# EEEN30052 Concurrent Systems

## Coursework Assignment

# Overview and Part 1

**Dr. Peter Green**
**Department of Electrical and Electronic Engineering**
**University of Manchester**
**Office: Sackville E8b**
**peter.green@manchester.ac.uk**

# 1. Introduction

The coursework for this unit is in three parts that will fit together into the **simulation** of **four-by-one hundred metres sprint relay race**[1]. The race consists of `NO_TEAMS` competing teams and each team has `NO_MEMBERS` members. `NO_MEMBERS` is, of course, four.

The three parts of the coursework are as follows:

Part 1: This is concerned with creating and starting a two dimensional array of threads, interrogating their properties, and using random numbers and time delays. Optionally it can also involve the use of C++ maps. **Part 1 is worth 40% of the assignment marks.**

Part 2: This involves synchronising threads at the start of the race, at the baton exchanges and ensuring that there is only one winner – photo-finishes are not allowed in this simulation! **This is worth 40% of the assignment marks.**

Part 3: Integrates the code from parts 1 and 2 into the compete simulation. **This is worth 20% of the assignment marks**.

This document provides the details for Part 1.

# 2. Overview of Part 1

The objective of this part is to write a C++ program that declares a two dimensional array of thread objects, each of which executes the function `run`. The initial version of `run` to be developed in Part 1 has the following prototype:

```
void run(Competitor& c);
```

Class `Competitor` is provided for you to use. It is discussed in Section 4 below. Note that it will require a small, but non-trivial extension. Objects of class `Competitor` identify the athletes in the race.

`run` should sleep for a random period that is compatible with the time taken to run 100 m by a professional athlete[2]. On waking, the thread should print the message specifying the name and team of the athlete identified in the Competitor object and the time duration of the sleep/sprint.

To create an array of threads, you will need to use **class `thread`'s default constructor** in the array declaration. The default constructor is briefly introduced near the end of Lecture 4 (slide Threads, Thread Objects and Move Assignment). A thread must then be assigned to each element of the array. **You are expected to do some Internet research on the exact details of how to accomplish this**, although it is straightforward.

[1] https://en.wikipedia.org/wiki/4_%C3%97_100_metres_relay.

[2] The women's world record for the 100 m sprint is 10.49 s, set by Florence Griffith-Joyner (US). The men's record is 9.58 s, set by Usain Bolt (Jamacia).

The slide mentioned above also provides an example of how to determine the **identifier** given to a thread by the underlying run-time system.

## 3. Random Numbers

Although the standard C/C++ `rand` and `srand` functions have limitations, they are suitable for use in this program. They can be accessed by:

```
#include <cstdlib>
```

Short notes on `rand` and `srand`, and an example program can be found in the Assignment folder on Blackboard.

## 4. `class Competitor`

This allows the program to specify the name of an athlete and the name of the team to which they belong. The basic version of this class, which is usable at the start of the coursework is as follows:

*Competitor.h*

```
#pragma once
#include <string>
using namespace std;
class Competitor { // created in main and never updated, passed to a thread, placed in map
private:
    string teamName;
    string personName;
public:
    Competitor();
    Competitor(string tN, string pN);
    void setTeam(string tN);
    string getTeam();
    void setPerson(string pN);
    string getPerson();
    static Competitor makeNull();
    void printCompetitor();
};
```

*Competitor.cpp*

```
#include "Competitor.h"
#include <iostream>
Competitor::Competitor() {}
Competitor::Competitor(string tN, string pN) : teamName(tN), personName(pN) {}
void Competitor::setTeam(string tN) { teamName = tN; }
string Competitor::getTeam() { return teamName; }
void Competitor::setPerson(string pN) { personName = pN; }
string Competitor::getPerson() { return personName; }
Competitor Competitor::makeNull() { return *(new Competitor(" ", " ")); }
void Competitor::printCompetitor() {
    std::cout << "Team = " << teamName << " Person = " << personName << std::endl;
}
```

The class has two data members of type string: `teamName` and `personName`, that enable individual athletes to be specified in terms of their team and name e.g. Jamacia and Bolt. There is a default constructor and a constructor that allows these data members to be initialised. `set` and `get` functions that are common in data holding classes to modify and return the values of data members are also included. `printCompetitor` simply prints the current values of `teamName` and `personName`.

The `makeNull` member function returns a 'null `Competitor`' object whose data members are both a single character of white space. It can be useful when writing a class to define and implement a null object, and this is the case here, as discussed below.

## 5. First Version of the Program

A skeleton of the first version of the program is shown and explained below

```
1.    #include <iostream>
2.    #include <string>
3.    #include //other .h files

4.    const int NO_TEAMS = 4;        // number of teams in the race
5.    const int NO_MEMBERS = 4;      // number of athletes in the team

6.    void run(Competitor& c) {
7.       // store thread id and competitor in a map
8.       // delay for random period
9.       // print message stating which competitor has just finished
10.   }

11.   int main() {
12.      thread theThreads[NO_TEAMS][NO_MEMBERS];
13.      Competitor teamsAndMembers[NO_TEAMS][NO_MEMBERS];
14.      // define elements of teamsAndMembers
15.      // create threads (elements of theThreads)
16.      // join threads
17.   }
```

Line 3: You will need to `#include` other header files to complete this part of the coursework.
Line 4: Declares a global constant representing the number of teams in the race.
Line 5: Declares a global representing the number of athletes in each team.
Line 6: This is the function executed by each of the threads. It must be passed a `Competitor` object that defines which team and athlete the thread represents.
Line 7: The thread id and Competitor should be stored in a **`map`** container. See Section 6
Line 8: This delay represents the time taken for an athlete to run 100 m. This will be a random number between the world record time and 15 s.
Line 9: This involves calling the `printCompetitor` member function for the Competitor object passed to run.
Line 12: The declaration of the two dimensional array of `threads`.
Line 13: The declaration of the two dimensional array of `Competitors`.
Line 14: This will be multiple lines in your code, each line defining a `Competitor` in terms of their team name and person (family) name.
Line 15: Again, this will be multiple lines within your code that actually create the `threads`.

Line 16: All the `threads` should be joined. Multiple lines in your code.

You should consider whether the Competitor class should be thread-safe. See Section 6.3.

The map in line 7 is not wholly necessary for the simulation. It can be omitted without but you will need to provide equivalent functionality. Not using a map will lead to a reduction in marks though.

## 6. Maps

Object Oriented Programming uses the idea of **Container Classes** – classes that store instances of objects of another class. Buffers and stacks are examples of Containers that you have already encountered, but there are many others, including sets, lists, trees and graphs.

Different Container Classes efficiently support different access patterns to the data stored in them, and a key programming skill is choosing a good container for a particular application. Buffers support FIFO access that is needed in Producer-Consumer problems, Stacks support LIFO access which is needed in compilers and navigation applications, amongst others.

C++ is supported by the **Standard Template Library (STL)** which provides a large library of classes, many of which are Container Classes. The library is based on **templates** so that the type of object stored can be customised for a particular application.

This part of the assignment makes use of the STL library **map class**. A map is an associative container that uses a **key** to locate a mapped value. In a sense, it provides an abstraction of an array. In an array, the desired element is specified by an integer index. In a map the 'index' is the key and can be of any type. Each mapped value is associated with a unique key.

An example of a map is shown below[3]. Each map entry is a pair – the first item (the key) is a Roman numeral between one and ten. The second item in the pair is the text representing the same number in decimal. In a program that used this map, both the Roman numeral and the text decimal number would be `strings`. The map allows the program to specify the Roman numeral and to find the corresponding text name.

| Roman numeral (key) | Text decimal number (mapped value) |
|---|---|
| i | one |
| ii | two |
| iii | three |
| iv | four |
| v | five |
| vi | six |
| vii | seven |
| viii | eight |
| ix | nine |
| x | ten |

[3] Not a very useful one!

In this assignment, **the key is the system thread id**, and the **data element associated with the key is the Competitor**. Why is this helpful? Well, a thread can discover its id via the `get_id` function from the `this_thread` namespace (see lecture 4). However, a thread cannot know the `Competitor` that it represents. Hence the 'mapping' between thread id and Competitor is stored in a map.

When a thread needs to know which Competitor it represents (e.g., for providing output that can be understood by users, such as printing the finishing order of the teams), it finds its id by calling `get_id` and then requests the map to provide the Competitor that corresponds to the thread id.

### 6.1 Using Maps in this Assignment

In order to use maps in this application it is necessary to use a 'wrapper class' – a class that is based on the STL `map` but which provides some extra functionality. This is called `ThreadMap`.

Like most classes in the STL, maps have many member functions. However, for this assignment you will only need to use[4]:

- `begin()` – Returns an **iterator** to the first element in the map
- `end()` – Returns an **iterator** to the notional element that follows last element in the map
- `size()` – Returns the number of elements in the map
- `insert(keyvalue, mapvalue)` – Adds a new pair to the map
- `find(keyvalue)` – Returns an **iterator** that indicates the map entry containing the key value. If the key value is not present in the map, find returns an iterator to end() .

An iterator can be thought of as a **pointer** which can be moved to point to each map element in turn. Hence iterators can be used to search for an entry (as with the `find` function), or to 'visit' every element e.g., if the contents of the map are to be printed out.

### 6.2 Wrapper Class – `ThreadMap`

Here is the header file for the wrapper class ThreadMap:

```
1. #include <map>
2. #include "Competitor.h"
3. ...
4. class ThreadMap {
5. private:
6.     std::map <std::thread::id, Competitor> threadComp;
7. public:
8.     ThreadMap();
9.     void insertThreadPair(Competitor c);
10.    Competitor getCompetitor();
11.    void printMapContents();
12.    int ThreadMapSize();
13. };
```

[4] See https://thispointer.com/stdmap-tutorial-part-1-usage-detail-with-examples/

Line 1: This must be included to enable the creation of a map objects.
Line 2: The map will store Competitor objects, so this is needed.
Line 6: This declares a map called `threadComp`, whose entries are thread id/Competitor pairs, as specified by the types within the angle brackets.
Line 8: Constructor.
Line 9: This function inserts a thread id/`Competitor` pair into the map `threadComp.`
Line 10: This member function returns the Competitor corresponding to the id of the thread that calls it.

The following is **part** of `ThreadMap.cpp`:

```
1. #include "ThreadMap.h"

2. ThreadMap::ThreadMap() {}; // constructor

3. void ThreadMap::insertThreadPair(Competitor c) {
       // create a threadID, Competitor pair using a call to std::make_pair
       // store the pair in the map using the map insert member function
   }

4. Competitor ThreadMap::getCompetitor() {
5.    std::map <std::thread::id, Competitor>::iterator
                                    it = threadComp.find(std::this_thread::get_id());
6.    if (it == threadComp.end())
7.        return Competitor::makeNull();
8.    else
9.        return it->second;       // the second item in the pair (the Competitor)
10.}

11.void ThreadMap::printMapContents() {
12.    std::cout << "MAP CONTENTS:" << std::endl;
13.    std::map <std::thread::id, Competitor>::iterator it = threadComp.begin();
       // you need to write the rest!
14.    cout << "END MAP CONTENTS" << endl;
15.}

16. int ThreadMap::ThreadMapSize() { return threadComp.size(); }
```

Line 3: Writing this function is part of the assignment

Line 4: This function searches the map for the Competitor corresponding to the id of the thread that calls it.

Line 5: This creates an iterator that will be used to search for thread id/Competitor pairs by calling the `find` function of class `map`. If `find` returns end() (see above) then the thread id is NOT present in the map and so a '`null Competitor`' object is returned (lines 6 and 7)[5]. If the thread id is found, then the second element of the pair (the `Competitor`) is returned. This is what `it->second` does (line 9).

[5] This is the reason for including `makeNull` in class `Competitor`.

Line 13. This creates an iterator that is used to move through the map, allowing each element to be printed out. It is initialised to indicate the first element of the map (`it = threadComp.begin();`).

The code to move through the map is part of the assignment. This is very straightforward, particularly if you recall that it is possible in C/C++ to iterate through an array using a pointer rather than an array index.

### 6.3 Thread Safety

Besides writing some parts of the `ThreadMap` class, you should consider whether part or all of the class needs to be **thread-safe**. You should also consider this issue with respect to class Competitor. Thread safety ensures that objects of a class can be used reliably in the presence of multiple threads without suffering from concurrency-related problems.

If you decide that Thread Safety is relevant to either or both of the above classes, then you should use appropriate techniques to ensure it.