Exercise 9.1

Verify explicitly the statements made about basic operation of using threads:

a) ill-defined order of printout

b) run-time errors

It is instructive to demonstrate explicitly the fact that two threads can execute in parallel

c) create a program that starts two threads, with different functions that are demonstrably executed in parallel:

- one printing some output, then sleeping some given amount of time, then printing something again
- one sleeping a small amount of time, then printing something else
- (in the code executed) within a thread, sleeping for n seconds can be achieved using

```
#include <chrono>
…
this_thread::sleep_for(std::chrono::seconds(n));
```

  - NB in a single-threaded application, the concept of a thread still applies

d) complete the example demonstrating how to modify an object passed by reference. Can you even pass an object by value?

Exercise 9.2

Like in exercise 9.1c, create a program that creates two threads, but in this case have them communicate in a simple “producer-consumer” model

a) let the “messages” that are passed between the two threads be variables of a type T of your choice (int, double, your favourite user-defined type), and create a std::queue<T> (defined in <queue>) that is accessible to both produce() (in the “sending” thread) and consume() (in the receiving thread)

- notable property of a queue: it has a first-in-first out property (contrary to a stack)

```
queue<double> q;
…
double a = 3.14159; q.push(a);
…
auto b = q.front(); q.pop();
```

Exercise 9.2 (continued)

b) have producer() generate/compute such variables, and add some randomness to the time it takes to do so, e.g. through a random number generator

```
#include <random>
…
random_device d;
mt19937 mt(d);
int t_max = 3000;
uniform_int_distribution<> distr(0., t_max);
…
while (true) {
    int n = distr(mt);
    this_thread::sleep_for(chrono::milliseconds(n));
    …
}
```

“Mersenne Twister” pseudo-random number generator

uniform distribution between 0 and 3000

generate one random number from the specified distribution

c) use a mutex, a unique_lock / lock_guard, and a condition_variable, as discussed during the lecture, to communicate the generated values

- i.e., have produce() add them to the queue and consume() remove them from it again

d) have consume() do something with the communicated values so as to demonstrate what happens

e) optionally, extend the setup to a single producer but multiple consumers

- only useful if it takes longer for a consumer to deal with a single “message” than for the producer to produce one