Document number: P0534R0
Date: 2017-02-04
Reply-to: Oliver Kowalke (oliver.kowalke@gmail.com)
Audience: SG1/LEWG

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

Abstract This document **supersedes P0099R1**[1] and proposes a C++ equivalent to the well-known concept **call-with-current-continuation** (abbreviation **call/cc**).
In fact, P0099R1's `std::execution_context<>` already represents a **one-shot continuation**, reminiscent of Scheme's[2] and Ruby's[3] **call/cc**. From this point of view the proposed API is an **advancement** of `std::execution_context<>`.
The benefits are:

- use of established, well-known concept
- no name clashes with *execution-context* in executor and network proposals
- relaxed constraints in the transfer of data
- eliminating false usage
- working implementation with boost.context[7]

Motivation

call/cc is an evolutionary step of `std::execution_context<>`; beside the name clashes with the executor and network proposals, `std::execution_context<>` has some drawbacks:

- The API described in P0099R1 allows calling `operator()(std::invoke_ontop_arg,fn &&,Args...)` on a newly created `std::execution_context<>`, which results in undefined behaviour: the context must be entered at least one time before it is permitted to invoke a function on top of the context.
- `std::execution_context<>` mandates data transfer in both directions, even when not required. For instance, generators must pass (and pay for) **dummy data** in one direction.
- `std::execution_context<>` transfers only data of a fixed type (the template argument). Implementation experience with boost.coroutine2 and boost.fiber shows that implementations of higher-level abstractions might require transferring data of different types, or no data at all (for instance during initialization phase).
- If the data type used by `std::execution_context<>` (template argument) is not default constructible and the context-function simply returns (no data transferred back) the expression `auto [ctx2,x2]=ctx1(x1)` is invalid: `x2` can not be default constructed.

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, and for those the written and maintained code can look and act sequential while using continuations.

C and C++ already use implicit continuations: if a routine calls a sub-routine, then a (hidden) continuation (the remaining steps after the sub-routine call) is created. This continuation is resumed when the sub-routine returns. For instance the x86 architecture stores the (hidden) continuation as 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** via 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 continuation of the computation step derived from the current point in a program's execution is called the **current continuation**. **call/cc** captures the **current continuation** and passes it as the argument of the function invoked by **call/cc**.

Continuations that can be called multiple times are named **full continuations**.
One-shot continuations can only resumed once (a resumed **one-shot continuation** becomes invalidated); control is transferred to an execution context where the continuation is no longer in scope.
Class `std::execution_context<>`, proposed in **P0099R1**[1] already represents a **one-shot-continuation**.
Full continuations are **not** considered in this proposal because of their nature, problematic in C++. Full continuations would require copies of the stack (including the 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**, as shown in Delimited continuations.

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 as an argument 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 `std::continuation::operator()` 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` (see The switch mechanism).

`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(); // (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(); // (3)
}
```

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

```
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()` 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()` at (3) resumes function `foo`, returning from the `operator()` call at (1) and executing the assignment at (4). Here we replace the `std::continuation` instance `caller` invalidated by the `operator()` call at (1) with the new instance returned by that same `operator()` call.
Function `std::callcc()` captures the **current continuation** and enters the given function immediately, while `operator()` 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()` as well as `operator()` 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(0); // (1)
                int next=a+b;
                a=b;
                b=next;
            }
            return std::move(caller);
        });
for (int j=0;j<10;++j) {
    int i=std::get_data<int>(lambda); // (2)
    std::cout << i << " ";
    lambda=lambda(); // (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 `operator()` 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.

The switch mechanism

Modern **micro-processors** are **register machines**; the content of processor registers represent the execution context of the program at a given point in time.
Operating systems simulate parallel execution of programs on a single processor by switching between programs (**context switch**) by **preserving** and **restoring** the content of **all registers**.

For **call/cc**, not all registers need be preserved because the context switch is effected by a visible function call. It need not be undetectable like an operating-system context switch; it only needs to be as transparent as a call to any other function. The calling convention – the part of the ABI that specifies how a function's arguments and return values are passed – determines which subset of micro-processor registers must be preserved by the called subroutine.

As a consequence a continuation preserves the execution context, i.e. state of the register machine (including the stack as well as the instruction pointer).

The **calling convention**[4] of **SYSV ABI** for **x86_64** architecture determines that general purpose registers R12, R13, R14, R15, RBX and RBP must be preserved by the sub-routine - the first arguments are passed to functions via RDI, RSI, RDX, RCX, R8 and R9 and return values are stored in RAX, RDX.

```
1  leaq   -0x38(%rsp), %rsp

3  movq   %r12, 0x8(%rsp)
4  movq   %r13, 0x10(%rsp)
5  movq   %r14, 0x18(%rsp)
6  movq   %r15, 0x20(%rsp)
7  movq   %rbx, 0x28(%rsp)
8  movq   %rbp, 0x30(%rsp)

10 movq   %rsp, %rax

12 movq   %rdi, %rsp

14 movq   0x38(%rsp), %r8

16 movq   0x8(%rsp), %r12
17 movq   0x10(%rsp), %r13
18 movq   0x18(%rsp), %r14
19 movq   0x20(%rsp), %r15
20 movq   0x28(%rsp), %rbx
21 movq   0x30(%rsp), %rbp

23 leaq   0x40(%rsp), %rsp

25 movq   %rsi, %rdx
26 movq   %rax, %rdi

28 jmp   *%r8
```

The code fragment above, taken from boost.context,[7] shows how the context switch might be implemented for **SYSV ABI/x86_64**.
Line (1) reserves space on the stack of the current context to hold the content of registers R12-R15, RBX and RBP. The address of the stack pointer is preserved in register RAX at line (10) for later use.
The **return address**, i.e. the address of the instruction that will be executed after this function returns, is left on the stack. Other architectures store the return address in a special register (link register) instead of the stack; in that case the link register must be preserved too.
At line (12) the stack pointer gets assigned to the address of the continuation that has to be resumed - in fact, the continuation represents a stack address (the stack pointer was passed in RDI as first argument).
The return address is loaded into register R8 at line (14); with the indirect jump at line (28) the **continuation** is **resumed**.
As required by the calling convention, registers R12-R15, RBX and RBP are restored at lines (16) - (21).
The stack address, preserved in RAX at line (10), of the **suspended continuation** is **returned** as a **one-shot continuation**.

In fact, **call/cc** is an extended function call. The **general purpose registers** specified by the calling convention are preserved. In addition, the **stack pointer** and **instruction pointer** are preserved and exchanged too – thus, from the point of view of calling code, **call/cc** behaves like an ordinary function call.
In other words, **call/cc** acts on the level of a simple function invocation – with the same performance characteristics (in terms of CPU cycles).

Design

Because `std::continuation` **contains** only its **stack pointer** as a member variable, it is proposed as a **pure data structure**.

Passing data Data are passed to another context as additional arguments of `std::callcc()` and `operator()`. With functions `data_available()` and `get_data()` the code can test for data and if desired retrieve the data.

```
int i=1;
std::continuation lambda=
    std::callcc( // (0)
        [](std::continuation && caller){
            int j=std::get_data<int>(caller); // (1)
            std::cout << "inside lambda,j==" << j << std::endl;
            caller=caller(j+1); // (2)
            return std::move(caller); // (5)
        },
        i);
i=std::get_data<int>(lambda); // (3)
std::cout << "i==" << i << std::endl;
lambda=lambda(); // (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(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()` at (4) (note that no data is passed) resumes the lambda, returning from the `caller(j+1)` call at (2). Here, too, we replace the `std::continuation` instance `caller` invalidated by the `operator()` call at (2) with the new instance returned by that same `operator()` 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()`.
However, since context `lambda` has now terminated, the updated `lambda` is invalid. Its `operator bool()` returns `false`; its `operator!()` returns `true`.
It may seem tricky to keep track of which `std::continuation` instance is currently valid, representing the state of the suspended context. Please bear in mind that this facility is intended as a high-performance foundation for higher-level libraries. It is not intended to be directly consumed by applications.
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]=std::get_data<int,int>(caller); // (1)
            std::cout << "inside lambda,i==" << i << ",j==" << j << std::endl;
            caller=caller(i+j); // (2)
            return std::move(caller); // (5)
        },
        i,
        j);
int k=std::get_data<int>(lambda); // (3)
std::cout << "k==" << k << std::endl;
lambda=lambda(); // (4)

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

`std::get_data<int,int>(caller)` 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.

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

Termination When the *entry-function* invoked by **call/cc** returns a valid `std::continuation` instance, the running context is terminated. Control switches to the continuation 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 instance returned from `std::callcc()` or the previous instance's `operator()`), control returns to the context that most recently resumed the running callable. However, the callable may return (switch to) any reachable valid `std::continuation` instance.

Exceptions If an uncaught exception escapes from the *entry-function*, `std::terminate` is called.

Invoke function on top of a continuation Sometimes it is useful to invoke a new function (for instance, to throw an exception) on top of a continuation. For this purpose you may pass to
`operator()(invoke_ontop_arg_t,Fn &&,Args ...)`:

- the special argument `invoke_ontop_arg`
- the function to execute
- any additional arguments.

Like an *entry-function* passed to `std::callcc()`, the function passed in this case must accept an rvalue reference to `std::continuation`. However, instead of necessarily returning a `std::continuation`, it may return a single type or a `std::tuple`. Whatever value(s) it returns will become available to the context referenced by `*this` as the data tested by `data_available()` and retrieved by `get_data()`.

Suppose that code running on the program's main context calls `callcc(f)`, thereby entering `f()`. This is the point at which `mc` is synthesized and passed into `f()`.
Suppose further that after doing some work, `f()` calls `mc()`, thereby switching context back to the main context. `f()` remains suspended in the call to `mc()`.
At this point the main context calls `f_ct(invoke_ontop_arg, 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()` directly called `g()`.
Function `g()` has the same range of possibilities as any function called on `f()`'s context. Its special invocation only matters when control leaves it in either of two ways:

1. If `g()` throws an exception, that exception unwinds all previous stack entries in that context (such as `f()`'s) as well, back to a matching `catch` clause.[*][†]
2. If `g()` returns, its return value provides data for `f()`'s suspended `mc()` call.

```
std::continuation f(std::continuation && mc) {
    int data=std::get_data<int>(mc); // (1)
    std::cout << "f: entered first time: " << data << std::endl;
    mc = // (5)
```

[*] As stated in Exceptions, if there is no matching `catch` clause in that context, `std::terminate()` is called.
[†] There are only two ways to terminate a given context without terminating the whole process. One is to switch to some context that will destroy the continuation passed (or returned) to it. The other is to return a valid continuation from the *entry-function*. If an `invoke_ontop_arg` function throws an exception, it is good practice to bind into the exception object the continuation passed into the `invoke_ontop_arg` function so that a `catch` clause in the *entry-function* can return that continuation.

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

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

int data=1;
std::continuation f_ct= // (3)
    std::callcc(f,data); // (0)
data=std::get_data<int>(f_ct);
std::cout << "f: returned first time: " << data << std::endl;
f_ct = // (7)
    f_ct(data+1); // (4)
data=std::get_data<int>(f_ct);
std::cout << "f: returned second time: " << data << std::endl;
f_ct = // (13)
    f_ct(std::invoke_ontop_arg,g,data+1); // (8)
data=std::get_data<int>(f_ct);
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(invoke_ontop_arg, g, data+1)` call at (8) passes control to `g()` on the context of `f()`.
The `return` statement at (9) causes the `operator()` 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.

Stack destruction On construction of a continuation 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 a valid `std::continuation` instance (`operator bool()` returns `true`) is called, the stack will be unwound and destroyed. *
The stack on which `main()` is executed, as well as the stack implicitly created by `std::thread`'s constructor, is allocated by the operating system. Such stacks are recognized by `std::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:

* An implementation is free to unwind the stack without throwing an exception.
† This concept, along with `std::callcc()` accepting `std::allocator_arg_t`, is an optional part of the proposal. It might be that implementations can reliably infer the optimal stack representation.

- `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.

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 operator()( Arg ... arg);

    template< typename Fn, typename ... Arg >
    continuation operator()( invoke_ontop_arg_t, Fn && fn, Arg ... arg);

    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 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

(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. As noted in Stack destruction, an implementation is free to unwind the stack either by throwing an exception or by intrinsics not requiring `throw`.*

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**

operator() resumes a continuation

`template< typename ...Args >` `continuation operator()( Args ... args)`	(1)
`template< typename Fn, typename ...Args >` `continuation operator()( invoke_ontop_arg_t, Fn && fn, Args ... args)`	(2)

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

2) suspends the active context, resumes continuation `*this` but invokes `fn(args ...)` in the resumed context (on top of the last stack frame)

Parameters

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

fn function invoked ontop of 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) calls `std::terminate` if an exception escapes *entry-function* `fn`

Preconditions

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

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

Postcondition

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

*If the implementation unwinds the stack by throwing an exception, it should throw an exception with a conventional name, e.g. `std::unwind`. 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, if the implementation's stack unwinding is implemented by throwing an exception, a correct *entry-function* `try` statement must `catch (std::unwind const&)` and rethrow it.

Notes
`operator()` 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 `operator()` invalidates the instance on which it is called, *no valid `std::continuation` instance ever represents the currently-running context.*
When calling `operator()`, it is conventional to replace the newly-invalidated instance – the instance on which `operator()` was called – with the new instance returned by that `operator()` call. This helps to avoid inadvertent calls to `operator()` on the old, invalidated instance.

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 (`operator()` called) - calling `operator()` 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.

*required only by some x86 ABIs

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*.

`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 escapes *entry-function* `fn`

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` has to specify a `std::continuation` to which the 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::data_available() test if data are present

`bool data_available( continuation && c)`	(1)

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

std::get_data() transfer of data

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

1) transfers single datum from continuation `c` into this context

2) transfers multiple data from continuation `c` into this 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 `operator()`.

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

†required only by some x86 ABIs

std::any_thread() test whether suspended continuation may be resumed on a different thread

`bool any_thread( continuation const& c)const noexcept`	(1)

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

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`.

Use cases

call/cc can be used to implement several higher-level abstractions.

Asymmetric coroutines: N3708 is implemented in boost.coroutine2[8] using **call/cc** from boost.context[7] as a building block. Each `push_type` and `pull_type` of a coroutine represents a continuation (i.e. a coroutine consists of two continuations).

```
boost::coroutines2::coroutine<int>::pull_type source(
    [](boost::coroutines2::coroutine<int>::push_type & sink){
        int first=1,second=1;
        sink(first);
        sink(second);
        for(int i=0;i<10;++i){
            int third=first+second;
            first=second;
            second=third;
            sink(third);
        }
    });
for(auto i: source){
    std::cout << i <<  " ";
}

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

Cooperative multi-tasking: boost.fiber[9] provides a framework for micro-/userland-threads (fibers) scheduled cooperatively. The library implements fibers using **call/cc** (boost.context[7]). The API contains classes and functions to manage and synchronize fibers, similar to the standard thread support library.
Each fiber is implemented using a continuation.

```
using channel_t=boost::fibers::buffered_channel<std::string>;
channel_t chan1{1},chan2{1};
boost::fibers::fiber fping(
    [&chan1,&chan2]{
        chan1.push("ping");
        std::cout << chan2.value_pop() << "\n";
        chan1.push("ping");
        std::cout << chan2.value_pop() << "\n";
        chan1.push("ping");
        std::cout << chan2.value_pop() << "\n";
    });
boost::fibers::fiber fpong(
    [&chan1,&chan2]{
        std::cout << chan1.value_pop() << "\n";
        chan2.push("pong");
        std::cout << chan1.value_pop() << "\n";
        chan2.push("pong");
        std::cout << chan1.value_pop() << "\n";
```

```
        chan2.push("pong");
    });
fping.join();
fpong.join();

output:
    ping
    pong
    ping
    pong
    ping
    pong
```

Delimited continuations can be implemented via **call/cc**. `reset` delimits the continuation and `shift` reifies the continuation, i.e. the code that follows after `shift` returns is passed as a continuation to `shift`.

On entry `1` is written to `std::cout` at (0). The `shift` operator at (1) wraps the continuation, that means the code at (2), and passes it as argument `cont` at (1). `cont()` is called two times, thus (2) is executed two times before (3) writes `2` to `std::cout`.

```
reset=[]{
    std::cout << "1\n"; // (0)
    shift=[](auto cont){ // (1)
        cont();
        cont();
        std::cout << "2\n"; // (3)
    };
    std::cout << "3\n"; // (2)
};

output:
    1 3 3 2
```

Backtracking or non-deterministic choice is the ability to specify certain *choice points* in the program used to for finding all (or some) solutions to some computational problems. The algorithm *backtracks* to a previous *choice point* as soon as it determines that the current execution path cannot reach a valid solution.
Backtracking could be implemented using two continuations, a success continuation that proceeds with the algorithm and a failure continuation that backtracks to a previous choice point.[10]

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 (**call/cc** triggers the context switch at its invocation).
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.
As shown in The switch mechanism, **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 Design).

[*]MS Windows x64 calling convention

A. Assembler: shortest list of mnemonics (ARM)

The code is taken from boost.context[7] (architecture: ARM 32bit, calling convention: AAPCS).

```
@ save LR as PC
push {lr}
@ save hidden,V1-V8,LR
push {a1,v1-v8,lr}

@ store RSP (pointing to context-data) in A1
mov  a1, sp

@ restore RSP (pointing to context-data) from A2
mov  sp, a2

@ restore hidden,V1-V8,LR
pop {a4,v1-v8,lr}

@ return transfer_t from jump
str  a1, [a4, #0]
str  a3, [a4, #4]
@ pass transfer_t as first arg in context function
@ A1 == FCTX, A2 == DATA
mov  a2, a3

@ restore PC
pop {pc}
```

B. Assembler: longest list of mnemonics (PPC)

The code is taken from boost.context[7] (architecture: PPC 32bit, calling convention: SYSV).

```
# reserve space on stack
subi  %r1, %r1, 92

stw  %r13, 0(%r1)   # save R13
stw  %r14, 4(%r1)   # save R14
stw  %r15, 8(%r1)   # save R15
stw  %r16, 12(%r1)  # save R16
stw  %r17, 16(%r1)  # save R17
stw  %r18, 20(%r1)  # save R18
stw  %r19, 24(%r1)  # save R19
stw  %r20, 28(%r1)  # save R20
stw  %r21, 32(%r1)  # save R21
stw  %r22, 36(%r1)  # save R22
stw  %r23, 40(%r1)  # save R23
stw  %r24, 44(%r1)  # save R24
stw  %r25, 48(%r1)  # save R25
stw  %r26, 52(%r1)  # save R26
stw  %r27, 56(%r1)  # save R27
stw  %r28, 60(%r1)  # save R28
stw  %r29, 64(%r1)  # save R29
stw  %r30, 68(%r1)  # save R30
stw  %r31, 72(%r1)  # save R31
stw  %r3,  76(%r1)  # save hidden

# save CR
mfcr  %r0
stw   %r0, 80(%r1)
# save LR
mflr  %r0
stw   %r0, 84(%r1)
# save LR as PC
```

```
stw   %r0, 88(%r1)

# store RSP (pointing to context-data) in R6
mr  %r6, %r1

# restore RSP (pointing to context-data) from R4
mr  %r1, %r4

lwz  %r13, 0(%r1)  # restore R13
lwz  %r14, 4(%r1)  # restore R14
lwz  %r15, 8(%r1)  # restore R15
lwz  %r16, 12(%r1)  # restore R16
lwz  %r17, 16(%r1)  # restore R17
lwz  %r18, 20(%r1)  # restore R18
lwz  %r19, 24(%r1)  # restore R19
lwz  %r20, 28(%r1)  # restore R20
lwz  %r21, 32(%r1)  # restore R21
lwz  %r22, 36(%r1)  # restore R22
lwz  %r23, 40(%r1)  # restore R23
lwz  %r24, 44(%r1)  # restore R24
lwz  %r25, 48(%r1)  # restore R25
lwz  %r26, 52(%r1)  # restore R26
lwz  %r27, 56(%r1)  # restore R27
lwz  %r28, 60(%r1)  # restore R28
lwz  %r29, 64(%r1)  # restore R29
lwz  %r30, 68(%r1)  # restore R30
lwz  %r31, 72(%r1)  # restore R31
lwz  %r3,  76(%r1)  # restore hidden

# restore CR
lwz   %r0, 80(%r1)
mtcr  %r0
# restore LR
lwz   %r0, 84(%r1)
mtlr  %r0
# load PC
lwz   %r0, 88(%r1)
# restore CTR
mtctr %r0

# adjust stack
addi  %r1, %r1, 92

# return transfer_t
stw  %r6, 0(%r3)
stw  %r5, 4(%r3)

# jump to context
bctr
```

References

[1] P0099R1: A low-level API for stackful context switching

[2] call/cc in Scheme

[3] call/cc in Ruby

[4] System V Application Binary Interface, AMD64 Architecture Processor Supplement, Draft Version 0.96

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

[6] Split Stacks / GCC

[7] Library *Boost.Context* (**call/cc** available in boost-1.64)

[8] Library *Boost.Coroutine2*

[9] Library *Boost.Fiber*

[10] Darrell Ferguson and Dwight Deugo, Call with Current Continuation Patterns. in 8th Conference on Pattern Languages of Programs (PLoP 2001) , 2001