Document Number: N4403
Date: 2015-04-10
Revises: None
Reply to: Gor Nishanov
Microsoft
gorn@microsoft.com

Draft wording for Resumable Functions

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 (1.2) that enables definition of resumable functions. 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 Normative references [intro.refs]

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

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

ISO/IEC 14882:2014 is hereafter called the *C++ Standard*. The numbering of Clauses, sections, and paragraphs in this document reflects the numbering in the C++ Standard. References to Clauses and sections not appearing in this Technical Specification refer to the original, unmodified text in the C++ Standard.

1.3 Implementation compliance [intro.compliance]

1 Conformance requirements for this specification are the same as those defined in 1.3 in the C++ Standard. [*Note:* Conformance is defined in terms of the behavior of programs. — *end note*]

1.4 Feature-testing recommendations (Informative) [intro.features]

1 For the sake of improved portability between partial implementations of various C++ standards, WG21 (the ISO Technical Committee for the C++ Programming Language) recommends that implementers and programmers follow the guidelines in this section concerning feature-test macros. [*Note:* WG21's SD-6 makes similar recommendations for the C++ Standard. — *end note*]

2 Implementers who provide a new standard feature should define a macro with the recommended name, in the same circumstances under which the feature is available (for example, taking into account relevant command-line options), to indicate the presence of support for that feature. Implementers should define that macro with the value specified in the most recent version of this technical specification that they have implemented. The recommended macro name is `__cpp_experimental_` followed by the string in the "Macro name suffix" column in Table 1.

3 No header files should be required to test macros describing the presence of support for language features.

Table 1 — Feature-test macros for resumable functions

Macro name suffix	Value
`resumable`	201599

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 resumable function (8.4.4), or receipt of a signal).

2 Lexical conventions [lex]

2.12 Keywords [lex.key]

In section 2.12, add the keywords `await` and `yield` to Table 4 "Keywords".

3 Basic concepts [basic]

3.6 Start and termination [basic.start]

3.6.1 Main function [basic.start.main]

Add paragraph 5.

5 The function `main` shall not be resumable.

5 Expressions [expr]

5.3 Unary expressions [expr.unary]

In this section add the `await` *cast-expression* to the rule for *unary-expression*.

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

Add subsection 5.3.9.

5.3.9 Await [expr.await]

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

2 The presence of an `await` operator in a potentially-evaluated expression makes the enclosing function a resumable function.

3 The `await` operator shall not appear in a potentially-evaluated expression in a catch clause of a try block.

4 An `await` expression of the form

`await` *cast-expression*

is equivalent to [1]

```
{
  auto && __expr = cast-expression;
  if ( !await-ready-expr ) {
    await-suspend-expr;
    suspend-resume-point
  }
  return await-resume-expr;
}
```

if the type of *await-suspend-expr* is `void`, otherwise it is equivalent to

```
{
  auto && __expr = cast-expression;
  if ( !await-ready-expr && await-suspend-expr ) {
    suspend-resume-point
  }
```

1) if it were possible to write an expression in terms of a block, where return from the block becomes the result of the expression

```
  return await-resume-expr;
}
```

where `__expr` is a variable defined for exposition only, and `_ExprT` is the type of the *cast-expression*, and `_ResumableHandle` is an object of the `resumable_handle` type specialized for an enclosing function, and *await-ready-expr*, *await-suspend-expr*, and *await-expr* are determined as follows:

(4.1) — if `_ExprT` is a class type, the *unqualified-id*s `await_ready`, `await_suspend` and `await_resume` are looked up in the scope of class `_ExprT` as if by class member access lookup (3.4.5), and if it finds at least one declaration, `await_ready`, `await_suspend`, and `await_resume` are `__expr.await_ready()`, `__expr.await_suspend(_ResumableHandle)` and `__expr.await_resume()`, respectively;

(4.2) — otherwise, *await_ready*, *await_suspend* and *await_resume* are `await_ready(__expr)`, `await_resume(__expr, _ResumableHandle)`, and `await_resume(__expr)` respectively, where `await_ready`, `await_suspend`, and `await_resume` are looked up in the associated namespaces (3.4.2). [*Note:* Ordinary unqualified lookup (3.4.1) is not performed. — *end note*]

5 An `await` expression may appear as an unevaluated operand (5.2.8, 5.3.3, 5.3.7, 7.1.6.2). The presence of such an `await` expression does not make the enclosing function resumable and can be used to examine the type of an `await` expression.

[*Example:*

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

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

— *end example*]

6 An `await` expression may only appear in a resumable function with an eventual return type, i.e a resumable function shall have the `set_result` member function defined in its *promise type* (8.4.4).

6 Statements [stmt.stmt]

6.5 Iteration statements [stmt.iter]

Add underlined text to paragraph 1.

1 Iteration statements specify looping.

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

Add subclause 6.5.5.

6.5.5 The await-for statement [stmt.for.await]

1 A `for await` statement of the form

`for await` `(` *for-range-declaration* `:` *expression* `)` *statement*

is equivalent to

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

where `__range`, `__begin`, *range-init*, *begin-expr*, and *end-expr* are defined as in the range-based for statement 6.5.4.

2 A `for await` statement may only appear in a resumable function with an eventual return type, i.e a resumable function shall have the `set_result` member function defined in its *promise type* (8.4.4).

6.6 Jump statements [stmt.jump]

In paragraph 1 add productions for `yield` statement.

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

6.6.3 The `return` statement [stmt.return]

Add a note:

> [*Note:* In this section a function refers to non-resumable functions only. The return statement in resumable functions is described in 6.6.4 — *end note*]

Add sections 6.6.4 and 6.6.5.

6.6.4 The `return` statement in resumable functions [stmt.return.resumable]

1 A resumable function returns to its caller by the `return` statement or when suspended at a *suspend-resume point*.

2 In this clause, `_Pr` refers to the *promise object* of the enclosing resumable function.

3 A `return` statement with neither an *expression* nor a *braced-init-list* can be used only in resumable functions that do not produce an eventual value or have an eventual return type of `void`. In the latter case, completion of the resumable function is reported by calling `_Pr.set_result()`. A `return` statement with an expression of non-`void` type can be used only in resumable functions producing an eventual value; the value of the expression is supplied to a promise of the resumable function by calling `_Pr.set_result(`*`expression`*`)` or `_Pr.set_result(`*`braced-init-list`*`)`.

[*Example:*

```
std::future<std::pair<std::string,int>> f(const char* p, int x) {
  await g();
  return {p,x};
}
```

— *end example*]

Flowing off the end of a function is equivalent to a `return` with no value; the program is not well-formed if this happens in an eventual-value-returning resumable function.

4 A `return` statement with an expression of type `void` can be used only in functions without an eventual return type or with an eventual return type of `void`; the expression is evaluated just before the function returns to its caller.

5 Prior to returning to the caller, a resumable function evaluates `_Pr.final_suspend()` predicate. If `_Pr.final_suspend()` contextually converted to `bool` evaluates to `true`, the resumable function suspends at *final suspend point* (8.4.4), otherwise, resumable function flows off the end of the function-body and destroys the *resumable function state* and frees any extra memory dynamically allocated to store the state.

6.6.5 Yield statement [stmt.yield]

1 A yield statement of the form

```
yield V;
```

where `V` is either an *expression* or a *braced-init-list* is equivalent to:

```
_Pr.yield_value(V);
suspend-resume-point
```

If a `_Pr.yield_value(V)` expression is of type `void`, otherwise it is equivalent to:

```
if (_Pr.yield_value(V)) {
  suspend-resume-point
}
```

Where `_Pr` is a *promise object* (8.4.4) of the enclosing resumable function.

2 A `yield` statement may only appear if `yield_value` member function is defined in the promise type of the enclosing resumable function.

3 A promise object may have more than one overload of `yield_value`.

[*Example:*

```
recursive_generator<int> flatten(node* n)
{
  if (n == nullptr)
    return;

  yield flatten(n->left);
  yield n->value;
  yield flatten(n->right);
}
```

The promise for `flatten` function should contain overloads that can accept a value of type `int` and a value of type `recursive_generator<int>`. In the former case, yielding a value is unconditional. In the latter case, the nested generator may produce an empty sequence of values and thus suspension at the yield point shall not happen and corresponding `yield_value` contextually converted to `bool` must evaluate to `false`. — *end example*]

7 Declarations [dcl.dcl]

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

Add underlined text to paragraph two.

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

add paragraphs 16 through 18.

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

17 If a resumable function with a declared return type that contains a placeholder type, then the return type of the resumable function is deduced as follows:

(17.1) — If a `yield` statement and either an `await` expression or a `for await` statement are present, then the return type is `std::experimental::async_stream<T>`, where `T` is deduced from the `yield` statements.

(17.2) — Otherwise, if an `await` expression or a `for await` statement are present in a function, then the return type is `std::experimental::task<T>` where type `T` is deduced from `return` statements.

(17.3) — Otherwise, if a `yield` statement is present in a function, then the return type is `std::experimental::generator<T>`, where `T` is deduced from the `yield` statements.

[*Example:*

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

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

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

— *end example*]

18 The templates `std::experimental::generator`, `std::experimental::task`, and `std::experimental::async_stream` are not predefined; if the appropriate headers are not included prior to a use — even an implicit use in which the type is not named (7.1.5.4) — the program is ill-formed.

[Editor's note: Class templates `std::experimental::generator`, `std::experimental::task`, and `std::experimental::async_stream` are not specified in this proposal and will be developed as more experience with resumable functions is accumulated.]

8 Declarators [dcl.decl]

8.3.5 Functions [dcl.fct]

Add paragraph 16.

16 A function can not be resumable (8.4.4) if the *parameter-declaration-clause* terminates with an ellipsis.

8.4 Function definitions [dcl.fct.def]

Add subsection 8.4.4

8.4.4 Resumable Functions [dcl.fct.def.resumable]

1 A function is *resumable* if it contains one or more *suspend-resume point*s.

2 *Suspend-resume points* are created by `await` operator (5.3.9) in potentially-evaluated expression, `yield` statement (6.6.5) or `for await` statement (6.5.5).

3 Resumable functions need a set of related types and functions to complete the definition of their semantics. These types and functions are provided as a set of member types or typedefs and functions in the instantiation of struct template `resumable_traits` (18.11.1).

4 For a resumable function `f`, Let `R` be a return type and P_1, P_2, ..., P_n be types of parameters. If `f` is a non-static member function then P_1 denotes the type of the implicit `this` parameter. Resumable traits for function `f` is an instantiation of struct template `std::experimental::resumable_traits<R,P1,...,PN>`. Let `F` be a *function-body* [2] of that function. Then, the resumable function should behave as if its body were:

```
{
   using _Tr = std::experimental::resumable_traits<R,P1,...,PN>;
   _Tr::promise_type _Pr;
   if (_Pr.initial_suspend()) {
     suspend-resume-point // initial suspend point
   }
   try { function-body }
   catch(...) {
     stop-or-propagate;
   }
   if (_Pr.final_suspend()) {
     suspend-resume-point // final suspend point
   }
}
```

where type alias `_Tr` and local variable `_Pr` are defined for exposition only and *stop-or-propate* is `throw` if promise_type does not have `set_exception` member function defined, and `_Pr.set_exception(std::current_exception())` otherwise. An object denoted as `_Pr` is a *promise object* of a resumable function and its type is a *promise type* of the resumable function.

5 Execution of a resumable function is suspended when it reaches a suspend-resume point. A suspended resumable function can be resumed to continue execution by invoking resumption member functions (18.11.2.4) of

2) Due to requirement of having suspend-resume points, *function-body* is either a *compound-statement* or *function-try-block*.

an object of `resumable_handle<P>` type associated with this instance of a resumable function, where type P is a *promise type* of the function.

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

7 A *resumable function state* consist of storage for objects with automatic storage duration that are live at the current point of execution or suspension of a resumable function. *Resumable function state* is destroyed when the control flows off the end of the function or `destroy` member function (18.11.2.4) of `resumable_handle` object associated with that function is invoked.

8 *Suspension* of a resumable function returns control to the current caller of the resumable function. For the first suspend, the return value is obtained by invoking member function `get_return_object` (18.11.3) of the *promise object* of the resumable function. For the subsequent suspends, if any, the resumable function is invoked via resumption members functions of `resumable_handle` (18.11.2) and no return value is expected.

9 An invocation of a resumable function may incur a move operation for the parameters that may be accessed in the *function-body* of resumable function after a resume. References to those parameters in the *function-body* of the resumable function are replaced with references to their copies .

10 If a parameter copy/move is required, class object moves are performed according to the rules described in Copying and moving class objects (12.8).

11 If a parameter move, a call to `get_return_object` or a promise object construction throws an exception, objects with automatic storage duration (3.7.3) that have been constructed are destroyed in the reverse order of their construction, any memory dynamically allocated for *resumable function state* is freed and the search for a handler starts in the scope of the calling function.

12 Special member functions [special]

Add paragraph 6.

6 Special member functions shall not be resumable.

18 Language support library [language.support]

18.1 General [support.general]

Add a row to Table 2 for `<experimental/resumable>`

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	Resumable functions support	`<experimental/resumable>`

18.10	Other runtime support	`<csignal>`
		`<csetjmp>`
		`<cstdalign>`
		`<cstdarg>`
		`<cstdbool>`
		`<cstdlib>`
		`<ctime>`

Add section 18.11

18.11 Resumable functions support library [support.resumable]

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

Header `<experimental/resumable>` synopsis

```
namespace std {
namespace experimental {
  // 18.11.1 resumable traits
  template <typename R, typename... ArgTypes>
    class resumable_traits;

  // 18.11.2 resumable handle
  template <typename Promise = void>
    class resumable_handle;

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

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

18.11.1 resumable traits [resumable.traits]

1 This subclause defines requirements on classes representing *resumable traits*, and defines a primary struct template `resumable_traits<R,Args...>` that satisfies those requirements.

2 The `resumable_traits` may be specialized by the user to customize semantics of resumable functions.

18.11.1.1 Resumable traits requirements [resumable.traits.requirements]

1 In Table 3, X denotes a trait class instantiated as described in 8.4.4; a_1, a_2, ... a_n denote parameters passed to a resumable function. If it is a member function, than a_1 denotes implicit `this` parameter.

Table 3 — `resumable_traits` requirements [tab:resumable.traits.requirements]

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

18.11.1.2 Struct template `resumable_traits` [resumable.traits.primary]

1 The header `<resumable>` shall define primary struct template `resumable_traits` as follows.

2 The requirements for the members are given in clause 18.11.1.1.

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

18.11.2 Struct template `resumable_handle` [resumable.handle]

```
namespace std {
  namespace experimental {
    template <>
```

```
    struct resumable_handle<void>
    {
      // 18.11.2.1 construct/reset
      resumable_handle() noexcept;
      resumable_handle(std::nullptr_t) noexcept;
      resumable_handle& operator=(nullptr_t) noexcept;

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

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

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

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

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

      // 18.11.2.6 export/import
      static resumable_handle from_promise(Promise*) noexcept;
      Promise& promise() noexcept;
      Promise const& promise() const noexcept;
    };
  }
}
```

1 The struct template `resumable_handle` can be used to refer to a suspended or executing resumable function. Such function is called a *target* of `resumable_handle`. A default constructed `resumable_handle` object has no target.

18.11.2.1 `resumable_handle` construct/reset [resumable.handle.con]

```
resumable_handle() noexcept;
resumable_handle(std::nullptr_t) noexcept;
```

1 *Postconditions:* `!*this`.

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

2 *Postconditions:* `!*this`.

3 *Returns:* `*this`.

18.11.2.2 `resumable_handle` export/import [resumable.handle.export]

```
static resumable_handle from_address(void* addr) noexcept;
```

```
void* to_address() const noexcept;
```

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

18.11.2.3 `resumable_handle` capacity [resumable.handle.capacity]

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

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

18.11.2.4 `resumable_handle` resumption [resumable.handle.resumption]

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

1 *Requires:* *this refers to a suspended resumable function

2 *Effects:* resumes execution of a target function. If function was suspended at final suspend point std::terminate is called.

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

3 *Requires:* *this refers to a suspended resumable function

4 *Effects:* objects with automatic storage duration that are in scope at the suspend point are destroyed in the reverse order of the construction. If resumable function required dynamic allocation for the objects with automatic storage duration, the memory is freed.

18.11.2.5 `resumable_handle` completion check [resumable.handle.completion]

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

1 *Requires:* *this refers to a suspended resumable function

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

18.11.2.6 `resumable_handle` to/from promise [resumable.handle.prom]

```
static resumable_handle from_promise(Promise* p) noexcept;
```

1 *Requires:* `p` points to a promise object of a resumable function.

2 *Returns:* `resumable_handle` referring to that resumable function.

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

3 *Requires:* *this refers to a resumable function

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

18.11.3 Resumable promise requirements [resumable.promise]

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

Table 4 — Descriptive variable definitions

Variable	Definition
`P`	a resumable promise type
`p`	a value of type `P`

Table 4 — Descriptive variable definitions (continued)

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

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

Expression	Note
`P{}`	Construct an object of type `P`
`p.get_return_object()`	`get_return_object` is invoked by the resumable function to construct the return object prior to reaching the first suspend-resume point, a `return` statement or flowing off the end of the function.
`p.set_result(v)`	If present, an enclosing resumable function supports an eventual value of a type that `v` can be converted to. `set_result` is invoked by a resumable function when a return statement with an *expression* or a *braced-init-list* is encountered in a resumable function. If a promise type does not satisfy this requirement, the presence of a return statement with an *expression* or a *braced-init-list* statement in the body results in a compile time error.
`p.set_result()`	If present, an enclosing resumable function supports an eventual value of type `void`. `set_result` is invoked by a resumable function when a return statement without an *expression* nor a *braced-init-list* is encountered in a resumable function or control flows to the end of the function. A promise type must contain at most one declaration of `set_result`.
`p.set_exception(e)`	`set_exception` is invoked by a resumable function when an unhandled exception occurs within a *function-body* of the resumable function. If promise does not provide `set_exception`, an unhandled exception will propagate from a resumable functions normally.
`p.yield_value(v)`	Must be present for the enclosing resumable function to support `yield` statement.
`p.initial_suspend()`	if `p.initial_suspend()` evaluates to `true`, resumable function will suspend at *initial suspend point* (8.4.4).
`p.final_suspend()`	if `p.final_suspend()` evaluates to `true`, resumable function will suspend at *final suspend point* (8.4.4).

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

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

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

```
  };

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

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

  generator() = default;
  ~generator() { if (coro) { coro.destroy(); } }
private:
  generator(promise_type* myPromise):
     coro(std::experimental::resumable_handle<promise_type>::from_promise(myPromise)) {
  }
  std::experimental::resumable_handle<promise_type> coro;
};

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

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

— *end example*]

30 Thread support library [thread]

30.3 Threads [thread.threads]

30.3.2 Namespace this_thread [thread.thread.this]

Rename `yield` function to `yield_execution`.

```
namespace std {
  namespace this_thread {
    thread::id get_id() noexcept;

    void yield_execution() noexcept;
    template <class Clock, class Duration>
      void sleep_until(const chrono::time_point<Clock, Duration>& abs_time);
    template <class Rep, class Period>
      void sleep_for(const chrono::duration<Rep, Period>& rel_time);
  }
}
```

`thread::id this_thread::get_id() noexcept;`

1 *Returns:* An object of type `thread::id` that uniquely identifies the current thread of execution. No other thread of execution shall have this id and this thread of execution shall always have this id. The object returned shall not compare equal to a default constructed `thread::id`.

`void this_thread::yield_execution() noexcept;`

2 *Effects:* Offers the implementation the opportunity to reschedule.

3 *Synchronization:* None.