async: Start Policy

This lesson gives an overview of the start policy used with std::async in C++ for concurrency.

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

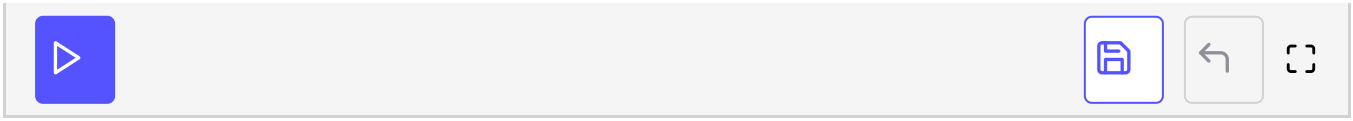
> **Eager versus lazy evaluation**
>
> Eager and lazy evaluations are two orthogonal strategies to calculate the result of an expression. In the case of eager evaluation, the expression will be evaluated immediately; In the case of lazy evaluation, the expression will only be evaluated if needed. Eager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

By default, `std::async` executes its work package immediately. This is why it's called an eager evaluation.

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, i.e., the promise will only run if the future asks via `fut.get()` for the result.

```
// asyncLazy.cpp

#include <chrono>
#include <future>
#include <iostream>

int main(){

  std::cout << std::endl;
```

```
  auto begin= std::chrono::system_clock::now();

  auto asyncLazy=std::async(std::launch::deferred,
                            []{ return std::chrono::system_clock::now(); });

  auto asyncEager=std::async(std::launch::async,
                             []{ return std::chrono::system_clock::now(); });

  std::this_thread::sleep_for(std::chrono::seconds(1));

  auto lazyStart= asyncLazy.get() - begin;
  auto eagerStart= asyncEager.get() - begin;

  auto lazyDuration= std::chrono::duration<double>(lazyStart).count();
  auto eagerDuration=  std::chrono::duration<double>(eagerStart).count();

  std::cout << "asyncLazy evaluated after : " << lazyDuration
            << " seconds." << std::endl;
  std::cout << "asyncEager evaluated after: " << eagerDuration
            << " seconds." << std::endl;

  std::cout << std::endl;

}
```



Both `std::async` calls (lines 13 and 16) return the current time point, but the first call is lazy while the second is eager; the short sleep of one second in line 19 makes that obvious. The call `asyncLazy.get()` in line 21 will trigger the execution of the promise in line 13 - the result will be available after a short nap of one second (line 19). This is not true for `asyncEager`, as `asyncEager.get()` gets the result from the immediately executed work package.

We do not have to bind a future to a variable.

In the next lesson, we'll discuss fire and forget, which are used with `std::async` in C++ for multithreading.