std::async

This lesson offers an introduction to std::async, which is used in C++ for multithreading.

WE'LL COVER THE FOLLOWING

Introduction

`std::async` behaves similar to an asynchronous function call. This function call takes a ***callable*** together with its arguments. `std::async` is a variadic template and can, therefore, take an arbitrary number of arguments. The call to `std::async` returns a future object `fut`. That's your handle for getting the result via `fut.get()`.

std::async should be your first choice

The C++ runtime decides if `std::async` is executed in a separate thread. The decision of the C++ runtime may depend on the available number of CPU cores, the utilization of your system, or the size of your work package. By using `std::async`, you only specify the task that will run. The C++ runtime automatically manages the creation and the lifetime of the thread.

Eager or Lazy Evaluation

Optionally, you can specify a launch policy for `std::async`.

With the launch policy, you can explicitly specify whether the asynchronous call should be executed in the same thread (`std::launch::deferred`) or in

another thread (`std::launch::async`).

> **Eager versus lazy evaluation**
>
> Eager and lazy evaluations are two orthogonal strategies to calculate the result of an expression. In the case of eager evaluation, the expression will be evaluated immediately - in the case of lazy evaluation, the expression will only be evaluated if needed. Eager evaluation is often called **greedy evaluation**, and lazy evaluation is often called **call-by-need**. With lazy evaluation, you save time and compute power as there is no evaluation on suspicion.

By default, the launch policy of `std::async` is `std::launch::async` | `std::launch::deferred`, meaning that the system determines if it will execute the work package eagerly or lazily.

What is special about the call `auto fut= std::async(std::launch::deferred, ... )` is that the promise will not be executed immediately. The call `fut.get()` starts the promise lazily, meaning that the promise will only run if the future asks for the result via `fut.get()`.

Let's take a look at an example of `std::async` in the next lesson.