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

## Contents

List of Tables

1. General
   1.1 Scope .................................................. 1
   1.2 Normative references ................................ 1
   1.3 Implementation compliance .......................... 1
   1.4 Feature-testing recommendations (Informative) .... 1
   1.9 Program execution .................................... 1

2. Lexical conventions ..................................... 3
   2.12 Keywords ............................................. 3

3. Basic concepts .......................................... 4
   3.6 Start and termination ................................. 4

5. Expressions .............................................. 5
   5.3 Unary expressions ...................................... 5

6. Statements ............................................... 7
   6.5 Iteration statements .................................. 7
   6.6 Jump statements ...................................... 7

7. Declarations ............................................. 10

8. Declarators ................................................ 11
   8.4 Function definitions .................................. 11

12. Special member functions ............................... 13

18. Language support library ............................... 14
   18.1 General ................................................ 14
   18.11 Resumable functions support library ............... 14

30. Thread support library .................................. 20
   30.3 Threads ................................................. 20
List of Tables

1  Feature-test macros for resumable functions ........................................... 1
2  Language support library summary .......................................................... 14
3  \texttt{resumable\_traits} requirements ....................................................... 15
4  Descriptive variable definitions ............................................................... 17
5  \texttt{ResumablePromise} requirements ..................................................... 18
1 General

1.1 Scope

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.

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

The following referenced document is indispensable for the application of this document. For dated refer-
ences, 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 para-
graphs 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

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)

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]

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
cmd-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.

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

<table>
<thead>
<tr>
<th>Macro name suffix</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>resumable</td>
<td>201599</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 resumable function (8.4.4), or receipt of a signal).
2 Lexical conventions

2.12 Keywords

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

3.6 Start and termination

3.6.1 Main function

Add paragraph 5.

5 The function main shall not be resumable.
5 Expressions

5.3 Unary expressions

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

\[
\text{unary-expression:}
\]

- 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

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

\[
\{
  \text{auto } \&\& \_\text{expr} = \text{cast-expression};
  \text{if ( !await-ready-expr ) }\{
    \text{await-suspend-expr;}
    \text{suspend-resume-point}
  \}\n  \text{return await-resume-expr;}
\}
\]

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

\[
\{
  \text{auto } \&\& \_\text{expr} = \text{cast-expression};
  \text{if ( !await-ready-expr } \&\& \text{await-suspend-expr } )\{
    \text{suspend-resume-point}
  \}
\}
\]

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-ids 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:

```cpp
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

6.5 Iteration statements

Add underlined text to paragraph 1.

Iteration statements specify looping.

iteration-statement:

while ( condition ) statement
do statement while ( expression )
for ( for-init-statement conditionopt; expressionopt ) 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

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.

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

In paragraph 1 add productions for yield statement.

jump-statement:

break ;
continue ;
return expressionopt;
return braced-init-list ;
yield expression ;
yield braced-init-list ;
goto identifier ;
6.6.3 The return statement

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

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:

```cpp
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

1 A yield statement of the form

```cpp
yield V;
```

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

```cpp
_Pr.yield_value(V);
```

suspend-resume-point

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

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

§ 6.6.5
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.

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.

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 points.

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 P₁, P₂, ..., Pₙ be types of parameters. If f is a non-static member function then P₁ 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 of that function. Then, the resumable function should behave as if its body were:

```cpp
{
    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 \texttt{resumable\_handle<P>} type associated with this instance of a resumable function, where type P is a \textit{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 \textit{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. \textit{Resumable function state} is destroyed when the control flows off the end of the function or \texttt{destroy} member function (18.11.2.4) of \texttt{resumable\_handle} object associated with that function is invoked.

8 \textit{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 \texttt{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 \texttt{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 \texttt{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 \textit{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.

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>
<thead>
<tr>
<th>Subclause</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>18.2 Types</td>
<td>&lt;cstdio&gt;</td>
</tr>
<tr>
<td>18.3 Implementation properties</td>
<td>&lt;climits&gt;</td>
</tr>
<tr>
<td>18.4 Integer types</td>
<td>&lt;cstdint&gt;</td>
</tr>
<tr>
<td>18.5 Start and termination</td>
<td>&lt;cstdlib&gt;</td>
</tr>
<tr>
<td>18.6 Dynamic memory management</td>
<td>&lt;cstdlib&gt;</td>
</tr>
<tr>
<td>18.7 Type identification</td>
<td>&lt;cstdlib&gt;</td>
</tr>
<tr>
<td>18.8 Exception handling</td>
<td>&lt;cstdlib&gt;</td>
</tr>
<tr>
<td>18.9 Initializer lists</td>
<td>&lt;cstdlib&gt;</td>
</tr>
<tr>
<td>18.11 Resumable functions support</td>
<td>&lt;experimental/resumable&gt;</td>
</tr>
<tr>
<td>18.10 Other runtime support</td>
<td>&lt;cstdlib&gt;</td>
</tr>
</tbody>
</table>

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;

```

§ 18.11
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

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

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

18.11.1.1 Resumable traits requirements

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>
<thead>
<tr>
<th>Expression</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>X::promise_type</td>
<td>X::promise_type must be a type satisfying resumable promise requirements (18.11.3)</td>
</tr>
</tbody>
</table>
| 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_allocation_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

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.

```cpp
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

```cpp
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;

§ 18.11.2.2
void* to_address() const noexcept;

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

18.11.2.3 resumable_handle capacity

explicit operator bool() const noexcept;

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

18.11.2.4 resumable_handle resumption

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

Requires: *this refers to a suspended resumable function
Effects: resumes execution of a target function. If function was suspended at final suspend point std::terminate is called.

void destroy() const;

Requires: *this refers to a suspended resumable function
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

bool done() const noexcept;

Requires: *this refers to a suspended resumable function
Returns: true if target function is suspended at final suspend point, otherwise false.

18.11.2.6 resumable_handle to/from promise

static resumable_handle from_promise(Promise* p) noexcept;

Requires: p points to a promise object of a resumable function.
Returns: resumable_handle referring to that resumable function.

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

Requires: *this refers to a resumable function
Returns: a reference to a promise of the target function.

18.11.3 Resumable promise requirements

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

<table>
<thead>
<tr>
<th>Variable</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>a resumable promise type</td>
</tr>
<tr>
<td>p</td>
<td>a value of type P</td>
</tr>
</tbody>
</table>
Table 4 — Descriptive variable definitions (continued)

<table>
<thead>
<tr>
<th>Variable</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>e</td>
<td>a value of std::exception_ptr type</td>
</tr>
<tr>
<td>h</td>
<td>a value of std::experimental::resumable_handle&lt;P&gt; type</td>
</tr>
<tr>
<td>v</td>
<td>an expression or braced-init-list</td>
</tr>
</tbody>
</table>

Table 5 — ResumablePromise requirements [ResumablePromise]

<table>
<thead>
<tr>
<th>Expression</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>P{}</td>
<td>Construct an object of type P</td>
</tr>
<tr>
<td>p.get_return_object()</td>
<td>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.</td>
</tr>
<tr>
<td>p.set_result(v)</td>
<td>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.</td>
</tr>
<tr>
<td>p.set_result()</td>
<td>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.</td>
</tr>
<tr>
<td>p.set_exception(e)</td>
<td>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.</td>
</tr>
<tr>
<td>p.yield_value(v)</td>
<td>Must be present for the enclosing resumable function to support yield statement.</td>
</tr>
<tr>
<td>p.initial_suspend()</td>
<td>if p.initial_suspend() evaluates to true, resumable function will suspend at initial suspend point (8.4.4).</td>
</tr>
<tr>
<td>p.final_suspend()</td>
<td>if p.final_suspend() evaluates to true, resumable function will suspend at final suspend point (8.4.4).</td>
</tr>
</tbody>
</table>

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

```cpp
#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; } 
    }

    promise_type promise;  
    generator() : promise() {}  
    ~generator() {
        promise.set_exception(new std::exception);  
    }  
    generator(int value) : promise() {
        promise.set_result(value);  
    }  
    generator(const generator&) = delete;  
    generator& operator=(const generator&) = delete;  
    int request(int n) {  
        return promise.set_exception(new std::exception{__FILE__, __LINE__});  
    }  
    int promise(int n) {  
        return promise.get_return_object().current_value;  
    }  
    static string description() {  
        return "generator";  
    }  
    static void validate(generation& gen) {  
        return;  
    }  
};
```

§ 18.11.3
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

30.3 Threads

30.3.2 Namespace this_thread

Rename yield function to yield_execution.

```cpp
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.