Document Number: P0057R2
Date: 2016-02-12
Revises: P0057R1
Audience: CWG / LEWG
Authors: Gor Nishanov <gorn@microsoft.com>
Jens Maurer <Jens.Maurer@gmx.net>
Richard Smith <richard@metafoo.co.uk>
Daveed Vandevoorde <daveed@edg.com>

Wording for Coroutines

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 a collaboration of researchers in industry and academia. We wish to thank people who made valuable contributions within and outside these groups, including Artur Laksberg, Chandler Carruth, David Vandevoorde, Deon Brewis, Gabriel Dos Reis, Herb Sutter, James McNellis, Jens Maurer, Jonathan Caves, Lawrence Crowl, Lewis Baker, Michael Wong, Nick Maliwacki, Niklas Gustafsson, Pablo Halpern, Richard Smith, Robert Schumacher, Shahms King, Slava Kuznetsov, Stephan T. Lavavej, Tongari J, Vladimir Petter, 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.

(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`	201602	*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 (5.3.8), or receipt of a signal).

2 Lexical conventions [lex]

2.12 Keywords [lex.key]

Add the keywords `co_await`, `co_yield`, and `co_return` 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.

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 (8.4.4). 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]

Add *await-expression* to the grammar production *unary-expression*:

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

5.3.8 Await [expr.await]

Add this section to 5.3.

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

await-expression:
  `co_await` *cast-expression*

2 An *await-expression* shall appear only in a potentially-evaluated expression within the *compound-statement* of a *function-body* outside of a *handler* (15). In a *declaration-statement* or in the *simple-declaration* (if any) of a *for-init-statement*, an *await-expression* shall appear only in an *initializer* of that *declaration-statement* or *simple-declaration*. An *await-expression* shall not appear in a default argument (8.3.6). A context within a function where an *await-expression* can appear is called a *suspension context* of the function.

3 Evaluation of an *await-expression* involves the following auxiliary expressions:

(3.1) — p is an lvalue naming the promise object (8.4.4) of the enclosing coroutine and P is the type of that object.

(3.2) — The *unqualified-id* `await_transform` is looked up within the scope of P by class member access lookup (3.4.5), and if this lookup finds at least one declaration, then a is *p*`.await_transform(`*cast-expression*`);` otherwise, a is *cast-expression*.

(3.3) — o is the result of the invocation of the unary `co_await` operator (13.5) applied to expression a.

(3.4) — If o is a prvalue, e is a temporary object copy-initialized from o, otherwise e is an lvalue referring to the result of evaluating o.

(3.5) — h is an object of type `std::coroutine_handle<`*P*`>` referring to the enclosing coroutine.

(3.6) — *await-ready* is the expression *e*`.await_ready()`, contextually converted to `bool`.

(3.7) — *await-suspend* is the expression *e*`.await_suspend(`*h*`)`, which shall be a prvalue of type `void` or `bool`.

(3.8) — *await-resume* is the expression e.`await_resume()`.

4 The *await-expression* has the same type and value category as the `await-resume` expression.

5 The *await-expression* evaluates the *await-ready* expression, then:

(5.1) — If the result is `false`, the coroutine is considered suspended. Then, the *await-suspend* expression is evaluated. If that expression has type `bool` and returns `false`, the coroutine is resumed. If that expression exits via an exception, the exception is caught, the coroutine is resumed, and the exception is immediately re-thrown (15.1). Otherwise, control flow returns to the current caller or resumer (8.4.4).

(5.2) — If the result is `true`, or when the coroutine is resumed, the *await-resume* expression is evaluated, and its result is the result of the *await-expression*.

6 [*Example:*

```
template <typename T>
struct my_future {
  ...
  bool await_ready();
  void await_suspend(std::coroutine_handle<>);
  T await_resume();
};

template <class Rep, class Period>
auto operator co_await(std::chrono::duration<Rep, Period> d) {
  struct awaiter {
    std::chrono::system_clock::duration duration;
    ...
    awaiter(std::chrono::system_clock::duration d) : duration(d){}
    bool await_ready() const { return duration.count() <= 0; }
    void await_resume() {}
    void await_suspend(std::coroutine_handle<> h){...}
  };
  return awaiter{d};
}

using namespace std::chrono;

my_future<void> g() {
  std::cout << "just about go to sleep...\n";
  co_await 10ms;
  std::cout << "resumed\n";
}
```

— *end example*]

5.18 Assignment and compound assignment operators [expr.ass]

Add *yield-expression* to the grammar production *assignment-expression*.

assignment-expression:
conditional-expression
logical-or-expression assignment-operator initializer-clause
throw-expression
yield-expression

5.21 Yield [expr.yield]

Add a new section to Clause 5.

yield-expression:
  `co_yield` *assignment-expression*
  `co_yield` *braced-init-list*

1 A *yield-expression* shall appear only within a suspension context of a function (5.3.8). Let e be the operand of the *yield-expression* and p be an lvalue naming the promise object of the enclosing coroutine (8.4.4), then the *yield-expression* is equivalent to the expression `co_await` p`.yield_value(`e`)`.

[*Example:*

```
template <typename T>
struct my_generator {
  struct promise_type {
    T current_value;
    ...
    auto yield_value(T v) {
      current_value = std::move(v);
      return std::suspend_always{};
    }
  };
  struct iterator { ... };
  iterator begin();
  iterator end();
};

my_generator<pair<int,int>> g1() {
  for (int i = i; i < 10; ++i) co_yield {i,i};
}
my_generator<pair<int,int>> g2() {
  for (int i = i; i < 10; ++i) co_yield make_pair(i,i);
}

auto f(int x = co_yield 5); // error: yield-expression outside of function suspension context
int a[] = { co_yield 1 }; // error: yield-expression outside of function suspension context

int main() {
  auto r1 = g1();
  auto r2 = g2();
  assert(std::equal(r1.begin(), r1.end(), r2.begin(), r2.end()));
}
```

— *end example*]

6 Statements [stmt.stmt]

6.5 Iteration statements [stmt.iter]

Add the 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` `co_await`$_{opt}$ `(` *for-range-declaration* `:` *for-range-initializer* `)` *statement*

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

Add the underlined text to paragraph 1.

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

`for` `co_await`$_{opt}$ `(` *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` `co_await`$_{opt}$ `(` *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 =  co_await_opt begin-expr,
  __end = end-expr;
  __begin != __end;
   co_await_opt ++__begin ) {
    for-range-declaration = *__begin;
    statement
  }
}
```

where `co_await` is present 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: ...

Add the following paragraph after paragraph 2.

3 A range-based `for` statement with `co_await` shall appear only within a suspension context of a function (5.3.8).

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 two productions to the grammar:

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

6.6.3 The `return` statement [stmt.return]

Add the underlined text to paragraph 1:

1 A function returns to its caller by the `return` statement; that function shall not be a coroutine (8.4.4).

Add the underlined text to the last sentence of paragraph 2:

2 ... Flowing off the end of a function that is not a coroutine is equivalent to a `return` with no value; this results in undefined behavior in a value-returning function.

6.6.3.1 The `co_return` statement [stmt.return.coroutine]

Add this section to 6.6.

coroutine-return-statement:
`co_return` *expression$_{opt}$*`;`
`co_return` *braced-init-list*`;`

1 A coroutine returns to its caller or resumer (8.4.4) by the `co_return` statement or when suspended (5.3.8).

2 The *expression* or *braced-init-list* of a `co_return` statement is called its operand. Let p be an lvalue naming the coroutine promise object (8.4.4) and P be the type of that object, then a `co_return` statement is equivalent to:

`{` S`; goto` $final\_suspend\_label$`; }`

where $final\_suspend\_label$ is as defined in 8.4.4 and S is an expression defined as follows:

(2.1) — S is p`.return_value(`*braced-init-list*`)`, if the operand is a *braced-init-list*;

(2.2) — S is p`.return_value(`*expression*`)`, if the operand is an expression of non-`void` type;

(2.3) — S is p`.return_void()`, otherwise;

S shall be a prvalue of type `void`.

3 [*Note:* See 8.4.4 about the flowing off the end of a coroutine. — *end note*]

7 Declarations [dcl.dcl]

7.1.5 The `constexpr` specifier [dcl.constexpr]

Insert a new bullet after paragraph 3 bullet 1.

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

(3.1) — it shall not be virtual (10.3);

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

(3.3) — . . .

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

Add the following paragraph.

15 A function declared with a return type that uses a placeholder type shall not be a coroutine (8.4.4).

8 Declarators [dcl.decl]

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 a *coroutine-return-statement* (6.6.3.1), an *await-expression* (5.3.8), a *yield-expression* (5.21), or a range-based `for` (6.5.4) with `co_await`. The *parameter-declaration-clause* of the coroutine shall not terminate with an ellipsis that is not part of a *parameter-declaration*.

2 [*Example:*

```
task<int> f();

task<void> g1() {
  int i = co_await f();
  std::cout << "f() => " << i << std::endl;
}

template <typename... Args>
task<void> g2(Args&&...) { // OK: ellipsis is a pack expansion
  int i = co_await f();
  std::cout << "f() => " << i << std::endl;
}

task<void> g3(int a, ...) { // error: variable parameter list not allowed
  int i = co_await f();
  std::cout << "f() => " << i << std::endl;
}
```

— *end example*]

3 For a coroutine f that 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 the type `std::coroutine_traits<`R`,`P_1`,...,`P_n`>`, and P be the class type denoted by T`::promise_type`. Then, the coroutine behaves as if its body were:

```
{
   P p;
   co_await p.initial_suspend(); // initial suspend point
   F'
final_suspend:
   co_await p.final_suspend(); // final suspend point
}
```

where *F'* is

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

if the *unqualified-id* `set_exception` is found in the scope of P by class member access lookup (3.4.5), and *F'* is F otherwise. An object denoted as *p* is the *promise object* of the coroutine and its type P is the *promise type* of the coroutine.

4 The *unqualified-ids* `return_void` and `return_value` are looked up in the scope of class P. If both are found, the program is ill-formed. If the *unqualified-id* `return_void` is found, flowing off the end of a coroutine is equivalent to a `co_return` with no operand. Otherwise, flowing off the end of a coroutine results in undefined behavior.

5 When a coroutine returns to its caller, the return value is obtained by a call to p`.get_return_object()`. A call to a `get_return_object` is sequenced before the call to `initial_suspend` and is invoked at most once.

6 A suspended coroutine can be resumed to continue execution by invoking a resumption member function (18.11.2.4) of an object of type `coroutine_handle<`P`>` associated with this instance of the coroutine. The function that invoked a resumption member function is called *resumer*. Invoking a resumption member function for a coroutine that is not suspended results in undefined behavior.

7 An implementation may need to allocate additional storage for the lifetime of a coroutine. This storage is known as the *coroutine state* and is obtained by calling a non-array allocation function (3.7.4.1). The allocation function's name is looked up in the scope of P. If this lookup fails, the allocation function's name is looked up in the global scope. If the lookup finds an allocation function that takes exactly one parameter, it will be used; otherwise, all parameters of the coroutine are passed to the allocation function after the size parameter in order. [*Note:* An allocation function template shall have two or more function parameters. A template instance is never considered to be an allocation function with exactly one parameter, regardless of its signature. — *end note*]

8 The coroutine state is destroyed when control flows off the end of the coroutine or the `destroy` member function (18.11.2.4) of an object of type `std::coroutine_handle<`P`>` associated with this 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. The dynamically allocated storage is released by calling a non-array deallocation function (3.7.4.2). If `destroy` is called for a coroutine that is not suspended, the program has undefined behavior.

9 The deallocation function's name is looked up in the scope of P. If this lookup fails, the deallocation function's name is looked up in the global scope. If deallocation function lookup finds both a usual deallocation function with only a pointer parameter and a usual deallocation function with both a pointer parameter and a size parameter, then the selected deallocation function shall be the one with two parameters. Otherwise, the selected deallocation function shall be the function with one parameter. If no usual deallocation function is found, the program is ill-formed.

10 When a coroutine is invoked, an implementation may create a copy of one or more coroutine parameters. Each such copy is direct-initialized from an lvalue referring to the corresponding parameter if it is an lvalue reference, and an xvalue referring to it otherwise. A reference to a parameter in the function-body of the coroutine is replaced by a reference to its copy.

11 The unqualified-id `get_return_object_on_allocation_failure` is looked up in the scope of class P by class member access lookup (3.4.5). If a declaration is found, then `std::nothrow_t` forms of allocation and deallocation functions are used. If an allocation function returns `nullptr`, the coroutine returns control to the caller of the coroutine and the return value is obtained by a call to P`::get_return_object_on_allocation_failure()`.

[*Example:*

```
// using nothrow operator new
struct generator {
  using handle = std::coroutine_handle<promise_type>;
  struct promise_type {
    int current_value;
```

```
    static auto get_return_object_on_allocation_failure() { return generator{nullptr}; }
    auto get_return_object() { return generator{handle::from_promise(*this)}; }
    auto initial_suspend() { return std::suspend_always{}; }
    auto final_suspend() { return std::suspend_always{}; }
    auto yield_value(int value) {
      current_value = value;
      return std::suspend_always{};
    }
  };
  bool move_next() { return coro ? (coro.resume(), !coro.done()) : false; }
  int current_value() { return coro.promise().current_value; }
  ~generator() { if(coro) coro.destroy(); }
private:
  generator(handle h) : coro(h) {}
  handle coro;
};
generator f() { co_yield 1;  co_yield 2; }

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

— *end example*]

12 [*Example:*

```
// using a stateful allocator
class Arena;
struct my_coroutine {
  struct promise_type {
    ...
    template <typename... TheRest>
    void* operator new(std::size_t size, Arena& pool, TheRest const&...) {
      return pool.allocate(size);
    }
    void* operator delete(void* p, std::size_t size) {
      // reference to a pool is not available
      // to the delete operator and should be stored
      // by the allocator as a part of the allocation
      return Arena::deallocate(p, size);
    }
  };
};

my_coroutine (Arena& a) {
  // will call my_coroutine::promise_type::operator new(<required-size>, a)
  // to obtain storage for the coroutine state
  co_yield 1;
}

int main() {
  Pool memPool;
  for (int i = 0; i < 1'000'000; ++i) my_coroutine(memPool);
};
```

— *end example*]

12 Special member functions [special]

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]

Modify paragraph 33 as follows:

33 When the criteria for elision of a copy/move operation are met, but not for an *exception-declaration*, and the object to be copied is designated by an lvalue, or when the *expression* in a `return` or `co_return` statement is a (possibly parenthesized) *id-expression* that names an object with automatic storage duration declared in the body or *parameter-declaration-clause* of the innermost enclosing function or *lambda-expression*, overload resolution to select the constructor for the copy or the `return_value` overload to call is first performed as if the object were designated by an rvalue. If the first overload resolution fails or was not performed, or if the type of the first parameter of the selected constructor or `return_value` overload is not an rvalue reference to the object's type (possibly cv-qualified), overload resolution is performed again, considering the object as an lvalue. [*Note:* This two-stage overload resolution must be performed regardless of whether copy elision will occur. It determines the constructor or `return_value` overload to be called if elision is not performed, and the selected constructor or `return_value` overload must be accessible even if the call is elided. — *end note*]

13 Overloading [over]

13.5 Overloaded operators [over.oper]

Add `co_await` to the list of operators in paragraph 1 before operators `()` and `[]`.

13.6 Built-in operators [over.built]

Add the underlined text to the note in paragraph 1.

1 The candidate operator functions that represent the built-in operators defined in Clause 5 are specified in this subclause. These candidate functions participate in the operator overload resolution process as described in 13.3.1.2 and are used for no other purpose. [*Note:* Because built-in operators except for operator `co_await` take only operands with non-class type, and operator overload resolution occurs only when an operand expression originally has class or enumeration type, operator overload resolution can resolve to a built-in operator only when an operand has a class type that has a user-defined conversion to a non-class type appropriate for the operator, or when an operand has an enumeration type that can be converted to a type appropriate for the operator. Also note that some of the candidate operator functions given in this subclause are more permissive than the built-in operators themselves. As described in 13.3.1.2, after a built-in operator is selected by overload resolution the expression is subject to the requirements for the built-in operator given in Clause 5, and therefore to any additional semantic constraints given there. If there is a user-written candidate with the same name and parameter types as a built-in candidate operator function, the built-in operator function is hidden and is not included in the set of candidate functions. — *end note*]

Add new paragraph after paragraph 25.

26 For every pair (T, CV), where T is a class type containing declarations of any of the following names: `await_ready`, `await_suspend`, `await_resume`, and CQ is *cv-qualifier-seq*, there exist candidate operator functions of the form

```
CV T& operator co_await(CV T&);
CV T&& operator co_await(CV T&&);
```

which return their operand as the result.

18 Language support library [language.support]

18.1 General [support.general]

Add a row to Table 2 for coroutine support header <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	<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 <coroutine> defines several types providing compile and run-time support for coroutines in a C++ program.

Header <coroutine> synopsis

```
namespace std {
  // 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;

  // 18.11.3 trivial awaitables
  struct suspend_never;
  struct suspend_always;

  // 18.11.2.8 hash support:
  template <class T> struct hash;
  template <class P> struct hash<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 Struct template `coroutine_traits` [coroutine.traits.primary]

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

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

18.11.2 Struct template `coroutine_handle` [coroutine.handle]

```
namespace std {
  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
    void* address() const noexcept;
    static coroutine_handle from_address(void* addr) 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;
    static coroutine_handle from_promise(Promise&) noexcept;
    coroutine_handle& operator=(nullptr_t) noexcept;

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

1 Let P be a promise type of the coroutine (8.4.4). An object of type `coroutine_handle<P>` is called a *coroutine handle* and can be used to refer to a suspended or executing coroutine. A default constructed `coroutine_handle` object does not refer to any coroutine.

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

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

1 *Postconditions:* `address() == nullptr`.

```
static coroutine_handle coroutine_handle::from_promise(Promise& p) noexcept;
```

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

3 *Returns:* coroutine handle h refering to the coroutine.

4 *Postconditions:* `addressof(`h`.promise()) == addressof(p)`.

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

5 *Postconditions:* `address() == nullptr`.

6 *Returns:* `*this`.

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

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

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

18.11.2.3 coroutine_handle capacity [coroutine.handle.capacity]

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

1 *Returns:* `true` if `address() != nullptr`, 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 the coroutine. If the coroutine was suspended at the final suspend point, behavior is undefined.

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

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

4 *Effects:* destroys the 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 coroutine 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 coroutine.

18.11.2.7 Comparison operators [coroutine.handle.compare]

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

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

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

2 *Returns:* `less<void*>()(x.address(), y.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<coroutine_handle<P>>;
```

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

18.11.3 Trivial awaitables [coroutine.trivial.awaitables]

The header `<coroutine>` shall define `suspend_never` and `suspend_always` as follows.

```
struct suspend_never {
  bool await_ready() { return true; }
  void await_suspend(coroutine_handle<>) {}
  void await_resume() {}
};
struct suspend_always {
  bool await_ready() { return false; }
  void await_suspend(coroutine_handle<>) {}
  void await_resume() {}
};
```