Document number: P0534R1
Date: 2017-06-18
Reply-to: Oliver Kowalke (oliver.kowalke@gmail.com)
Authors: Oliver Kowalke (oliver.kowalke@gmail.com), Nat Goodspeed (nat@lindenlab.com)
Audience: SG1/LEWG

call/cc (call-with-current-continuation): A low-level API for stackful context switching

Abstract

This document proposes a C++ equivalent to the well-known concept **call-with-current-continuation** (abbreviated **call/cc**). This facility permits a program written in portable C++ to subdivide processing into distinct **contexts:** units smaller than a thread.

Within this proposal, the unadorned term "thread" means a `std::thread` (or **kernel thread**). When the Standard's more general term "thread of execution" is intended, it is spelled out in full.

With *call/cc*, processing in a given thread may be further subdivided into multiple contexts. Each such context qualifies as a "thread of execution" according to the definition in the Standard. However, within a given thread, control is cooperatively passed from one context to another.

This has a couple of important implications:

- In each thread in a process, exactly one context is running at any given time. All others are **suspended.**
- The running context on a thread continues running until it explicitly **resumes** some other context. The act of resuming another context suspends the previously-running context. This transfer of control, in which one context suspends and another resumes, is **context-switching.**
- There are no data races between contexts running on the same thread.

The kind of context-switching presented in this proposal is called **stackful** because each context requires some implementation of the C++ stack. C++ code running on a particular context may transparently call ordinary C++ functions. In contrast to the **`co_await`** facility (proposed separately[4]), this permits encapsulation. A function that suspends (by resuming some other context) needs no special signature. Its caller need not be aware that it might suspend. It need not call that function in any special way.

This supports use cases that cannot be addressed with `co_await` alone.

Also in contrast to the `co_await` facility, this proposal requires no changes to the core C++ language. *call/cc* is presented as a library facility, albeit a library that cannot be implemented in portable C++. This is why it is desirable to incorporate it into the International Standard.

Consider the following bullets from P0559R0:[6]

- Avoid 'compiler magic' when possible
- Prefer library solutions over language changes if feasible

The proposed *call/cc* facility is intended to be foundational. While of course application coders are free to use the *call/cc* API, its real promise is in supporting higher-level abstractions.

This proposal describes the basic *call/cc* facility, presents some illustrative use cases and explains why the API is set at its present level.

Why should *call/cc* be standardized?

The *call/cc* facility cannot itself be implemented in portable C++. The present implementation,[12] maintained by a single author, supports a small set of current platforms available to that author. Should *call/cc* be integrated into the Standard, it will become universally available.

Moreover, correct support for certain platforms might involve undocumented complexity. The runtime vendor is best positioned to implement the specified functionality.

Compiler awareness of this facility could enable certain optimizations as well:

- The compiler might be able to analyze the code to be launched on a new *call/cc* context and determine an optimal stack size for that context.
- The compiler might be able to determine that not all registers need be preserved across a context switch.
- For certain use cases, the compiler might be able to optimize away context-switching altogether. Promising work has been done in this area for the `co_await` facility.[4]

Revision History

This document supersedes P0534R0.[5]

Changes since P0534R0:

- API modified
  - `operator()` renamed to `resume()`
  - `operator(invoke_ontop_arg_t)` renamed to `resume_with()`
  - `data_available()`,`get_data()` and `any_thread()` are now member functions instead of free functions
  - stack unwinding now explicitly requires specific `unwind_exception` exception
- added example use cases
- *call/cc* compared with *ucontext* and *longjmp*
- why *call/cc* is a low-level implementation
- details of stack destruction

Continuations

A continuation is an abstract concept that represents the context state at a given point during the execution of a program. That implies that a continuation represents the remaining steps of a computation.

As a **basic, low-level primitive** it can be used to implement control structures like coroutines, generators, lightweight threads, cooperative multitasking (fibers), backtracking, non-deterministic choice. In classic event-driven programs, organized around a main loop that fetches and dispatches incoming I/O events, certain asynchronous I/O sequences are logically sequential. Use of continuations permits writing and maintaining code that looks and acts sequential, even though from time to time it may suspend while asynchronous I/O is pending.

C and C++ already use implicit continuations: when running code calls a function, then a (hidden) continuation (the remaining steps after the function call) is created. This continuation is resumed when the function returns. For instance the x86 architecture stores the (hidden) continuation as a return address on the stack.[*]

Continuations exposed as **first-class continuations** can be passed to and returned from functions, assigned to variables or stored into containers. With first-class continuations, a language can explicitly **control the flow of execution** by suspending and resuming continuations, enabling control to pass into a function at exactly the point where it previously suspended. Making the program state visible via first-class continuations is known as **reification**.

The remainder of the computation derived from the current point in a program's execution is called the **current continuation**. *call/cc* captures the **current continuation** and passes it to the function invoked by *call/cc*.

Continuations that can be called multiple times are named **full continuations**.
One-shot continuations can only resumed once: once resumed, a **one-shot continuation** is invalidated.
Full continuations are **not** considered in this proposal because of their nature, which is problematic in C++. Full continuations would require copies of the stack (including all stack variables), which would violate C++'s RAII pattern.

In contrast to *call/cc* that captures the **entire remaining** continuation, the operators *shift* and *reset* create a so-called **delimited continuation**. A delimited continuation represents a slice of the program context. Operator *reset* delimits the continuation, i.e. it determines where the continuation starts and ends, while *shift* **reifies** the continuation.
Delimited continuations are **not** part of this proposal. However, delimited continuation functionality can be built on *call/cc*.

[*]Other (RISC) architectures use a special micro-processor register for this purpose.

Call with current continuation

call/cc (abbreviation of 'call with current continuation') is a universal control operator (well-known from languages like Scheme, Ruby, Lisp ...) that captures the **current continuation** (the sequence of instructions after *call/cc* returns) as a **first-class object** and passes it to a function that is executed in a newly-created execution context.

`std::callcc()` is the C++ equivalent to *call/cc*, preserving the **call state** and the **program state** (variables).

When code running in some *original* context calls `resume()` on some `std::continuation` instance `target`, the *original* context is saved and the `target` continuation is restored in its place, so that program flow will continue at the point at which the `target` continuation was originally captured. The captured *original* continuation then becomes the *return value* of the `std::callcc()` invocation in `target`.

`std::continuation` is a **one-shot continuation**: it can be resumed at most once, is only move-constructible and move-assignable.

```
std::continuation foo(std::continuation && caller) {
    while (caller) {
        std::cout << "foo\n";
        caller= // (4)
           caller.resume(); // (1)
    }
    return std::move(caller);
}

std::continuation foo_ct= // (2)
                     std::callcc(foo); // (0)
while (foo_ct) {
    std::cout << "bar\n";
    foo_ct= // (5)
       foo_ct.resume(); // (3)
}

output:
    foo
    bar
    ...
```

The `std::callcc(foo)` call at (0) captures the **current continuation**, entering function `foo()` while passing the captured continuation as argument `caller`.

As long as continuation `caller` is valid, `"foo"` is passed to standard output.

The expression `caller.resume()` at (1) resumes the original continuation represented within `foo()` by `caller` and transfers back the control of execution to `main()`. On return from `std::callcc(foo)`, the assignment at (2) sets `foo_ct` to the **current continuation** as of (1).

The call to `foo_ct.resume()` at (3) resumes function `foo`, returning from the `resume()` call at (1) and executing the assignment at (4). Here we replace the `std::continuation` instance `caller` invalidated by the `resume()` call at (1) with the new instance returned by that same `resume()` call.

Function `std::callcc()` captures the **current continuation** and enters the given function immediately, while `resume()` returns control back to the continuation saved in `*this`.

The presented code prints out `"foo"` and `"bar"` in a endless loop.

In order to transfer data, `std::callcc()` and `resume()` accept arguments. These are stored on the stack of the captured **current continuation**. Function `data_available()` tests whether data have been passed, and with `get_data()` the data can be retrieved.

```
std::continuation lambda=
    std::callcc(  // (0)
        [](std::continuation && caller){
            int a=0;
            int b=1;
            for(;;){
                caller=caller.resume(a); // (1)
                int next=a+b;
                a=b;
                b=next;
            }
            return std::move(caller);
        });
for (int j=0;j<10;++j) {
    int i=lambda.get_data<int>(); // (2)
    std::cout << i << " ";
    lambda=lambda.resume(); // (3)
}

output:
    0 1 1 2 3 5 8 13 21 34 55
```

The invocation of `std::callcc()` at (0) immediately enters the lambda, passing no data but the **current continuation**. The lambda calculates the fibonacci number using local variables `a`, `b` and `next`. The calculated fibonacci number is transferred via `resume()` at (1). The execution control returns; `lambda` now represents the continuation of the lambda. With `get_data()` at (2) the fibonacci number is transferred to the current context while at (3) the lambda is entered again in order to compute the next fibonacci number – without passing any parameter to the lambda.

Design

std::callcc() as a factory function Every valid std::continuation instance is synthesized by the std::callcc() facility:

- as a parameter passed into the function called by std::callcc() or resume_with()
- as the value returned by std::callcc(), resume() or resume_with().

This is intentional for consistency with the *call/cc* facility in other languages.[7,8]

Footprint std::continuation contains only its **stack pointer** as a member variable. It should typically be no larger than a pointer.

Passing data Data may be passed to another context as additional arguments of std::callcc() and resume().[*]

With functions data_available() and get_data() the code can test for data and, if desired, retrieve the data.

Any additional data arguments passed to std::callcc() or resume() can be retrieved by the created/resumed context using get_data(). The template arguments for get_data() must agree with the types passed to std::callcc() or resume():

- If you call std::callcc(fn) or resume() with no additional arguments, then data_available() will return false: fn() may not call get_data() at all.
- If you call std::callcc(fn, single_arg) or resume(single_arg), then data_available() will return true, and if fn() calls get_data(), its template argument must match get_data<decltype(single_arg)>().
- If you call std::callcc(fn, multiple_args...) or resume(multiple_args...), then data_available() will return true, and if fn() calls get_data(), its template arguments must match the types of multiple_args.... Specifically, get_data<Args...>() returns std::tuple<Args...>; that std::tuple must be compatible with std::make_tuple(multiple_args...).

```
int i=1;
std::continuation lambda=
    std::callcc( // (0)
        [](std::continuation && caller){
            int j=caller.get_data<int>(); // (1)
            std::cout << "inside lambda,j==" << j << std::endl;
            caller=caller.resume(j+1); // (2)
            return std::move(caller); // (5)
        },
        i);
i=lambda.get_data<int>(); // (3)
std::cout << "i==" << i << std::endl;
lambda=lambda.resume(); // (4)

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

The callcc() call at (0) enters the lambda and passes 1 into the new context. The value is retrieved as j, as shown by (1). The expression caller.resume(j+1) at (2) resumes the original context (represented within the lambda by caller) and transfers back an integer of j+1. The assignment at (3) sets i to j+1.

The call to lambda.resume() at (4) (note that no data is passed) resumes the lambda, returning from the caller.resume(j+1) call at (2). Here, too, we replace the std::continuation instance caller invalidated by the resume() call at (2) with the new instance returned by that same resume() call.

Finally the lambda returns (the updated) caller at (5), terminating its context.

Since the updated caller represents the continuation suspended by the call at (4), control returns to main().

[*] or returned from the function invoked by resume_with(); see section Inject function into suspended context

However, since context `lambda` has now terminated, the updated `lambda` is invalid. Its `operator bool()` returns `false`; its `operator!()` returns `true`.

Multiple arguments can be transferred into another continuation too.

```
int i=1,j=2;
std::continuation lambda=
    std::callcc( // (0)
        [](std::continuation && caller){
            auto [i,j]=caller.get_data<int,int>(); // (1)
            std::cout << "inside lambda,i==" << i << ",j==" << j << std::endl;
            caller=caller.resume(i+j); // (2)
            return std::move(caller); // (5)
        },
        i,
        j);
int k=lambda.get_data<int>(); // (3)
std::cout << "k==" << k << std::endl;
lambda=lambda.resume(); // (4)

output:
    inside lambda,i==1,j==2
    k==3
```

`caller.get_data<int,int>()` returns a `std::tuple<int,int>` containing the values passed by the `std::callcc()` call at (0).

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

Calling `any_thread()` on a `std::continuation` instance representing `main()`, or the *entry-function* of a thread, will return `false`: such a `std::continuation` must *only* be resumed on its original thread.

```
int main() {
    std::continuation lambda=
        std::callcc( // (0)
            [](std::continuation && caller){ // (1)
                return std::move(caller); // (2)
            });
    return 0;
}
```

The `callcc()` call at (0) enters the lambda. The `std::continuation caller` at (1) represents the execution context of `main()`. Returning `caller` at (2) resumes the original context, switching back to `main()`.

***call/cc* and std::thread** Any context represented by a valid `std::continuation` instance is necessarily suspended.

It is valid to resume a `std::continuation` instance on any thread – *except* that since the operating system is responsible for the stack allocated for `main()`, as well as each `std::thread`, you must not attempt to resume a `std::continuation` instance representing any such context on any thread other than its own. `any_thread()` tests for this.

If, for `std::continuation c`, `c.any_thread()` returns `false`, it is only valid to resume `c` on the thread on which it was initially launched.

Termination There are a few different ways to terminate a given context without terminating the whole process, or engaging undefined behavior.

- Return a valid continuation from the *entry-function*.
- Call `std::unwind_context()` with a valid continuation. This throws a `std::unwind_exception` instance that binds that continuation.

- **[LEWG: Should we publish the `std::unwind_exception` constructor that accepts `std::continuation`? Then another supported way would be to construct and throw `std::unwind_exception` "by hand," which is what `std::unwind_context()` does internally.]**
- Call `std::continuation::resume_with(std::unwind_context)`. This is what `~continuation()` does. Since `std::unwind_context()` accepts a `std::continuation`, and since `resume_with()` synthesizes a `std::continuation` and passes it to the subject function, this terminates the context referenced by the original `std::continuation` instance and switches back to the caller.
- Engage `~continuation()`: switch to some other context, which will receive a `std::continuation` instance representing the current context. Make that other context destroy the received `std::continuation` instance.

When the *entry-function* invoked by `std::callcc()` returns a valid `std::continuation` instance, the running context is terminated. Control switches to the context indicated by the returned `std::continuation` instance.

Returning an invalid `std::continuation` instance (`operator bool()` returns `false`) invokes undefined behavior.

If the *entry-function* returns the same `std::continuation` instance it was originally passed (or rather, the most recently updated `std::continuation` returned from `std::callcc()` or the previous instance's `resume()`), control returns to the context that most recently resumed the running context. However, the *entry-function* may return (switch to) any reachable valid `std::continuation` instance.

Calling `std::continuation::resume()` means: "Please switch to the indicated context; I am suspending; please resume me later."

Returning a particular `std::continuation` means: "Please switch to the indicated context; and by the way, I am done."

Exceptions In general, if an uncaught exception escapes from the *entry-function*, `std::terminate` is called. There is one exception: `std::unwind_exception`. The `std::callcc()` facility internally uses `std::unwind_exception` to clean up the stack of a suspended context being destroyed. This exception must be allowed to propagate out of an *entry-function*.

A correct *entry-function* `try / catch` block looks like this:

```
try {
    // ... body of context logic ...
} catch (std::unwind_exception const&) {
    // do not swallow unwind_exception
    throw;
} catch (...) {
    // ... log, or whatever ...
}
```

Of course, if you do not expect the *entry-function* or anything it calls to throw exceptions, you need no `try / catch` block.

If a `resume_with()` function throws an exception that you expect to catch in the context's *entry-function*, it is good practice to bind into the exception object the continuation passed to the `resume_with()` function so that the *entry-function*'s `catch` clause can return that continuation.

Inject function into suspended context Sometimes it is useful to inject a new function (for instance, to throw an exception) into a suspended context. For this purpose you may call `resume_with(Fn && fn,Args ... args)`, passing:

- the function `fn()` to execute
- additional data `args` to be retrieved by `fn()`.

Let's say that the context represented by the `std::continuation` instance `ctx` has suspended in a function `suspender()`. You intend to inject function `fn()` into context `ctx` as if `suspender()` had directly called `fn()` at the point where it suspended.

Like an *entry-function* passed to `std::callcc()`, `fn()` must accept `std::continuation&&`. However, instead of returning `std::continuation`, `fn()` must return a type corresponding to the `args` passed to the `resume_with()` call.

- If you call `ctx.resume_with(fn)` with no additional `args`, the return type of `fn()` is irrelevant: its return value is discarded.
- If you call `ctx.resume_with(fn, single_arg)`, then `fn()` must return `decltype(single_arg)`.
- If you call `ctx.resume_with(fn, multiple_args...)`, then `fn()` must return a `std::tuple` with corresponding types: specifically, `decltype(std::make_tuple(multiple_args...))`.

Any `args...` passed to `resume_with()` can be retrieved within `fn()` using `get_data()`. As with `std::callcc()` and `resume()`, the template arguments for `get_data()` must agree with the types passed to `resume_with()`:

- If you call `ctx.resume_with(fn)` with no additional `args`, then `data_available()` will return `false`: `fn()` may not call `get_data()` at all.
- If you call `ctx.resume_with(fn, single_arg)`, then `data_available()` will return `true`, and if `fn()` calls `get_data()`, its template argument must match `get_data<decltype(single_arg)>()`.
- If you call `ctx.resume_with(fn, multiple_args...)`, then `data_available()` will return `true`, and if `fn()` calls `get_data()`, its template arguments must match the types of `multiple_args...`. Specifically, `get_data<Args...>()` returns `std::tuple<Args...>`; that `std::tuple` must be compatible with `std::make_tuple(multiple_args...)`.

The above describes retrieving passed values *within* `fn()`. The value returned by `fn()` becomes available to `suspender()`. It is up to `fn()` whether to return the same value(s) it retrieved from the `resume_with()` call.

- If you call `ctx.resume_with(fn)` with no additional `args`, then `data_available()` will return `false` even within `suspender()`: `suspender()` may not call `get_data()` at all.
- If you call `ctx.resume_with(fn, args...)` – with one or more `args...` – then `data_available()` will return `true` within `suspender()`. `get_data()` within `suspender()` will retrieve the value returned by `fn()`.

Note that if you pass one or more `args...` to `resume_with()`, those `args...` *must* be compatible with the return type of `fn()`. However, it is permissible to call `resume_with(fn)` sometimes (`data_available()` returns `false`) but pass compatible data in other `resume_with()` calls.

Suppose that code running on the program's main context calls `std::callcc(f)`, thereby entering `f()`. This is the point at which `mc` is synthesized and passed into `f()`, as illustrated below.

Suppose further that after doing some work, `f()` calls `mc.resume()`, thereby switching back to the main context. `f()` remains suspended in the call to `mc.resume()`.

At this point the main context calls `f_ct.resume_with(g)` where `g()` is declared as `int g(continuation &&)`. `g()` is entered in the context of `f()`. It is as if `f()`'s call to `mc.resume()` directly called `g()`.

Function `g()` has almost the same range of possibilities as any function called on `f()`'s context. Its special invocation 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 provides data for `f()`'s suspended `mc.resume()` call.

```
std::continuation f(std::continuation && mc) {
    int data=mc.get_data<int>(); // (1)
    std::cout << "f: entered first time: " << data << std::endl;
    mc = // (5)
       mc.resume(data+1); // (2)
    data=mc.get_data<int>();
    std::cout << "f: entered second time: " << data << std::endl;
    mc = // (10)
       mc.resume(data+1); // (6)
    data=mc.get_data<int>(); // (11)
    std::cout << "f: entered third time: " << data << std::endl;
    return std::move(mc); // (12)
}

int g(std::continuation && mc) {
```

[*] As stated in Exceptions, if there is no matching `catch` clause in that context, `std::terminate()` is called.

```
    int data=mc.get_data<int>();
    std::cout << "g: entered: " << data << std::endl;
    return -1; // (9)
}

int data=1;
std::continuation f_ct= // (3)
    std::callcc(f,data); // (0)
data=f_ct.get_data<int>();
std::cout << "f: returned first time: " << data << std::endl;
f_ct = // (7)
    f_ct.resume(data+1); // (4)
data=f_ct.get_data<int>();
std::cout << "f: returned second time: " << data << std::endl;
f_ct = // (13)
    f_ct.resume_with(g,data+1); // (8)
data=f_ct.get_data<int>();
std::cout << "f: returned third time: " << data << std::endl;

output:
    f: entered first time: 1
    f: returned first time: 2
    f: entered second time: 3
    f: returned second time: 4
    g: entered: 5
    f: entered third time: -1
```

Control passes from (0) to (1) to (2), and so on.

The `f_ct.resume_with(g, data+1)` call at (8) passes control to `g()` on the context of `f()`.

The `return` statement at (9) causes the `resume()` call at (6) to return, executing the assignment at (10). The `int` returned by `g()` is accessed at (11).

Finally, `f()` returns its own `mc` variable, switching back to the main context.

There is one restriction on a function `fn()` passed to `resume_with()`: neither `fn()`, nor any function it calls, may perform a context switch back to the context that called `resume_with()` – whether directly, or indirectly via other contexts. `fn()` *must* return (or throw an exception) before `resume_with()` returns.

`std::callcc()` immediately enters new context `std::callcc()` creates a new context and immediately calls its passed *entry-function* on that new context.

This is intentional for consistency with the *call/cc* facility in other languages.[7,8]

Moreover, this behavior prevents a problematic usage. Suppose we had a `callcc_deferred()` which would create a new context but immediately return to its caller. The newly-created context would first be entered by calling `resume()` on the returned `std::continuation` instance.

```
    std::continuation newcontext = std::callcc_deferred(entry_function);
    newcontext = newcontext.resume();
```

But now consider this scenario:

```
    std::continuation newcontext = std::callcc_deferred(entry_function);
    newcontext = newcontext.resume_with(injected_function);
```

What should happen here?

`resume_with()` is supposed to call `injected_function()` as if it had been directly called by `entry_function()` – rather, by the context-switch operation most recently executed by `entry_function()`. But since `entry_function()` has never yet been entered, it hasn't executed any context-switch operation. Indeed, it does not yet have a stack frame.

Should `injected_function()` *become* the *entry-function* for `newcontext`, displacing `entry_function()` entirely?

That would encounter signature problems. The *entry-function* invoked by `std::callcc()` *must* return a `std::continuation` . Yet the `fn` passed to `resume_with()` returns – not a `std::continuation` – but arbitrary data to be retrieved by the suspended context!

To quickly resume the caller's context rather than prioritizing the new context, the *entry-function* passed to `std::callcc()` can immediately context-switch back to its caller by calling `resume()` on its passed-in `std::continuation`:

```
std::continuation entry_function(std::continuation&& caller) {
    caller = caller.resume();
    // ...
}
```

A more generic wrapper for that behavior could look something like this:

```
template <typename Fn>
std::continuation suspend_immediately(std::continuation&& caller) {
    Fn fn = caller.get_data<Fn>();
    return fn(caller.resume());
}

template <typename Fn>
std::continuation callcc_deferred(Fn&& fn) {
    return std::callcc(suspend_immediately<Fn>, std::forward<Fn>(fn));
}
```

Note that since `suspend_immediately()` *has* been entered, it is perfectly valid for the caller of `callcc_deferred()` to call `resume_with()` on the returned `std::continuation`.

Stack destruction On construction of a context with `std::callcc()` a stack is allocated. If the *entry-function* returns, the stack will be destroyed. If the function has not yet returned and the (destructor) of the `std::continuation` instance representing that context is called, the stack will be unwound and destroyed.

For this purpose member-function `resume_with()` is called with `std::unwind_context()` as argument. The execution context will be temporarily resumed and `std::unwind_context()` is invoked. Function `std::unwind_context()` throws exception `std::unwind_exception`.[*] The exception is caught by the first frame on the stack: the one created by `std::callcc()`. Control is switched back to the context that called `~continuation()` and the stack gets deallocated.

The StackAllocator's deallocate operation is called on the context that invoked `~continuation()`.

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::continuation` (`any_thread()` returns `false`), 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::callcc()`, the StackAllocator concept is an implementation detail, used only by the internal mechanisms of the *call/cc* 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.

[*] `std::unwind_exception` binds an instance of `std::continuation` that represents the continuation that called `resume_with()`.

- `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[9] with the specified initial size, which grows on demand.[*]

It is expected that the StackAllocator's allocation operation will run in the context of the `std::callcc()` call (before control is passed to the new context), and that the StackAllocator's deallocation operation will run in the context of the `~continuation()` call (after control returns from the destroyed context). No special constraints need apply to either operation.

Performance of *call/cc*

On modern architectures suspending/resuming continuations takes very few CPU cycles. [†]

[*] An implementation of the `segmented` StackAllocator necessarily interacts with the C++ runtime. For instance, with gcc, the Boost.Context[12] library invokes the `__splitstack_makecontext()` and `__splitstack_releasecontext()` intrinsic functions.[10],[11]

[†] `callcc()` from boost.context takes 18 CPU cycles on Intel E5 2620 v4, SYS V.

Use case: userland threads

`std::callcc()` can be used to implement userland threads. A userland thread resembles a `std::thread` in that it is launched and proceeds more or less independently. Its lifespan is not tied to the code that launched it.

The operating system kernel schedules `std::thread` s. When there are more threads on the system than processor cores (which is frequently the case), it gives every such thread a time slice, preemptively and transparently suspending it once it has consumed that time slice.

In contrast, a userland thread does not create additional work for the kernel scheduler. As far as the kernel is concerned, that userland thread is simply the work the kernel thread chooses to do at this moment. When context switches from one userland thread to another, the kernel is not engaged.

For present purposes, we will use the term **fiber** to mean "userland thread."

A fiber suspends by calling its own (userland) scheduler. The scheduler decides which of the other fibers are ready to run, picks one and resumes it. Typically a suspending fiber is waiting *for* something: a timer, asynchronous I/O, work performed by another fiber or another thread. The suspending fiber first arranges to be notified when the wait is over. That notification informs the scheduler that this fiber is once again ready to run.

Because each fiber has a C++ stack, the details of arranging notification and suspending can be encapsulated in a lower-level function. Consuming code calls that function, which returns with expected results once they become available. The fact that the lower-level function suspended, allowing other fibers to run in the meantime, is transparent to the caller.

In effect, fibers provide a way to organize code running within a kernel thread. Fibers provide no more parallelism than setting up a chain of callbacks, which is the conventional way to orchestrate a sequence of asynchronous operations. But code running on a fiber looks and behaves as if the underlying operations were classic blocking operations.[*]

Fibers permit structuring your code into layers of abstraction, even when the underlying I/O is asynchronous. Chains of callbacks do not permit that.

Fibers permit ordinary use of stack variables. Chains of callbacks do not permit that.

C++ code based on asynchronous operations can be (and has been) written in a startling variety of different ways. But when such code is written using fibers, it is both easier to write and easier to reason about – thus easier to maintain, and therefore more robust.

Asynchronous I/O is becoming increasingly pervasive in modern computer systems. It's clear that we need some better tactic than chains of callbacks, state machines, big switch statements, Duff's Device, etc. etc.

Fibers allow us to use the native control structures provided by C++ itself to organize and maintain our code.

The Boost.Fiber library[15] is an implementation of fibers based on the `callcc()` implementation in Boost.Context.[12]

Why not use `std::thread` s? There are a few reasons to prefer userland threads over `std::thread` s:

Performance: The Skynet benchmark results[16] illustrate that userland context-switching can be three orders of magnitude faster than kernel-mediated context-switching.

Scalability: You can productively run many more userland threads in a single process than you could run kernel threads. The Skynet benchmark[16] tests a million concurrent tasks. It is not reasonable (even when possible) to launch that many kernel threads. Kernel-mediated context-switching overhead starts to overwhelm the processor.

Thread safety: Refactoring an existing chain of callbacks into userland threads is guaranteed to introduce no new data races.

Why not use `co_await`[4]? The `co_await` facility is best suited for new code. The caller of a `co_await` function must itself be a `co_await` function, and so on all the way up to the launch point.

If you are modifying existing code to use `co_await`, you quickly find that introducing a new `co_await` operation into a given function requires transitively modifying every function that directly calls it, then also modifying every function that

[*] In a sense, they are – but they block only the calling *fiber,* not the entire host *thread.*

calls any of *those...*

Modification for co_await requires more than sprinkling co_await operations throughout your code base. It also typically requires altering the signature of every affected function. A function invoked by co_await must be able to communicate to the co_await operation whether it is suspending or returning with a result. This is often communicated by using a return type such as std::future, which can express the absence of data and pass control back to the co_await operation once data become available.

Of course, as has been pointed out,[3] std::future introduces overhead of its own.

It is not usually emphasized that each call to a co_await function implies a malloc() call to obtain a heap activation frame; each return implies a corresponding free() call. Under certain circumstances – specifically, the case of a function-local coroutine – that overhead can be optimized away. When co_await is used to emulate userland threads, it cannot.

One might consider the use of a memory pool for activation frames. This is an excellent idea. A memory pool for activation frames is called a "stack."

Userland threads built on callcc()

- Each fiber is reified as an object. That object contains the continuation representing its suspended context.
- The object representing the currently-running fiber contains an invalid continuation.
- The function that launches a fiber creates its context using callcc().
- Fiber objects are known to a central fiber manager object.
- The fiber manager keeps fiber objects in separate containers: those still waiting for something versus those that are ready to resume.
- Instead of directly resuming a specific other fiber, the running fiber suspends by calling a scheduler to pick one of the ready fibers. (Note that the scheduler can execute on the context of the fiber about to suspend; we do not need two separate context switches.)

Why not propose userland threads instead?

Consider the following bullet from P0559R0:[6]

- "Prefer generality over specificity: prefer standardizing general building blocks on top of which domain-specific semantics can be layered, as opposed to domain-specific facilities on top of which other domain-specific semantics can't be layered."

The std::callcc() facility proposed in this document is very low-level and very general. With a public implementation of this facility,[12] the author has built high-performance stackful coroutines[13] and high-performance userland threads.[15]

Both libraries, it should be noted, are built in portable C++ on top of the callcc() and continuation API. The callcc()-based implementation gives the best performance yet[16] for each of these libraries.

The API permits still other higher-level abstractions too. The author has also prototyped an implementation of delimited continuations (*shift* and *reset* operators).

Use case: stackful coroutines

The Boost.Coroutine2 library[13] is an implementation of stackful coroutines based on the `callcc()` implementation in Boost.Context.[12]

With this API, a `pull_type` asymmetric coroutine functions as a generator: it is resumed each time its invoker requests the next data item. The library provides input iterator support for `pull_type` coroutines; such a generator may provide data to any range operation compatible with input iterators, for example range `for`.

Similarly, a `push_type` asymmetric coroutine is a data sink. It is resumed each time its invoker passes the next data item. The library provides output iterator support for `push_type` coroutines.

The API permits chaining[14] of coroutines. Indeed, a function that accepts both `pull_type` and `push_type` coroutine endpoints can be chained without having to care whether it's in an "input chain" or an "output chain."

Invoking code instantiates a coroutine, then engages with it either by requesting data items or passing data items to it.

Why not use `co_await`?[4] This kind of use case is actually where `co_await` shines. Significant effort has already been invested in compiler optimizations that can make a small local `co_await` coroutine "vanish."

Even so, there remain significant cases that cannot readily be handled by `co_await`. Generally speaking, there are times when it's very useful to run your processing on a whole separate stack.

A few examples of such cases:

Recursive SAX parsing There are two common ways to parse an XML document.

With **DOM** parsing, you read the entire document into memory as a linked data structure. You may then traverse that structure any way you want, including recursively. Of course, the caveat is that you must have enough free memory to represent the entire document.

With **SAX** parsing, the document is streamed. Application code is notified (via callbacks) when the start and end of each element is encountered. The benefit is that this can handle XML documents of arbitrary size: you need not store the entire document in memory at the same time. The drawback is that you must manually track element nesting.

A C++Now 2014 talk[21] illustrates how stackful coroutines permit you to use an off-the-shelf SAX parser to structure your processing code as recursion over an XML document of arbitrary size being streamed through your application.

Essentially, it permits passing callback events from the SAX parser into recursive processing code. Having two different stacks allows the SAX parser to call application callbacks, which then return to the parser to continue, while the application performs recursive processing on the other stack.

Attempting to use `co_await` for this would require rewriting the SAX parser.

Lazy visitor-based processing Even when you do have an entire data structure in memory, a particular library might use callbacks – or visitors – to engage application-specific processing.

The Boost.Graph library[17] provides a number of useful graph traversals and processing algorithms. Its API relies heavily on visitors.[18]

The problem with that design is that the Boost.Graph algorithm drives the traversal. The visitor is called for every applicable node or edge on the graph. What if you want to stop? What if you want to pause? What if you want to proceed stepwise?

For example, consider the problem of finding a route from node A to node B in a densely-connected graph. You might consider taking a step outward from node A, taking a step outward from node B, taking another from node A, and so forth, until the two traversals reach the same node.

This is unreasonably difficult using visitors.

A C++Now 2016 talk[22] illustrates how stackful coroutines permit you to run two concurrent Boost.Graph traversal algorithms on separate stacks, "pulling" visitor events from each traversal on demand, interleaving calls to the two traversals.

Attempting to use `co_await` for this would require rewriting the Boost.Graph algorithm of interest.

Connecting output iterator to input iterator In February 2017, a question was asked on the Boost developers' mailing list:[23] Can a Boost.Spirit[19] Karma[20] generator, which takes an output iterator, be engaged lazily? Can consumer code request one item at a time?

The answer is straightforward: run the Karma generator in a Boost.Coroutine `pull_type` asymmetric coroutine. Such a coroutine receives, as a parameter, a synthesized `push_type` coroutine endpoint representing its invoker.

Boost.Coroutine provides an output iterator façade for a `push_type` coroutine endpoint. This output iterator is passed to the Karma generator.

The input iterator corresponding to the `pull_type` coroutine can then be dereferenced as desired. Every time a new item is requested, the Karma generator is resumed to produce it.

Attempting to use `co_await` for this would require rewriting the Boost.Spirit Karma library.

Stackful coroutines built on `callcc()`

- Symmetric coroutines map very directly to `callcc()` functionality. Each coroutine is reified as an object. The object contains the `continuation` representing its suspended context – or, if that coroutine is currently running, an invalid `continuation`.
- A symmetric coroutine suspends by specifying a particular other coroutine object to resume. The implementation calls `resume()` on that other coroutine object's `continuation`.
- An asymmetric coroutine "knows" its invoker: rather than explicitly resuming an arbitrary other coroutine, it *yields,* implicitly resuming its invoker. (In just the same way, `return` implicitly resumes a function's caller.)
- An asymmetric coroutine object could contain a reference to its invoker's coroutine object, permitting an anonymous *yield* operation.

Why not propose stackful coroutines instead?

In fact – we *did!*[1,2] We were directed to bring back a lower-level proposal. That lower-level proposal has evolved to this present form.

Use case: many small stacks, one deep stack

Proponents of `co_await` frequently describe a particular execution environment: a 32-bit Windows server process supporting millions of clients in a transiently-stateful way, preserving some amount of state data across some number of asynchronous operations.

It is pointed out that calls to opaque library, runtime and operating-system functions may consume arbitrary amounts of stack space. The inability to predict stack consumption in advance leads the Windows operating system to allocate a 1MB stack for each kernel thread.

Of course, that stack memory is not committed until actually used. Still, it does present a problem: in a 32-bit process, you quickly run out of address space. You are constrained to no more than

$$\frac{2^{32} - (\text{size of all code}) - (\text{size of all other data})}{(\text{stack size})}$$

stacks. Even if you set both `(size of all code)` and `(size of all other data)` to zero – impossible, in practice – you can allocate no more than 4096 1MB stacks. 4096 is very much smaller than "millions."

On another operating system, one could use segmented stacks, but those are not supported on Windows.

In a 64-bit process, the limitation would be actual memory consumption rather than the address space. But it is suggested that many people still use 32-bit server processes.

When considering userland threads, we might not be quite as conservative as the Windows operating system. We might decide that we need far less stack space than 1MB per userland thread. We might be so bold as to suggest 16KB stacks. But that *still* is constrained to a theoretical maximum of 262144 stacks – and that's without code or any other data. This falls short of "millions" by at least an order of magnitude.

We might be certain that our own code requires very little stack space – even less than 16KB. But does our code ever call library functions? runtime functions? operating-system functions? How much stack space do *they* consume? This brings us back to the original unanswerable question.

Proponents of the `co_await` facility explain that since each `co_await` function allocates a separate heap activation frame – in effect, each has its own tiny stack – the thread's main stack is left largely untouched. Calls to opaque library or runtime or operating-system functions that require arbitrary stack space consume the thread's main stack, which is presumed to be Big Enough.

With *call/cc*, we can use a similar trick. We can construct each new context with a very small stack: just big enough for the function calls in our own code. We can set aside one "big enough" stack as a common resource, shunting function calls of unknown depth onto the shared "big enough" stack.

We assume that opaque functions of this kind will not themselves suspend. Please note that the `co_await` scenario requires the same assumption.

```
// launch this context early in the thread as a service used by deep_call()
std::continuation deep_stack(std::continuation&& client) {
    for (;;) {
        // In effect, this context is a little server, running a function that
        // needs a deep stack and then context-switching back to whatever
        // invoked it. Because of this infinite loop, it won't terminate
        // voluntarily: we'll destroy it while it's suspended.
        client = client.resume();
        // if we were passed a std::function object
        if (client.data_available()) {
            // retrieve it
            std::function<void()> func =
                client.get_data<std::function<void()>>();
            // and call it
            func();
        }
    }
    return std::move(client);
}

// deep_call() continually updates this continuation as the way to run code in
// the context of deep_stack()
std::continuation deep_stack_cont;

// this function can be called from a context with a shallow stack to run some
// non-suspending function in the context of deep_stack()
void deep_call(std::function<void()> const& func) {
    deep_stack_cont = deep_stack_cont.resume(func);
}

// this is an example of a context with a shallow stack
std::continuation shallow_stack(std::continuation&& caller) {
    // pretend that emitting std::cout requires arbitrary stack depth
    deep_call([](){ std::cout << "Hello from deep_stack!" << std::endl; });
    deep_call([](){ std::cout << "Another visit to deep_stack" << std::endl; });
    return std::move(caller);
}

// main() simply launches the deep_stack() resource context, then runs
// shallow_stack() as an example of a consumer
int main() {
    // Launch the deep_stack() context. Pretend we're passing a StackAllocator
    // with a size "big enough for whatever." Since the first thing
    // deep_stack() does is to switch back to its invoker, this call should
    // bounce right back here.
    deep_stack_cont = std::callcc(deep_stack);
    // pretend we're passing a StackAllocator with a very small stack size
    std::callcc(shallow_stack);
    return 0;
}
```

(With `std::callcc()` replaced by `boost::context::callcc()` and `std::continuation` replaced by `boost::context::continuation`, and with lavish `std::cout` annotation, the program above compiles and runs as expected.)

Why *call/cc* should be preferred over *ucontext* or *longjmp*

stack represents the continuation In contrast to *ucontext*, *call/cc* uses the stack as storage for the suspended execution context (the content of the registers).

- only the target has to be provided at resumption (`swapcontext()` required source and target)
- current execution context is already represented by the stack to which the stack-pointer points
- suspended execution context is passed as continuation (parameter) to the resume operation
- no need for a global pointer that points to the current execution context
- data are transferred through a function call, no need for a global pointer
- `main()` and each thread's *entry-function* integrate seamlessly with *call/cc* because the stack of `main()` , or the thread, already represents the continuation of that context

aggregation of stack address A instance of `std::continuation` contains the stack address of a suspended execution. `std::continuation`:

- represents the continuation of a suspended context
- prevents accidentally copying the stack
- prevents accidentally resuming a context that was previously resumed
- prevents accidentally resuming the running execution context
- prevents accidentally resuming an execution context that has already terminated (computation has finished)
- manages lifespan of an explicitly-allocated stack: the stack gets deallocated when `std::continuation` goes out of scope
- context switch and data transfer via one function call

Of course a *ucontext*-like standard API would be possible, but in C++ we can do much better with very little abstraction cost.

Disadvantages of *ucontext*

- deprecated since POSIX.1-2004d and removed in POSIX.1-2008
- `makecontext` violates C99 standard (function pointer cast and integer arguments)
- `makecontext` arguments in var-arg list are required to be integers; passing pointers is not guaranteed to work (especially on platforms where pointers are larger than integers)
- `swapcontext` calls into the kernel, consuming many CPU cycles (two orders of magnitude)
- does not prevent accidentally copying the stack
- does not prevent accidentally resuming the running execution context
- does not prevent accidentally resuming an execution context that has already terminated (computation has finished)
- does not manage lifespan of an explicitly-allocated stack
- context switch (`swapcontext`) does not transfer data (requires global pointer)

Disadvantages of *longjmp*

- C99 defines *setjmp*/ *longjmp* for non-local jumps
- *longjmp* is not required to preserve the current stack frame, therefore jumping into a function which has exited (by return or by a different *longjmp* higher up the stack) is undefined behavior: only *longjmp* up the call stack is allowed
- does not prevent accidentally copying the stack
- does not prevent accidentally resuming the running execution context
- does not prevent accidentally resuming an execution context that has already terminated (computation has finished)
- does not manage lifespan of an explicitly-allocated stack
- context switch (*longjmp*) does not transfer data (requires global pointer)

How low-level is `std::callcc()`?

call/cc is a low-level implementation. `std::continuation` has the memory footprint of a pointer because it contains only a pointer to the stack of the managed context.

The assembly code generated for a typical function consists of a *function prologue* at the beginning and *function epilogue* at the end.

```
void foo() {
    PROLOGUE
    <computation, calling other sub-routines, etc.>
    EPILOGUE
}
```

The *prologue* (a few lines of assembler) prepares the stack and registers for use inside the function. The *epilogue* restores the stack and registers[*] to the state they were before the function was called.
Between prologue and epilogue the computation and calls to other functions are done.

Like ordinary functions, `resume()` and `resume_with()` contain prologue and epilogue. The only difference from ordinary functions is that `resume()` and `resume_with()` additionally exchange the stack- and instruction pointer [†] between prologue and epilogue.
The prologue and epilogue of `resume()` and `resume_with()` neither consume stack space nor do they call functions; only the stack-pointer is exchanged.

```
class continuation {
private:
    void    *   vp;

    explicit continuation( void * ptr) :
        vp{ ptr } {
    }

public:
    ~continuation() {
        if ( nullptr != vp) {
            // resume context by calling resume_with(),
            // passing std::unwind_context(), which throws
            // std::unwind_exception.
            // this exception is caught in the first stack-frame,
            // i.e. the stack-frame originally prepared by callcc().
            // from that stack frame switch back to the context that
            // invoked ~continuation().
            // deallocate the stack.
        }
    }

    continuation resume() {
        // PROLOGUE (+ store instruction-pointer)
        // exchange current stack with 'vp' (becomes current stack)
        // EPILOGUE (+ restore instruction-pointer)
        // stack-pointer of caller passed as std::continuation
    }

    template< typename Fn >
    continuation resume_with( Fn && fn) {
        // PROLOGUE (+ store instruction-pointer)
        // exchange current stack with 'vp' (becomes current stack)
        // EPILOGUE (+ restore instruction-pointer)
        // 'fn' is loaded into instruction-pointer: 'fn()' is executed by
        // the resumed execution context
```

[*] callee-saved registers as defined by the calling convention
[†] In fact on x86-architecture the instruction-pointer return-address is already stored on the stack. On some RISC-architectures the link-register must be preserved on the stack while the context is suspended, and is loaded into the instruction-pointer on resumption.

```
        // stack-pointer of caller passed as std::continuation to `fn()`
    }
};

template< typename StackAlloc, typename Fn >
continuation callcc( std::allocator_arg_t, StackAlloc salloc, Fn && fn) {
    // allocate memory for stack `vp`
    // prepare stack (generate special stack-frame)
    // construct std::continuation from stack `vp` and resume it
    return continuation{ vp }.resume();
}
```

API

std::continuation   declaration of class `std::continuation`

```
class continuation {
public:
    continuation() noexcept;
    ~continuation();
    continuation( continuation && other) noexcept;
    continuation & operator=( continuation && other) noexcept;
    continuation( continuation const& other) noexcept = delete;
    continuation & operator=( continuation const& other) noexcept = delete;

    template< typename ... Arg >
    continuation resume( Arg ... arg);
    template< typename Fn, typename ... Arg >
    continuation resume_with( Fn && fn, Arg ... arg);

    bool data_available() const noexcept;
    template< typename ... Arg >
    <unspecified> get_data();

    bool any_thread() const noexcept;

    explicit operator bool() const noexcept;
    bool operator!() const noexcept;
    bool operator==( continuation const& other) const noexcept;
    bool operator!=( continuation const& other) const noexcept;
    bool operator<( continuation const& other) const noexcept;
    bool operator>( continuation const& other) const noexcept;
    bool operator<=( continuation const& other) const noexcept;
    bool operator>=( continuation const& other) const noexcept;
    void swap( continuation & other) noexcept;
};

template< typename Fn, typename ...Arg >
continuation callcc( Fn &&, Arg ...);

template< typename StackAlloc, typename Fn, typename ...Arg >
continuation callcc( std::allocator_arg_t, StackAlloc, Fn &&, Arg ...);

void unwind_context( continuation && cont);

struct unwind_exception{};
```

member functions

(constructor)   constructs new continuation

`continuation() noexcept`	(1)
`continuation(continuation&& other)`	(2)
`continuation(const continuation& other)=delete`	(3)

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

2) moves underlying state to new `std::continuation`

3) copy constructor deleted

Notes

Every valid `std::continuation` instance is synthesized by the underlying facility – or move-constructed, or move-assigned, from another valid instance. There is no public `std::continuation` constructor that directly constructs a valid `std::continuation` instance.

The entry-function `fn` passed to `std::callcc()` is passed a synthesized `std::continuation` instance representing the suspended caller of `std::callcc()`.

The function `fn` passed to `resume_with()` is passed a synthesized `std::continuation` instance representing the suspended caller of `resume_with()`.

`std::callcc()` returns a synthesized `std::continuation` representing the previously-executing context, the context that suspended in order to resume the caller of `std::callcc()`. The returned `std::continuation` instance *might* represent the context created by `std::callcc()`, but need not: the context created by `std::callcc()` might have created (or resumed) yet another context, which might then have resumed the caller of `std::callcc()`.

Similarly, `resume()` returns a synthesized `std::continuation` instance representing the previously-executing context, the context that suspended in order to resume the caller of `resume()` .

Similarly, `resume_with()` returns a synthesized `std::continuation` instance representing the previously-executing context, the context that suspended in order to resume the caller of `resume_with()`.

(destructor) destroys a continuation

`~continuation()`	(1)

1) destroys a `std::continuation` 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 by throwing `std::unwind_exception`.[*]

operator= moves the continuation object

`continuation& operator=(continuation&& other)`	(1)
`continuation& operator=(const continuation& 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**

resume() resumes a continuation

`template< typename ...Args >`
`continuation resume( Args ... args)`
	(1)

`template< typename Fn, typename ...Args >`
`continuation resume_with( Fn && fn, Args ... args)`
	(2)

1) suspends the active context, resumes continuation `*this`

2) suspends the active context, resumes continuation `*this` but calls `fn()` in the resumed context (as if called by the suspended function)

Parameters

...args passed to the resumed continuation - see section Passing data

fn function invoked ontop of resumed continuation

[*] In a program in which exceptions are thrown, it is prudent to code a context's *entry-function* with a last-ditch `catch (...)` clause: in general, exceptions must *not* leak out of the *entry-function*. However, since stack unwinding is implemented by throwing an exception, a correct *entry-function* `try` statement must also `catch (std::unwind_exception const&)` and rethrow it.

Return value

continuation the returned instance represents the execution context (continuation) that has been suspended in order to resume the current context

Exceptions

1) `resume()` or `resume_with()` might throw `std::unwind_exception` if, while suspended, the calling context is destroyed

2) `resume()` or `resume_with()` might throw *any* exception if, while suspended:

- some other context calls `resume_with()` to resume this suspended context
- the function `fn` passed to `resume_with()` – or some function called by `fn` – throws an exception

3) Any exception thrown by the function `fn` passed to `resume_with()`, or any function called by `fn`, is thrown in the context referenced by `*this` rather than in the context of the caller of `resume_with()`.

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
`resume()` preserves the execution context of the calling context as well as stack parts like *parameter list* and *return address*.[*]
Those data are restored if the calling context is resumed.
A suspended `continuation` can be destroyed. Its resources will be cleaned up at that time.
The returned `continuation` indicates whether the suspended context has terminated (returned from *entry-function*) via `operator bool()`. If the returned `continuation` has terminated, no data may be retrieved.
Because `resume()` invalidates the instance on which it is called, *no valid `std::continuation` instance ever represents the currently-running context.*
When calling `resume()`, it is conventional to replace the newly-invalidated instance – the instance on which `resume()` was called – with the new instance returned by that `resume()` call. This helps to avoid inadvertent calls to `resume()` on the old, invalidated instance.

data_available() test if data are present

```
bool data_available()     (1)
```

1) returns `true` if `std::callcc()` or `resume()` have been invoked with additional data as argument (`args`)

get_data() transfer of data

```
template< typename Arg >
Arg get_data()                        (1)
```
```
template< typename ...Args >
std::tuple< Args... > get_data()      (2)
```

1) transfers single datum from continuation `*this` into calling context

2) transfers multiple data from continuation `*this` into calling context

Notes
The template argument(s) passed to `get_data()` must match in number and type the actual argument types passed to `std::callcc()` or `resume()`.

any_thread() test whether suspended continuation 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

[*] required only by some x86 ABIs

Notes
As stated in main() and thread functions, a `std::continuation` 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`.
Attempting to resume such a `std::continuation` instance on any thread other than its original thread invokes undefined behavior. `any_thread()` allows consumer code to distinguish this case by returning `false`.

operator bool   test whether continuation is valid

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

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

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

- It might have been default-constructed.
- It might have been assigned to another instance, or passed into a function. `std::continuation` instances are move-only.
- It might already have been resumed – calling `resume()` invalidates the instance.
- The *entry-function* might have voluntarily terminated the context by returning.

The essential points:

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

operator!   test whether continuation 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()`.

(comparisons)   establish an arbitrary total ordering for `std::continuation` instances

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

1) Every invalid `std::continuation` instance compares equal to every other invalid instance. But because the running context is never represented by a valid `std::continuation` 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::continuation` 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::continuation` instances

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

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

std::callcc() create and enter a new context, capturing the current execution context (the **current continuation**) in a `std::continuation` and passing it to the specified *entry-function*.
`std::callcc()` acts as a factory-function: it creates and starts a new execution context (stack etc.) and returns a continuation that represents the rest of the execution context's computation.
`std::callcc()` explicitly expresses the creation of a new execution context and the switch to the other execution path.

```
template< typename Fn, typename ...Args >
continuation callcc( Fn && fn, Args ...args)                                          (1)
template< typename StackAlloc, typename Fn, typename ...Args >
continuation callcc( std::allocator_arg_t, StackAlloc salloc, Fn && fn, Args ...args)   (2)
```

1) creates and immediately enters the new execution context (executing `fn`). The current execution context is suspended, wrapped in a continuation (`std::continuation`) and passed as argument to `fn`.

2) takes a callable as argument, requirements as for (1). The stack is constructed using *salloc* (see Stack allocators).

Parameters

fn callable (function, lambda, functor) executed in the new context; expected signature `continuation(continuation &&)`

...args data transferred to the new context - see section Passing data

Return value

continuation the returned instance represents the execution context (continuation) that was suspended in order to resume the current context

Exceptions

1) calls `std::terminate` if an exception other than `std::unwind_exception` escapes *entry-function* `fn`

2) `std::callcc()` might throw `std::unwind_exception` if, while suspended, the calling context is destroyed

3) `std::callcc()` might throw *any* exception if, while suspended:

- some other context calls `resume_with()` to resume this suspended context
- the function `fn` passed to `resume_with()` – or some function called by `fn` – throws an exception

Notes

`std::callcc()` preserves the execution context of the calling context as well as stack parts like *parameter list* and *return address*.[*] Those data are restored if the calling context is resumed.

A suspended `continuation` can be destroyed. Its resources will be cleaned up at that time.

On return `fn` must specify a `std::continuation` to which execution control is transferred.

If an instance with valid state goes out of scope and its `fn` has not yet returned, the stack is unwound and deallocated.

std::unwind_context() terminate the current running context, switching to the context represented by the passed `std::continuation`. This is like returning that `std::continuation` from the *entry-function*, but may be called from any function on that context.

```
void unwind_context( continuation && cont )   (1)
```

1) throws `std::unwind_exception`, binding the passed `std::continuation`. The running context's first stack entry – the one created by `std::callcc()` – catches `std::unwind_exception`, extracts the bound `std::continuation` and terminates the current context by returning that `std::continuation`.

Parameters

cont the `std::continuation` to which to switch once the current context has terminated

Preconditions

1) `cont` must be valid (`operator bool()` returns `true`)

Return value

1) None: `std::unwind_context()` does not return

Exceptions

1) throws `std::unwind_exception`

*required only by some x86 ABIs

Additional notes

GPU *call/cc* as proposed in this paper does not take GPUs into account. Later revisions will address this issue, once we have an overarching concept of how the various kinds of "lightweight execution agents" should interact.

SIMD does not interfere with *call/cc* and can be used as usual.

Of course, depending on the calling convention, some micro-processor registers dedicated to SIMD might be preserved and restored too [*].

TLS *call/cc* is TLS-agnostic - best practice related to TLS applies to *call/cc* too.

call/cc only preserves and restores micro-processor registers at its invocation.

Migration between threads `std::continuation` can be migrated between threads, except for instances of `std::continuation` representing `main()` or *entry-function* of a thread (see main() and thread functions).

[*] *MS Windows x64* calling convention

References

[1] N3708: A proposal to add coroutines to the C++ standard library

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

[3] N4045: Library Foundations for Asynchronous Operations, Revision 2

[4] N4649: Working Draft, Technical Specification on C++ Extensions for Coroutines

[5] P0534R0: call/cc (call-with-current-continuation): A low-level API for stackful context switching

[6] P0559R0: Operating principles for evolving C++

[7] call/cc in Scheme

[8] call/cc in Ruby

[9] Split Stacks / GCC

[10] Re: using split stacks

[11] `segmented_stack.hpp`: Boost.Context implementation of `segmented_stack` StackAllocator

[12] Library *Boost.Context*

[13] Library *Boost.Coroutine2*

[14] *Boost.Coroutine* example illustrating coroutine chaining

[15] Library *Boost.Fiber*

[16] *Boost.Fiber* performance using Skynet microbenchmark

[17] Library *Boost.Graph*

[18] *Boost.Graph* Visitor Concepts

[19] Library *Boost.Spirit*

[20] Library *Boost.Spirit* Karma

[21] C++Now 2014 talk: Coroutines, Fibers and Threads, Oh My

[22] C++Now 2016 talk: Pulling Visitors

[23] Boost developer mailing list: Lazy Spirit Generators?