Exercise 5 C++ Multitasking

1. Tasks 101: Running Functions asynchronously)

For this exercise you need to understand C++11 *futures*, `std::async` and thread/task launching policies. This exercise involves running synchronous and asynchronous functions based on various launch policies. The functions should have both `void` and non-`void` return types. The objective is in showing how the tasks execute and exchange information.

Answer the following questions:

a) Create two functions having the following signatures:

```
void func1()
{

}

double func2(double a, double b)
{

}
```

You may insert any code you see fit into the body of these functions for the purpose of this exercise.

b) Use `std::async` (default settings) to launch `func1` and `func2`. Get the results of the computations and print them when applicable. Check the validity of the associated future before and after getting the result.

c) What happens if you try to get the result of a `std::future` more than once?

d) Now test the same code using the launch parameter `std::launch::async`. Do you notice any differences?

e) We now wish to asynchronously call a function at some time later in the client code (*deferred/lazy evaluation*). Get the result of the function and check that it is the same as before.

2. (Shared Futures 101)

We process the outcome of a concurrent computation more than once, especially when multiple threads are running. To this end, we use `std::shared_future` so that we can make multiple calls to `get()`. Answer the following questions:

a) Create the following *shared future* by calling the appropriate constructor:
- Default instance.
- As a shared future that shares the same state as another shared state.
- Transfer shared state from a 'normal' future to a shared future.
- Transfer shared state from a shared future to a shared future.

b) Check the member functions in `std::future` that they are also applicable to `std::shared_future` .

c) Test what happens when you call `get()` twice on a shared future.

d) Create a shared future that waits for an infinite loop to finish (which it never does). To this end, use `wait_for` and `wait_until` to trigger a time out.

3. (C++11 Promises 101)

A *promise* is the counterpart of a future. Both are able to temporarily hold a shared state. Thus, a promise is a general mechanism to allow values and exceptions to be passed *out of threads*. A promise is the "push" end of the promise-future communication channel.

Answer the following questions:

a) Create a default promise, a promise with an empty shared state and a promise based on the move constructor.

b) Create a promise with `double` as stored value. Then create a future that is associated with the promise.

c) Start a thread with the new future from part b). Create a thread function that uses the value of the shared data.
d) Use the promise to set the value of the shared data.

4. (Concurrency versus Parallelism, Quiz)
Which of the following statements are applicable to concurrency or parallelism (or both)?
a) Use of shared resources.
b) Goals are correctness, performance (throughput) and robustness.
c) Reduce latency (*latency* is the fixed cost of servicing a request).
d) Prevent *thread starvation* (never allow a program to become idle).
e) Concurrency is an optimisation, parallelism is a a functional requirement.

5. (Data Parallelism versus Task Parallelism, Quiz)
Which of the following statements accurately describe data parallelism best and which accurately describe task parallelism?
a) Synchronous/asynchronous computation.
b) Optimum load balancing/load balancing depending on hardware availability and scheduling algorithms.
c) Different operations performed on the same data.
d) Degree of parallelisation depends on number of independent tasks/input data size.
e) Less/more speedup use cases.