Coroutines

Yehezkel Bernat

yehezkelshb@gmail.com

Core C++ Meetup, Sep. 2018

The Future of `future`

Yehezkel Bernat

yehezkelshb@gmail.com

Core C++ Meetup, Sep. 2018

Intro

Agenda

- Today we'll learn about coroutines
  - Coroutines are a new tool for async operations
  - It's been published as a TS in 2017-12-05
  - Updated a few times since then
  - Considered to be merged into C++20
- We'll focus on the user side
- Non-agenda items:
  - How to write a library type for using coroutines
  - How the compiler transforms our code
  - (See the links in the references for this kind of info)

Playground rules / assumptions

- I assume you know about:
  - threads and multi-threading environment,
  - data races (race conditions), and
  - the importance of synchronization tools
- Please ask questions

Motivation

`std::future` - background

- Currently, the best existing tool in C++ for passing a value from one thread to another
  - And it doesn't say much
- In short:
  - `std::future` is the reader end
  - Can be created from `std::promise`, which is the writer end
  - They share a shared state that can hold a value or an exception (both coming from the `std::promise` side)
  - It's possible to wait on the `std::future`, to check if it's ready and of course to get the value (or exception) from it
  - There are tools, `std::package_task` and `std::async`, to wrap around an existing function and insert the returned value (or the thrown exception) into a shared state that can be read from an `std::future`

Example

```
int divide(int dividend, int divisor) {
    if (divisor == 0)
        throw std::range_error("div by zero");
    return dividend / divisor;
}

void f() {
                int  answer =

    // do a lot of other things while the
    // computation is done in the background
    assert(answer       == 42);
}
```

Two async operations

```
void g() {
    std::future<int> answer1 =
        std::async(std::launch::async,
                   divide, 126, 3);
    std::future<int> answer2 =
        std::async(std::launch::async,
                   divide, 168, 4);
    // ...
    assert(answer1.get() == answer2.get());
}
```

- Life is good

Two **depended** async operations

- (From here on we assume `divide` is inherently async and returns `std::future<int>`)

```
void h() {
  auto answer1 = divide(168, 2);
  auto answer2 = divide(              , 2);
  // ...
  assert(answer2.get() == 42);
}
```

- Oops, now it waits for `answer1` to be ready before starting the next async operation
- We can create an overload of `divide()` that takes a `std::future<int>`
  - but where is the fun in it?
- Seriously, what about `std::future<short>`?

Two depended async operations – solution

```
void h() {
    auto answer = std::async([] {
        auto ans = divide(168, 2);
        return divide(ans.get(), 2);
    });
    // ...
    assert(answer.get() == 42);
}
```

- It works, but costs us an additional thread!
- This thread is just sitting there, waiting to pass the value around

Proposed std::future::then

- Adds the notion of “continuation”
- We can attach the next async operation to the future to run when it’s ready

```
void h() {
    auto answer = divide(168, 2)
        return divide(ans.get(), 2);
    });
    // ...
    assert(answer.get() == 42);
}
```

- Now it may even reuse the first thread
- But there are a lot of issues around `.then()` (see P0701)
- And it gets complicated and uglier with more complex cases

Conclusion so far

- We need better tools for handling concurrency

Coroutines TS

TS

- TS = Technical Specification
- “Feature branch” of the standard, things that can be changed drastically, including breaking compatibility, changing or removing functionality or maybe not “merged” to IS (International Standard) at all
- Not mandatory to implement to be standard compliant
- But the whole point is with the hope that implementation will start appear, users will start using it in their use-cases and feedback from the field will start gathering up
- Library headers are under `experimental` directory and names are inside `std::experimental` namespace, but it doesn’t say anything about the quality of the implementation

Coroutines TS

- Originally published in 2017-12-05
- But implementation started while ago
- MSVC (VS)
  - VS2015 (or even VS2013 with Nov. 2013 CTP) – pre-TS experimental implementation
  - VS2017 – TS Compliant implementation (e.g. adding `co_` prefix)
  - Just add `/await` to the compiler flags
- Clang
  - Since clang 5
  - Use `-fcoroutines-ts -stdlib=libc++`
- gcc
  - Not yet (as of Sep. 2018)
- The relevant include is `<experimental/coroutine>`

Two depended async operations – coroutines-based solution

```
void h() {
    auto answer = []()                    {
        auto ans =          divide(168, 2);
            return         divide(ans, 2);
    }();
    // ...
    assert(answer.get() == 42);
}
```

- (Assuming that this `future<T>` is usable for coroutines (*Awaitable*))

So what happens there?

- The function starts to run until it encounters the first `co_await`
  - Which means it also started the async `divide()` call which returned `future<int>`
- Then, it's ***suspended*** and the caller can continue to run (or suspend or wait)
- When the `future<int>` returned by `divide()` is ready, it (possibly) ***resumes*** the coroutine running, which extracts the result from the `future<int>` into `ans`
- Then it calls the second `divide()` with this result…
- …and is suspended again on the new `future<int>` returned from this second call
- Eventually, when the second `divide()` returns, it extracts the result (which is part of what `co_await` does)…
- …and stores it (with `co_return`) in the `future<int>` it returned in the original invocation, where the caller can get the result from

What we have seen?

- 2 new keywords: `co_await` and `co_return`
  - We'll see a third one later
- co_await is usable on *Awaitable* types
  - For our discussion, it tautologically means "types that support using co_await on them"
- co_return knows how to pass a result into an appropriate *Promise* type
  - (It's not exactly like the `std::promise` we already know, but is close enough for our discussion)
- Any usage of any of these keywords makes the function a *coroutine*
- The compiler applies well-defined transformations on such a function, mainly around the enter/exit and with each occurrence of these `co_` keywords
- The semantics of the coroutine behavior is controller by the library type it returns!

Compiler + Library = WIN

- The compiler gives only a “low-level API”
- It’s almost unusable directly
- Higher-level API and semantics are defined by library writers
- We simply use these library types as suits our needs
- This separation between compiler work and library work enables wide variety of usages and semantics
  - Instead of baking a few specific semantics and use-cases into the language

Common use-cases

- There are operations that are async already, e.g. I/O (file or network) interface supplied by the OS, which are found at the end of the call stack
- The call stack starts with a thread, it can be `std::thread`, `std::async` or even the main thread itself, to wait for the chain of the async operations to complete
- In the middle, now we have much simpler (and usually more efficient) code, using coroutines

How to write a coroutine?

- Use a coroutine type as the return type (even if it doesn't return anything)
- Use `co_await` to wait on lengthy operations inside it
- Use `co_return` to return the result (if any)
- Or just throw on error
- Beware the life-time of the parameters!
  - By-value parameters are copied (except of some optimization cases) into the *coroutine frame*, which is controlled by the return type (or something related to it)
  - Pass by-ref only if you sure about it!
  - (The same as with any concurrency tool)
- Note that `co_await` may continue immediately if the result is ready

cppcoro

The Freedom to Choose

cppcoro

- The poster boy of what libraries can do with coroutines
- `task<T>` – lazily executed task that returns `T` (`void` by default)
  - Lazily executed means that it starts execution only when the task is awaited
- `shared_task<T>` – a `task<T>` that can be copied and awaited in multiple places
- Previously there were (eager) `task<T>` and `lazy_task<T>`, but the eager version was removed for safety and performance reasons
- cppcoro documentation states "[…] the only way to start the first/top-level `task`" is to use a thread (possibly the current one) to wait on it:

```
auto res = sync_wait(foo());
```

cppcoro – more awaitable types

- `single_consumer_event`
  - manual-reset event that supports awaiting
- `single_consumer_async_auto_reset_event`
  - the auto-reset version
- `async_manual_reset_event`
  - the variation that allows multiple waiting coroutines
- More…
  - (E.g. tools for file I/O)

Generators

Generators

- Very different use-case, also included in coroutines TS
- What if we need a function to generate more than a single value?
- It’s possible to (co_)return a container with all the values, but what if we want to generate them one at a time? What if we don’t know right ahead how many will be needed?
- This is where generators are coming
- And the promised third keyword: `co_yield`

Simple generator example

```
cppcoro::generator<std::uint64_t> fibonacci() {
    std::uint64_t a = 0, b = 1;
    while (true) {
        co_yield b;
        auto tmp = a; a = b; b += tmp;
    }
}

void usage() {
    for (auto i : fibonacci()) {
        if (i > 1'000'000) break;
        std::cout << i << std::endl;
    }
}
```

(Based on cppcoro documentation)

Combining `co_await` and `co_yield`

```
cppcoro::async_generator<int> ticker(int count, threadpool& tp) {
    for (int i = 0; i < count; ++i) {
        co_await tp.delay(std::chrono::seconds(1));
        co_yield i;
    }
}

cppcoro::task<> consumer(threadpool& tp) {
    auto sequence = ticker(10, tp);
    for co_await(std::uint32_t i : sequence) {
        std::cout << "Tick " << i << std::endl;
    }
}
```

(Based on cppcoro documentation)

cppcoro – generator types

- As we saw, there are:
- `generator<T>` – to produce values lazily and synchronously
  - It isn't possible to `co_await` in a function that returns it!
- `async_generator<T>` – similarly but asynchronously
  - `co_await` is possible inside the generator function and on the generator itself (actually, on its iterator operations)
- Additionally, it includes `recursive_generator<T>`
  - Like `generator<T>` but useful when the generator returns results (also) from another generator called inside

How to write a generator coroutine?

- Use an appropriate generator type as a return type
- Have a loop (possibly endless one) inside, probably
- Use `co_yield` to return the generated value(s)
- Use range-based for loop (possibly with `co_await`) to consume the generated values
  - Or use iterators directly, as

```
for co_await(declaration : range_init) statement
```

  - is equivalent to

```
auto && __range = range_init;
auto __begin = co_await begin_expr;
auto __end = end_expr;
for ( ; __begin != __end; co_await ++__begin) {
    declaration = *__begin;
    statement
}
```

  - (and for regular generator you don't even need `co_await`, it's all encapsulated in the `begin()` and `++operator()` methods)

Bonus Sections

Interesting/Amusing Usages

Qt events (signals and slots) with coroutines

- Instead of:

```
bool got_first_point{ false }; QPointF first_point;
QObject::connect(&w, &MyWidget::click, [&](QPointF p) {
  if (got_first_point) {
    got_first_point = false; w.setLine(first_point, p);
  } else {
    got_first_point = true; first_point = p;
  }
});
```

- Use something as simple as:

```
QPointF first_point  = co_await make_awaitable_signal(&w, &MyWidget::click);
QPointF second_point = co_await make_awaitable_signal(&w, &MyWidget::click);
w.setLine(first_point, second_point);
```

- (Link in the references)

Using `std::optional<T>` with `co_await`

- It's possible to write the needed machinery to enable `co_await` to take the value out of an `std::optional<T>` or return `std::nullopt` if it's empty

```
std::optional<Point> parse_point() {
    co_await parse_lit('(');
    auto x = co_await parse_int();
    co_await parse_lit(',');
    auto y = co_await parse_int();
    co_await parse_lit(')');
    co_return Point{ x, y };
}
```

- Instead of a code full with if-not-return-nullopt conditions
- (Link in the references)
- Actually, there is a better, more general, solution proposed
  - P0798 – Monadic operations for `std::optional` (e.g. `and_then()`, `or_else()`)
- And either way will be even better usable with things like `expected<T,E>` (P0323)

Possible Radical Changes

Core Coroutines proposal

- Not everyone likes the current state of coroutines
- Core Coroutines proposal (P1063) suggested using an operator instead of `co_await`
- The suggested operator is `[<-]`
  - (It also suggests as an extension to add `[->]` operator too)
  - Andrew Pardoe said (on reddit) that he suggested `co_co_await` instead
- It also radically changes the “compiler API”
- Presented in Rapperswil meeting (last June)
- But the concepts are still the same

Summary && References

Summary

- We saw today a new way to handle concurrency
- It simplifies a lot many use-cases
- It has a lot of possible performance improvements
- It’s already available to try it or even use

References

- https://blogs.msdn.microsoft.com/vcblog/category/coroutine
  - a few interesting blog posts on various usages of coroutines
- https://github.com/lewissbaker/cppcoro
  - rich library with many types for using with coroutines
- https://lewissbaker.github.io/
  - in-depth technical (but easy to follow) explanation of the mechanisms and interfaces defined in the TS and used by the library writers to interact with the compiler, by the author of cppcoro library
- http://jefftrull.github.io/qt/c++/coroutines/2018/07/21/coroutines-and-qt.html
  - using `co_await` with Qt signals
- https://www.reddit.com/r/cpp/comments/6ly4rz/it_had_to_be_done_abusing_co_await_for_optionals/
  - using `co_await` on `std::optional<T>`
- https://wg21.link/N4402
  - Resumable Functions (revision 4) - includes rational, explaining various terms etc.
- https://wg21.link/N4760
  - Latest TS draft (2018-06-24)