Username: Pralay Patoria **Book:** C++ Concurrency in Action: Practical Multithreading. No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.

1.4. Getting started

OK, so you have a nice, shiny C++11-compatible compiler. What next? What does a multithreaded C++ program look like? It looks pretty much like any other C++ program, with the usual mix of variables, classes, and functions. The only real distinction is that some functions might be running concurrently, so you need to ensure that shared data is safe for concurrent access, as described in chapter 3. Of course, in order to run functions concurrently, specific functions and objects must be used to manage the different threads.

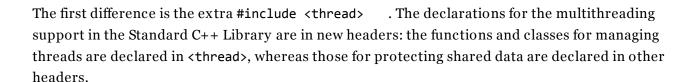
1.4.1. Hello, Concurrent World

Let's start with a classic example: a program to print "Hello World." A really simple Hello, World program that runs in a single thread is shown here, to serve as a baseline when we move to multiple threads:

```
#include <iostream>
int main()
{
    std::cout<<"Hello World\n";
}</pre>
```

All this program does is write "Hello World" to the standard output stream. Let's compare it to the simple Hello, Concurrent World program shown in the following listing, which starts a separate thread to display the message.

Listing 1.1. A simple Hello, Concurrent World program



Second, the code for writing the message has been moved to a separate function . This is because every thread has to have an *initial function*, which is where the new thread of execution begins. For the initial thread in an application, this is main(), but for every other thread it's specified in the

constructor of a std::thread object—in this case, the std::thread object named t has the new function hello() as its initial function.

This is the next difference: rather than just writing directly to standard output or calling hello() from main(), this program launches a whole new thread to do it, bringing the thread count to two—the initial thread that starts at main() and the new thread that starts at hello().

6

After the new thread has been launched , the initial thread continues execution. If it didn't wait for the new thread to finish, it would merrily continue to the end of main() and thus end the program

-possibly before the new thread had had a chance to run. This is why the call to join() is there - as described in <u>chapter 2</u>, this causes the calling thread (in main()) to wait for the thread associated with the std::thread object, in this case, t.

If this seems like a lot of work to go to just to write a message to standard output, it is—as described previously in section 1.2.3, 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. Later in the book, we'll work through examples that show scenarios where there's a clear gain to using multiple threads.