Attaching Extended Futures

This lesson discusses how one extended future can be attached to another in C++20.

WE'LL COVER THE FOLLOWING

- `std::async`, `std::packaged_task`, and `std::promise`
- Creating new Futures
  - `std::make_ready_future` and `std::make_exceptional_future`
  - `std::when_any` and `std::when_all`
  - `std::when_all`
  - `std::when_any`

The method `then` empowers you to attach a future to another future. It often happens that a future will be packed into another future. The job of the unwrapping constructor is to unwrap the outer future.

The proposal N3721

Before I show the first code snippet, I have to say a few words about proposal N3721. Most of this section is from the proposal on "Improvements for std::future and Related APIs", including my examples. Strangely, the original authors frequently did not use the final `get` call to get the result from the future. Therefore, I added the `res.get` call to the examples and saved the result in a variable `myResult`. Additionally, I fixed a few typos.

```
#include <future>
using namespace std;
int main() {
  future<int> f1 = async([]() { return 123; });
  future<string> f2 = f1.then([](future<int> f) {

    return to_string(f.get());      // here .get() won't block
  });

  auto myResult= f2.get();

}
```

There is a subtle difference between the `to_string(f.get())` call (line 7) and the `f2.get()` call in line 10. As I already mentioned in the code snippet, the first call is non-blocking/asynchronous and the second call is blocking/synchronous. The `f2.get()` call waits until the result of the future-chain is available. This statement will also hold for chains such as `f1.then(...).then(...).then(...).then(...)` as it will hold for the composition of extended futures. The final `f2.get()` call is blocking.

`std::async`, `std::packaged_task`, and `std::promise`

There is not much to say about the extensions of `std::async`, `std::package_task`, and `std::promise`. I only have to add that in C++20, all three return extended futures.

The composition of futures is more exciting. In the next lesson, I will discuss how we can compose asynchronous tasks.

Creating new Futures

C++20 gets four new functions for creating special futures. These functions are `std::make_ready_future`, `std::make_execptional_future`, `std::when_all`, and `std::when_any`.

First, let's look at the functions `std::make_ready_future` and `std::make_exceptional_future`.

`std::make_ready_future` and `std::make_exceptional_future`

Both functions create a future that is immediately ready. In the first case, the future has a value; in the second case, an exception. Therefore, what seems to be strange at first actually makes a lot of sense. In C++11 the creation of a ready future requires a promise. This is necessary even if the shared state is immediately available.

```
future<int> compute(int x) {
  if (x < 0) return make_ready_future<int>(-1);
  if (x == 0) return make_ready_future<int>(0);
  future<int> f1 = async([]() { return do_work(x); });
  return f1;
}
```

Hence, the result must only be calculated by a promise if `(x > 0)` holds. Now let's begin with future composition. A short remark: both functions are the pendant to the return function in a ***monad***.

`std::when_any` and `std::when_all`

Both functions have a lot in common. First, let's look at the input.

```
template < class InputIt >
auto when_any(InputIt first, InputIt last)
    -> future<when_any_result<std::vector<typename std::iterator_traits<InputIt>::value_type>

template < class... Futures >
auto when_any(Futures&&... futures)
    -> future<when_any_result<std::tuple<std::decay_t<Futures>...>>>;

template < class InputIt >
auto when_all(InputIt first, InputIt last)
    -> future<std::vector<typename std::iterator_traits<InputIt>::value_type>>;

template < class... Futures >
auto when_all(Futures&&... futures)
    -> future<std::tuple<std::decay_t<Futures>...>>;
```

Both functions accept a pair of iterators for a future range or an arbitrary number of futures. The big difference is that in the case of the pair of iterators, the futures have to be of the same type; while in the case of the arbitrary number of futures, the futures can have different types and even `std::future` and `std::shared_future` can be used.

The output of the function depends on whether a pair of iterators or an arbitrary number of futures (variadic template) was used; either way, both functions return a future. If a pair of iterators were used, you would get a future of futures in an `std::vector`: `future<vector<future<R>>>`. If you use a variadic template, you will get a future of futures in a `std::tuple`: `future<tuple<future<R0>, future<R1>, ... >>`.

This covers their commonalities. The future that both functions return will be ready if all (`when_all`) or any (`when_any`) of the input futures are ready. The next two examples show the usage of `std::when_all` and `std::when_any`.

`std::when_all`

```
#include <future>

using namespace std;

int main() {

  shared_future<int> shared_future1 = async([] { return intResult(125); });
  future<string> future2 = async([]() { return stringResult("hi"); });

  future<tuple<shared_future<int>, future<string>>> all_f =
        when_all(shared_future1, future2);

  future<int> result = all_f.then(
        [](future<tuple<shared_future<int>, future<string>>> f){
          return doWork(f.get());
        });

  auto myResult = result.get();

}
```

The future `all_f` (line 10) composes both the future `shared_future1` (line 7) and `future2` (line 8). The future `result` in line 13 will be executed if all underlying futures are ready. In this case, the future `all_f` in line 15 will be executed. The result is in the future `result` and can be used in line 18.

`std::when_any`

The future in `when_any` can be taken by `result` in line 11 below. `result` provides the information indicating which input future is ready. If you don't use `when_any_result`, you have to ask each future if it is ready - which is tedious.

```
#include <future>
#include <vector>

using namespace std;

int main(){

  vector<future<int>> v{ .... };
  auto future_any = when_any(v.begin(), v.end());

  when_any_result<vector<future<int>>> result = future_any.get();

  future<int>& ready_future = result.futures[result.index];

  auto myResult = ready_future.get();

}
```

`future_any` is the future that will be ready if one of its input futures is ready. `future_any.get()` in line 11 returns the future `result`. By using `result.futures[result.index]` (line 13) you have the `ready_future`, and thanks to `ready_future.get()`, you can ask for the result of the job.