Document number: N4397
Date: 2015-04-08
Project: WG21, SG1
Author: Oliver Kowalke (oliver.kowalke@gmail.com)

A low-level API for stackful coroutines

Abstract . . . 1
Background . . . 2
Definitions . . . 2
- Execution context . . . 2
- Activation record . . . 2
- Stack frame . . . 2
- Heap-allocated activation record . . . 2
- Processor stack . . . 2
- Application stack . . . 2
- Thread stack . . . 2
- Side stack . . . 3
- Linear stack . . . 3
- Linked stack . . . 3
- Non-contiguous stack . . . 3
- Parent pointer tree . . . 3
- Coroutine . . . 3
- Toplevel context function . . . 3
- Asymmetric coroutine . . . 3
- Symmetric coroutine . . . 4
- Fiber/user-mode thread . . . 4
- Resumable function . . . 4
- Suspend by return . . . 4
- Suspend by call . . . 4

Introduction . . . 4
Discussion . . . 5
- Memory requirements . . . 5
- Calling subroutines . . . 6
- Asymmetric vs. symmetric . . . 7
- Passing data . . . 7
- Stack strategies . . . 9

Design . . . 9
- Class `std::execution_context` . . . 9
- First-class object . . . 10
- Capture record . . . 10
- Suspend-on-call . . . 10
- Exceptions . . . 10

Acknowledgement . . . 12
References . . . 12
A. recursive descent parser using `std::execution_context` . . . 13

Revision History This document supersedes N3985 and elaborates a low-level API for stackful coroutines.

Abstract

This paper proposes a *low-level* API for a stackful execution context, suitable to act as **building-block** for **high-level** constructs like stackful coroutines from N3985[1] (Boost.Coroutine2[9]) as well as to implement effective cooperative multitasking (Boost.Fiber[10]).
Based on the proposed low-level API, the follow-up proposal N4398[5] introduces a unified syntax for stackless and stackful coroutines.

The most important features are:

- first-class object that can be stored in variables or containers
- introduction of new keyword `resumable` together with a lambda-like expression
- symmetric transfer of execution control, i.e. suspend-by-call - enables a richer set of control flows than asymmetric transfer of control (i.e. suspend-by-return as described in N4134)
- benefits of traditional stack management retained
- ordinary function calls and returns not affected
- working implementation in Boost.Context[8]

Background

At the November 2014 meeting in Urbana the committee discussed proposals for *stackless* (N4134[3] and N4244[4]) and *stackful* (N3985[1]) coroutines. The members of the committee concluded that stackless and stackful coroutines are distinct use cases and decided to pursue both kinds of coroutines.
The committee encouraged the community to evaluate the possibility of a unified syntax for coroutines.

Definitions

This section gives an overview about the wording used in this proposal.

Execution context environment in which program logic is evaluated (CPU registers, stack).

Activation record also known as *activation frame* or *stack frame* (a special activation record).
An *activation record* is a data structure containing the state of a particular function call. The following data are part of an activation record:

- a set of registers (defined by the calling convention of the ABI *, e.g. the return address)
- local variables
- parameters to the function
- eventually return value

Stack frame an activation record allocated on the processor stack; stack frames are allocated in strict last-in-first-out order - placed by incrementing/decrementing the stack pointer register on function call/return.

Heap-allocated activation record an activation record allocated in heap storage; every heap allocated activation record holds a pointer to its parent activation record (parent pointer tree).

*Application Binary Interface

Processor stack is a chunk of memory into which the processor's stack pointer register is pointing. The processor stack might belong to:

- an application's initial (or only) thread
- an explicitly-launched thread
- a sub-thread execution context (e.g. a coroutine or fiber)

Application stack The processor stack assigned to function `main()` is generally a *linear stack*. Often, the memory management system is used to detect stack overflow and to allocate more memory. This stack is ideally placed well away from any other data in the process's address space so it can be extended as needed.

Thread stack The processor stack assigned to an explicitly-launched thread is generally a *linear stack*. In a 32-bit (or smaller) address space, the operating system cannot, in general, guarantee that such a stack can be arbitrarily extended on overflow: the range of available addresses is constrained by the next adjacent allocated memory block. Instead, stacks with a fixed size are used - usually 1MB (Windows) up to 2MB (Linux), but some platforms use smaller stacks (64KB on HP-UX/PA and 256KB on HP-UX/Itanium).

Side stack is a processor stack that is used for *stackful* context switching. Each execution context gets its own stack. The stack belonging to an inactive context remains unchanged, while function calls and returns push and pop stack frames on the currently-active stack.
A side stack is generally allocated by the running process (library code) rather than by the operating system. It might either be a *linear stack* or a *linked stack*.

Linear stack is a contiguous memory area into which the processor's stack pointer register points. On a processor with a stack pointer register, this is the traditional stack model for C++. Allocation and deallocation of activation records (stack frames in this context) is performed by incrementing/decrementing the stack pointer.

Linked stack also known as *split stack*[6] or *segmented stack*.[7] This is a linked list of contiguous memory blocks of intermediate size, each of which might hold several function stack frames. The processor's stack pointer points into the head block on this list; normally a new stack frame is allocated by adjusting the stack pointer, as usual. When compiler-generated code detects near overflow, it links on a new head block. Thus, the effective stack space can grow arbitrarily without requiring a single contiguous address range.
This mechanism has a low performance penalty - typically adding around 5-10% overhead to the instruction sequence implementing a function call. Only an application consisting solely of empty functions would see that impact overall. Small functions would see the most overhead – but modern C++ compilers work really hard to inline small functions, and of course the overhead vanishes entirely for inline functions.
Applications compiled with support for linked stacks can use (link against) libraries not supporting linked stacks. (See GCC's documentation,[6] chapter 'Backward compatibility'.)

Non-contiguous stack is a *parent pointer tree*-structure used to store heap-allocated activation records. It is a stack with branches; an activation record can outlive the context in which it was created. With this mechanism, the processor's stack pointer register does not point to (or into) any of these activation records. Compare to *Linked stack*.
A heap activation record is allocated on entry to a function and freed on return, as with N4134.[3]
This structure is also called *cactus stack*[11] or *spaghetti stack*.

Parent pointer tree data structure in which each node has a pointer to its parent, but no pointer to its children. Traversal from any node to its ancestors is possible but not to any other node.

Coroutine function that enables explicit suspend and resume of its progress by preserving execution state (*activation record*), thus providing an enhanced control flow. Coroutines have the following characteristics:[1]

- values of local data persist across context switches
- execution is suspended as control leaves coroutine and resumed at certain time later
- symmetric or asymmetric control-transfer mechanism
- first-class object or compiler-internal structure
- stackless or stackful

Toplevel context function is a function executed as a coroutine (stackless or stackful). This term is particularly useful when discussing library-supported context switching, as the execution context must be constructed explicitly and passed a function.

Asymmetric coroutine provides two distinct operations for the context switch - one operation to resume and one operation to suspend the coroutine.
An asymmetric coroutine is tightly coupled with its caller, i.e. suspending the coroutine transfers execution control back to the point in the code from which the coroutine was last resumed. The asymmetric control-transfer mechanism is usually used in the context of generators.

Symmetric coroutine only one operation to resume/suspend the context is needed.
A symmetric coroutine does not know its caller; control can be transferred to any other symmetric coroutine (which must be explicitly specified). The symmetric control-transfer mechanism is usually used to implement cooperative multitasking.

Fiber/user-mode thread execute tasks in a cooperative multitasking environment involving a scheduler. Coroutines and fibers are distinct (N4024[2]).
Code running in a user-mode thread (or "fiber") suspends by passing control to the scheduler, which selects the next fiber to resume. Such code is not inherently coupled to the code that launched the fiber (as with an asymmetric coroutine); nor must it explicitly select the next fiber (as with a symmetric coroutine).

Resumable function N4134[3] describes resumable functions as an efficient language-supported mechanism for stackless context switching introducing two new keywords - `await` and `yield`. A resumable function is a kind of asymmetric coroutine: suspending passes control back to its caller.
Characteristics of resumable functions:

- stackless
- uses heap allocated activation records (referred as activation frames in N4134)
- tight coupling between caller and resumable function (asymmetric control-transfer mechanism)
- implicit *return*-statement[3] (code transformation)

Suspend by return The terms *stackless* and *stackful* can become confusing as different implementations are discussed. An N4134[3] "stackless" resumable function might consume a bit of the current processor stack on entry, cleaning it off on suspension. A *cactus stack* coroutine implementation might provide "stackful" semantics while completely avoiding the processor stack.
Suspend by return describes a mechanism such as described in N4134[3] and N4244,[4] in which suspending is implemented by returning from the function body.

Suspend by call describes a mechanism such as described in N3985,[1] in which suspending is implemented by calling some other (library) function.

Introduction

Traditionally C++ code is run on a linear stack, i.e. the activation records are allocated in strict *last-in-first-out* order. This stack model allocates activation records on function call/return by incrementing/decrementing the stack pointer.
But in the context of coroutines, that is, switching between different execution contexts, a linear stack introduces problems. Calling a function creates an activation record on the stack which is removed if the function returns. But for a suspended coroutine the activation record **must not** be **removed**!
Consider the following scenario:

- Assume the processor stack is built in descending order: that is, a PUSH instruction decrements the stack pointer register. Call the stack pointer's initial value SP0.
- Function `main()` enters coroutine `C()`.
- `C()`'s prolog allocates a stack frame of size `sizeof(C::frame)` by decrementing SP. SP is now at SP1 = (SP0 - `sizeof(C::frame)`).
- `C()` suspends, returning control to `main()`. `main()` must find its stack frame at SP = SP0.
- `main()` now calls function `F()`.
- `F()`'s prolog allocates a stack frame of size `sizeof(F::frame)` by decrementing SP. SP is now at SP2 = (SP0 - `sizeof(F::frame)`).
- Unless either `(sizeof(C::frame)== 0)` or `(sizeof(F::frame)== 0)`, any data written by `F()` to its own stack frame will necessarily overwrite any data saved by `C()` in its stack frame.
- `F()` returns. SP is back to SP0.
- `main()` resumes `C()`. SP is set to SP1.
- `C()` attempts to access data in its stack frame – which has been overwritten by `F()`. We are now in the realm of Undefined Behavior.

In order to prevent stack corruption, a stackless coroutine uses a heap-allocated activation record (N4134[3]), while stackful coroutines use a side stack (N3985[1]).
Since an N4134 stackless resumable function uses *suspend by return*, when it suspends, the stack pointer is restored to its position before the resumable function was called. While executing, a resumable function can consume additional space in the linear processor stack; it can call traditional functions. But they must all return before the resumable function suspends. If resumable function `A()` calls function `B()`, and `B()` wishes to suspend, `B()` must also be a resumable function. Thus the term *stackless*: a suspended resumable function leaves no activation record on the linear processor stack. A single linear stack can be reused by an arbitrary number of suspended resumable functions.
Using a side stack permits *stackful* coroutines to use *suspend by call*. An arbitrary number of ordinary stack frames can be left on the side stack for a suspended coroutine context; arbitrary stack frames can be pushed or popped on the currently-active stack, independently of any suspended stack.
This is the fundamental difference between stackless and stackful coroutines.

Traditional stack management – a single linear stack per thread – is inadequate for coroutines because coroutines must outlive the context hey were created in.

Discussion

Memory requirements In the case of a simple toplevel context-function, stackless and stackful coroutines have the same memory requirements for their activation records.

As described in N4134, the compiler analyses the body of the toplevel context function and determines the required size and allocates the activation record on the heap.

```
// N4134: stackless resumable function
generator<int> fib(int n){
    int a=0,b=1;
```

```
    while(n-->0){
        yield a;
        auto next=a+b;
        a=b;
        b=next;
    }
}

int main(){
    for(auto v:fib(35)){
        std::cout<<v<<std::endl;
        if(v>10)break;
    }
}
```

In the example of calculating Fibonacci numbers using a resumable function, the compiler reserves space in the activation record for local variables *n*, *a*, *b* and *next*. Those variables are not accessed via the stack pointer. The processor stack is not used and the stack pointer is not changed*.

Each stackful coroutine owns a side stack (assigned to the stack pointer). The activation record of the toplevel context function is stored on the side stack, thus the original stack remains unchanged.

```
// N4397: stackful execution context
#define yield(x) p=x; mctx();
int main(){
    int n=35;
    int p=0;
    auto mctx(std::execution_context::current());
    auto ctx([n,&p,mctx]()mutable resumable(fixedsize(1024)){
            int a=0,b=1;
            while(n-->0){
                yield(a);
                auto next=a+b;
                a=b;
                b=next;
            }
        });
    for(int i=0;i<10;++i){
        ctx();
        std::cout<<p<<std::endl;
    }
}
```

In the case of calculating Fibonacci numbers using a stackful execution context, the compiler stores the local variables *n*, *a*, *b* and *next* on the side stack. The local variables are accessed via the stack pointer.

The memory requirements for both types of coroutines are equal.

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

If a traditional function (not another resumable function) is invoked inside the body of a resumable function, then the activation record belonging to the traditional function is allocated on the processor stack (so it is called a stack frame). As a consequence, stack frames of called functions must be removed from the processor

*Depending on the calling convention, typical *x86* code stores parameters and return address on the stack. This must be cleaned off by the time control is returned to the caller. Other architectures/ABIs do not have this requirement.

stack before the resumable function yields back to its caller.
In other words: the calling convention of the **ABI dictates** that, after the resumable function returns (suspends), the stack pointer contains the same address as before the resumable function was entered.
Hence a yield from nested call is **not permitted** – unless every called function down to the yield point is also a resumable function.
The benefit of stackless coroutines consists in reusing the processor stack for called subroutines: no separate stack memory need be allocated.

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

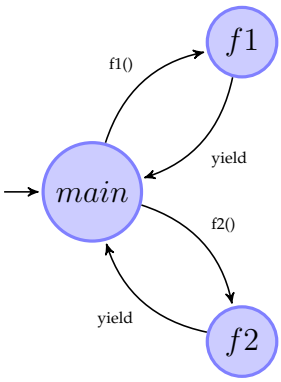
In stackful context switching, each execution context owns a distinct side stack which is assigned to the stack pointer (thus the stack pointer must be exchanged during each context switch).
All activation records (stack frames) of subroutines are placed on the side stack. Hence each stackful execution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

```
// N4397: stackful execution context
// access current execution context l1
auto l1=std::execution_context::current();
// create stackful execution context l2
auto l2=[&l1]()resumable(segmented()){
    std::printf("inside l2\n");
    // suspend l2 and resume l1
    l1();
}
// resume l2
l2();
```

Variable *l1* is passed to stackful execution context *l2* via the capture list of *l2*. The stack frames of `std::printf()` are allocated on the side stack owned by *l2*.

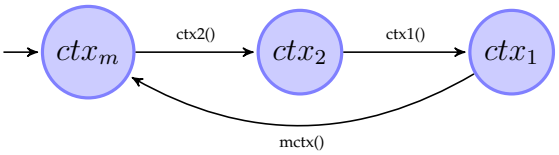
Asymmetric vs. symmetric As a building block for user-mode threads, symmetric control transfer is more efficient than the asymmetric mechanism.

Resumable functions (as proposed in N4134[3]) provide two operations for context switching: asymmetric control transfer. The caller and the resumable function are coupled, that is, a resumable function can only jump back to the point in the code where it was entered.



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

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



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

Passing data Because of the asymmetric resume/suspend operations of N4134, the proposal applies well to generator examples, e.g. returning data from the resumable function.

Passing data into the body of resumable functions (N4123) requires helper classes like `channel<>`.

```
// N4134: stackless resumable function
goroutine pusher(channel<int>& left, channel<int>& right){
    for(;;){
        auto val=await left.pull();
        await right.push(val+1);
    }
}

int main(){
    static const int N = 1000*1000;
    std::vector<channel<int>> c(N+1);
    for(int i=0;i<N;++i){
        goroutine::go(pusher(c[i],c[i+1]));
    }
    c.front().sync_push(0);
    std::cout<<c.back().sync_pull()<<std::endl;
}
```

In this case, the way data are passed into the body is not intuitive and introduces some problems.

Experience with `execution_context` from Boost.Context[8] turned up a common pattern: to pass a lambda to the `execution_context`, using its capture list to transfer data into and to return data from the body. Parameters (input/output) are accessed via captured references or pointers.

```
// N4397: stackful execution context
class X{
private:
    int* inp_;
    std::string outp_;
    std::execution_context caller_;
    std::execution_context callee_;

public:
    X():
        inp_(nullptr),outp_(),
        caller_(std::execution_context::current()),
```

```
        callee_([=]()resumable(segmented(1024)){
                        outp_=lexical_cast<std::string>(*inp_);
                        caller_(); // context switch to main()
                    })
    {}

    std::string operator()(int i){
        inp_=&i;
        callee_(); // context switch for conversion
        return outp_;
    }
};

int main(){
    X x;
    try{
        std::cout<<x(7)<<std::endl;
    }catch(const std::exception& e){
        std::cout<<"exception: "<<e.what()<<std::endl;
    }
}
```

Class `X` (rudimentary coroutine) demonstrates how input and output parameters are transferred between contexts. Member variable *callee_* represents a new execution context and captures the *this*-pointer of `X`. The body converts a integer variable (input) into a string (output). Any exception thrown by the conversion is transported and re-thrown in main().

Stack strategies For stackful coroutines two strategies are typical: a contiguous, fixed-size stack (as used by threads), or a linked stack (grows on demand).
The advantage of a fixed-size stack is the fast allocation/deallocation of activation records. A disadvantage is that the required stacksize has to be guessed.
The benefit of using a linked stack is that only the initial size of the stack is required. The stack itself grows on demand, by means of an overflow handler. The performance penalty is low. The disadvantage is that code executed inside a stackful coroutine must be rebuilt for this purpose. In the case of GCC's split stacks, special compiler/linker flags must be specified - no changes to source code are required.
When calling a library function not compiled for linked stacks (expecting a traditional contiguous stack), GCC's implementation uses link-time code generation to change the instructions in the caller. The effect is that a reasonably large contiguous stack chunk is temporarily linked in to handle the deepest expected chain of traditional function stack frames (see GCC's documentation[6]).

Design

Class `std::execution_context` is derived from the work on Boost.Context.[8] It provides a **small, basic API** on which to build **higher-level APIs** such as stackful coroutines (N3985,[1] working implementation Boost.Coroutine2[9]) and user-mode threads (executing tasks in a cooperative multitasking environment, working implementation Boost.Fiber[10]).
In effect, `execution_context` is a copyable handle to an underlying, noncopyable *capture record*.
`execution_context::operator()` preserves the CPU register set * + stack pointer: the content of those registers is stored in the capture record associated with `execution_context::current()`. Then `operator()` loads the CPU register set and stack pointer from the capture record associated with `*this`.
Finally, `execution_context::operator()` stores `execution_context::current()` in the capture record associated with `*this` – to permit backtracking through the history of context switches. It makes `*this` become `current()`.

*defined by ABI's calling conveintion

Class `std::execution_context` Based on implementation experience with `execution_context` in Boost.Coroutine2[9] and Boost.Fiber,[10] the author noticed that `execution_context` is almost always passed a lambda as its constructor argument. Lambda captures are especially useful for transporting data between different execution contexts (address of lvalue, residing on stack/heap).
Why not implicitly construct `std::execution_context` with a 'resumable lambda'-like syntax,[4] instead of explicitly passing a lambda as a constructor argument? Even if `std::execution_context` is constructed like a lambda, actually it isn't a lambda (it is constructible from `current()` too).
`std::execution_context` is more like a handle to a *capture record* of an execution context.

```
// N4397: stackful execution context
std::execution_context l1=[]()resumable(fixedsize(1024)){
    ...
};
auto l2=[&l1]()resumable(segmented(1024)){
    ...
};
```

The keyword `resumable` together with an hint (attribute) about the type and size of the side stack tells the compiler to generate a stackful execution context.
Because of the symmetric context switching (only one operation transfers control), the target `execution_context` must be explicitly specified.
Exchanging data between different execution contexts requires the use of lambda captures.

First-class object As first-class object the execution context can be stored in a variable or container.

Capture record Each instance of `std::execution_context` owns a toplevel activation record, the capture record. The capture record is a special activation record that stores additional data such as stack pointer, instruction pointer and a link to its parent execution context. That means that during a context switch, the execution state of the running context is captured and stored in the capture record while the content of the resumed execution context is loaded (into CPU registers etc.).

Parent context An `execution_context`'s pointer to its *parent* `execution_context` allows one to traverse the chain of ancestor contexts, that is, to locate the `execution_context` which resumed (called `operator()()` on) the currently active `execution_context`.

Active context The static member function `current()` returns a `std::execution_context` pointing to the current capture record. The current active capture record is stored in an internal, thread local pointer.

Toplevel capture records On entering `main()` as well as the *entry-function* of a thread, an execution context (capture record) is created and stored in the internal pointer underlying `current()`. (In practice, this could be lazily instantiated.)

Suspend-on-call `std::execution_context` is designed to suspend on call of `operator()()`. The *prologue* of `operator()()` captures (preserves) the active execution context and loads the data from the capture record of the resumed context (**this*) into the CPU.
The previous running context becomes the parent of the resumed context and gets suspended. This forms a parent-child relationship between the suspended and the resumed context. The parent may vary between successive calls of `operator()()`.

Termination If the body of the toplevel context function reaches its end, the parent execution context (pointer in the capture record) is resumed. That means that in the parent context the function `operator()()` returns. For this purpose the *epilogue* of `operator()()` loads the capture record (instruction pointer, stack pointer etc.) of the parent context, so that it is resumed.

Exceptions An uncaught exception thrown from the execution context is caught, the parent execution context is resumed and the exception is re-thrown inside the parent context (that is, the exception is emitted by `operator()()`).

member functions

(constructor) constructs new execution context

`[captures](params) mutable resumable(hint) attrs -> ret {body}`	(1)
`execution_context(const execution_context& other)=default`	(2)
`execution_context(execution_context&& other)=default`	(3)

1) the constructor does not take a lambda as argument, instead the compiler evaluates the lambda-like syntax and constructs a `std::execution_context` directly

captures list of captures

params only empty parameter-list allowed

mutable allows to modify parameters captured by copy

resumable identify resumable context

hint stack type hint:

- `fixedsize(x=default_stacksize)`: fixed size stack (`default_stacksize` is platform depended)
- `segmented(x=default_stacksize)`: stack grows on demand (`default_stacksize` is platform depended)

attrs attributes for `operator()`

ret only `void` allowed (use of capture list instead)

body function body

2) copies `std::execution_context`, e.g. underlying capture record is shared

3) moves underlying capture record to new `std::execution_context`

Notes
If an instance of `std::execution_context` is copied, both instances share the same underlying capture record.

(destructor) destroys an execution context

`~execution_context()`	(1)

1) destroys a `std::execution_context`. If associated with a context of execution and holds the last reference to the internal capture record, then the context of execution is destroyed too. Specifically, the stack is unwound.

operator= copies/moves the coroutine object

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

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

2) copies the state of *other* to **this*, state (capture record) is shared

Parameters

other another execution context to assign to this object

Return value

***this**

operator() jump context of execution

```
execution_context& operator()()   (1)
```

1) suspends the active context, resumes the execution context

Exceptions

1) re-throws the exception of the resumed context in the parent context

Notes
An exception thrown inside execution context is caught, the parent execution context is resumed and the exception is re-thrown in the parent context (out of `operator()()`).
If the toplevel context function terminates (reaches end), the parent context is resumed (return of `operator()()` in the parent execution context).
The behaviour is undefined if `operator()` is called while `current()` returns **this* (attempting to resume an already running context).

explicit bool operator test if context has not reached its end

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

1) returns *true* if context is not terminated

Exceptions

1) noexcept specification: `noexcept`

operator! test if context has reached its end

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

1) returns *true* if context is terminated

Exceptions

1) noexcept specification: `noexcept`

current accesses the current active execution context

```
static execution_context current()   (1)
```

1) construct an instance of `std::execution_context` pointing to the capture record of the current, active execution context

Notes
The current active execution context is thread-specific, e.g. for each thread (including `main()`) an execution context is created at start-up.

Acknowledgement

I'd like to thank Nat Goodspeed for reviewing earlier drafts of this proposal.

References

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

[2] N4024: Distinguishing coroutines and fibers

[3] N4134: Resumable Functions v.2

[4] N4244: Resumable Lambdas

[5] N4398: A unified syntax for stackless and stackful coroutines

[6] Split Stacks / GCC

[7] Segmented Stacks / LLVM

[8] Library *Boost.Context*: git repo, documentation

[9] Library *Boost.Coroutine2*: git repo, documentation

[10] Library *Boost.Fiber*: git repo, documentation

[11] *cactus stack*: cactus stack, parent pointer tree

A. recursive descent parser using `std::execution_context`

This example (taken from Boost.Context[8]) uses a recursive descent parser for processing a stream of characters. The parser (a SAX parser or other event-dispatching framework would be equivalent) uses callbacks to push parsed data to the user code. With `std::execution_context` the user code inverts the control-flow. In this case, the user code *pulls* data from the parser.

```
// N4397: stackful execution context
// grammar:
//   P ---> E '\0'
//   E ---> T {('+'|'-') T}
//   T ---> S {('*'|'/') S}
//   S ---> digit | '(' E ')'
class Parser{
   int level;
   char next;
   istream& is;
   function<void(char)> cb;

   char pull(){
```

```
        return char_traits<char>::to_char_type(is.get());
    }

    void scan(){
        do{
            next=pull();
        }
        while(isspace(next));
    }

public:
    Parser(istream& is_,function<void(char)> cb_) :
        level(0),next(),is(is_),cb(cb_)
     {}

    void run() {
        scan();
        E();
    }

private:
    void E(){
        T();
        while (next=='+'||next=='-'){
            cb(next); // callback; signal new symbol
            scan();
            T();
        }
    }

    void T(){
        S();
        while (next=='*'||next=='/'){
            cb(next); // callback; signal new symbol
            scan();
            S();
        }
    }

    void S(){
        if (isdigit(next)){
            cb(next); // callback; signal new symbol
            scan();
        }
        else if(next=='('){
            cb(next); // callback; signal new symbol
            scan();
            E();
            if (next==')'){
                cb(next); // callback; signal new symbol
                scan();
            }else{
                throw std::runtime_error("parsing failed");
            }
        }
        else{
```

```
            throw std::runtime_error("parsing failed");
        }
    }
};

int main(){
    std::istringstream is("1+1");
    char c;
    // access current execution context
    auto m=std::execution_context::current();
    // use of linked stack (grows on demand) with initial size of 1KB
    std::execution_context l(
    auto l=[&is,&m,&c]()resumable(segmented(1024)){
            Parser p(is,
                    // callback, used to signal new symbol
                    [&m,&c](char ch){
                       c=ch;
                       m(); // resume main-context
                    });
            p.run(); // start parsing
        });
    try{
        // inversion of control: user-code pulls parsed symbols from parser
        while(l()){
            std::cout<<"Parsed: "<<c<<std::endl;
        }
    }catch(const std::exception& e){
        std::cerr<<"exception: "<<e.what()<<std::endl;
    }
}
```

Code like this example is not possible without transforming all functions in the call chain into resumable functions - which is impossible for closed source code or for a too large/complex code base.