C++ course

Coroutines primer

Eduard Drusa

Nov 25th - 27th 2024

COROUTINES

Coroutines

Coroutine is a concept created back in ’50s

Coroutine is any construct that bears two main characteristics:

- Data local to the function execution are persisted across successive calls (alike `static` local variables)
- Function execution can be suspended and without reaching its end.
- Function execution can later be resumed continuing where it was suspended before.

Provides cleaner way to implementat some inherently concurrent tasks.

Officially available in C++ since C++20. Makeshift solutions available for years.

Can be implemented even in plain ANSI C.

Coroutines (2)

A coroutine in C++ is a function that uses any of the following keywords:

`co_return` similarly to `return`, completely quits the coroutine and optionally returns a value. Coroutine may not resume after `co_return` has been called.

`co_yield` suspends coroutine execution and optionally returns a value. Coroutine may be resumed where it left.

`co_await` suspends coroutine execution and optionally waits for input. Coroutine will be resume where it left once value is supplied.

While not executing, coroutine does not have any stack. Thus coroutine object is copyable.

Coroutines example

```
struct promise;
struct range : std::coroutine_handle<promise> { using promise_type = ::promise; };

struct promise {
    range get_return_object() { return {range::from_promise(*this)}; }
    std::suspend_always initial_suspend() noexcept { return {}; }
    std::suspend_always final_suspend() noexcept { return {}; }
    std::suspend_always yield_value(int i) { value = i; return {}; }
    void return_value(int i) { value = i; }
    void unhandled_exception() {}

    int value;
};
/* ... */
range r = [](int from, int to) -> range {
    while (from < to) { co_yield from++; } co_return from;
}(0, 100);

while (!r.done()) { r.resume(); std::cout << r.promise().value << std::endl; }
```

Code without coroutines

```
struct Range {
    Range(int from, int to): m_from{from}, m_to{to} {}
    bool operator()(int & out) {
        if (m_from < m_to) {
            out = m_from++;
            return true;
        }
        return false;
    }
    int m_from, m_to;
}
```

Sometimes we need code that is inherently concurrent in its execution.

Plain C or C++ do not provide elegant way to deal with such problems (at least from the caller’s point of view).

The `promise` type

Not to be mistaken with `std::promise`.

Controls the behavior of a coroutine.

Contains following members (at least):

- `initial_suspend` - controls what happens upon coroutine object creation.
- `final_suspend` - controls what happens upon coroutine termination via `co_return`.
- `yield_value` - controls behavior when coroutine suspends via `co_yield`.
- `return_value` or `return_void` - controls coroutine termination via `co_return`.
- `unhandled_exception` - controls what happens when unhandled exception is thrown.
- `get_return_object` - controls what object is returned to the caller upon coroutine creation.

The `std::coroutine_handle` type

Is standardized handle provided by the STL to interact with coroutines.

Provides following utilities:

- `operator()` / `resume()` - allows resuming coroutine from non-coroutine.
- `done()` / `operator bool()` - controls what happens upon coroutine termination via `co_return`.
- `promise()` - provides access to the ”promise” object of the coroutine.

Other than that, the coroutine handle is copyable.

Creating a coroutine instance

```
range r = [](int from, int to) -> range { /* ... */ }
```

Keyword `coroutine` is never used.

Compiler realizes that this *lambda* is a *coroutine* because it contains some of coroutine-exclusive keywords.

Here the return type of the lambda does not match apparent return type of the `co_yield` and `co_return` statements.

Compiler expects that the type returned by coroutine will contains `promise_type` type declaration.

At this point, the coroutine is optionally executed. If this happens or not, is driven by the `initial_suspend` member of the promise type.

What is `std::suspend_always`?

`std::suspend_always`

Is one of default *awaitables*. Its use indicates that upon reaching given suspension point, coroutine suspends.

Has a friend - `std::suspend_never`. Its use indicates that upon reaching given suspension point, coroutine **does not suspend**.

If `initial_suspend` returns `std::suspend_never` then the coroutine starts execution immediately after creation.

Developer can create their own awaitables.

Suspending a coroutine instance

```
co_yield 42;
```

Happens then `co_yield` is hit inside the coroutine.

`yield_value` method of promise type is called with argument to `co_yield` to handle what happens.

`yield_value` is expected to return *awaitable*.

Awaitable controls what happens next:

- If `std::suspend_always` is returned, then coroutine is suspended and control is returned to the caller.
- If `std::suspend_never` is returned, then coroutine continues executing.
- If custom awaiter is returned, magic can happen.

You may have more than one `yield_value` method accepting different types, potentially yielding different types from coroutine.

Terminating a coroutine instance

```
co_return 42;
```

Happens then `co_return` is hit inside the coroutine.

`return_value` method of promise type is called if argument to `co_return` was passed.

`return_void` method of promise type is called if no argument was passed to co$_r$*eturn*.*Theyarebothexpectedtoreturn*`awaitable`.

Awaitable controls what happens next:

- If `std::suspend_always` is returned, then coroutine is suspended and control is returned to the caller.
- If `std::suspend_never` will corrupt the coroutine state and should never be used here.
- If custom awaiter is returned, magic can happen.

You may have more than one `return_value` method accepting different types, potentially returning different types from coroutine.

The `awaitable` type

Is a custom type that implements *awaitable* semantics.

Allows fine control (and real magic) on what happens when.

Makes sense to be used with `co_await` keyword.

Must contain following members:

- `await_ready` - informs if the result being awaited for is already ready.
- `await_suspend` - controls what will happen upon coroutine awaiting input.
- `await_resume` - provides the return value of the `co_await` expression.

This is not syntactically checked (remember the named rules?), so failing to comply with these requirements ends up with weird errors.

Custom awaiter

```
struct awaitable {
        bool await_ready();
        auto await_suspend(std::coroutine_handle<> h);
        auto await_resume();
}
```

The return type of `await_suspend` determines what happens upon suspending:

`void` - control is immediately transferred to the caller (`std::suspend_always`).

`bool` - if `true` is returned, behaves as `void` type. If false is returned then coroutine is not suspended and continues execution.

If a coroutine handle of a coroutine is passed, then this coroutine is resumed instead.

Argument to the `await_suspend` is a coroutine handle of the currently suspended coroutine.

Simple awaiter to "call" coroutine from another

```
struct call_gate {
    bool await_ready() { return false; }
    auto await_suspend(std::coroutine_handle<> h) { return handle; }
    void await_resume() {}
    Coroutine_t handle;
}
```

Structure `call_gate` holds one data member `handle`.

This data member holds a coroutine object.

Call to `await_suspend` returns this coroutine object effectively passing control to the ”nested” coroutine.

Note that the current coroutine is still suspended, so the control does not need to return back to this coroutine. It is entirely up on programmer to define the behavior.

Simple coroutine to "return" to the calling coroutine

```
struct ReturnPromise { // available via coroutine.promise()
    std::coroutine<MyPromise> parent; /* default nullptr */
    struct final_awaitable {
        bool await_ready() { return false; }
        std::coroutine_handle<> await_suspend(handle_type h) {
            return h.promise().parent;
        }
        void await_resume() {}
    };
    final_awaitable final_suspend() { return {}; }
};
struct call_gate {
    bool await_ready() { return false; }
    auto await_suspend(std::coroutine_handle<> h) { handle.promise().parent = h; return handle; }
    void await_resume() {}
    Coroutine_t handle;
};
```

`await_suspend` stores the current coroutine into awaitable’s handle.
`await_resume` of nested coroutine returns stored coroutine when suspended.
Note that awaitable and promise can be mixed into one object.

Conclusion

Coroutines offer interesting mechanism to fine-grain control over execution of code.

You can implement cooperative multitasking based on `co_await`.

Execution of awaitables may happen in another threads.

The traditional caller - callee relationship can be broken and represented in any way desired.

Coroutines can yield control without actually terminating. Thus keeping execution context is easier while communicating between coroutines.

This makes concurrent programming a bit more readable (well, at least in theory).