Document Number: N4499
Date: 2015-05-22
Revises: N4403
Authors: Gor Nishanov <gorn@microsoft.com>
Daveed Vandevoorde <daveed@edg.com>

Draft wording for Coroutines (Revision 2)

Note: this is an early draft. It's known to be incomplet and incorrekt, and it has lots of bad formatting.

Contents

List of Tables

1 General [intro]

1.1 Scope [intro.scope]

1 This Technical Specification describes extensions to the C++ Programming Language (1.3) that enable definition of coroutines. These extensions include new syntactic forms and modifications to existing language semantics.

2 The International Standard, ISO/IEC 14882, provides important context and specification for this Technical Specification. This document is written as a set of changes against that specification. Instructions to modify or add paragraphs are written as explicit instructions. Modifications made directly to existing text from the International Standard use underlining to represent added text and ~~strikethrough~~ to represent deleted text.

1.2 Acknowledgements [intro.ack]

This work is the result of collaboration of researchers in industry and academia, including CppDes Microsoft group and the WG21 study group SG1. We wish to thank people who made valuable contributions within and outside these groups, including Jens Maurer, Artur Laksberg, Chandler Carruth, Gabriel Dos Reis, Deon Brewis, Jonathan Caves, James McNellis, Stephan T. Lavavej, Herb Sutter, Pablo Halpern, Robert Schumacher, Michael Wong, Niklas Gustafsson, Nick Maliwacki, Vladimir Petter, Shahms King, Slava Kuznetsov, Tongari J, Lawrence Crowl, and many others not named here who contributed to the discussion.

1.3 Normative references [intro.refs]

1 The following referenced document is indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

(1.1) — ISO/IEC 14882:2014, *Programming Languages – C++*

ISO/IEC 14882:2014 is hereafter called the *C++ Standard*. Beginning with section 1.9 below, all clause and section numbers, titles, and symbolic references in [brackets] refer to the corresponding elements of the C++ Standard. Sections 1.1 through 1.5 of this Technical Specification are introductory material and are unrelated to the similarly-numbered sections of the C++ Standard.

1.4 Implementation compliance [intro.compliance]

1 Conformance requirements for this specification are the same as those defined in section 1.4 of the C++ Standard. [*Note:* Conformance is defined in terms of the behavior of programs. — *end note*]

1.5 Feature testing [intro.features]

An implementation that provides support for this Technical Specification shall define the feature test macro in Table 1.

Table 1 — Feature-test macro

Name	Value	Header
`__cpp_coroutines`	201510	*predeclared*

1.9 Program execution [intro.execution]

Modify paragraph 7 to read:

7 An instance of each object with automatic storage duration (3.7.3) is associated with each entry into its block. Such an object exists and retains its last-stored value during the execution of the block and while the block is suspended (by a call of a function, suspension of a coroutine (8.4.4), or receipt of a signal).

2 Lexical conventions [lex]

2.12 Keywords [lex.key]

[Editor's note: In Lenexa's EWG session there was a brief discussion on possible keywords. In this document we use placeholder keywords with suffix -keyword to be replaced with real ones in Kona. A companion paper discussing keyword alternatives is to appear in pre-Kona mailing.]

Add the keyword placeholders `await-keyword`, `coroutine-return-keyword`, and `yield-keyword` to Table 4 "Keywords".

3 Basic concepts [basic]

3.6 Start and termination [basic.start]

3.6.1 Main function [basic.start.main]

Add underlined text to paragraph 3.

> The function `main` shall not be used within a program. The linkage (3.5) of `main` is implementation-defined. A program that defines `main` as deleted or that declares `main` to be `inline, static`, or `constexpr` is ill-formed. The function `main` shall not be a coroutine. The name `main` is not otherwise reserved. [*Example:* member functions, classes, and enumerations can be called `main`, as can entities in other namespaces. — *end example*]

5 Expressions [expr]

5.3 Unary expressions [expr.unary]

In this section change the grammar for *unary-expression* as follows:

unary-expression:
postfix-expression
`++` *cast-expression*
`--` *cast-expression*
await-expression
unary-operator cast-expression
`sizeof` *unary-expression*
`sizeof (` *type-id* `)`
`sizeof ... (` *identifier* `)`
`alignof (` *type-id* `)`
noexcept-expression
new-expression
delete-expression

5.3.7 `noexcept` operator [expr.unary.noexcept]

In this section, add a new paragraph after paragraph 3.

4 If in a potentially-evaluated context the *expression* would contain a potentially-evaluated *await-expression*, the program is ill-formed.

5.3.8 Await [expr.await]

Add this section to 5.3.

1 The `await-keyword` operator is used to suspend evaluation of the enclosing coroutine (8.4.4) while awaiting completion of the computation represented by the operand expression.

await-expression:
`await-keyword` *cast-expression*

2 A potentially-evaluated *await-expression* shall only appear within the *compound-statement* of a *function-body* outside of a *handler* (15.3). In a *declaration-statement* or in the *simple-declaration* (if any) of a *for-init-statement*, a potentially-evaluated *await-expression* shall only appear in an *initializer* of that *declaration-statement* or *simple-declaration*. A potentially-evaluated *await-expression* shall not appear in a default argument (8.3.6).

3 Let T be the type of the *cast-expression*. If the *cast-expression* is a prvalue, let e be a temporary initialized as-if by `T e = cast-expression`; otherwise let e be an lvalue designating the value of the *cast-expression*. Let p be the promise object (8.4.4) of the enclosing coroutine, P be the type of the promise object, h be an object of `std::experimental::coroutine_handle<P>` referring to the enclosing coroutine, then *await-ready-expr*, *await-suspend-expr*, and *await-resume-expr* are expressions defined as follows:

(3.1) — if T is a class type, the *unqualified-id*s `await_ready`, `await_suspend`, `await_resume` are looked up in the scope of that class as if by class member access lookup (3.4.5), and if it finds at least one declaration, *await-ready-expr*, *await-suspend-expr*, and *await-resume-expr* are `e.await_ready()`, `e.await_suspend(h)`, and `e.await_resume()`, respectively;

(3.2) — otherwise, *await-ready-expr*, *await-suspend-expr*, and *await-resume-expr* are `await_ready(`*e*`)`, `await_suspend(`*e*`, `*h*`)`, and `await_resume(`*e*`)` respectively, where *await-ready-expr*, *await-suspend-expr*, and *await-resume-expr* are looked up in the associated namespaces (3.4.2). [*Note:* Ordinary unqualified lookup (3.4.1) is not performed. — *end note*]

If the type of *await-suspend-expr* is *cv* `void`, then

`await-keyword` *cast-expression*

is equivalent to:

```
(
    await-ready-expr ? await-resume-expr
    : (await-suspend-expr, suspend-resume-point, await-resume-expr)
)
```

otherwise, it is equivalent to:

```
(
    (await-ready-expr && !await-suspend-expr) ? await-resume-expr
    : (suspend-resume-point, await-resume-expr)
)
```

where *suspend-resume-point*s are treated as expressions of type `void`. Suspend-resume-points are defined in (8.4.4).

4 An *await-expression* may only appear in a coroutine with an eventual return type (6.6.4).

5 [*Note:* An *await-expression* may appear as an unevaluated operand (5.2.8, 5.3.3, 5.3.7, 7.1.6.2). The presence of such an *await-expression* does not make the enclosing function a coroutine and can be used to examine the type of an *await-expression*.

[*Example:*

```
std::future<int> f();

int main() {
  using t = decltype(await-keyword f()); // t is int
  static_assert(sizeof(await-keyword f()) == sizeof(int));
  cout << typeid(await-keyword f()).name() << endl;
}
```

— *end example*] — *end note*]

6 Statements [stmt.stmt]

6.5 Iteration statements [stmt.iter]

Add underlined text to paragraph 1.

1 Iteration statements specify looping.

iteration-statement:
`while (` *condition* `)` *statement*
`do` *statement* `while (` *expression* `) ;`
`for (` *for-init-statement condition*$_{opt}$`;` *expression*$_{opt}$`)` *statement*
`for` `await-keyword`$_{opt}$ `(` *for-range-declaration* `:` *for-range-initializer* `)` *statement*

6.5.4 The range-based `for` statement [stmt.ranged]

Add underlined text to paragraph 1.

1 For a range-based `for` statement of the form

`for` `await-keyword`$_{opt}$ `(` *for-range-declaration : expression* `)` *statement*

let *range-init* be equivalent to the *expression* surrounded by parentheses[1]

`(` *expression* `)`

and for a range-based `for` statement of the form

`for` `await-keyword`$_{opt}$ `(` *for-range-declaration* `:` *braced-init-list* `)` *statement*

let *range-init* be equivalent to the *braced-init-list*. In each case, a range-based `for` statement is equivalent to

```
{
  auto && __range = range-init;
  for ( auto __begin = await-keyword_opt begin-expr,
  __end = end-expr;
  __begin != __end;
  await-keyword_opt ++__begin ) {
    for-range-declaration = *__begin;
    statement
  }
}
```

where `await-keyword` appears if and only if it appears immediately after the `for` keyword, and `__range`, `__begin`, and `__end` are variables defined for exposition only, and `_RangeT` is the type of the expression, and *begin-expr* and *end-expr* are determined as follows:

[Editor's note: The remainder of paragraph 1 remains unchanged and is not included here.]

1) this ensures that a top-level comma operator cannot be reinterpreted as a delimiter between *init-declarator*s in the declaration of `__range`.

6.6 Jump statements [stmt.jump]

In paragraph 1 add four productions to the grammar:

> *jump-statement:*
> `break ;`
> `continue ;`
> `return` *expression*$_{opt}$ `;`
> `return` *braced-init-list* `;`
> `coroutine-return-keyword` *expression*$_{opt}$ `;`
> `coroutine-return-keyword` *braced-init-list* `;`
> `yield-keyword` *expression* `;`
> `yield-keyword` *braced-init-list* `;`
> `goto` *identifier* `;`

6.6.3 The `return` statement [stmt.return]

Add underlined text to paragraph 1:

1 A function returns to its caller by the `return` statement. A `return` statement shall not appear in a coroutine.

6.6.4 The `coroutine-return-keyword` statement [stmt.coreturn]

Add this section to 6.6.

1 A coroutine returns to its caller by the `coroutine-return-keyword` statement or when suspended at a suspend-resume point (8.4.4). A `coroutine-return-keyword` statement shall not appear in a function other than a coroutine.

2 If the promise type (8.4.4) of the coroutine defines the member function `return_void`, the coroutine is considered to have an *eventual return type* of `void`, if the promise type (8.4.4) of the coroutine defines the member function `return_value`, the coroutine is considered to have a non-`void` eventual return type, otherwise, the coroutine is considered not to have an eventual return type. If the promise type defines both `return_value` and `return_void` member functions, the program is ill-formed.

3 In this section, *p* refers to the promise object (8.4.4) of the enclosing coroutine.

4 A `coroutine-return-keyword` statement with neither an *expression* nor a *braced-init-list* can be used only in coroutines that do not have an eventual return type or have an eventual return type of `void`. In the latter case, completion of the coroutine is signaled to the promise of the coroutine by calling `p.return_void()`. A `coroutine-return-keyword` statement with an expression of non-`void` type can be used only in coroutines producing an eventual value; the value of the expression is supplied to the promise of the coroutine by calling `p.return_value(expression)` or `p.return_value(braced-init-list)`. Flowing off the end of a coroutine is equivalent to a `coreturn` with no value; this results in undefined behavior in a coroutine with non-`void` return type.

5 Prior to returning to the caller, a coroutine evaluates the `p.final_suspend()` predicate. If `p.final_suspend()` contextually converted to `bool` evaluates to `true`, the coroutine suspends at final suspend point (8.4.4), otherwise, the coroutine destroys the coroutine state (8.4.4) and frees the memory dynamically allocated (if any) to store the state.

6 A `coroutine-return-keyword` statement with an expression of type *cv* `void` can be used only in functions without an eventual return type or with an eventual return type of `void`; the expression is evaluated just before the call to `p.final_suspend()` and `p.return_void()` respectively.

6.6.5 The `yield` statement [stmt.yield]

Add this section to 6.6.

Let *yielded value* be the operand of the `yield-keyword` statement and *p* be the promise object of the enclosing coroutine. If the result type of *p*.`yield_value(`*yielded-value*`)` is of type *cv* `void`, then the `yield-keyword` statement is equivalent to:

```
p.yield_value(yielded-value);
suspend-resume-point
```

otherwise, it is equivalent to:

```
if (p.yield_value(yielded-value)) {
  suspend-resume-point
}
```

7 Declarations [dcl.dcl]

7.1.5 The `constexpr` specifier [dcl.constexpr]

Add the underlined text as the last item in the list in paragraph 3. Note that the preceding (unmodified) items in the C++ Standard are elided in this document.

3 The definition of a `constexpr` function shall satisfy the following constraints:

(3.1) — ...

(3.2) — ...

(3.3) — ...

(3.4) — ...

(3.5) — it shall not be a coroutine (8.4.4);

7.1.6.4 `auto` specifier [dcl.spec.auto]

Add the underlined text to paragraph 2.

2 The placeholder type can appear with a function declarator in the *decl-specifier-seq*, *type-specifier-seq*, *conversion-function-id*, or *trailing-return-type*, in any context where such a declarator is valid. If the function declarator includes a *trailing-return-type* (8.3.5), that specifies the declared return type of the function. If the declared return type of the function contains a placeholder type, the return type of the function is deduced from `return`, `coroutine-return-keyword`, and `yield-keyword` statements in the body of the function, if any.

Add the underlined text to paragraph 9.

9 If a function with a declared return type that contains a placeholder type has multiple `return`, `coroutine-return-keyword`, and `yield-keyword` statements, the return type is deduced for each `return`, `coroutine-return-keyword`, and `yield-keyword` statement. If the type deduced is not the same in each deduction, the program is ill-formed.

Add paragraphs 16 through 18.

16 If a coroutine has a declared return type that contains a placeholder type, then the return type of the coroutine is deduced as follows:

(16.1) — If a `yield-keyword` statement and an *await-expression* are present, then the return type is `std::experimental::async_stream<T>`, where `T` is deduced from the `yield-keyword` statements as if a `yield-keyword` statement were a `return` statement in a function with declared type `auto` without a *trailing-return-type*.

(16.2) — Otherwise, if an *await-expression* is present in a function, then the return type is `std::experimental::task<T>` where type `T` is deduced from `coroutine-return-keyword` statements as if a `coroutine-return-keyword` statement were a `return` statement in a function with declared type `auto` without a *trailing-return-type*.

(16.3) — Otherwise, if a `yield-keyword` statement is present in a function, then the return type is `std::experimental::generator<T>`, where `T` is deduced from the `yield-keyword` statements as if a `yield-keyword` statement were a `return` statement in a function with declared type `auto` without a *trailing-return-type*.

[*Example:*

```
// deduces to std::experimental::generator<char>
auto f() { for(auto ch: "Hello") yield-keyword ch; }

// deduces to std::experimental::async_stream<int>
auto ticks() {
  for(int tick = 0;; ++tick) {
    yield-keyword tick;
    await-keyword sleep_for(1ms);
  }
}

future<void> g();

// deduces to std::experimental::task<void>
auto f2() {  await-keyword g(); }
```

— *end example*]

17 The templates `std::experimental::generator`, `std::experimental::task`, and `std::experimental::async_stream` are not predefined; if the appropriate headers are not included prior to a use — even an implicit use in which the type is not named (7.1.6.4) — the program is ill-formed.

8 Declarators [dcl.decl]

8.3.5 Functions [dcl.fct]

Add paragraph 16.

16 If the *parameter-declaration-clause* terminates with an ellipsis that is not part of *abstract-declarator*, a function shall not be coroutine (8.4.4).

8.4 Function definitions [dcl.fct.def]

8.4.4 Coroutines [dcl.fct.def.coroutine]

Add this section to 8.4.

1 A function is a *coroutine* if it contains one or more suspend-resume-points introduced by a potentially-evaluated *await-expression* (5.3.8) and a `yield-keyword` statement (6.6.5). Every coroutine also has an implicit initial and final suspend-resume point as described later in this section.

2 [*Note:* From the perspective of the caller, a coroutine is just a function with that particular signature. The fact that a function is implemented as a coroutine is unobservable by the caller. — *end note*]

3 A coroutine needs a set of related types and functions to complete the definition of its semantics. These types and functions are provided as a set of member types or typedefs and member functions in the specializations of class template `std::experimental::coroutine_traits` (18.11.1).

4 For a coroutine *f*, if *f* is a non-static member function, let P_1 denote the type of the implicit object parameter (13.3.1) and P_2 ... P_n be the types of the function parameters; otherwise let P_1 ... P_n be the types of the function parameters. Let R be the return type and F be the *function-body* of *f*, T be a type `std::experimental::coroutine_traits<`R`,`P_1`,...,`P_n`>`, and P be the type denoted by *T::*`promise_type`. Then, the coroutine behaves as if its body were:

```
{
  P p;
  if (p.initial_suspend()) {
    suspend-resume-point // initial suspend point
  }
  F'
  if (p.final_suspend()) {
    suspend-resume-point // final suspend point
  }
}
```

where local variable *p* is defined for exposition only and *F'* is *F* if *P* does not define a `set_exception` member function, and

```
try { F } catch(...) { p.set_exception(std::current_exception()); }
```

otherwise. No header needs to be included for this use of the function `std::current_exception`. An object denoted as *p* is the *promise object* of the coroutine *f* and its type is a *promise type* of the coroutine. An execution of a coroutine is suspended when it reaches a suspend-resume-point.

5 A *suspension* of a coroutine returns control to the current caller of the coroutine. For the first return of control from the coroutine, the return value is obtained by invoking the member function `get_return_object` (18.11.3) of the promise object.

6 A suspended coroutine can be resumed to continue execution by invoking a resumption member functions (18.11.2.4) of an object of `coroutine_handle<`*P*`>` type associated with this instance of the coroutine, where type *P* is the promise type of the coroutine.

7 A coroutine may need to allocate memory to store objects with automatic storage duration local to the coroutine. If so, it must use the allocator object obtained as described in Table 3 in clause 18.11.1.

8 A *coroutine state* consists of storage for objects with automatic storage duration that are live at the current point of execution or suspension of a coroutine. The coroutine state is destroyed when the control flows off the end of the function or the `destroy` member function (18.11.2.4) of an object of `std::experimental::coroutine_handle<`*P*`>` associated with that coroutine is invoked. In the latter case objects with automatic storage duration that are in scope at the suspend point are destroyed in the reverse order of the construction. If the coroutine state required dynamic allocation, the memory is freed. If `destroy` is called for a coroutine that is not suspended, the program has undefined behavior.

9 When a coroutine is invoked, each of its parameters is copied/moved to the coroutine state, as specified in 12.8. The copy/move operations are indeterminately sequenced with respect to each other. A reference to a parameter in the function-body of the coroutine is replaced by a reference to the copy of the parameter.

10 If the coroutine state initialization, a call to `get_return_object`, or a promise object construction throws an exception, any memory dynamically allocated for the coroutine state is freed.

11 If type T defines static member function `get_return_object_on_allocation_failure` (18.11.1) and the coroutine state is allocated dynamically, the result of an allocation call needs to be compared with `nullptr`, and if it is `nullptr`, coroutine must return control to the current caller of the coroutine and the return value is obtained by a call to `T::get_return_object_on_allocation_failure()`. [*Note:* This provision allows coroutines to be used in environments where exception use is not possible to report allocation failures. — *end note*]

12 [*Example:*

```
// coroutine hello world
std::experimental::generator<char> hello_fn() {
  for (auto ch: "Hello, world") yield-keyword ch;
}

int main() {
  // coroutine as a lambda
  auto hello_lambda = []{ for (auto ch: "Hello, world") yield-keyword ch; };

  for (auto ch : hello_lambda())
     cout << ch;
  for (auto ch : hello_fn())
     cout << ch;
}
```

— *end example*]

12 Special member functions [special]

In this section add new paragraph after paragraph 5.

6 A special member function shall not be a coroutine.

12.8 Copying and moving class objects [class.copy]

Add underlined text to paragraph 31.

31 When certain criteria are met, an implementation is allowed to omit the copy/move construction of a class object, even if the constructor selected for the copy/move operation and/or the destructor for the object have side effects. In such cases, the implementation treats the source and target of the omitted copy/move operation as simply two different ways of referring to the same object, and the destruction of that object occurs at the later of the times when the two objects would have been destroyed without the optimization.[2] This elision of copy/move operations, called *copy elision*, is permitted in the following circumstances (which may be combined to eliminate multiple copies):

(31.1) — in a `return` statement in a function with a class return type, when the expression is the name of a non-volatile automatic object (other than a function or catch-clause parameter) with the same cv-unqualified type as the function return type, the copy/move operation can be omitted by constructing the automatic object directly into the function's return value

(31.2) — When a parameter would be copied/moved to the coroutine state (8.4.4) copy move can be omitted by continuing to refer to the function parameters instead of referring to their copies in the coroutine state.

2) Because only one object is destroyed instead of two, and one copy/move constructor is not executed, there is still one object destroyed for each one constructed.

18 Language support library [language.support]

18.1 General [support.general]

Add a row to Table 2 for coroutine support header <experimental/coroutine>.

Table 2 — Language support library summary

Subclause		Header(s)
18.2	Types	<cstddef>
18.3	Implementation properties	<limits> <climits> <cfloat>
18.4	Integer types	<cstdint>
18.5	Start and termination	<cstdlib>
18.6	Dynamic memory management	<new>
18.7	Type identification	<typeinfo>
18.8	Exception handling	<exception>
18.9	Initializer lists	<initializer_list>
18.11	Coroutines support	<experimental/coroutine>
18.10	Other runtime support	<csignal> <csetjmp> <cstdalign> <cstdarg> <cstdbool> <cstdlib> <ctime>

18.10 Other runtime support [support.runtime]

Add underlined text to paragraph 4.

4 The function signature `longjmp(jmp_buf jbuf, int val)` has more restricted behavior in this International Standard. A `setjmp`/`longjmp` call pair has undefined behavior if replacing the `setjmp` and `longjmp` by `catch` and `throw` would invoke any non-trivial destructors for any automatic objects. A call to `setjmp` or `longjmp` has undefined behavior if invoked in a coroutine.

SEE ALSO: ISO C 7.10.4, 7.8, 7.6, 7.12.

18.11 Coroutines support library [support.coroutine]

Add this section to clause 18.

1 The header <experimental/coroutine> defines several types providing compile and run-time support for coroutines in a C++ program.

Header <experimental/coroutine> synopsis

```
namespace std {
  namespace experimental {
```

```
    inline namespace coroutines_v1 {
      // 18.11.1 coroutine traits
      template <typename R, typename... ArgTypes>
        class coroutine_traits;

      // 18.11.2 coroutine handle
      template <typename Promise = void>
        class coroutine_handle;

      // 18.11.2.7 comparison operators:
      bool operator==(coroutine_handle<> x, coroutine_handle<> y) noexcept;
      bool operator<(coroutine_handle<> x, coroutine_handle<> y) noexcept;
      bool operator!=(coroutine_handle<> x, coroutine_handle<> y) noexcept;
      bool operator<=(coroutine_handle<> x, coroutine_handle<> y) noexcept;
      bool operator>=(coroutine_handle<> x, coroutine_handle<> y) noexcept;
      bool operator>(coroutine_handle<> x, coroutine_handle<> y) noexcept;
    } // namespace coroutines_v1
    } // namespace experimental

    // 18.11.2.8 hash support:
    template <class T> struct hash;
    template <class P> struct hash<experimental::coroutine_handle<P>>;
  } // namespace std
```

18.11.1 coroutine traits [coroutine.traits]

1 This subclause defines requirements on classes representing *coroutine traits*, and defines the class template `coroutine_traits` that satisfies those requirements.

2 The `coroutine_traits` may be specialized by the user to customize the semantics of coroutines.

18.11.1.1 Coroutine traits requirements [coroutine.traits.requirements]

1 In Table 3, X denotes a trait class instantiated as described in 8.4.4; If a coroutine is a member function, then a_1 denotes the implicit `this` parameter, a_2, ... a_n refer to explicit parameters of the coroutine, otherwise, a_1, a_2, ... a_n denote the parameters of the coroutine.

Table 3 — Coroutine traits requirements [**tab:coroutine.traits.requirements**]

Expression	Behavior
`X::promise_type`	`X::promise_type` must be a type satisfying coroutine promise requirements (18.11.3)
`X::get_allocator(`a_1`,` a_2`, ...` a_n`)`	*(optional)* Given a set of arguments passed to a coroutine, returns an allocator (17.6.3.5) that the implementation shall use to dynamically allocate memory for coroutine state if dynamic allocation is required. If `get_allocator` is not present, the implementation shall use `allocator<char>`.
`X::get_return_object_on_allocation_failure()`	*(optional)* If present, it is assumed that an allocator's `allocate` function will return `nullptr` in case of an allocation failure. If a coroutine requires dynamic allocation, it must check if an `allocate` returns `nullptr`, and if so it shall use the expression `X::get_return_object_on_allocation_failure()` to construct the return value and return back to the caller.

18.11.1.2 Struct template coroutine_traits [coroutine.traits.primary]

1 The header `<experimental/coroutine>` shall define the class template `coroutine_traits` as follows:

```
namespace std {
namespace experimental {
inline namespace coroutines_v1 {
  template <typename R, typename... Args>
  struct coroutine_traits {
    using promise_type = typename R::promise_type;
  };
} // namespace coroutines_v1
} // namespace experimental
} // namespace std
```

18.11.2 Struct template coroutine_handle [coroutine.handle]

```
namespace std {
  namespace experimental {
  inline namespace coroutines_v1 {
    template <>
    struct coroutine_handle<void>
    {
      // 18.11.2.1 construct/reset
      constexpr coroutine_handle() noexcept;
      constexpr coroutine_handle(nullptr_t) noexcept;
      coroutine_handle& operator=(nullptr_t) noexcept;

      // 18.11.2.2 export/import
      static coroutine_handle from_address(void* addr) noexcept;
      void* to_address() const noexcept;

      // 18.11.2.3 capacity
      explicit operator bool() const noexcept;

      // 18.11.2.4 resumption
      void operator()() const;
      void resume() const;
      void destroy() const;

      // 18.11.2.5 completion check
      bool done() const noexcept;
    };

    template <typename Promise>
    struct coroutine_handle : coroutine_handle<>
    {
      // 18.11.2.1 construct/reset
      using coroutine_handle<>::coroutine_handle;
      coroutine_handle(Promise*) noexcept;
      coroutine_handle& operator=(nullptr_t) noexcept;

      // 18.11.2.6 promise access
      Promise& promise() noexcept;
      Promise const& promise() const noexcept;
```

```
    };
  } // namespace coroutines_v1
  } // namespace experimental
} // namespace std
```

1 Let *P* be a promise type of the coroutine (8.4.4). An object of the type `coroutine_handle<P>` is called a *coroutine handle* and can be used to refer to a suspended or executing coroutine. Such a function is called a *target* of a coroutine handle. A default constructed `coroutine_handle` object has no target.

18.11.2.1 coroutine_handle construct/reset [coroutine.handle.con]

```
constexpr coroutine_handle() noexcept;
constexpr coroutine_handle(nullptr_t) noexcept;
```

1 *Postconditions:* `!*this`.

```
coroutine_handle(Promise* p) noexcept;
```

2 *Requires:* `p` points to a promise object of a coroutine.

3 *Postconditions:* `!*this` and `addressof(this->promise()) == p`.

```
coroutine_handle& operator=(nullptr_t) noexcept;
```

4 *Postconditions:* `!*this`.

5 *Returns:* `*this`.

18.11.2.2 coroutine_handle export/import [coroutine.handle.export]

```
static coroutine_handle from_address(void* addr) noexcept;
void* to_address() const noexcept;
```

1 *Postconditions:* `coroutine_handle<>::from_address(this->to_address()) == *this`.

18.11.2.3 coroutine_handle capacity [coroutine.handle.capacity]

```
explicit operator bool() const noexcept;
```

1 *Returns:* `true` if `*this` has a target, otherwise `false`.

18.11.2.4 coroutine_handle resumption [coroutine.handle.resumption]

```
void operator()() const;
void resume() const;
```

1 *Requires:* `*this` refers to a suspended coroutine.

2 *Effects:* resumes the execution of a target function. If the function was suspended at the final suspend point, `terminate` is called (15.5.1).

```
void destroy() const;
```

3 *Requires:* `*this` refers to a suspended coroutine.

4 *Effects:* destroys the target coroutine (8.4.4).

18.11.2.5 coroutine_handle completion check [coroutine.handle.completion]

```
bool done() const noexcept;
```

1 *Requires:* `*this` refers to a suspended coroutine.

2 *Returns:* `true` if the target function is suspended at final suspend point, otherwise `false`.

18.11.2.6 `coroutine_handle` promise access [coroutine.handle.prom]

```
Promise& promise() noexcept;
Promise const& promise() const noexcept;
```

1 *Requires:* `*this` refers to a coroutine.

2 *Returns:* a reference to a promise of the target function.

18.11.2.7 Comparison operators [coroutine.handle.compare]

```
bool operator==(coroutine_handle<> x, coroutine_handle<> y) noexcept;
```

1 *Returns:* `x.to_address() == y.to_address()`.

```
bool operator<(coroutine_handle<> x, coroutine_handle<> y) noexcept;
```

2 *Returns:* `x.to_address() < y.to_address()`.

```
bool operator!=(coroutine_handle<> x, coroutine_handle<> y) noexcept;
```

3 *Returns:* `!(x == y)`.

```
bool operator>(coroutine_handle<> x, coroutine_handle<> y) noexcept;
```

4 *Returns:* `(y < x)`.

```
bool operator<=(coroutine_handle<> x, coroutine_handle<> y) noexcept;
```

5 *Returns:* `!(x > y)`.

```
bool operator>=(coroutine_handle<> x, coroutine_handle<> y) noexcept;
```

6 *Returns:* `!(x < y)`.

18.11.2.8 Hash support [coroutine.handle.hash]

```
template <class P> struct hash<experimental::coroutine_handle<P>>;
```

1 The template specializations shall meet the requirements of class template hash (20.9.12).

18.11.3 Coroutine promise requirements [coroutine.promise]

1 A user supplies the definition of the coroutine promise to implement desired high-level semantics associated with a coroutines discovered via instantiation of struct template `coroutine_traits`. The following tables describe the requirements on coroutine promise types.

Table 4 — Descriptive variable definitions

Variable	Definition
`P`	a coroutine promise type
`p`	a value of type `P`
`e`	a value of `exception_ptr` type
`h`	a value of `experimental::coroutine_handle<P>` type
`v`	an *expression* or *braced-init-list*

Table 5 — `CoroutinePromise` requirements [**CoroutinePromise**]

Expression	Note
`P{}`	Construct an object of type `P`
`p.get_return_object()`	The `get_return_object` is invoked by the coroutine to construct the return object prior to reaching the first suspend-resume point, a `return` statement, or flowing off the end of the function.
`p.return_value(v)`	Invoked by a coroutine when a `coroutine-return-keyword` statement with an *expression* or a *braced-init-list* is encountered in a coroutine (6.6.4).
`p.return_void()`	If present, invoked when a `coroutine-return-keyword` statement is encountered as described in (6.6.4). A promise type shall not define both `return_void` and `return_value` member functions.
`p.set_exception(e)`	The `set_exception` is invoked by a coroutine when an unhandled exception occurs within a *function-body* of the coroutine. If the promise does not provide `set_exception`, an unhandled exception will propagate from the coroutine normally.
`p.yield_value(v)`	The `yield_value` is invoked when `yield-keyword` statement is encountered in the coroutine. If promise does not define `yield_value`, `yield-keyword` statement may not appear in the coroutine body.
`p.initial_suspend()`	if `p.initial_suspend()` evaluates to `true`, the coroutine will suspend at *initial suspend point* (8.4.4).
`p.final_suspend()`	if `p.final_suspend()` evaluates to `true`, the coroutine will suspend at *final suspend point* (8.4.4).

2 [*Example:* This example illustrates full implementation of a promise type for a simple generator.

```
#include <iostream>
#include <experimental/coroutine>

struct generator {
  struct promise_type {
    int current_value;
    auto get_return_object() { return generator{this}; }
    auto initial_suspend() { return true; }
    auto final_suspend() { return true; }
    void yield_value(int value) { current_value = value; }
  };

  bool move_next() {
    coro.resume();
    return !coro.done();
  }

  int current_value() { return coro.promise().current_value; }

  ~generator() { coro.destroy(); }
private:
  explicit generator(promise_type* myPromise) : coro(myPromise)
  {
  }
```

```
  std::experimental::coroutine_handle<promise_type> coro;
};

generator f() {
  yield-keyword 1;
  yield-keyword 2;
}

int main() {
  auto g = f();
  while (g.move_next()) std::cout << g.current_value() << std::endl;
}
```

— *end example*]