Document number: P0099R1
Supersedes: P0099R0
Date: 2016-10-16
Reply-to: Oliver Kowalke (oliver.kowalke@gmail.com), Nat Goodspeed (nat@lindenlab.com)
Audience: SG1

A low-level API for stackful context switching

Abstract . . . 2
Motivation . . . 2
  Content . . . 2
  Why Bother? . . . 2
Notes on *suspend-up* and *suspend-down* terminology . . . 2
Discussion . . . 3
  Calling subroutines . . . 3
  Why symmetric transfer of control matters . . . 3
  First class object . . . 4
  Stack strategies . . . 4
Design . . . 4
  Suspend-by-call . . . 4
  Call semantics . . . 5
  `std::execution_context<void>` . . . 5
  Passing data . . . 5
  Toplevel functions: main() and thread functions . . . 7
  `std::execution_context<>` and std::thread . . . 7
  Termination . . . 7
  Exceptions . . . 7
  Invoke function on top of a context . . . 7
  Stack destruction . . . 9
  Stack allocators . . . 9
  API . . . 10
A. Existing practice . . . 15
B. *std::execution_context<>* as Thread of Execution (ToE) . . . 15
References . . . 16

Revision History This document supersedes P0099R0.

Changes since P0099R0:

- `std::execution_context<>` with new API
- type-safe transfer of data
- static member function `current()` removed
- invoke function on top of a resumed execution context.

Abstract

This paper proposes a **low-level** and **minimal API** for a stackful execution context, suitable to act as **building-block** for **high-level** constructs such as stackful coroutines as well as cooperative multitasking (aka fibers/userland threads/green threads).

The most important features are:

- first-class object that can be stored in variables or containers
- symmetric API, in which the target context is explicit - enables a richer set of control flows than asymmetric API, in which a subordinate context can only switch back to its invoker
- benefits of traditional stack management retained
- ordinary function calls and returns not affected
- working implementation in Boost.Context[13]

Motivation

Content This paper proposes `std::execution_context<>` that serves as the crucial foundation on which a number of valuable higher-level facilities can be built using pure C++.

`std::execution_context<>` itself is, however, impossible to code in portable C++. Having it provided with the runtime library will be enabling technology for both third-party libraries and user applications.

Why Bother? Given P0057 resumable functions, why should we even consider a completely different mechanism for suspending and resuming a function? Isn't this completely redundant with that?

The answer is "no," for several different reasons.*

- With resumable functions, the markup to support suspension propagates virally through a code base. Resumable functions suspend by returning. Every caller must therefore distinguish between "callee returned value" and "callee suspended, you must also suspend." The new `co_await` keyword makes that distinction, suspending the containing function until the callee produces a value. Every call to a resumable function must itself use `co_await` – which then imposes the same requirement on *its* caller, and so on.
- Forgetting to annotate a call to a resumable function with `co_await` (perhaps because the caller is unaware of its nature, perhaps because the callee changed its nature) can result in subtle timing bugs. Suppose function `f()` returns a `std::future<void>`. The statement `co_await f();` suspends the containing function until the `future` returned by `f()` is fulfilled. Coding simply `f();` merely *discards* that future. Both forms are perfectly legal; both are likely to survive desk-checking and code review. It's even worse if `f()` *usually* fulfills its `future` by the time it returns, only *occasionally* taking longer. That way the error will survive most QA as well, and escape into production.
- The need to `co_await` called functions means that we must evolve a whole new family of `co_await`-aware STL algorithms.†
- Even if one is willing to accept the viral `co_await` markup of resumable functions, using normal encapsulation to manage layers of abstraction could become expensive in runtime. Every entry to a resumable function requires a heap allocation, freed on return. By contrast, once you have allocated a side stack, function call and return on that side stack is just as efficient as function call and return on the original application stack. It's a classic time/space tradeoff. (If you determine to address that issue by drastically pruning local variables from your resumable functions, note that the same tactic could allow you to use a far smaller side stack – which would still be faster for nontrivial call chains.)

*The authors are indebted to Christopher Kohlhoff for his excellent summary in N4453,[4] sections 4.1 and 4.2.
†N4453[4] section 4.2 explains this more fully.

Notes on *suspend-up* and *suspend-down* terminology

The terms *suspend-up* and *suspend-down* were introduced in paper N4232[2] and carried forward in P0158[9] to distinguish stackless (*suspend-up*) and stackful (*suspend-down*) context switching.
These terms rely on a particular visualization of the C++ function call operation in which calling a function passes control "downwards," whereas returning from a function passes control "upwards."
The authors recommend the terms *suspend-by-return* instead of *suspend-up*, and *suspend-by-call* instead of *suspend-down*. The recommended terminology directly references the underlying C++ operations, without requiring a particular visualization.
suspend-by-return (*suspend-up*, or "stackless" context switching) is based on returning control from a called function to its caller, along with some indication as to whether the called function has completed and is returning a result or is merely suspending and expects to be called again. The called function's body is coded in such a way that – if it suspended – calling it again will direct control to the point from which it last returned. This describes both P0057[6] resumable functions and earlier technologies such as Boost.Asio coroutines.[12]
suspend-by-call (*suspend-down*, or "stackful" context switching) is based on calling a function which, transparently to its caller, switches to some other logical chain of function activation records. (This may or may not be a contiguous stack area. The processor's stack pointer register, if any, may or may not be involved.)
This describes N4397[3] coroutines as well as Boost.Context,[13] Boost.Coroutine2[14] and Boost.Fiber.[15]
`std::execution_context<>::operator()()` requires *suspend-by-call* semantics.

Discussion

Calling subroutines The advantage of stackless coroutines is that they reuse the same linear processor stack for stack frames for called subroutines. The advantage of stackful context switching is that it permits **suspending from nested calls**.

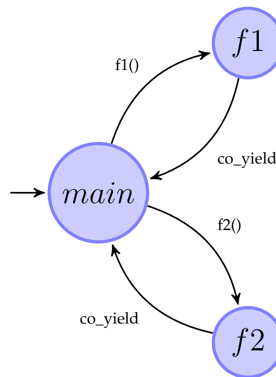
If a resumable function calls a traditional function (rather than another resumable function), then the activation record belonging to the traditional function is allocated on the processor stack (so it is called a stack frame). As a consequence, stack frames of called functions must be removed from the processor stack before the resumable function yields back to its caller.
In other words: the calling convention of the **ABI dictates** that, when the resumable function returns (suspends), the stack pointer must contain the same address as before the resumable function was entered.
Hence a yield from nested call is **not permitted** – unless every called function down to the yield point is also a resumable function.
The benefit of stackless coroutines consists in reusing the processor stack for called subroutines: no separate stack memory need be allocated.

Of course even a stackless resumable function might fail if its called functions exhaust the available stack.

In stackful context switching, each execution context owns a distinct side stack which is assigned to the stack pointer (thus the stack pointer must be exchanged during each context switch).
All activation records (stack frames) of subroutines are placed on the side stack. Hence each stackful execution context requires enough memory to hold the stack frames of the longest call chain of subroutines. Therefore, to support calling subroutines, stackful context switching has a higher memory footprint than resumable functions.
On the other hand, it is beneficial to use side stacks because the stack frames of active subroutines remain intact while the execution context is suspended. This is the reason why stackful context switching permits **yielding from nested calls**.

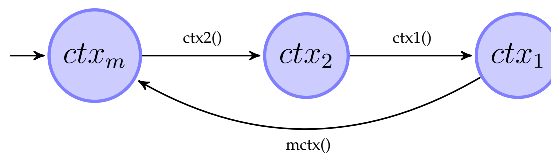
Why symmetric transfer of control matters As a building block for user-mode threads, symmetric control transfer is more efficient than the asymmetric mechanism.

Asymmetric coroutines (as implied with keywords like `resumable`, `co_await` or `co_yield`) provide two operations for context switching. The caller and the coroutine are coupled, that is, such a coroutine can only jump back to the code that most recently resumed it.



For **N** asymmetric coroutines, **2N** context switches are required. This is sufficient in the case of generators, but in the context of cooperative multitasking it is inefficient.

The proposed stackful execution context (`std::execution_context<>`) provides only one operation to resume/suspend the context (`operator()()`). Control is directly transferred from one execution context to another (symmetric control transfer) - no jump back to the caller. In addition to supporting generators, this enables an efficient implementation of cooperative multitasking: **no additional context switch** back to caller, direct context switch to next task.
The next execution context must be explicitly specified.



Resuming **N** instances of `std::execution_context<>` takes **N+1** context switches.

First class object Symmetric control transfer requires that the context is represented by a **first class object** (context that has to be resumed next must be selectable).

Stack strategies For stackful coroutines two strategies are typical: a contiguous, fixed-size stack (as used by threads), or a linked stack (grows on demand).
The advantage of a fixed-size stack is the fast allocation/deallocation of activation records. A disadvantage is that the required stacksize must be guessed.
The benefit of using a linked stack is that only the initial size of the stack is required. The stack itself grows on demand, by means of an overflow handler. The performance penalty is low. The disadvantage is that code executed inside a stackful coroutine must be compiled for this stack model. In the case of GCC's split stacks, special compiler/linker flags must be specified - no changes to source code are required.
When calling a library function not compiled for linked stacks (expecting a traditional contiguous stack), GCC's implementation uses link-time code generation to change the instructions in the caller. The effect is that a reasonably large contiguous stack chunk is temporarily linked in to handle the deepest expected chain of traditional function stack frames (see GCC's documentation[10]).

Design

Class `std::execution_context<>` provides a **low-level** and **minimal API** on which to build **higher-level APIs** such as stackful coroutines (N3985[1]) and user-mode threads (such as Boost.Fiber[15]).

Suspend-by-call `std::execution_context<>::operator()()` preserves the CPU register set*: the content of those registers is pushed at the end of the stack of the current context (at the current stack-pointer). Then `operator()()` restores the stack-pointer register stored in `*this` and pops the CPU register set from the newly-restored stack. Because the context state is preserved on the context's stack, a `std::execution_context<>` instance need only store the stack-pointer register.

Call semantics When `std::execution_context<>::operator()()` is called, a new instance of `std::execution_context<>` is synthesized representing the current state of the running context (e.g. the stack-pointer). This new instance is passed to the resumed context. On initial entry, it is passed as the first argument to the top-level function. On every subsequent resumption, the new `std::execution_context<>` instance is returned by the suspended `operator()()` call.
On completion of a successful context switch, the `std::execution_context<>` instance on which `operator()()` was called is invalidated. The data member from which the stack pointer was just restored is set to `nullptr`.
At most one instance of `std::execution_context<>` can represent a given execution context. The currently-running execution context is not represented by *any* `std::execution_context<>` instance. Only when `operator()()` is called on some *other* `std::execution_context<>` instance is the state of the running execution context captured in a synthesized `std::execution_context<>` instance.
As mentioned in the (destructor) section, `~execution_context<>()` on a suspended (not terminated) instance destroys the stack managed by that instance. Thus, the stack must be managed by only one `std::execution_context<>` instance.†
Because of the symmetric context switching (only one operation transfers control), the target execution context must be explicitly specified.

`std::execution_context<void>` With `std::execution_context<void>` no data will be transferred, only the context switch is executed.
The function or lambda passed to the constructor of a `std::execution_context<void>` must accept a single `std::execution_context<void>` parameter and return `std::execution_context<void>`.

```
std::execution_context<void> ctx1([](std::execution_context<void>&& ctx2){
            std::cout << "inside ctx1" << std::endl;
            return std::move(ctx2);
        });
ctx1();

output:
    inside ctx1
```

`ctx1()` resumes `ctx1`, that is, control enters the lambda passed to the constructor of `ctx1`. Argument `ctx2` represents the previous context: the context that was suspended by the call to `ctx1()`. When the lambda returns `ctx2`, context `ctx1` will be terminated while the context represented by `ctx2` is resumed, hence control returns from `ctx1()`.

Passing data You may pass data between contexts by constructing a `std::execution_context<>` with template arguments other than `void`. Consider `std::execution_context<args...>` (where `args...` here represents any list of type template arguments other than `void`), for purposes of discussion. The function or lambda that initializes that instance must accept parameters `(std::execution_context<args...>, args...)`.
It must return `std::execution_context<args...>`.
The initial `std::execution_context<>` argument is synthesized by `operator()()`. All other arguments must be passed explicitly to `operator()()`.
The first call to `operator()()` with those arguments populates the parameter list for the newly-entered function or lambda.

*defined by ABI's calling convention
†An earlier design used reference counting, but that subverts the intended role of this facility as an extremely fast substrate for higher-level libraries.

That function or lambda switches context back to the original context by calling the passed `std::execution_context<>::operator()()`, passing appropriate arguments.
A `std::tuple<std::execution_context<args...>, args...>` is returned by the *original* context's call to `operator()()`. The returned `std::execution_context<>` is a synthesized instance representing the context that just suspended. The rest of the `args...` are as passed by that context to `operator()()`.
So, for instance:

```
std::execution_context<int> ctx1([](std::execution_context<int>&& ctx2,int j){
            std::cout << "inside ctx1,j==" << j << std::endl; // (b)
            std::tie(ctx2,j) = // (f)
                ctx2(j+1); // (c)
            return std::move(ctx2); // (g)
        });
int i=1;
std::tie(ctx1,i) = // (d)
    ctx1(i); // (a)
std::cout << "i==" << i << std::endl;

output:
    inside ctx1,j==1
    i==2
```

The `ctx1(i)` call at (a) enters the lambda in context `ctx1` with argument `j=1`, as shown by the output at (b). The expression `ctx2(j+1)` at (c) resumes the original context (represented within the lambda by `ctx2`) and transfers back an integer of `j+1`. On return from `ctx1(i)`, the assignment at (d) sets `i` to `j+1`, or 2.
The assignment at (d) illustrates a recommended idiom: since the call to `operator()()` at (a) has invalidated `ctx1`, it should be replaced by the newly-synthesized `std::execution_context<>` instance returned by `operator()()`.
To continue the example:

```
std::tie(ctx1,i) = // (h)
    ctx1(i); // (e)
assert(! ctx1); // (i)
// ignore i: value unspecified
```

The call to `ctx1(i)` at (e) (the *updated* `ctx1`) resumes the `ctx1` lambda, returning from the `ctx2()` call at (c) and executing the assignment at (f). Here, too, we replace the `std::execution_context<>` instance `ctx2` invalidated by the `operator()()` call at (c) with the new instance returned by that same `operator()()` call. Moreover, we replace `j` with the value passed by the call at (e).
Finally the lambda returns (the updated) `ctx2` at (g), terminating its context.
Since the updated `ctx2` represents the context suspended by the call at (e), control returns to the assignment at (h). Once again we replace the invalidated `ctx1` with the one returned by `operator()()`.
However, since context `ctx1` has now terminated, the updated `ctx1` is invalid. Its `operator bool()` returns `false`; its `operator!()` returns `true`.
This is important, since in that case the values of any remaining fields of the returned `std::tuple` are indeterminate.
It may seem tricky to keep track of which `std::execution_context<>` instance is currently valid, representing the state of the suspended context. Please bear in mind that this facility is intended as a high-performance foundation for higher-level libraries. It is not intended to be directly consumed by applications.
We can extend the example to multiple arguments.

```
std::execution_context<int,int> ctx1(
        [](std::execution_context<int,int>&& ctx2,int i,int j){
            std::cout << "inside ctx1,i==" << i << " j==" << j << std::endl;
            std::tie(ctx2,i,j)=ctx2(i+j,i-j);
            return std::move(ctx2);
        });
int i=3,j=2;
std::tie(ctx1,i,j)=ctx1(i,j);
```

```
std::cout << "i==" << i << " j==" << j << std::endl;

output:
    inside ctx1,i==3 j==2
    i==5 j==1
```

`operator()()` accepts the parameters specified by `std::execution_context<>`'s template parameters. It returns a `std::tuple` of that `std::execution_context<>` specialization, prepended to those types.

Toplevel functions: main() and thread functions `main()` as well as the *entry-function* of a thread can be represented by an execution context. That `std::execution_context<>` instance is synthesized when the running context suspends, and is passed into the newly-resumed context.

```
int main() {
    std::execution_context<void> ctx1(
        [](std::execution_context<void>&& ctx2){ // (b)
            return std::move(ctx2); // (c)
        });
    ctx1(); // (a)
    return 0;
}
```

The `ctx1()` call at (a) enters the lambda in context ctx1.
The `std::execution_context<> ctx2` at (b) represents the execution context of `main()`.
Returning `ctx2` at (c) resumes the original context (switch back to `main()`).

`std::execution_context<>` and std::thread Any execution context represented by a valid `std::execution_context<>` instance is necessarily suspended.
It is valid to resume a `std::execution_context<>` instance on any thread – *except* that you must not attempt to resume a `std::execution_context<>` instance representing `main()`, or the *entry-function* of some other `std::thread`, on any thread other than its own. `std::execution_context<>` provides a method to test for this. If `std::execution_context<>::any_thread()` returns `false`, it is only valid to resume that `std::execution_context<>` instance on the thread on which it was initially launched.

Termination When you explicitly construct a particular `std::execution_context<args...>` specialization, passing its constructor a function or lambda, that callable must accept that same `std::execution_context<args...>` specialization as its first parameter. It must return that same `std::execution_context<args...>` specialization as well.
When that toplevel callable returns a `std::execution_context<>` instance, the running context is voluntarily terminated. Control switches to the context indicated by the returned `std::execution_context<>` instance.
Returning an invalid `std::execution_context<>` instance (`operator bool()` returns `false`) invokes undefined behavior.
If the toplevel callable returns the same `std::execution_context<>` instance it was originally passed (or rather, the most recently updated instance returned from the previous instance's `operator()()`), control returns to the context that most recently resumed the running callable. However, the callable may return (switch to) any reachable valid `std::execution_context<>` instance with the correct type signature.

Exceptions If an uncaught exception escapes from the toplevel context function, `std::terminate` is called.

Invoke function on top of a context Sometimes it is useful to invoke a new function (for instance, to trigger unwinding the stack) on top of a suspended context. For this purpose you may pass to `std::execution_context<>::operator()()`:

- the special argument `invoke_ontop_arg`

- the function to execute
- any additional arguments required by the `std::execution_context<>` specialization.

The function passed in this case must accept as parameters the `std::execution_context<>` specialization for that context plus any arguments specified by that specialization. It must return a tuple of that `std::execution_context<>` specialization plus the same set of arguments.*
For purposes of discussion, consider two `std::execution_context<void>` instances: `mctx` and `fctx`. Suppose that code running on the program's main context instantiates `fctx` with function `f(std::execution_context<void>&& mctx)` and calls `fctx()`, thereby entering `f()`. This is the point at which `mctx` is synthesized and passed into `f()`.
Suppose further that after doing some work, `f()` calls `mctx()`, thereby switching context back to the main context. `f()` remains suspended in the call to `mctx.operator()()`.
At this point the main context calls `fctx(std::invoke_ontop_arg, g);` where `g()` is declared as:
`std::execution_context<void> g(std::execution_context<void> gmctx);`
`g()` is entered in the context of `f()`. It is as if `f()`'s call to `mctx.operator()()` directly called `g()`.
However, as usual, the `fctx.operator()()` call synthesizes a `std::execution_context<>` instance representing the main context and passes it to `g()` as `gmctx`.
Function `g()` has the same range of possibilities as any function called on `f()`'s context. It can context-switch back to the main context, or to any other reachable context. Its special invocation only matters when control leaves it in either of two ways:

1. If `g()` throws an exception, that exception unwinds all previous stack entries in that context (such as `f()`'s) as well, back to a matching `catch` clause.†
2. If `g()` returns, its return value becomes the value returned by `f()`'s suspended `mctx.operator()()` call. This is why `g()`'s return type must be the same as that of `operator()()`, rather than that of an ordinary toplevel context function.

Consider the following example:

```
// f1() is the toplevel context function:
// it returns only std::execution_context<args...>
std::execution_context<int> f1(std::execution_context<int>&& ctx,int data) {
    std::cout << "f1: entered first time: " << data << std::endl; // (b)
    std::tie(ctx,data) = // (g)
        ctx(data+1); // (c)
    std::cout << "f1: entered second time: " << data << std::endl; // (h)
    std::tie(ctx,data) = // (o)
        ctx(data+1); // (i)
    std::cout << "f1: entered third time: " << data << std::endl; // (p)
    return std::move(ctx); // (q)
}

// f2() is an invoke_ontop_arg function:
// though its parameter list is very like that of a toplevel context function,
// it must return std::tuple<std::execution_context<args...>, args...>
std::tuple<std::execution_context<int>,int>
f2(std::execution_context<int>&& ctx,int data) {
    std::cout << "f2: entered: " << data << std::endl; // (m)
    return std::make_tuple(std::move(ctx),-1); // (n)
}

int data=0;
std::execution_context<int> ctx(f1);
```

*But in the case of `std::execution_context<void>`, the return type is simply `std::execution_context<void>`.
†As stated in Exceptions, if there is no matching `catch` clause in that context, `std::terminate()` is called.

```
std::tie(ctx,data) = // (d)
    ctx(data+1); // (a)
std::cout << "f1: returned first time: " << data << std::endl; // (e)
std::tie(ctx,data) = // (j)
    ctx(data+1); // (f)
std::cout << "f1: returned second time: " << data << std::endl; // (k)
std::tie(ctx,data) = // (r)
    ctx(std::invoke_ontop_arg,f2,data+1); // (l)

output:
    f1: entered first time: 1
    f1: returned first time: 2
    f1: entered second time: 3
    f1: returned second time: 4
    f2: entered: 5
    f1: entered third time: -1
```

Control passes from (a) to (b) to (c), and so on.
The `ctx(std::invoke_ontop_arg, f2, data+1)` call at (l) passes control to `f2()` on the context originally created for `f1()`.
The `return` statement at (n) causes the `operator()()` call at (i) to return, executing the assignment at (o).
The `std::tuple` returned by `f2()` is directly returned to that assignment at (o).
So in this example, the call at (l) synthesizes a `std::execution_context<>` instance representing the main context and passes it to `f2()` as `ctx`. `f2()` returns that `ctx` instance, which is received by `f1()` and assigned to *its* `ctx` variable. Finally, `f1()` returns its own `ctx` variable, switching back to the main context.

Stack destruction On construction of `execution_context` a stack is allocated. If the toplevel context-function returns, the stack will be destroyed. If the context-function has not yet returned and the (destructor) of a valid `execution_context` instance (`operator bool()` returns `true`) is called, the stack will be unwound and destroyed.*
The stack on which `main()` is executed, as well as the stack implicitly created by `std::thread`'s constructor, is allocated by the operating system. Such stacks are recognized by `std::execution_context<>`, and are not deallocated by its destructor.

Stack allocators are used to create stacks.† Stack allocators might implement arbitrary stack strategies. For instance, a stack allocator might append a guard page at the end of the stack, or cache stacks for reuse, or create stacks that grow on demand.
Because stack allocators are provided by the implementation, and are only used as parameters of `std::execution_context<>`'s constructor, the StackAllocator concept is an implementation detail, used only by the internal mechanisms of the `std::execution_context<>` implementation. Different implementations might use different StackAllocator concepts.
However, when an implementation provides a stack allocator matching one of the descriptions below, it should use the specified name.
Possible types of stack allocators:

- `protected_fixedsize`: The constructor accepts a `size_t` parameter. This stack allocator constructs a contiguous stack of specified size, appending a guard page at the end to protect against overflow. If the guard page is accessed (read or write operation), a segmentation fault/access violation is generated by the operating system.

*An implementation is free to unwind the stack without throwing an exception. However, if an exception is thrown, it should be named `std::execution_context_unwind`. Portable consumer code *must* permit `std::execution_context_unwind` exceptions to propagate, even if all other exceptions are caught with `catch (...)`.

†This concept, along with the `std::execution_context<>` constructor accepting `std::allocator_arg_t`, is an optional part of the proposal. It might be that implementations can reliably infer the optimal stack representation.

- `fixedsize`: The constructor accepts a `size_t` parameter. This stack allocator constructs a contiguous stack of specified size. In contrast to `protected_fixedsize`, it does not append a guard page. The memory is simply managed by `std::malloc()` and `std::free()`, avoiding kernel involvement.
- `segmented`: The constructor accepts a `size_t` parameter. This stack allocator creates a segmented stack with the specified initial size, which grows on demand.

API declaration of class `std::execution_context<>`

```
template<typename ...Args>
class execution_context {
public:
    execution_context() noexcept;

    template<typename Fn,
             typename ...Params>
    execution_context(Fn&& fn, Params&& ...params);

    template<typename StackAlloc,
             typename Fn,
             typename ...Params>
    execution_context(std::allocator_arg_t, StackAlloc salloc,
                      Fn&& fn, Params&& ...params);

    ~execution_context();

    execution_context(execution_context&& other) noexcept;

    execution_context&
    operator=(execution_context&& other) noexcept;

    execution_context(const execution_context& other)=delete;

    execution_context&
    operator=(const execution_context& other)=delete;

    std::tuple<execution_context, Args ...>
    operator()(Args ...args);

    template<typename Fn>
    std::tuple<execution_context, Args ...>
    operator()(std::invoke_ontop_arg_t, Fn&& fn, Args ...args);

    explicit operator bool() const noexcept;

    bool operator!() const noexcept;

    bool any_thread() const noexcept;

    bool operator==(const execution_context& other) const noexcept;

    bool operator!=(const execution_context& other) const noexcept;

    bool operator<(const execution_context& other) const noexcept;

    bool operator>(const execution_context& other) const noexcept;
```

```
    bool operator<=(const execution_context& other) const noexcept;

    bool operator>=(const execution_context& other) const noexcept;

    void swap(execution_context& other) noexcept;
};
```

member functions

(constructor) constructs new execution context

`execution_context() noexcept`	(1)
`template<typename Fn, typename ...Params>` `explicit execution_context(Fn&& fn, Params&& ...params)`	(2)

`template<typename StackAlloc, typename Fn, typename ...Params>`
`execution_context(std::allocator_arg_t, StackAlloc salloc,`
`Fn&& fn, Params&& ...params)`
	(3)

`execution_context(execution_context&& other)`	(4)
`execution_context(const execution_context& other)=delete`	(5)

1) This constructor instantiates an invalid `std::execution_context<>`. Its `operator bool()` returns `false`; its `operator!()` returns `true`.

2) takes a callable (function, lambda, object with `operator()()`) as argument. The callable must have signature as described in `std::execution_context<void>` or Passing data. The stack is constructed using either `fixedsize` or `segmented` (see Stack allocators). An implementation may infer which of these best suits the code in `fn`. If it cannot infer, `fixedsize` will be used.

3) takes a callable as argument, requirements as for (2). The stack is constructed using *salloc* (see Stack allocators).*

4) moves underlying state to new `std::execution_context<>`

5) copy constructor deleted

Notes
When a `std::execution_context<>` is constructed using either of the constructors accepting `fn`, control is *not* immediately passed to `fn`. Resuming (entering) `fn` is performed by calling `operator()()` on the new `std::execution_context<>` instance.

(destructor) destroys an execution context

`~execution_context()`	(1)

1) destroys a `std::execution_context<>` instance. If this instance represents a context of execution (`operator bool()` returns `true`), then the context of execution is destroyed too. Specifically, the stack is unwound. As noted in Stack destruction, an implementation is free to unwind the stack either by throwing `std::execution_context_unwind` or by intrinsics not requiring `throw`.

*This constructor, along with the Stack allocators section, is an optional part of the proposal. It might be that implementations can reliably infer the optimal stack representation.

operator= moves the context object

`execution_context& operator=(execution_context&& other)`	(1)
`execution_context& operator=(const execution_context& other)=delete`	(2)

1) assigns the state of `other` to `*this` using move semantics

2) copy assignment operator deleted

Parameters

other another execution context to assign to this object

Return value

***this**

operator() resume context of execution

`std::tuple<execution_context, Args ...> operator()(Args ...args)`	(1)
`execution_context<void> operator()()`	(2)

`template<typename Fn>`
`std::tuple<execution_context, Args ...>`
`operator()(invoke_ontop_arg_t, Fn&& fn, Args ...args)`
	(3)

`template<typename Fn>`
`execution_context<void>`
`operator()(invoke_ontop_arg_t, Fn&& fn)`
	(4)

1) suspends the active context, resumes the execution context

2) specialization of (1) for `execution_context<void>`

3) suspends the active context, resumes the execution context but executes `fn(args ...)` in the resumed context (on top of the last stack frame)

4) specialization of (3) for `execution_context<void>`

Parameters

...args If the toplevel context-function represented by `*this` has not yet been entered, the arguments you pass are passed to the context-function as its parameters, following the `std::execution_context<>` first parameter.
If the context represented by `*this` suspended by calling `operator()()`, the arguments you pass are constructed into a `std::tuple<std::execution_context<args...>, args...>` and returned by that suspended `operator()()` call.
See section Passing data.

fn The `fn` passed to (3) must accept `std::execution_context<args...>, args...`. It must return `std::tuple<std::execution_context<args...>, args...>`.
The `fn` passed to (4) must accept `std::execution_context<void>`.
It must return `std::execution_context<void>`.

Return value

void When called on a `std::execution_context<void>` instance, `operator()()` returns a different `std::execution_context<void>` instance. This new instance represents the context that suspended in order to resume the current context. That may or may not be the same context that was previously represented by `*this`, depending on other context switches executed in the meantime.

tuple When called on a `std::execution_context<args...>` instance, `operator()()` returns a `std::tuple<std::execution_context<args...>, args...>` containing a different `std::execution_context<args...>` instance. This new instance represents the context that suspended in order to resume the current context, as above.
If the context represented by the new `std::execution_context<args...>` instance suspended by calling `operator()()`, the arguments passed to `operator()()` are used to populate the rest of the members of the returned `std::tuple`.
See section Passing data.
If the context represented by the new `std::execution_context<args...>` instance voluntarily terminated by returning from its toplevel context-function, the rest of the members of the returned `std::tuple` are indeterminate.

Exceptions

1) calls `std::terminate` if an exception escapes toplevel `fn`

Preconditions

1) `*this` represents a context of execution (`operator bool()` returns `true`)

2) `any_thread()` returns `true`, or the running thread is the same thread on which `*this` ran previously.

Postcondition

1) `*this` is invalidated (`operator bool()` returns `false`)

Notes
The *prologue* preserves the execution context of the calling context as well as stack parts like *parameter list* and *return address*.* Those data are restored by the *epilogue* if the calling context is resumed.
A suspended `execution_context` can be destroyed. Its resources will be cleaned up at that time.
The returned `execution_context` indicates whether the suspended context has terminated (returned from toplevel context-function) via `operator bool()`. If the returned `execution_context` has terminated, no data are transferred in the returned tuple.
Because `operator()()` invalidates the instance on which it is called, *no valid `std::execution_context<>` instance ever represents the currently-running context.*
When calling `operator()()`, it is conventional to replace the newly-invalidated instance – the instance on which `operator()()` was called – with the new instance returned by that `operator()()` call. This helps to avoid inadvertent calls to `operator()()` on the old, invalidated instance.
For any `std::execution_context<>` specialization other than `std::execution_context<void>`, when `operator()()` returns, it is important to test the returned `std::execution_context<args...>` instance using `operator bool()` or `operator!()` before referencing any of the `args...` in the returned `std::tuple<std::execution_context<args...>, args...>`. If that context voluntarily terminated by returning from the toplevel context-function, only the `std::execution_context<args...>` member of the `std::tuple` will be populated. The rest of the members will have indeterminate values.

operator bool   test whether context is valid

`explicit operator bool()const noexcept`	(1)

1) returns `true` if `*this` represents a context of execution, `false` otherwise.

Notes
A `std::execution_context<>` instance might not represent a context of execution for any of a number of reasons.

- It might have been default-constructed.

*required only by some x86 ABIs

- It might have been assigned to another instance, or passed into a function. `std::execution_context<>` instances are move-only.
- It might already have been resumed (`operator()()` called). Calling `operator()()` invalidates the instance.
- The toplevel context-function might have voluntarily terminated the context by returning.

The essential points:

- Regardless of the number of `std::execution_context<>` declarations, exactly one `std::execution_context<>` instance represents each suspended context.
- No `std::execution_context<>` instance represents the currently-running context.

operator! test whether context is invalid

`bool operator!() const noexcept`	(1)

1) returns `false` if `*this` represents a context of execution, `true` otherwise.

Notes
See **Notes** for `operator bool()`.

any_thread test whether suspended context may be resumed on a different thread

`bool any_thread() const noexcept`	(1)

1) returns `false` if `*this` must be resumed on the same thread on which it previously ran, `true` otherwise

Notes
As stated in Toplevel functions: main() and thread functions, a `std::execution_context<>` instance can represent the initial context on which the operating system runs `main()`, or the context created by the operating system for a new `std::thread`.
It is not permitted to attempt to resume such a `std::execution_context<>` instance on any thread other than its original thread. `any_thread()` allows consumer code to distinguish this case.

(comparisons) establish an arbitrary total ordering for `std::execution_context<>` instances

`bool operator==(const execution_context& other) const noexcept`	(1)
`bool operator!=(const execution_context& other) const noexcept`	(1)
`bool operator<(const execution_context& other) const noexcept`	(2)
`bool operator>(const execution_context& other) const noexcept`	(2)
`bool operator<=(const execution_context& other) const noexcept`	(2)
`bool operator>=(const execution_context& other) const noexcept`	(2)

1) Every invalid `std::execution_context<>` instance compares equal to every other invalid instance. But because the running context is never represented by a valid `std::execution_context<>` instance, and because every suspended context is represented by exactly one valid instance, *no valid instance can ever compare equal to any other valid instance.*

2) These comparisons establish an arbitrary total ordering of `std::execution_context<>` instances, for example to store in ordered containers. (However, key lookup is meaningless, since you cannot construct a search key that would compare equal to any entry.) There is no significance to the relative order of two instances.

swap swaps two `std::execution_context<>` instances

`void swap(execution_context& other)noexcept`	(1)

1) Exchanges the state of `*this` with `other`.

A. Existing practice

As a library-only facility, `boost::context::execution_context<>`[13] is existing practice, used to implement Boost.Coroutine2[14] and the Boost.Fiber library.[15]

The facility described in this proposal adds significant semantics beyond resumable functions. For instance, you cannot build cooperative user-mode threads on resumable functions.

These libraries based on Boost.Context can seamlessly interoperate with Boost.Asio:[11] Asio's `async_result` mechanism (proposed for standardization in N4588[5]) supports adapters that can, using a natural syntax, suspend the caller for the duration of an asynchronous network call. For instance, using Boost.Coroutine2:

```
void read_msg(yield_context yield) {
    try {
        array< char, 64 > bf1;
        async_read(socket,buffer(b1),yield);
        header hdr(bf1);
        std::size_t n=hdr.payload_size();
        std::vector< char > b2(n,'\0');
        async_read(socket,buffer(bf2),yield);
        payload pld(bf2);
        // process message ...
    } catch (exception const&) {
        // ...
    }
}
```

It is important to realize that in these examples, the symbol `yield` is neither a keyword nor a macro, but simply the name of an ordinary C++ object.
The same example recast using Boost.Fiber:

```
void read_msg() {
    try {
        array< char, 64 > bf1;
        async_read(socket,buffer(b1),boost::fibers::asio::yield);
        header hdr(bf1);
        std::size_t n=hdr.payload_size();
        std::vector< char > b2(n,'\0');
        async_read(socket,buffer(bf2),boost::fibers::asio::yield);
        payload pld(bf2);
        // process message ...
    } catch (exception const&) {
        // ...
    }
}
```

Libraries based on `std::execution_context<>` will similarly interoperate cleanly with the proposed standard Networking Library.[5]

At C++Now 2016[16] and CppCon 2016,[17] Nat Goodspeed presented the Fiber library and described some use cases.

B. *std::execution_context<>* as Thread of Execution (ToE)

As described in P0073,[8] a `std::execution_context<>` represents a ToE.
Each `std::execution_context<>` has its own execution path (sequence of instructions). (In the terminology used by P0072,[7] this ToE runs on a "weakly parallel" execution agent.) An operating system thread (`std::thread`) consists of at least one `std::execution_context<>` representing the main-stack/thread-stack.
Multiple `std::execution_context<>` instances might interact in following manner:

- an **asymmetric coroutine** involves two `std::execution_context<>` s simply passing control back and forth to each other
  - the two are strongly coupled; the callee `std::execution_context<>` can only resume its caller
  - they directly exchange data (in general this could be bidirectional; a generator restricts data flow to a single direction)
- each **fiber** runs on a `std::execution_context<>` of its own
  - fibers are loosely coupled: a scheduler selects the next ready fiber
  - no direct data transfer
- **shift/reset operators** involve a few interacting `std::execution_context<>` s
  - coupled
  - data exchanged
  - interleaved transfer of execution

References

[1] N3985: A proposal to add coroutines to the C++ standard library

[2] N4232: Stackful Coroutines and Stackless Resumable Functions

[3] N4397: A low-level API for stackful coroutines

[4] N4453: Resumable Expressions

[5] N4588: Working Draft, C++ extensions for Networking

[6] P0057R5: Wording for Coroutines

[7] P0072R1: Light-Weight Execution Agents

[8] P0073R2: On unifying the coroutines and resumable functions proposals

[9] P0158R0: Coroutines belong in a TS

[10] Split Stacks / GCC

[11] Library *Boost.Asio*

[12] *Boost.Asio* Coroutines

[13] Library *Boost.Context*

[14] Library *Boost.Coroutine2*

[15] Library *Boost.Fiber*

[16] C++Now 2016: Nat Goodspeed, *The Fiber Library*

[17] CppCon 2016: Nat Goodspeed, *Elegant Asynchronous Code*