Document Number: N4775
Date: 2018-10-07
Revises: N4760
Reply to: Gor Nishanov <gorn@microsoft.com>

Working Draft, C++ Extensions 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 Scope [intro.scope]

1 This document describes extensions to the C++ Programming Language (Clause 2) that enable definition of coroutines. These extensions include new syntactic forms and modifications to existing language semantics.

2 The International Standard, ISO/IEC 14882:2017, provides important context and specification for this document. 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.

2 Normative references [intro.refs]

1 The following documents are referred to in the text in such a way that some or all of their content constitutes requirements 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 14882:2017, *Programming Languages – C++*

ISO/IEC 14882:2017 is hereafter called the *C++ Standard*. Beginning with Clause 5, all clause and subclause numbers, titles, and symbolic references in [brackets] refer to the corresponding elements of the C++ Standard. Clauses 1 through 4 of this document are unrelated to the similarly-numbered clauses and subclauses of the C++ Standard.

3 Terms and definitions [intro.defs]

No terms and definitions are listed in this document. ISO and IEC maintain terminological databases for use in standardization at the following addresses:

- — ISO Online browsing platform: available at `http://www.iso.org/obp`
- — IEC Electropedia: available at `http://www.electropedia.org/`

4 General [intro]

4.1 Implementation compliance [intro.compliance]

Conformance requirements for this specification shall be the same as those defined in subclause 4.1 of the C++ Standard. [*Note:* Conformance is defined in terms of the behavior of programs. — *end note*]

4.2 Feature testing [intro.features]

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

Table 1 — Feature-test macro

Name	Value	Header
`__cpp_coroutines`	201806	*predeclared*

4.3 Program execution [intro.execution]

In subclause 4.6 of the C++ Standard modify paragraph 6 to read:

7 An instance of each object with automatic storage duration (6.7.3) 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 (8.3.8), or receipt of a signal).

5 Lexical conventions [lex]

5.11 Keywords [lex.key]

Add the keywords `co_await`, `co_yield`, and `co_return` to Table 5 "Keywords".

6 Basic concepts [basic]

6.6.1 Main function [basic.start.main]

Add underlined text to paragraph 3.

3 The function `main` shall not be used within a program. The linkage (6.5) 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 (11.4.4). ...

6.7.4.1 Allocation functions [basic.stc.dynamic.allocation]

Modify paragraph 4 as follows:

4

A global allocation function is only called as the result of a new expression (8.3.4), or called directly using the function call syntax (8.2.2), or called indirectly to allocate storage for a coroutine frame (11.4.4), or called indirectly through calls to the functions in the C++ standard library. [*Note:* In particular, a global allocation function is not called to allocate storage for objects with static storage duration (6.7.1), for objects or references with thread storage duration (6.7.2), for objects of type `std::type_info` (8.2.8), or for an exception object (18.1). — *end note*]

7 Standard Conversions [conv]

No changes are made to Clause 7 of the C++ Standard.

8 Expressions [expr]

8.3 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

8.3.8 Await [expr.await]

Add this subclause to 8.3.

1 The `co_await` expression is used to suspend evaluation of a coroutine (11.4.4) 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* (Clause 18). 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 (11.3.6). 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 types, expressions, and objects:

(3.1) — p is an lvalue naming the promise object (11.4.4) of the enclosing coroutine and P is the type of that object.

(3.2) — a is the *cast-expression* if the *await-expression* was implicitly produced by a *yield-expression* (8.21), an initial suspend point, or a final suspend point (11.4.4). Otherwise, the *unqualified-id* `await_transform` is looked up within the scope of P by class member access lookup (6.4.5), and if this lookup finds at least one declaration, then a is *p*`.await_transform(`*cast-expression*`);` otherwise, a is the *cast-expression*.

(3.3) — o is determined by enumerating the applicable `operator co_await` functions for an argument a (16.3.1.2), and choosing the best one through overload resolution (16.3). If overload resolution is ambiguous, the program is ill-formed. If no viable functions are found, o is a. Otherwise, o is a call to the selected function.

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

(3.5) — h is an object of type `std::experimental::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`, `bool`, or `std::experimental::coroutine_handle<Z>` for some type Z.

(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 `std::experimental::coroutine_handle<Z>` and evaluates to a value s, the coroutine referred to by s is resumed as if by a call s`.resume()`. [*Note:* Any number of coroutines may be successively resumed in this fashion, eventually returning control flow to the current coroutine caller or resumer (11.4.4). — *end note*] If that expression has type `bool` and evaluates to `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 (18.1). Otherwise, control flow returns to the current coroutine caller or resumer (11.4.4) without exiting any scopes (9.6).

(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::experimental::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::experimental::coroutine_handle<> h){...}
  };
  return awaiter{d};
}

using namespace std::chrono;

my_future<int> h();

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

```
}

auto f(int x = co_await h()); // error: await-expression outside of function suspension context
int a[] = { co_await h() }; // error: await-expression outside of function suspension context
```

— *end example*]

8.18 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

8.20 Constant expressions [expr.const]

Add bullets prohibiting *await-expression* and *yield-expression* to paragraph 2.

— an *await-expression* (8.3.8);

— a *yield-expression* (8.21);

8.21 Yield [expr.yield]

Add a new subclause to Clause 8.

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 (8.3.8). Let *e* be the operand of the *yield-expression* and *p* be an lvalue naming the promise object of the enclosing coroutine (11.4.4), 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::experimental::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*]

9 Statements [stmt.stmt]

9.5 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` <u>`co_await`$_{opt}$</u> `(` *for-range-declaration* `:` *for-range-initializer* `)` *statement*

9.5.4 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` <u>`co_await`$_{opt}$</u> `(` *for-range-declaration : for-range-initializer* `)` *statement*

is equivalent to

```
{
    auto &&__range = for-range-initializer ;
    auto __begin = co_await_opt begin-expr ;
    auto __end = end-expr ;
    for ( ; __begin != __end; co_await_opt ++__begin ) {
        for-range-declaration = *__begin;
        statement
    }
}
```

Insert a new bullet after paragraph 1 bullet 1.

(1.1) — if the *for-range-initializer* is an *expression*, it is regarded as if it were surrounded by parentheses (so that a comma operator cannot be reinterpreted as delimiting two *init-declarator*s);

(1.2) — <u>co_await is present if and only if it appears immediately after the `for` keyword;</u>

(1.3) — `__range`, `__begin`, and `__end` are variables defined for exposition only; and ...

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 (8.3.8).

9.6 Jump statements [stmt.jump]

Add *coroutine-return-statement* to the grammar production *jump-statement*:

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

Add the underlined text to paragraph 2:

2 On exit from a scope (however accomplished), objects with automatic storage duration (6.7.3) that have been constructed in that scope are destroyed in the reverse order of their construction. [*Note:* A suspension of a coroutine (8.3.8) is not considered to be an exit from a scope. — *end note*] ...

9.6.3 The `return` statement [stmt.return]

Add the underlined text to paragraph 2:

2 ... Flowing off the end of a constructor, a destructor, or a function that is not a coroutine with a *cv* `void` return type is equivalent to a `return` with no operand. Otherwise, flowing off the end of a function other than main (6.6.1) or a coroutine (11.4.4) results in undefined behavior.

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

Add this subclause to 9.6.3.

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

1 A coroutine returns to its caller or resumer (11.4.4) by the `co_return` statement or when suspended (8.3.8). A coroutine shall not return to its caller or resumer by a `return` statement (9.6.3).

2 The *expr-braced-init-list* of a `co_return` statement is called its operand. Let p be an lvalue naming the coroutine promise object (11.4.4) and P be the type of that object, then a `co_return` statement is equivalent to:

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

where $final\_suspend$ is as defined in 11.4.4 and S is defined as follows:

(2.1) — S is p`.return_value(`*expr-or-braced-init-list*`)`, if the operand is a *braced-init-list* or an expression of non-`void` type;

(2.2) — S is `{` *expression$_{opt}$* `;` p`.return_void(); }`, otherwise;

S shall be a prvalue of type `void`.

3 If p`.return_void()` is a valid expression, 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.

10 Declarations [dcl.dcl]

10.1 Specifiers [dcl.spec]

10.1.5 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 (13.3);

(3.2) — it shall not be a coroutine (11.4.4);

(3.3) — . . .

10.1.6.4 `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 (11.4.4).

11 Declarators [dcl.decl]

11.4 Function definitions [dcl.fct.def]

11.4.4 Coroutines [dcl.fct.def.coroutine]

Add this subclause to 11.4.

1 A function is a *coroutine* if it contains a *coroutine-return-statement* (9.6.3.1), an *await-expression* (8.3.8), a *yield-expression* (8.21), or a range-based `for` (9.5.4) 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 (16.3.1) 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 p_1 ... p_n be lvalues denoting those objects. Let R be the return type and F be the *function-body* of *f*, T be the type `std::experimental::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 promise-constructor-arguments ;
   co_await p.initial_suspend(); // initial suspend point
   try { F } catch(...) { p .unhandled_exception(); }
final_suspend :
   co_await p.final_suspend(); // final suspend point
}
```

where an object denoted as p is the *promise object* of the coroutine and its type P is the *promise type* of the coroutine, and *promise-constructor-arguments* is determined as follows: overload resolution is performed on a promise constructor call created by assembling an argument list with lvalues p_1 ... p_n. If a viable constructor is found (16.3.2), then *promise-constructor-arguments* is `(`p_1,...,p_n`)`, otherwise *promise-constructor-arguments* is empty.

4 The *unqualified-id*s `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 produced 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 (21.11.2.4) 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 a coroutine. This storage is known as the *coroutine state* and is obtained by calling a non-array allocation function (6.7.4.1). 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 in the scope of P, overload resolution is performed on a function call created by assembling an argument list. The first argument is the amount of space requested, and has type `std::size_t`. The lvalues p_1 ... p_n are the succeeding arguments. If no viable function is found (16.3.2), overload resolution is performed again on a function call created by passing just the amount of space required as an argument of type `std::size_t`.

8 The *unqualified-id* `get_return_object_on_allocation_failure` is looked up in the scope of class P by class member access lookup (6.4.5). If a declaration is found, then the result of a call to an allocation function used to obtain storage for the coroutine state is assumed to return `nullptr` if it fails to obtain storage, and if a global allocation function is selected, the `::operator new(size_t, nothrow_t)` form shall be 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()`. The allocation function used in this case must have a non-throwing *noexcept-specification*.

[*Example:*

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

// ::operator new(size_t, nothrow_t) will be used if allocation is needed
struct generator {
  struct promise_type;
  using handle = std::experimental::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::experimental::suspend_always{}; }
    auto final_suspend() { return std::experimental::suspend_always{}; }
    void unhandled_exception() { std::terminate(); }
    void return_void() {}
    auto yield_value(int value) {
      current_value = value;
      return std::experimental::suspend_always{};
    }
  };
  bool move_next() { return coro ? (coro.resume(), !coro.done()) : false; }
  int current_value() { return coro.promise().current_value; }
  generator(generator const&) = delete;
  generator(generator && rhs) : coro(rhs.coro) { rhs.coro = nullptr; }
  ~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*]

9 The coroutine state is destroyed when control flows off the end of the coroutine or the `destroy` member function (21.11.2.4) of an object of type `std::experimental::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 storage for the coroutine state is released by calling a non-array deallocation function (6.7.4.2). If `destroy` is called for a coroutine that is not suspended, the program has undefined behavior.

10 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. The selected deallocation function shall be called with the address of the block of storage to be reclaimed as its first argument. If a deallocation function with a parameter of type `std::size_t` is used, the size of the block is passed as the corresponding argument.

11 When a coroutine is invoked, a copy is created for each coroutine parameter. Each such copy is an object with automatic storage duration that is direct-initialized from an lvalue referring to the corresponding parameter if the parameter is an lvalue reference, and from an xvalue referring to it otherwise. A reference to a parameter in the function-body of the coroutine and in the call to the coroutine promise constructor is replaced by a reference to its copy. The initialization and destruction of each parameter copy occurs in the context of the called coroutine. Initializations of parameter copies are sequenced before the call to the coroutine promise constructor and indeterminately sequenced with respect to each other. The lifetime of parameter copies ends immediately after the lifetime of the coroutine promise object ends. [*Note:* If a coroutine has a parameter passed by reference, resuming the coroutine after the lifetime of the entity referred to by that parameter has ended is likely to result in undefined behavior. — *end note*]

12 Classes [class]

No changes are made to Clause 12 of the C++ Standard.

13 Derived classes [class.derived]

No changes are made to Clause 13 of the C++ Standard.

14 Member Access Control [class.access]

No changes are made to Clause 14 of the C++ Standard.

15 Special member functions [special]

15.1 Constructors [class.ctor]

Add new paragraph after paragraph 10.

11 A constructor shall not be a coroutine.

15.4 Destructors [class.dtor]

Add new paragraph after paragraph 16.

17 A destructor shall not be a coroutine.

15.8 Copying and moving class objects [class.copy]

15.8.3 Copy/move elision [class.copy.elision]

Add a bullet to paragraph 1:

— in a coroutine (11.4.4), a copy of a coroutine parameter can be omitted and references to that copy replaced with references to the corresponding parameter if the meaning of the program will be unchanged except for the execution of a constructor and destructor for the parameter copy object

Modify paragraph 3 as follows:

3 In the following copy-initialization contexts, a move operation might be used instead of a copy operation:

(3.1) — If the *expression* in a `return` or `co_return` statement (9.6.3) 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*, or

(3.2) — if the operand of a *throw-expression* is the name of a non-volatile automatic object (other than a function or catch-clause parameter) whose scope does not extend beyond the end of the innermost enclosing *try-block* (if there is one),

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. *Remark:* 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.

16 Overloading [over]

16.5 Overloaded operators [over.oper]

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

Add the following paragraph after paragraph 5.

6 The `co_await` operator is described completely in 8.3.8. The attributes and restrictions found in the rest of this subclause do not apply to it unless explicitly stated in 8.3.8.

17 Templates [temp]

No changes are made to Clause 17 of the C++ Standard.

18 Exception handling [except]

No changes are made to Clause 18 of the C++ Standard.

19 Preprocessing directives [cpp]

No changes are made to Clause 19 of the C++ Standard.

20 Library introduction [library]

20.5.1.3 Freestanding implementations [compliance]

Add a row to Table 19 for coroutine support header `<experimental/coroutine>`.

Table 19 — C++ headers for freestanding implementations

Subclause		Header(s)
		`<ciso646>`
21.2	Types	`<cstddef>`
21.3	Implementation properties	`<cfloat>` `<limits>` `<climits>`
21.4	Integer types	`<cstdint>`
21.5	Start and termination	`<cstdlib>`
21.6	Dynamic memory management	`<new>`
21.7	Type identification	`<typeinfo>`
21.8	Exception handling	`<exception>`
21.9	Initializer lists	`<initializer_list>`
21.10	Other runtime support	`<cstdalign>` `<cstdarg>` `<cstdbool>`
21.11	Coroutines support	`<experimental/coroutine>`
23.15	Type traits	`<type_traits>`
32	Atomics	`<atomic>`

21 Language support library [language.support]

21.1 General [support.general]

Add a row to Table 32 for coroutine support header <experimental/coroutine>.

Table 32 — Language support library summary

Subclause		Header(s)
21.2	Types	<cstddef>
21.3	Implementation properties	<limits> <climits> <cfloat>
21.4	Integer types	<cstdint>
21.5	Start and termination	<cstdlib>
21.6	Dynamic memory management	<new>
21.7	Type identification	<typeinfo>
21.8	Exception handling	<exception>
21.9	Initializer lists	<initializer_list>
21.10	Other runtime support	<csignal> <csetjmp> <cstdalign> <cstdarg> <cstdbool> <cstdlib> <ctime>
21.11	Coroutines support	<experimental/coroutine>

21.10.2 Header <csetjmp> synopsis [csetjmp.syn]

Add underlined text to paragraph 2.

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 suspension context of a coroutine (8.3.8).

See also: ISO C 7.13.

21.11 Coroutines support library [support.coroutine]

Add this subclause to Clause 21.

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 {

  // 21.11.1 coroutine traits
  template <class R, class... ArgTypes>
    struct coroutine_traits;

  // 21.11.2 coroutine handle
  template <class Promise = void>
    struct coroutine_handle;

  // 21.11.3 noop coroutine promise
  struct noop_coroutine_promise;
  template <> struct coroutine_handle<noop_coroutine_promise>;
  // noop coroutine handle
  using noop_coroutine_handle = coroutine_handle<noop_coroutine_promise>;

  // 21.11.4 noop coroutine
  noop_coroutine_handle noop_coroutine() noexcept;

  // 21.11.2.6 comparison operators:
  constexpr bool operator==(coroutine_handle<> x, coroutine_handle<> y) noexcept;
  constexpr bool operator!=(coroutine_handle<> x, coroutine_handle<> y) noexcept;
  constexpr bool operator<(coroutine_handle<> x, coroutine_handle<> y) noexcept;
  constexpr bool operator>(coroutine_handle<> x, coroutine_handle<> y) noexcept;
  constexpr bool operator<=(coroutine_handle<> x, coroutine_handle<> y) noexcept;
  constexpr bool operator>=(coroutine_handle<> x, coroutine_handle<> y) noexcept;

  // 21.11.5 trivial awaitables
  struct suspend_never;
  struct suspend_always;

} // namespace coroutines_v1
} // namespace experimental

// 21.11.2.7 hash support:
template <class T> struct hash;
template <class P> struct hash<experimental::coroutine_handle<P>>;

} // namespace std
```

21.11.1 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.

21.11.1.1 Class template `coroutine_traits` [coroutine.traits.primary]

1 The header `<experimental/coroutine>` defines the primary template `coroutine_traits` such that if `ArgTypes` is a parameter pack of types and if the *qualified-id* `R::promise_type` is valid and denotes a type (17.9.2), then `coroutine_traits<R,ArgTypes...>` has the following publicly accessible member:

```
using promise_type = typename R::promise_type;
```

Otherwise, `coroutine_traits<R,ArgTypes...>` has no members.

2 Program defined specializations of this template shall define a publicly accessible nested type named `promise_type`.

21.11.2 Class template `coroutine_handle` [coroutine.handle]

```
namespace std {
namespace experimental {
inline namespace coroutines_v1 {

  template <>
  struct coroutine_handle<void>
  {
    // 21.11.2.1 construct/reset
    constexpr coroutine_handle() noexcept;
    constexpr coroutine_handle(nullptr_t) noexcept;
    coroutine_handle& operator=(nullptr_t) noexcept;

    // 21.11.2.2 export/import
    constexpr void* address() const noexcept;
    constexpr static coroutine_handle from_address(void* addr);

    // 21.11.2.3 observers
    constexpr explicit operator bool() const noexcept;
    bool done() const;

    // 21.11.2.4 resumption
    void operator()() const;
    void resume() const;
    void destroy() const;

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

  template <class Promise>
  struct coroutine_handle : coroutine_handle<>
  {
    // 21.11.2.1 construct/reset
    using coroutine_handle<>::coroutine_handle;
    static coroutine_handle from_promise(Promise&);
    coroutine_handle& operator=(nullptr_t) noexcept;

    // 21.11.2.2 export/import
    constexpr static coroutine_handle from_address(void* addr);

    // 21.11.2.5 promise access
    Promise& promise() const;
  };

  template <> struct coroutine_handle<noop_coroutine_promise> : coroutine_handle<>
  {
  // 21.11.2.8 noop observers
  constexpr explicit operator bool() const noexcept;
  constexpr bool done() const noexcept;
```

```
    // 21.11.2.9 noop resumption
    constexpr void operator()() const noexcept;
    constexpr void resume() const noexcept;
    constexpr void destroy() const noexcept;

    // 21.11.2.10 noop promise access
    noop_coroutine_promise& promise() const noexcept;

    // 21.11.2.11 noop address
    constexpr void* address() const noexcept;
  private:
    coroutine_handle(unspecified);
  };

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

1 Let P be the promise type of a coroutine (11.4.4). 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.

2 If a program declares an explicit or partial specialization of `coroutine_handle`, the behavior is undefined.

21.11.2.1 `coroutine_handle` construct/reset [coroutine.handle.con]

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

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

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

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

3 *Returns:* A coroutine handle h referring to the coroutine.

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

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

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

6 *Returns:* `*this`.

21.11.2.2 `coroutine_handle` export/import [coroutine.handle.export.import]

```
constexpr void* address() const noexcept;
```

1 *Returns:* `ptr`.

```
constexpr static coroutine_handle<> coroutine_handle<>::from_address(void* addr);
constexpr static coroutine_handle<Promise> coroutine_handle<Promise>::from_address(void* addr);
```

2 *Requires:* `addr` was obtained via a prior call to `address`.

3 *Postconditions:* `from_address(address()) == *this`.

21.11.2.3 `coroutine_handle` observers [coroutine.handle.observers]

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

1 *Returns:* `address() != nullptr`.

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

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

3 *Returns:* `true` if the coroutine is suspended at its final suspend point, otherwise `false`.

21.11.2.4 `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 its final suspend point, behavior is undefined.

3 [*Note:* A concurrent resumption of the coroutine via `resume`, `operator()`, or `destroy` may result in a data race. — *end note*]

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

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

5 *Effects:* Destroys the coroutine (11.4.4).

6 [*Note:* A concurrent resumption of the coroutine via `resume`, `operator()`, or `destroy` may result in a data race. — *end note*]

21.11.2.5 `coroutine_handle` promise access [coroutine.handle.promise]

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

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

2 *Returns:* A reference to the promise of the coroutine.

21.11.2.6 Comparison operators [coroutine.handle.compare]

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

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

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

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

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

3 *Returns:* `less<>()(x.address(), y.address())`.

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

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

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

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

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

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

21.11.2.7 Hash support [coroutine.handle.hash]

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

1 The specialization is enabled (23.14.15).

21.11.2.8 noop_coroutine_handle observers [coroutine.handle.noop.observers]

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

1 *Returns:* true.

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

2 *Returns:* false.

21.11.2.9 noop_coroutine_handle resumption [coroutine.handle.noop.resumption]

```
constexpr void operator()() const noexcept;
constexpr void resume() const noexcept;
constexpr void destroy() const noexcept;
```

1 *Effects:* None.

2 *Remarks:* If noop_coroutine_handle is converted to coroutine_handle<>, calls to operator(), resume and destroy on that handle will also have no observable effects.

21.11.2.10 noop_coroutine_handle promise access [coroutine.handle.noop.promise]

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

1 *Returns:* A reference to the promise object associated with this coroutine handle.

21.11.2.11 noop_coroutine_handle address [coroutine.handle.noop.address]

```
constexpr void* address() const noexcept;
```

1 *Returns:* ptr.

2 *Remarks:* A noop_coroutine_handle's ptr is always a non-null pointer value.

21.11.3 Class noop_coroutine_promise [coroutine.promise.noop]

```
struct noop_coroutine_promise {};
```

1 The class noop_coroutine_promise defines the promise type for the coroutine referred to by noop_coroutine_handle (21.11).

21.11.4 Function noop_coroutine [coroutine.noop]

```
noop_coroutine_handle noop_coroutine() noexcept;
```

1 *Returns:* A handle to a coroutine that has no observable effects when resumed or destroyed.

2 *Remarks:* A handle returned from noop_coroutine may or may not compare equal to a handle returned from another invocation of noop_coroutine.

21.11.5 Trivial awaitables [coroutine.trivial.awaitables]

The header <experimental/coroutine> defines suspend_never and suspend_always as follows.

```
namespace std {
namespace experimental {
inline namespace coroutines_v1 {

  struct suspend_never {
    constexpr bool await_ready() const noexcept { return true; }
    constexpr void await_suspend(coroutine_handle<>) const noexcept {}
    constexpr void await_resume() const noexcept {}
  };
```

```
  struct suspend_always {
    constexpr bool await_ready() const noexcept { return false; }
    constexpr void await_suspend(coroutine_handle<>) const noexcept {}
    constexpr void await_resume() const noexcept {}
  };

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