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 delegate processing to distinct lightweight execution agents nested within a thread. Within this proposal, the terms context, execution context and context of execution all refer interchangeably to an execution agent of this form.

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.

It may be desirable to invent a less overloaded term for lightweight stackful mutually-exclusive execution agents nested within a thread. The term “context” is used for now because “context-switching” is a well-established term of art.

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), 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:
- 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.

Why should call/cc be standardized?

The call/cc facility cannot itself be implemented in portable C++. The present implementation, 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 particular 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.

Revision History

This document supersedes P0534R2.

Changes since P0534R2:

- simpler API (transfer of data discarded)
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.

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

```cpp
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.

Data can be transferred between two continuations via global pointer, calling wrappers (like std::bind) or lambda captures.

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

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

Variable a is captured by reference by the lambda and used to transfer the computed fibonacci number.
The invocation of std::callcc() at (0) immediately enters the lambda, passing the current continuation. The lambda calculates the fibonacci number using local variables a, b and next. The calculated fibonacci number is stored in a and the execution control returns via resume() at (1). lambda now represents the continuation of the lambda. The computed fibonacci number is printed at (2). At (3) the lambda is entered again in order to compute the next fibonacci number.
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.  

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

Passing data  Data can be transferred between two continuations via global pointer, calling wrappers (like std::bind) or lambda captures.

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

Output:  
inside lambda,i==1
i==2

The callcc() call at (0) enters the lambda and passes 1 into the new context. The value is incremented by one, as shown by (1). The expression caller.resume() at (2) resumes the original context (represented within the lambda by caller). The call to lambda.resume() at (3) resumes the lambda, returning from the caller.resume() call at (2). 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 (4), terminating its context.

Since the updated caller represents the continuation suspended by the call at (3), control returns to main().

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

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.

```cpp
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 only valid to resume a std::continuation instance 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.
- Call std::unwind_context(). This throws a std::unwind_exception "by hand," which is what std::unwind_context() 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:

```cpp
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), passing the function fn() to execute.

Let’s say that the context represented by the std::continuation instance ctx has called a function suspender(), which has called std::continuation::resume() and is now suspended. You intend to inject function fn() into context ctx as if suspender()’s resume() call had directly called fn().
Like an entry-function passed to `std::callcc()`, `fn()` must accept `std::continuation&&` and return `std::continuation`. The `std::continuation` instance returned by `fn()` will, in turn, be returned to `suspender()` by `resume()`. Suppose that code running on the program’s main context calls `std::callcc()`, thereby entering the first lambda shown below. This is the point at which `mc` is synthesized and passed into the lambda at (0).

Suppose further that after doing some work ((1) through (5)), the lambda calls `mc.resume()`, thereby switching back to the main context. The lambda remains suspended in the call to `mc.resume()` at (6).

At (8) the main context calls `f_ct.resume_with()` where the passed lambda accepts `continuation &&`. That new lambda is entered in the context of the suspended lambda. It is as if the `mc.resume()` call at (6) directly called the second lambda.

The function passed to `resume_with()` has almost the same range of possibilities as any function called on the context represented by `f_ct`. Its special invocation matters when control leaves it in either of two ways:

1. If it throws an exception, that exception unwinds all previous stack entries in that context (such as the first lambda’s) as well, back to a matching `catch` clause.*

2. If the function returns, the returned `std::continuation` instance is returned by the suspended `mc.resume()` (or `std::callcc()`, or `resume_with()`) call.

```cpp
int data = 0;
std::continuation f_ct = // (3)
    std::callcc([&data](std::continuation && mc) { // (0)
        std::cout << "f1: entered first time: " << data << std::endl; // (1)
        data+=1;
        mc = // (5)
            mc.resume(); // (2)
        std::cout << "f1: entered second time: " << data << std::endl;
        data+=1;
        mc = // (10)
            mc.resume(); // (6)
        std::cout << "f1: entered third time: " << data << std::endl; // (11)
        return std::move(mc); // (12)
    });
std::cout << "f1: returned first time: " << data << std::endl;
data+=1;
    f_ct = // (7)
        f_ct.resume(); // (4)
std::cout << "f1: returned second time: " << data << std::endl;
data+=1;
    f_ct = // (13)
        f_ct.resume_with([&data](std::continuation && mc){ // (8)
            std::cout << "f2: entered: " << data << std::endl;
            data=-1;
            return std::move( mc); // (9)
        });
std::cout << "f1: returned third time" << std::endl;
```

output:

```
f1: entered first time: 0
f1: returned first time: 1
f1: entered second time: 2
f1: returned second time: 3
f2: entered: 4
f1: entered third time: -1
f1: returned third time
```

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

The `f_ct.resume_with(<lambda>)` call at (8) passes control to the second lambda on the context of the first lambda.

As usual, `resume_with()` synthesizes a `std::continuation` instance representing the calling context, passed into the lambda as `mc`. This particular lambda returns `mc` unchanged at (9); thus that `mc` instance is returned by the `resume()`

*As stated in Exceptions, if there is no matching `catch` clause in that context, `std::terminate()` is called.
call at (6) and assigned at (10).

Finally, the first lambda returns at (12) the `mc` variable updated at (10), switching back to the main context.

**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.5,5

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.

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

But now consider this scenario:

```cpp
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?

With the present API, 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:

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

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

```cpp
template<typename Fn>
std::continuation callcc_deferred(Fn&& fn) {
    return std::callcc([auto fn=std::forward<Fn>(fn)](std::continuation&& caller){
        return fn(caller.resume());
    });
}
```

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.

**resume() invalidates std::continuation**  
The academic literature distinguishes full continuations from one-shot continuations. A full continuation can be resumed multiple times, which would be not merely difficult to implement but semantically problematic in C++. The std::continuation proposed in this paper is a one-shot continuation. Its resume() method immediately invalidates the instance; that instance may no longer be resumed.

Consider an implementation in which std::continuation stores a pointer into the processor’s stack area. Once that std::continuation is resumed, the formerly-suspended function will eventually return, destroying its stack frame. Some other function will reuse the space in some completely different way. Or perhaps not; that location in the reserved stack area may remain uninitialized memory.

It would be dramatically bad if that std::continuation instance retained the old pointer into the stack area, and consuming code mistakenly attempted to resume that same std::continuation again. Invalidating the std::continuation allows consuming code to detect the difference between a std::continuation that has not yet been resumed and one that has.
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`, 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.
- `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.

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.

---

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

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

‡ `callcc()` from boost.context takes 16 CPU cycles on Intel E5 2620 v4, SYS V.
Why call/cc should be preferred over ucontext

**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
- 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 is deallocated when std::continuation goes out of scope

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

std::continuation stores a pointer.

std::continuation::resume():

- optionally tests for nullptr
- stores nullptr to invalidate *this
- calls assembly code for stack switch.

**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
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;
        continuation resume();
        template< typename Fn >
        continuation resume_with( Fn && fn);
        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;
    }

member functions

(constructor) constructs new execution context

continuation() noexcept (1)
continuation(continuation && other) noexcept (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().

12
(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)noexcept (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

continuation resume() (1)

template< typename Fn >
continuation resume_with( Fn && fn) (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
fn function injected into resumed continuation

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) the current std::thread is the same as the thread on which *this was originally launched

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

* 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.
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().
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.
An injected function fn() must accept std::continuation&& and return std::continuation. The returned
std::continuation instance is, in turn, used as the return value for the suspended function: std::callcc(),
resume() or resume_with().

operator bool test whether continuation is valid

```cpp
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

```cpp
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

```cpp
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, 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.

*required only by some x86 ABIs
swap swaps two std::continuation instances

```cpp
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.

```cpp
template< typename Fn >
continuation callcc( Fn && fn) (1)
```

```cpp
template< typename StackAlloc, typename Fn >
continuation callcc( std::allocator_arg_t, StackAlloc && salloc, Fn && fn) (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 &&)`

**Return value**

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

**Preconditions**

std::callcc() may only be called by code running on a std::thread, or on an execution agent created by a previous std::callcc() call.

**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

4) if entry-function fn contains a catch(...) clause, it should also catch and rethrow std::unwind_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. Returning an invalid std::continuation instance (operator bool() returns false) invokes undefined behavior.

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

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 fn.
- 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.]

* required only by some x86 ABIs
• 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.

```cpp
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`

`std::unwind_exception` is the exception used to unwind the stack referenced by a `std::continuation` being destroyed. It is thrown by `std::unwind_context()`. `std::unwind_exception` binds a `std::continuation` referencing the context to which control should be passed once the current context is unwound and destroyed.

Stack allocators are the means by which stacks with non-default properties may be requested by the caller of `std::callcc()`. The stack allocator concept is implementation-dependent; the means by which an implementation’s stack allocators communicate with `std::callcc()` is unspecified.

An implementation may provide zero or more stack allocators. However, a stack allocator with semantics matching any of the following must use the corresponding name.

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.

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\(^5\) with the specified initial size, which grows on demand.
Additional notes

**GPU** *call/cc* as proposed in this paper is solely a CPU operation. It cannot be used to create a GPU execution agent, or to create or resume a CPU context from code running on a GPU.

**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. (But see P0772R0.)

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

**Migration between threads** is forbidden. A `std::continuation` may only be resumed on the `std::thread` on which it was launched.

**Relationship to executors** The *call/cc* facility is intended to compose with executors. The authors envision an executor implementation that runs each passed work item on a *call/cc* context. But neither *call/cc* nor the executors proposal depend on each other.

* MS Windows x64 calling convention
References

[3] P0559R0: Operating principles for evolving C++
[4] call/cc in Scheme
[5] call/cc in Ruby
[7] Re: using split stacks