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";
}
```

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

```
#include <iostream>
#include <thread>                    <-1

void hello()                         <-2
{
    std::cout<<"Hello Concurrent World\n";
}

int main()
{
    std::thread t(hello);            <-3
    t.join();                        <-4
}
```

❶

The first difference is the extra `#include <thread>` . The declarations for the multithreading support in the Standard C++ Library are in new headers: the functions and classes for managing threads are declared in `<thread>`, whereas those for protecting shared data are declared in other headers.

❷

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

❸

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()`.

❸

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

❹

—possibly before the new thread had had a chance to run. This is why the call to `join()` is there — as described in chapter 2, 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.