Document number: N4397
Date: 2015-03-23
Project: WG21, SG1
Author: Oliver Kowalke (oliver dot kowalke at gmail dot com)

A low-level API for a stackful coroutine

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 coroutine, 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 syntax
- symmetric transfer of execution control, e.g. suspend-by-call - enables a richer set of control flows than asymmetric transfer of comtrol (e.g. suspend-by-return as described in N4134)
- benefits of traditional stack management retained
- ordinary function calls and returns not affected
- working reference implementation in boost.context[8]

Background

At the 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 that stackless and stackful coroutines are distinct use cases and decided to pursue both kinds of coroutines.
The committee encouraged the authors of the three proposals 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 where program logic is evaluated in (CPU registers, stack).

Activation record also known as *activation frame*, *control block* (*stack frame* is a special activation record). An *activation record* is a data structure containing the state of a particular function call. Folowing data are part of a activation record:

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

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

Heap-allocated activation record each activation record is allocated on a separate heap. The activation record holds a pointer to its parent activation record (parent pointer tree).

Linear stack is a contiguous stack and the traditional stack model of C++. The allocation, deallocation and linkage of activation records (stack frames in this context) are done by incrementing/decrementing the stack pointer.

Non-contiguous stack is a *parent pointer tree*-structure used to store activation records. It is a stack with branches, e.g. activation records can outlive the context they were created in.

*Application Binary Interface

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

- an application's inital (or only) thread
- an explicitly-launched thread
- a sub-thread execution context (coroutine)

The processor stack assigned to function `main()` is a linear (contiguous) stack. Often, the memory managment system is used to detect a stack overflow (and allocate more memory). Stacks assigned to threads can not easily extended if an overflow occurs; instead stacks with a fixed size are used - usually 1MBs (Windows) till 2MB (Linux), but some platforms use smaller stacks (64KB on HP-UX/PA and 256KB on HP-UX/Itanium).

Side stack is a call stack that is used in the case of stackful context switching, e.g. each execution context gets its own stack (assigned to stack pointer). Thus stack frames of subroutines are allocated on the side stack, the application stack remains unchanged.

Linked stack also known as *split stack*[6] or *segmented stack*,[7] is a non-contiguous stack, that grows on demand. An overflow handler allocates new chunk of memory as necessary. It has a low performance penalty - ca. 5-10% overhead.
Applications compiled with support for linked stacks can use (link against) libraries not supporting linked stacks (see GCC's documentation,[6] chapter 'Backward compatibility').

Parent pointer tree data structure in which each node has a pointer to its parent, but no pointer to its children. Traversion from any node to its ancestors is possible but not to any other node.
Used as call stack, the structure is called *cactus stack*.

Coroutine enables explicit suspend and resume of its progress via additional operations by preserving execution state (*activation record*) and thus provides an enhanced control flow. Coroutines have following characteristics:[1]

- values of local data persist between successive 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 by a coroutine (stackless or stackful).

Asymmetric coroutine provides two distingt 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, e.g. suspending the coroutine transferes the execution control back to the point in the code were it was called. The asymmetric control transfer-mechanism is usually used in the context of generators.

Symmetric coroutine only one operation to resume/suspend the context is available.
A symmetric coroutine does not know its caller, e.g. the execution control can be transfered to any other symmetric coroutine (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]).

Resumable function N4134[3] describes resumable functions as an efficient language supported mechanism for stackless context switching introducing two new keywords - `await` and `yield`. Resumable functions are equivalent to asymmetric coroutines.
Characteristics of resumable functions:

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

Introduction

In the past multiple proposals were published to support coroutines in C++.
It is astonishing that each proposal distinguishes coroutines into *two* kinds: *stackless* and *stackful*.

Traditionally C++ code is compiled on a linear stack, e.g. 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, e.g. switching between different execution context's, a linear stack introduces problems. Calling a function creates a activation record on the stack which is removed if the function returns. But for a suspended coroutine the activation record **must not** be **removed**! Additionaly, if the activation record of a suspended coroutine remains on the linear processor stack and other code uses the stack in the meanwhile, then all stack frames allocated after suspending the coroutine might be corrupted, if the coroutine is resumed.
In order to prevent stack corruption stackless coroutines use one heap-allocated activation record (N4134) while stackful coroutines use a side stack. Because stackless coroutines use only one heap-allocated activation record for their toplevel context function (e.g. body of resumable function) but leave the stack pointer unchanged (points still into linear processor stack), called subroutines use the linear processor stack.
Because stackful coroutines modify the stack pointer (pointing into coroutines own side stack), the activation records of the toplevel context function as well as the activation records of called subroutines are allocated on the side stack, not on the processor stack.
This is the fundamental difference between stackless and stackful coroutines.

Traditional stack management is inadequate for coroutines.

Discussion

Memory requirements In the case simple toplevel context-functions, 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 allocate 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 (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). Activation record of the toplevel context function is stored on the side stack, thus the application 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 the 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.

It is obvious that the memory requirements for both types of coroutines are equal.

Calling subroutines Stackless coroutines do not need to allocate memory for stack frames for called subroutines while (in contrast) stackful context switching permits **suspending from nested call stack**.

Resumable functions share the same processor stack, e.g. the stack pointer is unchanged and holds an address pointing into the processor stack.
If a subroutine is invoked inside the body of a resumable function, then the activation record belonging to the subroutine is allocated on the processor stack (so it is called stack frame). As a consequence stack frames of subroutines have to be removed from the processor stack before the resumable function yields back to its caller (suspend to its resumption point). Otherwise the stack frames would be corrupted (overwritten) by other code using the processor stack.
In other words: the calling convention of the ABI dictates that, after the resumable function returns (suspended), the stack pointer contains the same address as before the resumable function was resumed.
Hence a yield from nested call stack is **not permitted**.
The benefit of stackless coroutines consists in sharing the processor stack for called subroutines so that no additional memory for the stack frames of those subroutines has to be allocated.

Of course even stackless resumable functions might fail if the available stack overflows.

*depending on the calling convention, *x86* requires to store parameters and return address on the stack - other architectures/ABIs do not have this requirement

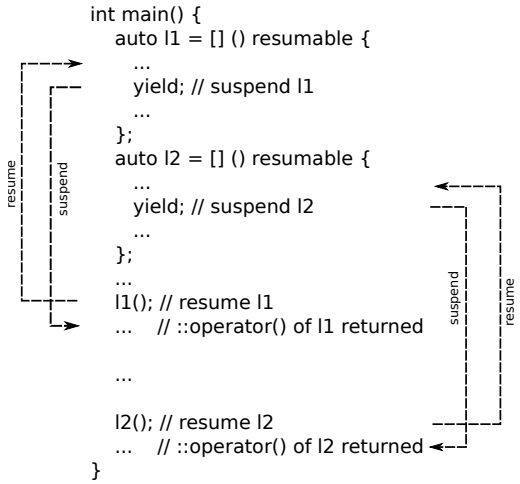
In the context of stackful context switching each execution context owns a distinct side stack which is assigned to the stack pointer (e.g. the stack pointer has to be exchanged during each context switch).
As a consequence, all activation records (stack frames) of subroutines are placed on the side stack. Hence each stackful execution context requires enough memory to hold the stack frames of the longest call chain of subroutines. Therefore stackful context switching has a higher memory footprint than resumable functions (but this applies only in the context of calling subroutines).
On the other side, it is beneficial to use side stacks because the stack frames of subroutines remain intact while the execution context gets suspended. This is the reason why stackful context switching permits **yielding from nested call stack**.

```
// 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 that *l2* owns.

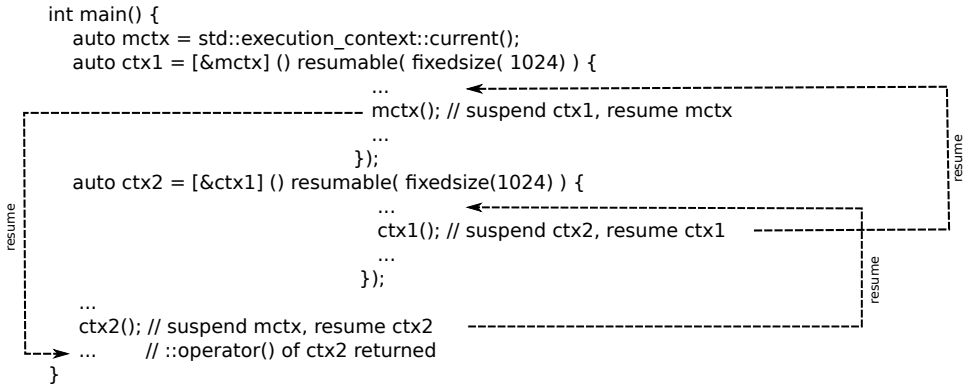
Asymmetric vs. symmetric Symmetric transfer of execution control is more efficient than the asymmetric mechanism.

Resumable functions (as proposed in N4134) provide two operations for context switching (asymmetric transfer of execution control). The caller and the resumable function are coupled, e.g. a resumable function can only jump back to the point in the code where it was resumed.



For N resumable functionss $2N$ 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()()`). The execution control is directly transferred from one execution context to another (symmetric transfer-mechanism) - no jump back to the caller. This enables a efficient implementation of cooperative multitasking (no additional context switch back to caller, e.g. direct context switch to next task) as well as the implementation of generators.
As an consequence, the next execution context has to 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 how data are passed into the body is not intuitive and provides some problems.

During the use of `execution_context` from boost.context[8] a common pattern was to pass lambdas to the execution context and use the capture list to transfer data into and to return data from the body. Exceptions are transferred via `std::exception_ptr` and parameters (input/output) are accessed via captured addresses (reference or pointer).

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

public:
```

```
    X():
        inp_(nullptr),outp_(),excptr_(),
        caller_(std::execution_context::current()),
        callee_([=]()resumable(segmented(1024)){
                    try{
                        outp_=lexical_cast<std::string>(*inp_);
                        caller_(); // context switch to main()
                    }catch (...){
                        excptr_=std::current_exception();
                    }
                })
    {}

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

int main(){
    X x;
    std::cout<<x(7)<<std::endl;
}
```

Class `X` (rudimentary coroutine) demonstrates how input and output parameter are transferred between a context switch. 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). Exceptions, thrown by the conversion, are transported via member variable *exptr_*.

Stack strategies for stackful coroutines two strategies apply - the use of a contiguous, fixed-size stack (this strategy is used by threads) or use of a non-contiguous (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 on behalf of an overflow handler. The performance penalty is low - ca. 5-10% overhead. The disadvantage is that the code, executed inside a stackful coroutine, needs to be rebuild for this purpose. In the case of GCC's split-stacks special compiler/linker flags have be specified - no changes on the source code are required.
In the case of calling a library function not compiled for linked stacks (e.g. expecting a traditional contiguous stack) GCC's implementation let change the linker the instructions in the caller function. The effect is that the required stacksize is increased large enough to reasonably work. This part of the stack will be released when the toplevel context function returns.

Design

Class `std::execution_context` is derived from the work on boost.context[8] - it provides a small, basic API, suitable to implement high-level APIs for 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]).

Class `std::execution_context` Based on the implementation experience with `execution_context` in boost.coroutine2[9] and boost.fiber,[10] the author noticed that `execution_context` is almost always used

together with lambdas (passed as argument to the constructor of `execution_context`). Especially the lambda captures are suitable to transport data between different execution context's (address of lvalue, reside on stack/heap).
Why not construct `std::execution_context` with an 'resumable lambda'-like syntax, instead of passing a lambda as 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 an 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 the control of execution) the target context must be explicitly specified.
Exchanging data between different execution context's 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 addtional data like stack pointer, instruction pointer and a link to its parent execution context. That means that during a execution 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 The pointer to its *parent* execution context allows to traverse the chain of ancestor context's, e.g. the execution context which has resumed (`operator()()` called) the running context.

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

Toplevel capture records At entering `main()` as well as the *thread-function* of a thread, a execution context (capture record) is created and stored in the internally. `current()`.

Suspend-on-call `std::execution_context` is designed to suspend on call of `operator()()`. The *prolog* of `operator()()` captures (preserves) the active execution context and loads the data from the capture record of the resumed context (e.g. `operator()()` called on **this*) into the CPU.
The previous running context becomes the parent of the resumed context and gets suspended. Thus forming a parent-child relationship between suspended and 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 *epilog* of `operator()()` loads the capture record (instruction pointer, stack pointer etc.) of the parent context, so that it is resumed.

Exceptions A exception thrown inside the execution context is catched, the parent execution context is resumed and the exception is re-thrown inside the parent context (e.g. the exception is emitted by `operator()()`).

member functions

(constructor) constructs new execution context

`[captures](params) mutable resumable(hint) exceptions 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)

exceptions only `noexcept` allowed; emitted exceptions trigger `std::terminate()`

attrs attributes for `operator()`

ret only `void` allowed; resumable lambda returns nothing (use of capture list instead)

body function body

2) copies `std::execution_context`, e.g. underlying control block is shared

3) moves underlying control block to new `std::execution_context`

Notes

If an instance of `std::execution_context` is copied, both instances share the same underlying control block. Resuming one instance modifies the control block of the other `std::execution_context` too.

(destructor) destroys a 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 control block, 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 (control block) 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) resumes the execution context

Exceptions

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

Notes
A exception thrown inside execution context is catched, the parent execution context is resumed and the exception is re-throw 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* (e.g. resuming 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 a 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()`) a 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

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.