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

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Contents</strong></td>
<td>ii</td>
</tr>
<tr>
<td>List of Tables</td>
<td>iii</td>
</tr>
<tr>
<td>1 General</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Scope</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Acknowledgements</td>
<td>1</td>
</tr>
<tr>
<td>1.3 Normative references</td>
<td>1</td>
</tr>
<tr>
<td>1.4 Implementation compliance</td>
<td>1</td>
</tr>
<tr>
<td>1.5 Feature testing</td>
<td>1</td>
</tr>
<tr>
<td>1.9 Program execution</td>
<td>1</td>
</tr>
<tr>
<td>2 Lexical conventions</td>
<td>3</td>
</tr>
<tr>
<td>2.12 Keywords</td>
<td>3</td>
</tr>
<tr>
<td>3 Basic concepts</td>
<td>4</td>
</tr>
<tr>
<td>5 Expressions</td>
<td>5</td>
</tr>
<tr>
<td>5.3 Unary expressions</td>
<td>5</td>
</tr>
<tr>
<td>5.17 Assignment and compound assignment operators</td>
<td>7</td>
</tr>
<tr>
<td>5.19 Constant expressions</td>
<td>7</td>
</tr>
<tr>
<td>5.20 Yield</td>
<td>7</td>
</tr>
<tr>
<td>6 Statements</td>
<td>9</td>
</tr>
<tr>
<td>6.5 Iteration statements</td>
<td>9</td>
</tr>
<tr>
<td>6.6 Jump statements</td>
<td>10</td>
</tr>
<tr>
<td>7 Declarations</td>
<td>11</td>
</tr>
<tr>
<td>8 Declarators</td>
<td>12</td>
</tr>
<tr>
<td>8.4 Function definitions</td>
<td>12</td>
</tr>
<tr>
<td>12 Special member functions</td>
<td>16</td>
</tr>
<tr>
<td>12.1 Constructors</td>
<td>16</td>
</tr>
<tr>
<td>12.4 Destructors</td>
<td>16</td>
</tr>
<tr>
<td>12.8 Copying and moving class objects</td>
<td>16</td>
</tr>
<tr>
<td>13 Overloading</td>
<td>17</td>
</tr>
<tr>
<td>13.5 Overloaded operators</td>
<td>17</td>
</tr>
<tr>
<td>17 Library introduction</td>
<td>18</td>
</tr>
<tr>
<td>18 Language support library</td>
<td>19</td>
</tr>
<tr>
<td>18.1 General</td>
<td>19</td>
</tr>
<tr>
<td>18.10 Other runtime support</td>
<td>19</td>
</tr>
<tr>
<td>18.11 Coroutines support library</td>
<td>19</td>
</tr>
</tbody>
</table>
# List of Tables

<table>
<thead>
<tr>
<th>Number</th>
<th>Table Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Feature-test macro</td>
<td>1</td>
</tr>
<tr>
<td>16</td>
<td>C++ headers for freestanding implementations</td>
<td>18</td>
</tr>
<tr>
<td>30</td>
<td>Language support library summary</td>
<td>19</td>
</tr>
</tbody>
</table>
1 General

1.1 Scope

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.

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

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 Vangool, Deon Brewis, Eric Fisler, Gabriel Dos Reis, Herb Sutter, James McNellis, Jens Maurer,
Jonathan Caves, Lawrence Crowl, Lewis Baker, Michael Wong, Nick Miliwacki, 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

The following referenced document is indispensable for the application of this document. For dated refer-
ences, only the edition cited applies.

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

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

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

Table 1 — Feature-test macro

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Header</th>
</tr>
</thead>
<tbody>
<tr>
<td>__cpp_coroutines</td>
<td>201606</td>
<td>predeclared</td>
</tr>
</tbody>
</table>

1.9 Program execution

Modify paragraph 7 to read:

§ 1.9
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

2.12 Keywords

Add the keywords `co_await`, `co_yield`, and `co_return` to Table 4 "Keywords".
3 Basic concepts

3.6.1 Main function

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]

3.7.4.1 Allocation functions

Modify paragraph 4 as follows:

A global allocation function is only called as the result of a new expression (5.3.4), called directly using the function call syntax (5.2.2), called indirectly to allocate storage for a coroutine frame (8.4.4), 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 (3.7.1), for objects or references with thread storage duration (3.7.2), for objects of type `std::type_info` (5.2.8), or for an exception object (15.1). — end note]
5 Expressions

5.3 Unary expressions

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

Add this section to 5.3.

1 The `co_await` expression 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 types, expressions, and objects:

   (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) — `a` is the `cast-expression` if the `await-expression` was implicitly produced by a `yield-expression` (5.20), an initial suspend point, or a final suspend point (8.4.4). Otherwise, 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 the `cast-expression`.

   (3.3) — `o` is determined by enumerating the applicable `operator co_await` functions for an argument `a` (13.3.1.2), and choosing the best one through overload resolution (13.3). 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`.

§ 5.3.8
— $h$ is an object of type `std::experimental::coroutine_handle<P>` referring to the enclosing coroutine.

— `await-ready` is the expression $e\text{.await_ready}()$, contextually converted to `bool`.

— `await-suspend` is the expression $e\text{.await_suspend}(h)$, which shall be a prvalue of type `void` or `bool`.

— `await-resume` is the expression $e\text{.await_resume}()$.

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

The `await-expression` evaluates the `await-ready` expression, then:

— If the result is `false`, the coroutine is considered suspended. Then, the `await-suspend` expression is evaluated. 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 (15.1). Otherwise, control flow returns to the current caller or resumer (8.4.4) without exiting any scopes (6.6).

— 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`.

[Example:

```c++

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]

§ 5.3.8
5.17 Assignment and compound assignment operators

Add \textit{yield-expression} to the grammar production \textit{assignment-expression}.

\begin{verbatim}
assignment-expression:
  conditional-expression
  logical-or-expression assignment-operator initializer-clause
  throw-expression
  yield-expression
\end{verbatim}

5.19 Constant expressions

Add bullets prohibiting \textit{await-expression} and \textit{yield-expression} to paragraph 2.

\begin{itemize}
  \item an \textit{await-expression} (5.3.8);
  \item a \textit{yield-expression} (5.20);
\end{itemize}

5.20 Yield

Add a new section to Clause 5.

\begin{verbatim}
yield-expression:
  co_yield assignment-expression
  co_yield braced-init-list
\end{verbatim}

A \textit{yield-expression} shall appear only within a suspension context of a function (5.3.8). Let \( e \) be the operand of the \textit{yield-expression} and \( p \) be an lvalue naming the promise object of the enclosing coroutine (8.4.4), then the \textit{yield-expression} is equivalent to the expression \texttt{co\_await \( p \).yield\_value(\( e \)).}

\textit{Example:}

\begin{verbatim}
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
\end{verbatim}

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

— end example]
6 Statements

6.5 Iteration statements

Add the underlined text to paragraph 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

Add the underlined text to paragraph 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 parentheses¹

( 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-exp, __begin = __begin; __end = end-exp; __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-exp and end-exp are determined as follows: ...

Add the following paragraph after paragraph 2.

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-declarators in the declaration of __range.
6.6 Jump statements

In paragraph 1 add two productions to the grammar:

```
jump-statement:
  break ;
  continue ;
  return expressionopt;
  return braced-init-list ;
  coroutine-return-statement
  goto identifier ;
```

Add the underlined text to paragraph 2:

1. On exit from a scope (however accomplished), objects with automatic storage duration (3.7.3) that have been constructed in that scope are destroyed in the reverse order of their construction.  

   [Note: A suspension of a coroutine (5.3.8) is not considered to be an exit from a scope.  
   —end note] ...

6.6.3 The return statement

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

Add this section to 6.6.

```
coroutine-return-statement:
  co_return expressionopt;
  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). A coroutine shall not return to its caller or resumer by a return statement (6.6.3).

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; }
   ```

   where final_suspend 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. 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.
7 Declarations

7.1.5 The constexpr specifier

Insert a new bullet after paragraph 3 bullet 1.

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

Add the following paragraph.

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

8.4 Function definitions

8.4.4 Coroutines

Add this section to 8.4.

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 (5.20), 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.

[Example:

```c++
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]

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 \( P_2 \ldots P_n \) be the types of the function parameters; otherwise let \( P_1 \ldots P_n \) be the types of the function parameters. Let \( p_1 \ldots p_n \) be lvalues denoting those objects. Let \( R \) be the return type and \( F \) be the function-body of \( f \). \( T \) be the type \( \text{std::experimental::coroutine_traits}<R,P_1,\ldots,P_n> \), and \( P \) be the class type denoted by \( T::\text{promise\_type} \). Then, the coroutine behaves as if its body were:

```c++
{
    P p;
    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.

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
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.

A suspended coroutine can be resumed to continue execution by invoking a resumption member function (18.11.2.5) 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.

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 (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 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 values `p1 ... pn` are the succeeding arguments. If no matching function is found, 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`.

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 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`.

The coroutine state is destroyed when control flows off the end of the coroutine or the `destroy` member function (18.11.2.5) 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 (3.7.4.2). If `destroy` is called for a coroutine that is not suspended, the program has undefined behavior.

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.

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. 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 coroutine promise object lifetime 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]

[Example:

```cpp
// ::operator new(size_t, nothrow_t) will be used if allocation is needed
struct generator {
    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{}; }
        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() { 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]

[Example:

```cpp
// 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
            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

12.1 Constructors

A constructor shall not be a coroutine.

12.4 Destructors

A destructor shall not be a coroutine.

12.8 Copying and moving class objects

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

13.5 Overloaded operators

Add `co_await` to the list of operators in paragraph 1 before operators `()` and `[]`. Add the following paragraph after paragraph 5.

The `co_await` operator is described completely in 5.3.8. The attributes and restrictions found in the rest of this subclause do not apply to it unless explicitly stated in 5.3.8.
17 Library introduction

17.6.1.3 Freestanding implementations

Add a row to Table 16 for coroutine support header `<experimental/coroutine>`.

Table 16 — C++ headers for freestanding implementations

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>18.2 Types</td>
<td><code>&lt;ciso646&gt;</code></td>
</tr>
<tr>
<td>18.3 Implementation properties</td>
<td><code>&lt;cfloat&gt; &lt;limits&gt; &lt;climits&gt;</code></td>
</tr>
<tr>
<td>18.4 Integer types</td>
<td><code>&lt;cstdint&gt;</code></td>
</tr>
<tr>
<td>18.5 Start and termination</td>
<td><code>&lt;cstdlib&gt;</code></td>
</tr>
<tr>
<td>18.6 Dynamic memory management</td>
<td><code>&lt;new&gt;</code></td>
</tr>
<tr>
<td>18.7 Type identification</td>
<td><code>&lt;typeinfo&gt;</code></td>
</tr>
<tr>
<td>18.8 Exception handling</td>
<td><code>&lt;exception&gt;</code></td>
</tr>
<tr>
<td>18.9 Initializer lists</td>
<td><code>&lt;initializer_list&gt;</code></td>
</tr>
<tr>
<td>18.10 Other runtime support</td>
<td><code>&lt;cstlalign&gt; &lt;cstldarg&gt; &lt;cstdbool&gt;</code></td>
</tr>
<tr>
<td>18.11 Coroutines support</td>
<td><code>&lt;experimental/coroutine&gt;</code></td>
</tr>
<tr>
<td>20.10 Type traits</td>
<td><code>&lt;type_traits&gt;</code></td>
</tr>
<tr>
<td>Atomics</td>
<td><code>&lt;atomic&gt;</code></td>
</tr>
</tbody>
</table>
18 Language support library
[language.support]

18.1 General

Add a row to Table 30 for coroutine support header `<experimental/coroutine>`.

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

18.10 Other runtime support

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 suspension context of a coroutine (5.3.8).

See also: ISO C 7.10.4, 7.8, 7.6, 7.12.

18.11 Coroutines support library

Add this section to clause 18.

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>
  struct coroutine_traits;

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

} // namespace coroutines_v1
} // namespace experimental

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

} // namespace std

18.11.1 Coroutine traits [coroutine.traits]

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

Users may specialize coroutine_traits to customize the semantics of coroutines.

18.11.1.1 Struct template coroutine_traits [coroutine.traits.primary]

The header <experimental/coroutine> defines the primary template coroutine_traits such that if ArgTypes is a parameter pack of types and if R is a type that has a valid (14.8.2) member type promise_type, 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.

18.11.2 Struct template coroutine_handle [coroutine.handle]

namespace std {
namespace experimental {
inline namespace coroutines_v1 {

§ 18.11.2 20
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
  constexpr void* address() const noexcept;
  constexpr static coroutine_handle from_address(void* addr);

  // 18.11.2.4 observers
  constexpr explicit operator bool() const noexcept;
  bool done() const;

  // 18.11.2.5 resumption
  void operator()();
  void resume();
  void destroy();

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

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

  // 18.11.2.3 import
  constexpr static coroutine_handle from_address(void* addr);

  // 18.11.2.6 promise access
  Promise& promise() const;
};

} // namespace coroutines_v1
} // namespace std

Let \(P\) be the promise type of a coroutine (8.4.4). An object of type \(\text{coroutine
d_handle} < P>\) is called a \textit{coroutine handle} and can be used to refer to a suspended or executing coroutine. A default constructed \textit{coroutine_handle} object does not refer to any coroutine.

\begin{itemize}
\item[18.11.2.1] \textbf{coroutine_handle} construct/reset \hspace{1em} [coroutine.handle.con]
\end{itemize}

\begin{verbatim}
constexpr coroutine_handle() noexcept;
constexpr coroutine_handle(nullptr_t) noexcept;

Postconditions: address() == nullptr.

static coroutine_handle from_promise(Promise& p);

Requires: \(p\) is a reference to a promise object of a coroutine.
\end{verbatim}
Returns: a coroutine handle \( h \) referring to the coroutine.

Postconditions: \( \text{addressof}(h.\text{promise}()) == \text{addressof}(p) \).

coroutine_handle\& operator=(nullptr\_t) noexcept;
Postconditions: address() == nullptr.

Returns: \*this.

18.11.2.2 coroutine_handle export/import  

constexpr void* address() const noexcept;

Returns: ptr.

static coroutine_handle coroutine_handle\<>::from_address(void* addr);
Requires: addr was obtained via a prior call to address().

Postconditions: from_address(address()) == \*this.

18.11.2.3 coroutine_handle import  

static coroutine_handle from_address(void* addr);
Requires: addr was obtained via a prior call to address().

Postconditions: from_address(address()) == \*this.

18.11.2.4 coroutine_handle observers  

constexpr explicit operator bool() const noexcept;

Returns: true if address() != nullptr, otherwise false.

bool done() const;
Requires: \*this refers to a suspended coroutine.

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

18.11.2.5 coroutine_handle resumption  

void operator()();
void resume();

Requires: \*this refers to a suspended coroutine.

Effects: resumes the execution of the coroutine. If the coroutine was suspended at its final suspend point, behavior is undefined.

Synchronization: a concurrent resumption of a coroutine by multiple threads may result in a data race.

void destroy();
Requires: \*this refers to a suspended coroutine.

Effects: destroys the coroutine (8.4.4).

Synchronization: a concurrent resumption of a coroutine by multiple threads may result in a data race.

18.11.2.6 coroutine_handle promise access  

Promise& promise() const;

Requires: \*this refers to a coroutine.

Returns: a reference to the promise of the coroutine.
18.11.2.7 Comparison operators [coroutine.handle.compare]

bool operator==(coroutine_handle<> x, coroutine_handle<> y) noexcept;
Returns: x.address() == y.address().

bool operator<(coroutine_handle<> x, coroutine_handle<> y) noexcept;
Returns: less<void*>()(x.address(), y.address()).

bool operator!=(coroutine_handle<> x, coroutine_handle<> y) noexcept;
Returns: !(x == y).

bool operator>(coroutine_handle<> x, coroutine_handle<> y) noexcept;
Returns: y < x).

bool operator<=(coroutine_handle<> x, coroutine_handle<> y) noexcept;
Returns: !(x > y).

bool operator>=(coroutine_handle<> x, coroutine_handle<> y) noexcept;
Returns: !(x < y).

18.11.2.8 Hash support [coroutine.handle.hash]

template <class P> struct hash<experimental::coroutine_handle<P>>;
The template specialization shall meet the requirements of class template hash (20.9.12).

18.11.3 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 noexcep {}
constexpr void await_resume() const noexcept {} }
};
struct suspend_always {
constexpr bool await_ready() const noexcept { return false; }
constexpr void await_suspend(coroutine_handle<>)) const noexcep {}
constexpr void await_resume() const noexcept {} }
};
// namespace coroutines_v1
} // namespace experimental
} // namespace std

§ 18.11.3