1 Coroutine body

Every time, the compiler encounters one of the following keywords:

- co_return
- co_yield
- co_await

function in which body the keyword was encountered is transformed into the coroutine. This is performed in the following schema:

```
{
  promise_type promise;
  auto&& return_object = promise.get_return_object();
  co_await promise.initial_suspend();

  try{
    //our coroutine_body
  }catch(...) {
    promise.unhandled_exception();
  }

  final_suspend:
  co_await promise.final_suspend();

  return return_object;
}
```

2 promise_type

Promise_type is the type used by the compiler to control behavior of the coroutine. It should be defined (as publicly available) either as a member of the coroutine type like:

```
returned_type::promise_type
```

or as a member of the specialization of the coroutine_traits:

```
namespace std{
        template <>
  struct coroutine_traits<returned_type>{
    struct promise_type{
      //definition
    };
  };
}
```

The functions, that steer the coroutine behavior are listed below:

```
struct promise_type{
  // creating coroutine object -mandatory
  auto get_return_object();

  // returns awaitable object - mandatory
  auto initial_suspend();
  auto final_suspend();

  void unhandled_exception(); // mandatory

  // one of below is mandatory
  // and only one must be present
  void return_value(/*type*/);
  void return_void();

  // support for yielding values - returns awaitable
  auto yield_value();

  // modification of the awaitable
  auto await_transform(/*co\_await operand*/);
};
```

3 co_yield

Each time, when compiler sees co_yield keyword, the following code is generated:

```
co_await promise.yield_value(<expression>);
```

If your type is not meant to support yield expression, you can suppress it with:

```
auto yield_value() = delete;
```

in the definition of your promise_type.

4 co_return

To finish the coroutine and optionally return the value from it one can use the co_return expression. Such expression is also translated by the compiler depending on the operand of the co_return keyword.

Void expressions and co_return without any expression is translated into following form:

```
<optional_expression>;
promise.return_void();
```

while for the non-void expressions, following code is generated:

```
promise.return_value(<expression>);
```

5 coroutine_handle

Coroutine handle is our object, that directly operates on the coroutine (it can for example resume it or delete it). It's API is as follows:

```
namespace std {
  template<>
  struct coroutine_handle<void>
  {
    // [coroutine.handle.con], construct/reset
    constexpr coroutine_handle() noexcept;
    constexpr coroutine_handle(nullptr_t) noexcept;
    coroutine_handle& operator=(nullptr_t) noexcept;

    // [coroutine.handle.export.import], export/import
    constexpr void* address() const noexcept;
    constexpr static coroutine_handle from_address(void* addr);

    // [coroutine.handle.observers], observers
    constexpr explicit operator bool() const noexcept;
    bool done() const;

    // [coroutine.handle.resumption], resumption
    void operator()() const;
    void resume() const;
    void destroy() const;

  private:
    void* ptr;  // exposition only
  };

  template<class Promise>
  struct coroutine_handle : coroutine_handle<>
  {
    // [coroutine.handle.con], construct/reset
    using coroutine_handle<>::coroutine_handle;
    static coroutine_handle from_promise(Promise&);
    coroutine_handle& operator=(nullptr_t) noexcept;

    // [coroutine.handle.export.import], export/import
    constexpr static coroutine_handle from_address(void* addr);

    // [coroutine.handle.promise], promise access
    Promise& promise() const;
  };
}
```

6 Awaitable primitives

The standard library defined two primitives, that can be operands of the co_await operator, namely:

- std::suspend_always - causes suspension of the coroutine
- std::suspend_never - is a no-op

7 Creating awaiter

The co_await operator needs so called awaiter object to know how should a coroutine behave on awaiting an awaitable object.

The awaiter object is created in following way:

- The await_transform function form the promise_type is executed on the co_await operand,
- co_await operator is searched in the body of the awaitable,
- if not found global co_await operator is searched for,
- if not found awaitable becomes the awaiter

8 Awaiter

Awaiter object must have following functions defined in it's body:

```
struct awaiter{
  bool await_ready();
  auto await_suspend(coro_handle_t);
  auto await_resume();
}
```

Their responsibility:

- await_ready - knows whether the awaitable is finished and result can be fetched from it,
- await_suspend - knows how to await on the awaitable (usually how to resume it),
- await_resume - result of this function evaluation is the result of the whole co_await expression.

9 co_await transformation

Whenever co_await keyword is encountered by compiler, compiler generates following code (besides the procedure for acquiring awaiter)

```
{
  std::exception_ptr exception = nullptr;
  if (not a.await_ready()) {
    suspend_coroutine();

    <await_suspend>
  }

  resume_point:
  if(exception)
    std::rethrow_exception(exception);
  /*return*/ a.await_resume();
}
```

where await_suspend expression is one of the following:

- when await_suspend returns void

```
try {
  a.await_suspend(coroutine_handle);
  return_to_the_caller();
} catch (...) {
  exception = std::current_exception();
  goto resume_point;
}
```

- when await_suspemd returns bool

```
bool await_suspend_result;
try {
  await_suspend_result = a.await_suspend(
                         coroutine_handle);
} catch (...) {/*...*/}

if (not await_suspend_result)
  goto resume_point;
return_to_the_caller();
```

- when await_suspemd returns another coroutine_handle

```
decltype(a.await_suspend(
  std::declval<coro_handle_t>()))
another_coro_handle;
try {
  another_coro_handle = a.await_suspend(
                        coroutine_handle);
} catch (...) {/*...*/}

another_coro_handle.resume();
return_to_the_caller();
```