Coroutines (C++20)

A coroutine is a function that can suspend execution to be resumed later. Coroutines are stackless: they suspend execution by returning to the caller and the data that is required to resume execution is stored separately from the stack. This allows for sequential code that executes asynchronously (e.g. to handle non-blocking I/O without explicit callbacks), and also supports algorithms on lazy-computed infinite sequences and other uses.

A function is a coroutine if its definition does any of the following:

- uses the `co_await` operator to suspend execution until resumed

```
task<> tcp_echo_server() {
  char data[1024];
  while (true) {
    size_t n = co_await socket.async_read_some(buffer(data));
    co_await async_write(socket, buffer(data, n));
  }
}
```

- uses the keyword `co_yield` to suspend execution returning a value

```
generator<int> iota(int n = 0) {
  while(true)
    co_yield n++;
}
```

- uses the keyword `co_return` to complete execution returning a value

```
lazy<int> f() {
  co_return 7;
}
```

Every coroutine must have a return type that satisfies a number of requirements, noted below.

Restrictions

Coroutines cannot use variadic arguments, plain return statements, or placeholder return types (`auto` or `Concept`).

Constexpr functions, constructors, destructors, and the main function cannot be coroutines.

Execution

Each coroutine is associated with

- the *promise object*, manipulated from inside the coroutine. The coroutine submits its result or exception through this object.
- the *coroutine handle*, manipulated from outside the coroutine. This is a non-owning handle used to resume execution of the coroutine or to destroy the coroutine frame.
- the *coroutine state*, which is an internal, heap-allocated (unless the allocation is optimized out), object that contains
  - the promise object
  - the parameters (all copied by value)
  - some representation of the current suspension point, so that resume knows where to continue and destroy knows what local variables were in scope
  - local variables and temporaries whose lifetime spans the current suspension point

When a coroutine begins execution, it performs the following:

- allocates the coroutine state object using `operator new` (see below)
- copies all function parameters to the coroutine state: by-value parameters are moved or copied, by-reference parameters remain references (and so may become dangling if the coroutine is resumed after the lifetime of referred object ends)

- calls the constructor for the promise object. If the promise type has a constructor that takes all coroutine parameters, that constructor is called, with post-copy coroutine arguments. Otherwise the default constructor is called.
- calls `promise.get_return_object()` and keeps the result in a local variable. The result of that call will be returned to the caller when the coroutine first suspends. Any exceptions thrown up to and including this step propagate back to the caller, not placed in the promise.
- calls `promise.initial_suspend()` and `co_await`s its result. Typical Promise types either return a `suspend_always`, for lazily-started coroutines, or `suspend_never`, for eagerly-started coroutines.
- when `co_await promise.initial_suspend()` resumes, starts executing the body of the coroutine

When a coroutine reaches a suspension point

- the return object obtained earlier is returned to the caller/resumer, after implicit conversion to the return type of the coroutine, if necessary.

When a coroutine reaches the `co_return` statement, it performs the following:

- calls `promise.return_void()` for
  - `co_return;`
  - `co_return expr` where expr has type void
  - falling off the end of a void-returning coroutine. The behavior is undefined if the Promise type has no `Promise::return_void()` member function in this case.
- or calls `promise.return_value(expr)` for `co_return expr` where expr has non-void type
- destroys all variables with automatic storage duration in reverse order they were created.
- calls `promise.final_suspend()` and `co_await`s the result.

If the coroutine ends with an uncaught exception, it performs the following:

- catches the exception and calls `promise.unhandled_exception()` from within the catch-block
- calls `promise.final_suspend()` and `co_await`s the result (e.g. to resume a continuation or publish a result). It's undefined behavior to resume a coroutine from this point.

When the coroutine state is destroyed either because it terminated via co_return or uncaught exception, or because it was destroyed via its handle, it does the following:

- calls the destructor of the promise object.
- calls the destructors of the function parameter copies.
- calls `operator delete` to free the memory used by the coroutine state
- transfers execution back to the caller/resumer.

Heap allocation

coroutine state is allocated on the heap via non-array `operator new`.

If the `Promise` type defines a class-level replacement, it will be used, otherwise global `operator new` will be used.

If the `Promise` type defines a placement form of `operator new` that takes additional parameters, and they match an argument list where the first argument is the size requested (of type `std::size_t`) and the rest are the coroutine function arguments, those arguments will be passed to `operator new` (this makes it possible to use leading-allocator-convention for coroutines)

The call to `operator new` can be optimized out (even if custom allocator is used) if

- The lifetime of the coroutine state is strictly nested within the lifetime of the caller, and
- the size of coroutine frame is known at the call site

in that case, coroutine state is embedded in the caller's stack frame (if the caller is an ordinary function) or coroutine state (if the caller is a coroutine)

If allocation fails, the coroutine throws `std::bad_alloc`, unless the Promise type defines the member function `Promise::get_return_object_on_allocation_failure()`. If that member function is defined, allocation uses the `nothrow` form of `operator new` and on allocation failure, the coroutine immediately returns the object obtained from `Promise::get_return_object_on_allocation_failure()` to the caller.

Promise

The Promise type is determined by the compiler from the return type of the coroutine using `std::coroutine_traits`.

Formally, let `R` and `Args...` denote the return type and parameter type list of a coroutine respectively, `ClassT` and `/*cv-qual*/` (if any) denote the class type to which the coroutine belongs and its cv-qualification respectively if it is defined as a non-static member function, its `Promise` type is determined by:

- `std::coroutine_traits<R, Args...>::promise_type`, if the coroutine is not defined as a non-static member function,
- `std::coroutine_traits<R, ClassT /*cv-qual*/&, Args...>::promise_type`, if the coroutine is defined as a non-static member function that is not rvalue-reference-qualified,
- `std::coroutine_traits<R, ClassT /*cv-qual*/&&, Args...>::promise_type`, if the coroutine is defined as a non-static member function that is rvalue-reference-qualified.

For example:

- If the coroutine is defined as `task<float> foo(std::string x, bool flag);`, then its `Promise` type is `std::coroutine_traits<task<float>, std::string, bool>::promise_type`.
- If the coroutine is defined as `task<void> my_class::method1(int x) const;`, its `Promise` type is `std::coroutine_traits<task<void>, const my_class&, int>::promise_type`.
- If the coroutine is defined as `task<void> my_class::method1(int x) &&;`, its `Promise` type is `std::coroutine_traits<task<void>, my_class&&, int>::promise_type`.

co_await

The unary operator `co_await` suspends a coroutine and returns control to the caller. Its operand is an expression whose type must either define `operator co_await`, or be convertible to such type by means of the current coroutine's `Promise::await_transform`

`co_await` *expr*

First, *expr* is converted to an awaitable as follows:

- if *expr* is produced by an initial suspend point, a final suspend point, or a yield expression, the awaitable is *expr*, as-is.
- otherwise, if the current coroutine's Promise type has the member function `await_transform`, then the awaitable is `promise.await_transform(expr)`
- otherwise, the awaitable is *expr*, as-is.

Then, the awaiter object is obtained, as follows:

- if overload resolution for `operator co_await` gives a single best overload, the awaiter is the result of that call (`awaitable.operator co_await()` for member overload, `operator co_await(static_cast<Awaitable&&>(awaitable))` for the non-member overload)
- otherwise, if overload resolution finds no operator co_await, the awaiter is awaitable, as-is
- otherwise, if overload resolution is ambiguous, the program is ill-formed

If the expression above is a prvalue, the awaiter object is a temporary materialized from it. Otherwise, if the expression above is an glvalue, the awaiter object is the object to which it refers.

Then, `awaiter.await_ready()` is called (this is a short-cut to avoid the cost of suspension if it's known that the result is ready or can be completed synchronously). If its result, contextually-converted to bool is `false` then

> The coroutine is suspended (its coroutine state is populated with local variables and current suspension point). `awaiter.await_suspend(handle)` is called, where handle is the coroutine handle representing the current coroutine. Inside that function, the suspended coroutine state is observable via that handle, and it's this function's responsibility to schedule it to resume on some executor, or to be destroyed (returning false counts as scheduling)
>
> - if await_suspend returns void, control is immediately returned to the caller/resumer of the current coroutine (this coroutine remains suspended), otherwise
> - if await_suspend returns bool,
>   - the value `true` returns control to the caller/resumer of the current coroutine
>   - the value `false` resumes the current coroutine.
> - if await_suspend returns a coroutine handle for some other coroutine, that handle is resumed (by a call to `handle.resume()`) (note this may chain to eventually cause the current coroutine to resume)

- if await_suspend throws an exception, the exception is caught, the coroutine is resumed, and the exception is immediately re-thrown

Finally, `awaiter.await_resume()` is called, and its result is the result of the whole `co_await expr` expression.

If the coroutine was suspended in the co_await expression, and is later resumed, the resume point is immediately before the call to `awaiter.await_resume()`.

Note that because the coroutine is fully suspended before entering `awaiter.await_suspend()`, that function is free to transfer the coroutine handle across threads, with no additional synchronization. For example, it can put it inside a callback, scheduled to run on a threadpool when async I/O operation completes. In that case, since the current coroutine may have been resumed and thus executed the awaiter object's destructor, all concurrently as await_suspend() continues its execution on the current thread, await_suspend() should treat `*this` as destroyed and not access it after the handle was published to other threads.

Example

Run this code

```
#include <coroutine>
#include <iostream>
#include <stdexcept>
#include <thread>

auto switch_to_new_thread(std::jthread& out) {
  struct awaitable {
    std::jthread* p_out;
    bool await_ready() { return false; }
    void await_suspend(std::coroutine_handle<> h) {
      std::jthread& out = *p_out;
      if (out.joinable())
        throw std::runtime_error("Output jthread parameter not empty");
      out = std::jthread([h] { h.resume(); });
      // Potential undefined behavior: accessing potentially destroyed *this
      // std::cout << "New thread ID: " << p_out->get_id() << '\n';
      std::cout << "New thread ID: " << out.get_id() << '\n'; // this is OK
    }
    void await_resume() {}
  };
  return awaitable{&out};
}

struct task{
  struct promise_type {
    task get_return_object() { return {}; }
    std::suspend_never initial_suspend() { return {}; }
    std::suspend_never final_suspend() noexcept { return {}; }
    void return_void() {}
    void unhandled_exception() {}
  };
};

task resuming_on_new_thread(std::jthread& out) {
  std::cout << "Coroutine started on thread: " << std::this_thread::get_id() << '\n';
  co_await switch_to_new_thread(out);
  // awaiter destroyed here
  std::cout << "Coroutine resumed on thread: " << std::this_thread::get_id() << '\n';
}

int main() {
  std::jthread out;
  resuming_on_new_thread(out);
}
```

Possible output:

```
Coroutine started on thread: 139972277602112
New thread ID: 139972267284224
Coroutine resumed on thread: 139972267284224
```

Note: the awaiter object is part of coroutine state (as a temporary whose lifetime crosses a suspension point) and is destroyed before the co_await expression finishes. It can be used to maintain per-operation state as required by some async I/O APIs without resorting to additional heap allocations.

The standard library defines two trivial awaitables: `std::suspend_always` and `std::suspend_never`.

This section is incomplete
Reason: examples

co_yield

Yield-expression returns a value to the caller and suspends the current coroutine: it is the common building block of resumable generator functions

`co_yield` *expr*

`co_yield` *braced-init-list*

It is equivalent to

```
co_await promise.yield_value(expr)
```

A typical generator's yield_value would store (copy/move or just store the address of, since the argument's lifetime crosses the suspension point inside the co_await) its argument into the generator object and return `std::suspend_always`, transferring control to the caller/resumer.

This section is incomplete
Reason: examples

Library support

Coroutine support library defines several types providing compile and run-time support for coroutines.