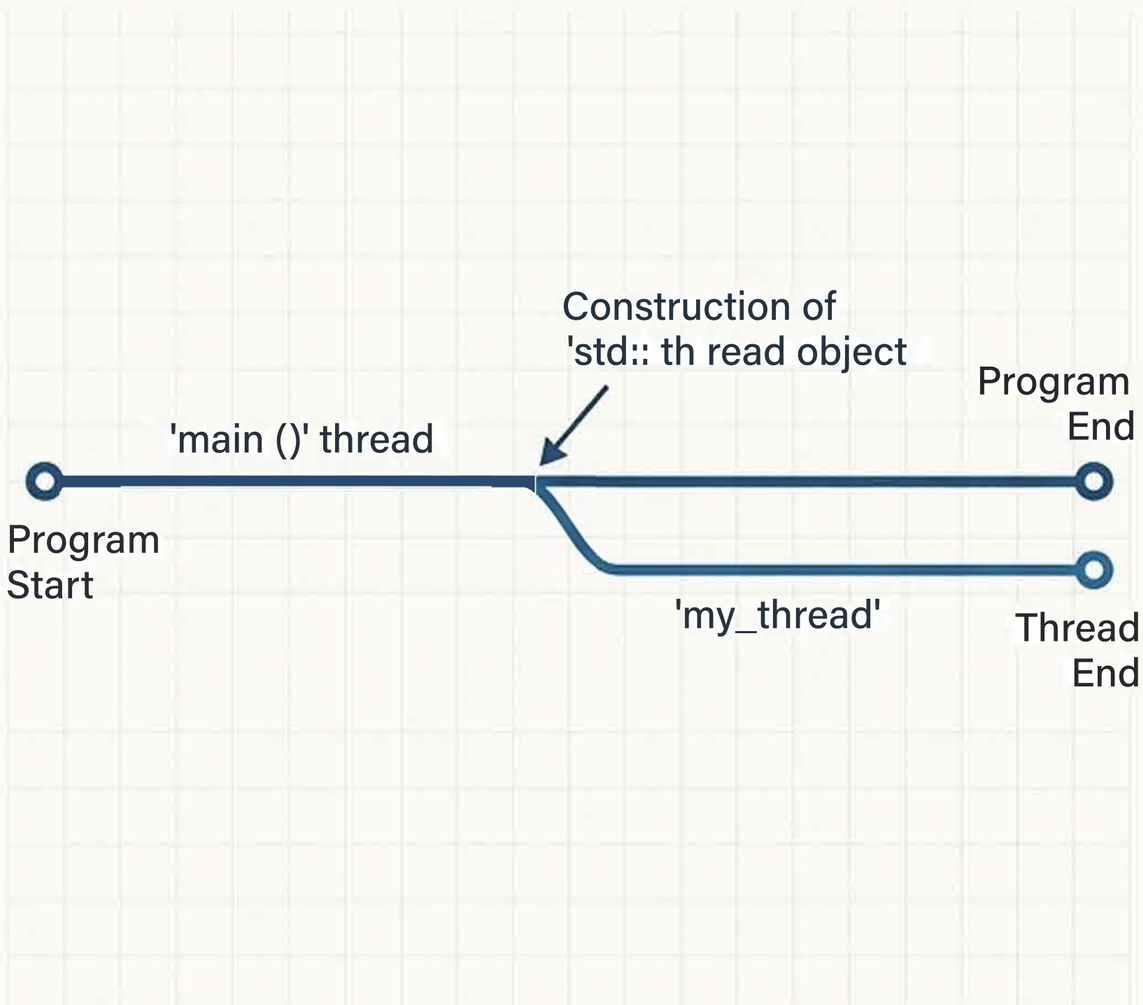
Core Concept

Every C++ program has at least one thread, which is started by the C++ runtime: the thread running main ().

Explanation

This initial thread is the foundation from which all concurrency is launched. New threads run concurrently with this main thread. Just as a program exits when it returns from main(), a new thread exits when its entry point function returns.

Forging a New Path: Construction is Execution



Core Concept

Launching a new thread boils down to one action: **constructing** a std:: th read object.

Explanation

The **moment** a std:: thread object is created and given a task, a new thread of execution may begin running. There is no separate . start () method; the constructor does it all.

```
#include <thread>

void do_some_work();

//Anew thread is born and begins execution immediately.
std::thread my_thread(do_some_work);
```

Defining the Thread, s Purpose

std: : thread works with any callable type. You can provide its task as a free function, a function object (functor), or a lambda expression.

Free Function

```
void task_function();

std::thread t1(task_function);
```

Function Object

```
class background_task
{
  public:
    void operator()() const;
};

std::thread t2((background_task()));
```

Lambda Expression

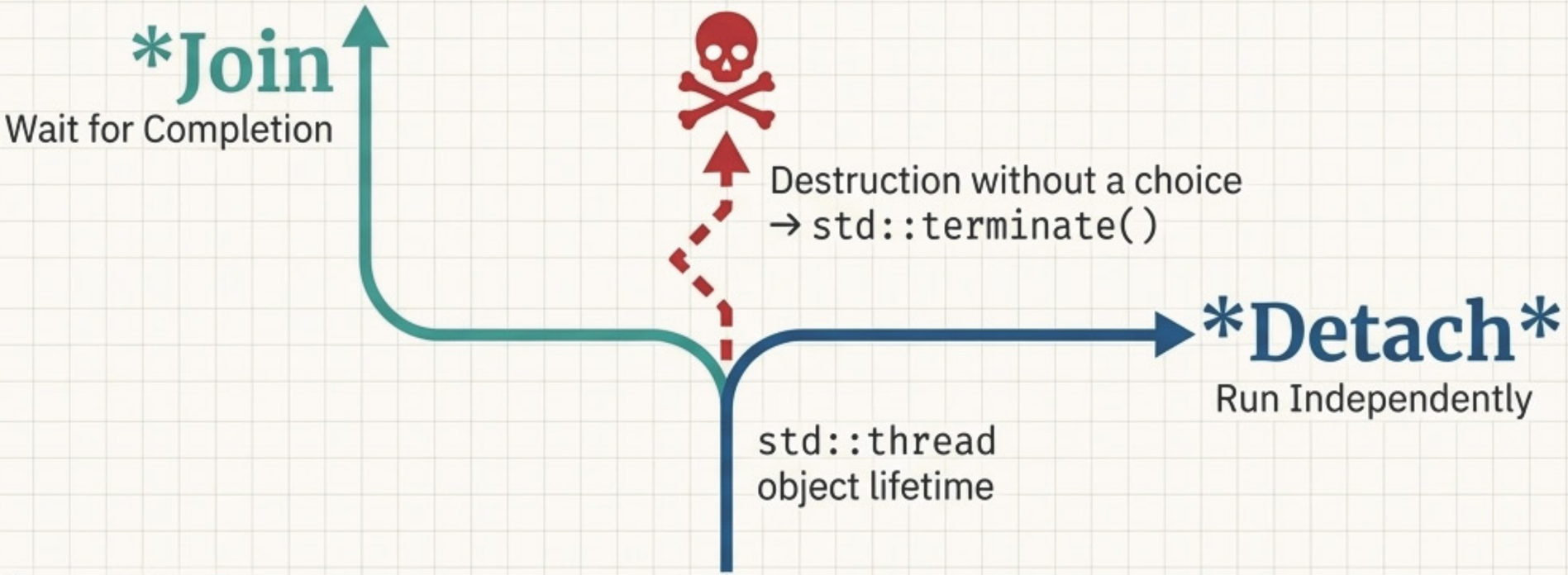
```
std::thread t3([]
{
/* do work */
});
```

Expert Tip: Beware C++'s Most Vexing Parse!

Problem: std: :thread my_thread(background_task());
is parsed as a function declaration, not an object definition.
Solutions: Use an extra set of parentheses or uniform initialization with braces.

```
std::thread my_thread((background_task())); // OK
std::thread my_thread{background_task()}; // OK (C++ll)
```

The Crossroads: An Unavoidable Decision



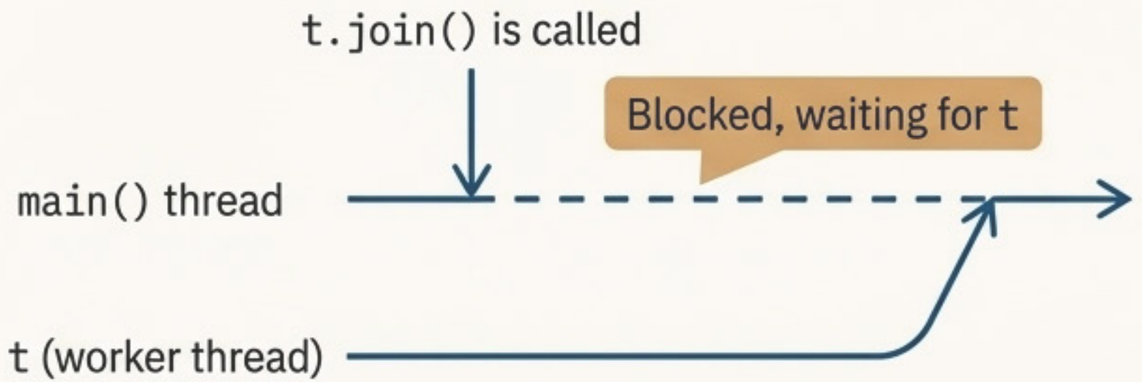
The Golden Rule

Before a `std::thread` object is destroyed, you *must* explicitly decide its fate.

Explanation

If you don't decide before the `std::thread` object is destroyed, then your program is terminated (the `std::thread` destructor calls `std::terminate()`). You **cannot** simply let a `std::thread` object go out of scope if it's still associated with a running thread.

Path 1: Wait for Completion with `join()`



Core Concept

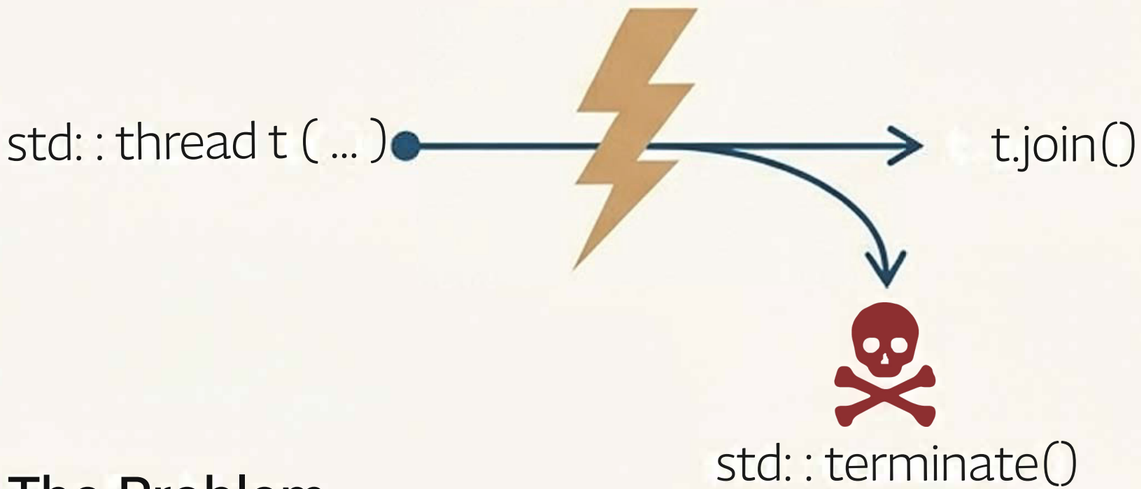
Calling `join()` blocks the calling thread until the thread of execution associated with the `std::thread` object has finished.

Key Properties

- It's a blocking call.
- `join()` cleans up all storage associated with the thread.
- A thread can only be joined once. After `join()`, the `std::thread` object is no longer joinable (`joinable()` will return `false`).

```
std::thread t(do_background_work);
// ... do other things in the main thread ...

// Pause here and wait for t to finish.
t.join();

// t has now completed. It is safe to proceed.
```

The Danger of the Unexpected Exit



The Problem: If an exception is thrown after a thread is launched but before the call to join(), the join() call will be skipped. This leads directly to std: : terminate() when the std: : thread object's destructor is called during stack unwinding.
The Naive Solution: Manually wrapping the code in a try/catch block.

The Problem

```
// From Listing 2.2
std::thread t(my_func);
try
{
      do_something_in_current_thread();
}
t.join(); // ... and on the normal path.
catch ( ... )
{
   t.join(); // Must remember to join on error path ...
   throw;
}
```

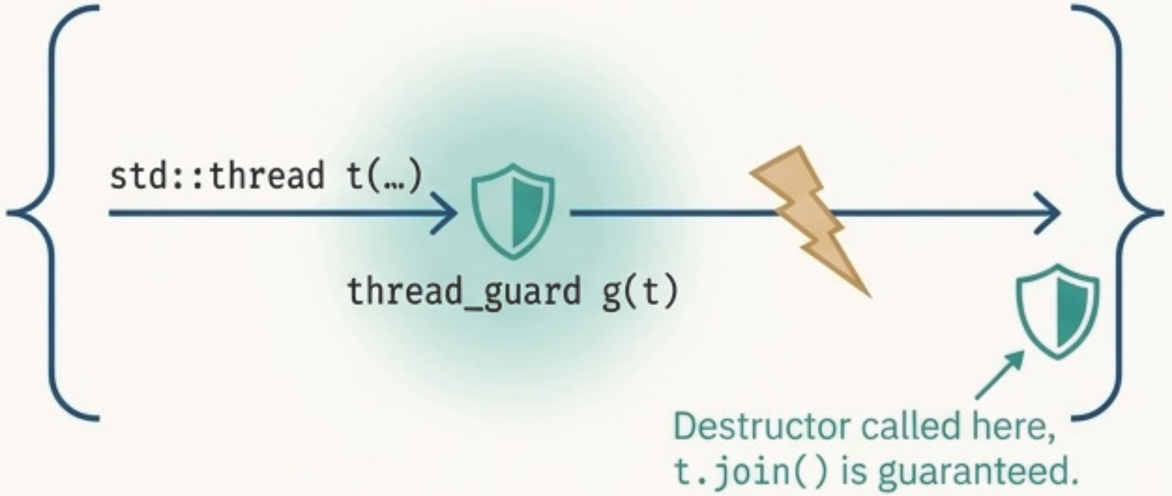
This approach is verbose, error-prone, and not idiomatic C++. We need a more robust solution.

The Guardian: Automating Safety with RAII

The Idiomatic Solution

Use the RAII idiom to create a `thread_guard` class. Its destructor guarantees that `join()` is called on the managed thread, no matter how the scope is exited (normally or by exception).

```
// From Listing 2.3
class thread_guard {
  std::thread& t;
public:
  explicit thread_guard(std::thread& t_): t(t_) {}
  ~thread_guard() {
    if(t.joinable()) { t.join(); }
  }
  // Deleted copy constructor and assignment operator
  thread_guard(thread_guard const&)=delete;
  thread_guard& operator=(thread_guard const&)=delete;
};
```

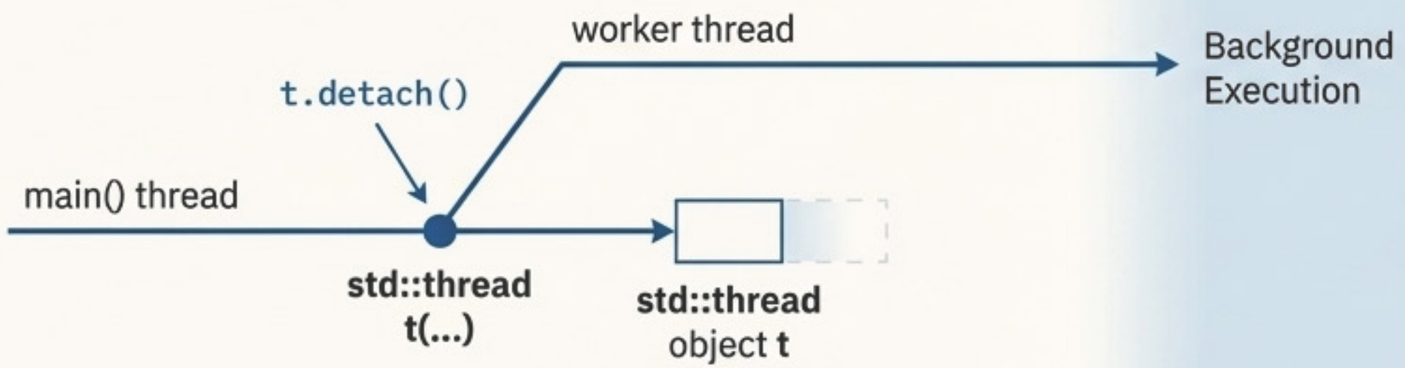


The elegant usage

```
void f() {
  std::thread t(my_func);
  thread_guard g(t); // The thread is now "guarded"
  do_something_in_current_thread();
} // g's destructor is called here, automatically joining the thread.
```

RAII transforms a manual cleanup task into an automatic, compile-time guarantee. This is the preferred method for managing joinable threads.

Path 2: Run in the Background with `detach()`



Core Concept

Calling `detach()` separates the thread of execution from the `std::thread` object. The C++ Runtime Library becomes responsible for cleaning up the thread's resources when it completes.

Explanation

The thread becomes a “daemon thread,” running in the background with no direct means of communication. After `detach()` is called, the `std::thread` object is no longer associated with the thread and is not joinable.

Use Cases

- Long-running background tasks (monitoring, cache cleaning).
- “Fire-and-forget" tasks where the result is not immediately needed.

Code Example

```
std::thread t(do_background_work);
t.detach();
assert(!t.joinable());
// The thread object is no longer joinable.
```

Cautionary Tale: The Peril of Dangling References

Code Example (The Pitfall)

```
// From Listing 2.1
void oops()
{
    int some_local_state = 0;
    struct func {
        int& i;
        func(int& i_): i(i_) {}
        void operator()() { /* uses i */ }
    };

    std::thread my_thread(func(some_local_state));
    my_thread.detach(); // Don't wait for thread to finish.
} // 'some_local_state' is destroyed, but my_thread may still be running.
```

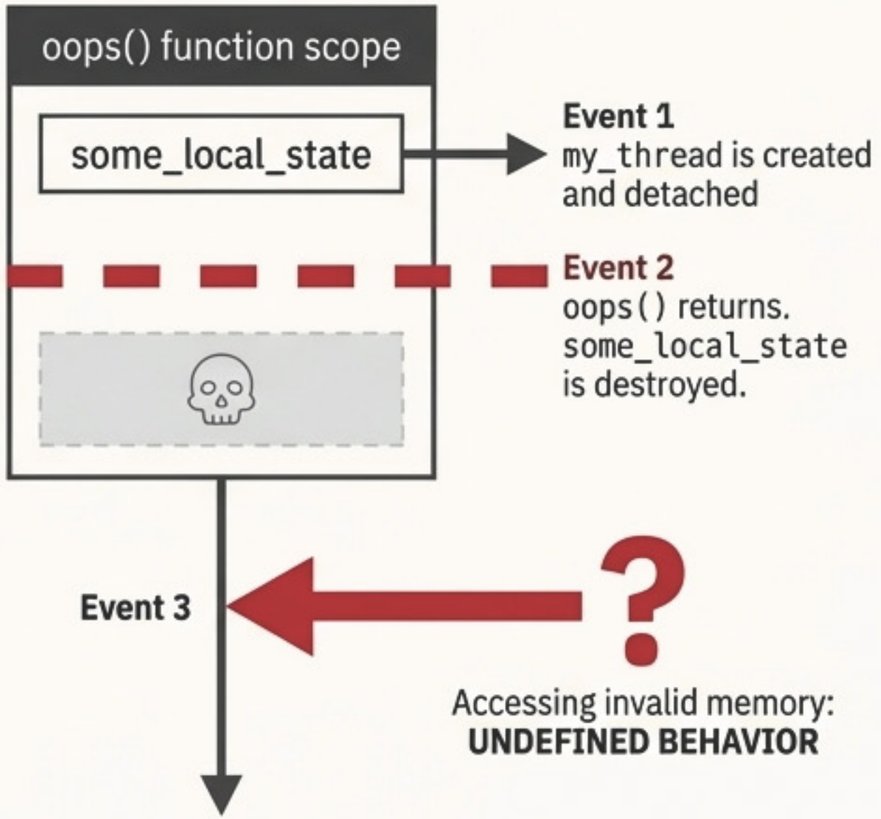
This local variable will be destroyed when `oops()' exits.

But this detached thread might still be using a reference to it!

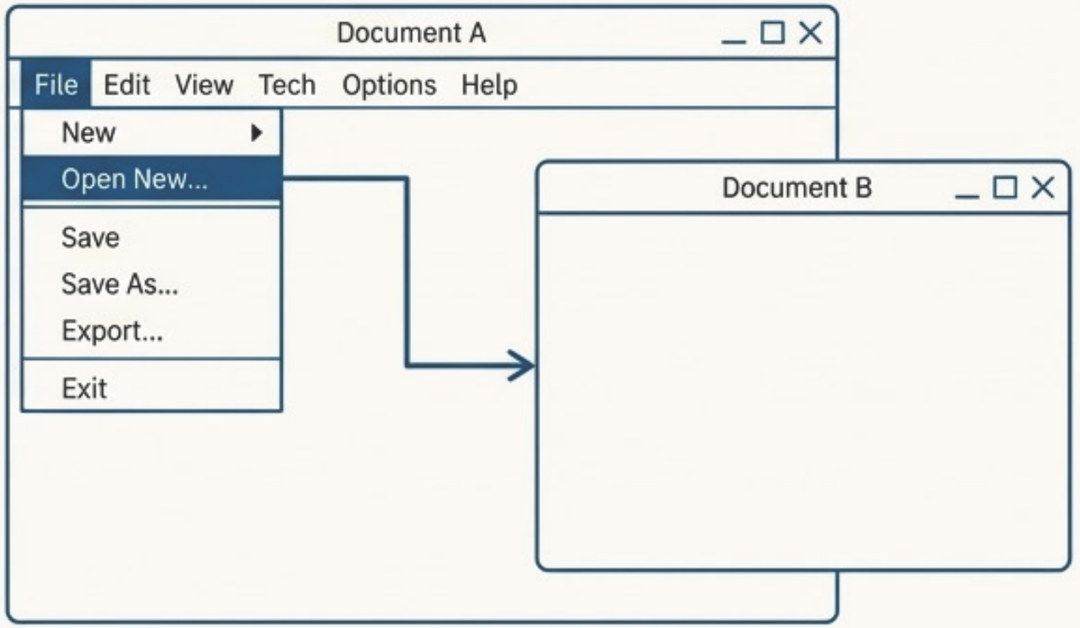
When you detach, you are solely responsible for data lifetime. Ensure the thread is self-contained or that its data is guaranteed to outlive it.

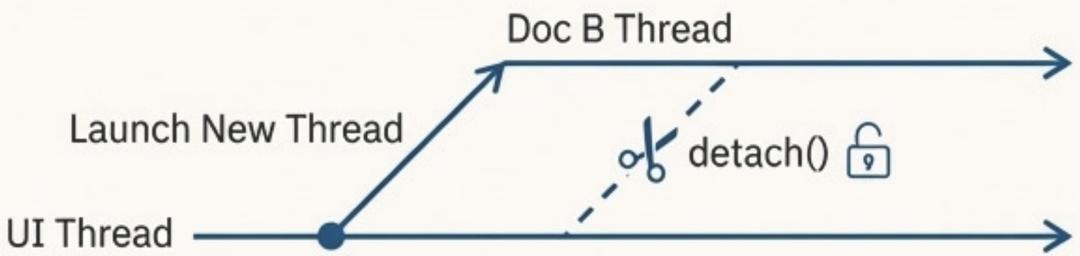
The Danger

If you don't wait for your thread to finish, you need to ensure that the data accessed by the thread is valid until the thread has finished with it.



A Valid Use Case: A Multi-Document Editor





The Scenario

A word processor allows users to edit multiple documents in separate windows. When a user opens a new document, the main UI thread must remain responsive and not block waiting for the new document to load.

The Solution

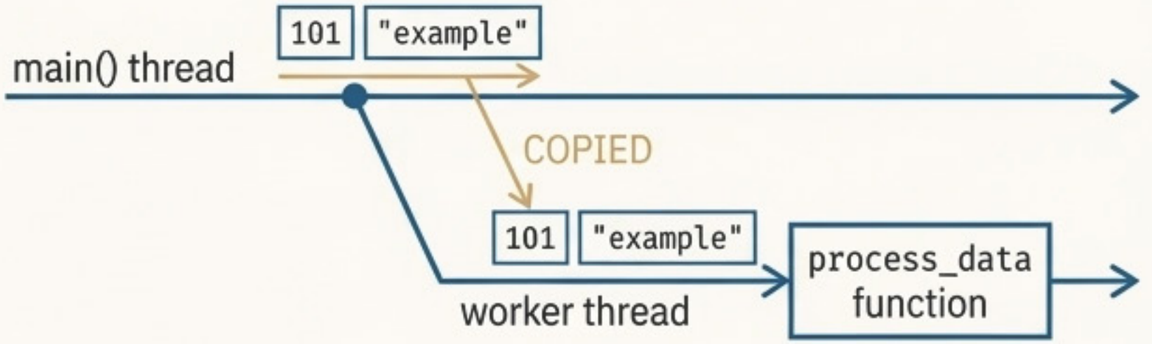
Launch a new thread to handle the new document window and immediately `detach()` it. The new thread becomes a peer to the original, managing its own document and window independently.

```
// Based on Listing 2.4
void edit_document(std::string const& filename);

// In response to a user command to open a new file...
std::string const new_name = get_filename_from_user();
std::thread t(edit_document, new_name);
t.detach(); // Let the new window's thread manage itself.
```

Key Message: `detach()` is powerful for creating independent, peer-level tasks within a single application, enhancing responsiveness.

Giving Threads Their Mission: Passing Arguments

```
void process_data(int id, std::string const& name);

// The arguments 101 and "example" are copied and passed
// to process_data on the new thread.
std::thread t(process_data, 101, "example");
t.join();
```



Core Concept

Additional arguments can be passed to the `std::thread` constructor. These arguments are then forwarded to the thread's entry-point function.

The Critical Detail

By default, arguments are *copied* into internal storage accessible to the new thread.

Why This Matters

Copying by default is a safety feature. It helps prevent accidental lifetime issues (like the dangling reference problem) by giving the thread its own copy of the data, decoupling it from the lifetime of the original variable

The Rules of the Road

A Checklist for Safe Thread Management

The Golden Rule: Every `std::thread` must be either `joined` or `detached` before its destructor is called. No exceptions.

Default to Safety: Prefer `join()`. Use the RAII pattern (`thread_guard`) to ensure your threads are joined correctly, even in the presence of exceptions.

Use `detach()` with Extreme Caution: Reserve it for truly independent, long-running 'daemon' threads or 'fire-and-forget' tasks.

Own Your Lifetimes: When detaching, you become solely responsible for data lifetimes. Ensure any data a detached thread accesses will outlive the thread itself, often by copying data into the thread.