Running your first multithread C++ program

Document name: “BDiF2015 – TN0001.pdf”
Date/revision: Wed 28th Jan 2015 / Rev A.
Author: Andrew Sheppard, © 2015.

1. Introduction

This technical note is part of the learning materials for the Baruch MFE “Big Data in Finance” (BDiF) course.

The BDiF course puts great emphasis on programs that execute in parallel. Many forms of data analysis follow a divide-and-conquer methodology and the only reasonable way to tackle Big Data is often by working on it in parallel in one way or another.

There are many ways in which to implement parallelism and the BDiF course covers most of them. This technical note covers multithreading on a single processor using the language features available in C++11. Support for parallelism in C++11 (hereafter just C++) is quite strong and that is one of the reasons that you will be using C++ throughout the BDiF course. Another reason for using C++ is because it is very common in quantitative finance and being a good C++ programmer is a valuable skill. This is another area of emphasis in the course. Namely, building practical skills that you can start applying today to Big Data problems (other types of problems too!).

2. Running Your First C++ Multithread Program

Here are step-by-step instructions to compiling, running and looking at the output of your first program:

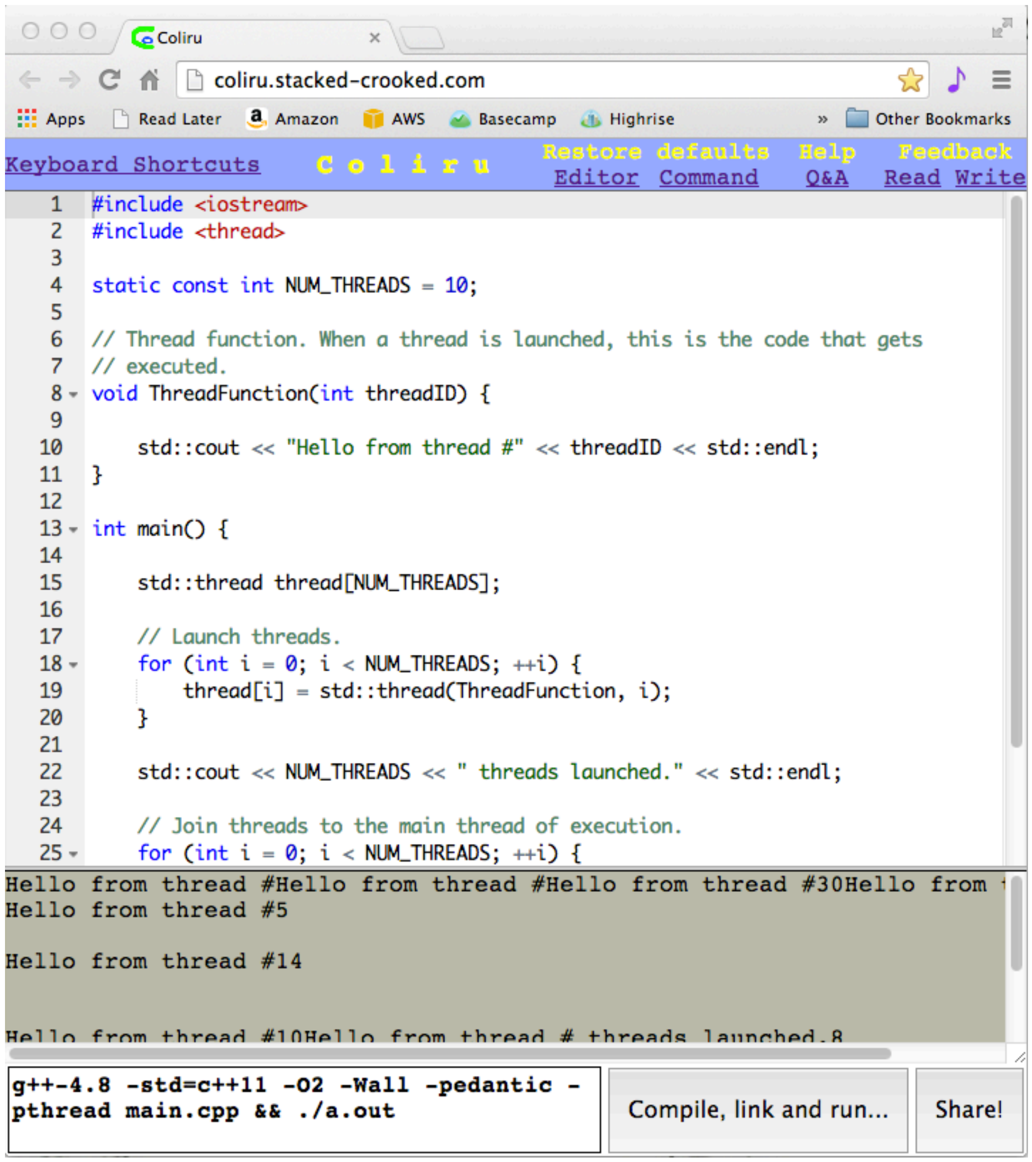
1. Open a web browser and type in the address http://coliru.stacked-crooked.com/ . This will open the window shown in Figure 1.
2. Cut & paste the code from **Appendix I** into the main panel.
3. Then click on the button labeled “Compile, link and run …”. This will run the program and the output will appear in the panel directly below the code.
4. That’s it! You’ve run your first multithread C++ program.

Alternatively, follow these steps:

1. Click on this link: load code in browser.
2. Click on the button labeled “Edit”.

© 2015 Andrew Sheppard, all rights reserved.

3. Then click on the button labeled “Compile, link and run ...”.
4. That’s it! You’ve run your first multithread C++ program.



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

static const int NUM_THREADS = 10;

// Thread function. When a thread is launched, this is the code that gets
// executed.
void ThreadFunction(int threadID) {

    std::cout << "Hello from thread #" << threadID << std::endl;
}

int main() {

    std::thread thread[NUM_THREADS];

    // Launch threads.
    for (int i = 0; i < NUM_THREADS; ++i) {
        thread[i] = std::thread(ThreadFunction, i);
    }

    std::cout << NUM_THREADS << " threads launched." << std::endl;

    // Join threads to the main thread of execution.
    for (int i = 0; i < NUM_THREADS; ++i) {
```

```
Hello from thread #Hello from thread #Hello from thread #30Hello from t
Hello from thread #5

Hello from thread #14


Hello from thread #10Hello from thread # threads launched.8
```

```
g++-4.8 -std=c++11 -O2 -Wall -pedantic -pthread main.cpp && ./a.out
```

Figure 1. Online C++ compiler “Coliru”.

Here’s the output that I got from the program (you’re output will likely be different for reasons that will become clear):

```
Hello from thread #Hello from thread #Hello from thread #0
3
Hello from thread #Hello from thread #24

1
Hello from thread #5
Hello from thread #6
Hello from thread #7
```

© 2015 Andrew Sheppard, all rights reserved.

```
Hello from thread #8
10Hello from thread #9
threads launched.
```

Probably not what you expected! The output is all over the place and the reason why is that the different threads of execution ran all at once and you cannot assume any particular order of execution. That is to say, one thread could be in the process of writing to output when another thread comes along and starts writing to output too. All the output is there, it's just garbled together because when several threads are running at the same time, each can be writing to output at different or the same time. This leads to our first rule of parallel programming.

Rule 1: In a multithread program you cannot assume any particular order of execution of the threads, unless you enforce it programmatically.

3. **Running Your Second C++ Multithread Program**

Here are step-by-step instructions to compiling, running and looking at the output of your second program:

1. As before, open a web browser and type in the address http://coliru.stacked-crooked.com/ .
2. Cut & paste the code from **Appendix II** into the main panel.
3. Then click on the button labeled "Compile, link and run …".
4. That's it! You've now run your second multithread C++ program.

Alternatively, follow these steps:

1. Click on this link: load code in browser.
2. Click on the button labeled "Edit".
3. Then click on the button labeled "Compile, link and run …".
4. That's it! You've now run your second multithread C++ program.

Let's again look at the output:

```
10 threads launched.
Hello from thread #0
Hello from thread #1
Hello from thread #2
Hello from thread #3
Hello from thread #4
Hello from thread #5
Hello from thread #6
Hello from thread #7
Hello from thread #8
Hello from thread #9
```

This time the output looks a lot more sensible and the reason why is because we have allocated space for the output of each thread, and only at the end of the

© 2015 Andrew Sheppard, all rights reserved.

program have we output all the data as though each thread had run in sequence. We know that all 10 threads run but we don't know in what order; and in this particular case we don't really care. This leads to our second rule of parallel programming.

Rule 2: A parallel program's output should ensure two things: 1) that all the threads of execution completed successfully (and if not, tell you why they failed), and 2) that the output is structured in a way that is meaningful and useful to users.

4. **Summary**

The two code examples in this technical note are small and largely self-explanatory. Even so, you would do well to study the code in detail because they provide the absolute basics of how to write parallel code in C++. By replacing the `ThreadFunction()` function you could do meaningful work in parallel rather than just saying "Hello" many times, albeit in a parallel way.

There's a lot more to come as the BDiF course will give you lots of practice in perfecting your parallel programming skills, but this is a start.

© 2015 Andrew Sheppard, all rights reserved.

Appendix I

Source code for your first multithread C++11 program:

_____ SNIP ~ Cut below this line when cutting & pasting _____

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

static const int NUM_THREADS = 10;

// Thread function. When a thread is launched, this is the code
// that gets executed.
void ThreadFunction(int threadID) {

      std::cout << "Hello from thread #" << threadID << std::endl;
}

int main() {

      std::thread thread[NUM_THREADS];

      // Launch threads.
      for (int i = 0; i < NUM_THREADS; ++i) {
            thread[i] = std::thread(ThreadFunction, i);
      }

      std::cout << NUM_THREADS << " threads launched." << std::endl;

      // Join threads to the main thread of execution.
      for (int i = 0; i < NUM_THREADS; ++i) {
            thread[i].join();
      }

      return 0;
}
```

_____ SNIP ~ Cut above this line when cutting & pasting _____

© 2015 Andrew Sheppard, all rights reserved.

Appendix II

Source code for your second multithread C++11 program:

_____ SNIP ~ Cut below this line when cutting & pasting _____

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

static const int NUM_THREADS = 10;
static std::string output[NUM_THREADS];

// Thread function. When a thread is launched, this is the code that
// gets executed.
void ThreadFunction(int threadID) {

      output[threadID] = "Hello from thread #" +
std::to_string(threadID) + '\n';
}

int main() {

      std::thread thread[NUM_THREADS];

      // Launch threads.
      for (int i = 0; i < NUM_THREADS; ++i) {
            thread[i] = std::thread(ThreadFunction, i);
      }

      std::cout << NUM_THREADS << " threads launched." << std::endl;

      // Join threads to the main thread of execution.
      for (int i = 0; i < NUM_THREADS; ++i) {
            thread[i].join();
      }

      // Even though threads ran independently and asynchronously,
      // output the results as though they had run in serial fashion.
      for (int i = 0; i<NUM_THREADS; i++) std::cout << output[i];

      return 0;
}
```

_____ SNIP ~ Cut above this line when cutting & pasting _____

© 2015 Andrew Sheppard, all rights reserved.