Coroutine

Oliver Kowalke

Table of Contents

Overview

Boost.Coroutine provides templates for generalized subroutines which allow suspending and resuming execution at certain locations. It preserves the local state of execution and allows re-entering subroutines more than once (useful if state must be kept across function calls).

Coroutines can be viewed as a language-level construct providing a special kind of control flow.

In contrast to threads, which are pre-emptive, *coroutine* switches are cooperative (programmer controls when a switch will happen). The kernel is not involved in the coroutine switches.

The implementation uses **Boost.Context** for context switching.

In order to use the classes and functions described here, you can either include the specific headers specified by the descriptions of each class or function, or include the master library header:

```
#include <boost/coroutine/all.hpp>
```

which includes all the other headers in turn.

All functions and classes are contained in the namespace *boost::coroutines*.

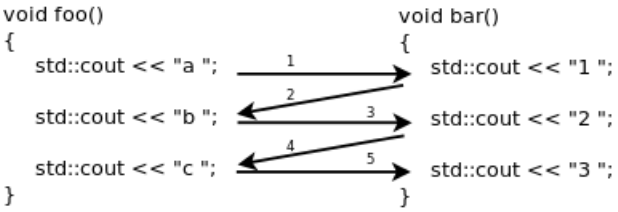
Introduction

Definition

In computer science routines are defined as a sequence of operations. The execution of routines forms a parent-child relationship and the child terminates always before the parent. Coroutines (the term was introduced by Melvin Conway [1]), are a generalization of routines (Donald Knuth [2]. The principal difference between coroutines and routines is that a coroutine enables explicit suspend and resume of its progress via additional operations by preserving execution state and thus provides an **enhanced control flow** (maintaining the execution context).

How it works

Functions foo() and bar() are supposed to alternate their execution (leave and enter function body).




```
output:
   a 1 b 2 c 3

int main()
{ ? }
```

If coroutines were called exactly like routines, the stack would grow with every call and would never be popped. A jump into the middle of a coroutine would not be possible, because the return address would be on top of stack entries.

The solution is that each coroutine has its own stack and control-block (*boost::contexts::fcontext_t* from **Boost.Context**). Before the coroutine gets suspended, the non-volatile registers (including stack and instruction/program pointer) of the currently active coroutine are stored in the coroutine's control-block. The registers of the newly activated coroutine must be restored from its associated control-block before it is resumed.

The context switch requires no system privileges and provides cooperative multitasking convenient to C++. Coroutines provide quasi parallelism. When a program is supposed to do several things at the same time, coroutines help to do this much more simply and elegantly than with only a single flow of control. The advantages can be seen particularly clearly with the use of a recursive function, such as traversal of binary trees (see example 'same fringe').

characteristics

Characteristics [3] of a coroutine are:

- values of local data persist between successive calls (context switches)
- execution is suspended as control leaves coroutine and is resumed at certain time later
- symmetric or asymmetric control-transfer mechanism; see below
- first-class object (can be passed as argument, returned by procedures, stored in a data structure to be used later or freely manipulated by the developer)
- stackful or stackless

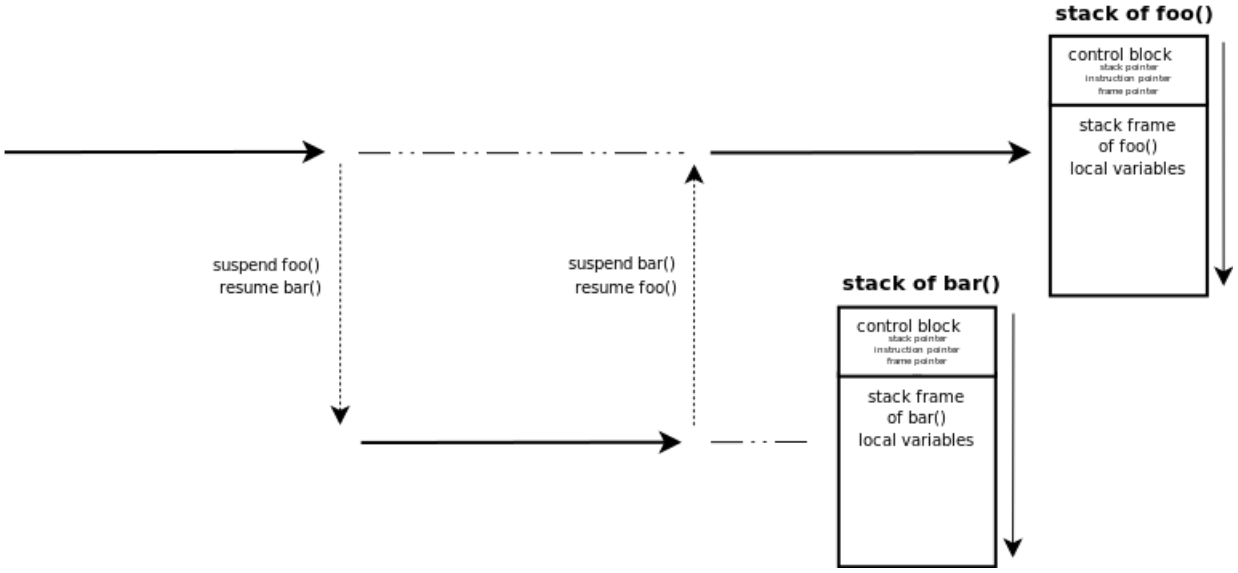
[1] Conway, Melvin E.. "Design of a Separable Transition-Diagram Compiler". Commun. ACM, Volume 6 Issue 7, July 1963, Article No. 7
[2] Knuth, Donald Ervin (1997). "Fundamental Algorithms. The Art of Computer Programming 1", (3rd ed.)
[3] Moura, Ana Lucia De and Ierusalimschy, Roberto. "Revisiting coroutines". ACM Trans. Program. Lang. Syst., Volume 31 Issue 2, February 2009, Article No. 6

Coroutines are useful in simulation, artificial intelligence, concurrent programming, text processing and data manipulation, supporting the implementation of components such as cooperative tasks (fibers), iterators, generators, infinite lists, pipes etc.

execution-transfer mechanism

Two categories of coroutines exist: symmetric and asymmetric coroutines.

An asymmetric coroutine knows its invoker, using a special operation to implicitly yield control specifically to its invoker. By contrast, all symmetric coroutines are equivalent; one symmetric coroutine may pass control to any other symmetric coroutine. Because of this, a symmetric coroutine *must* specify the coroutine to which it intends to yield control.



Both concepts are equivalent and a fully-general coroutine library can provide either symmetric or asymmetric coroutines. For convenience, Boost.Coroutine provides both.

stackfulness

In contrast to a stackless coroutine a stackful coroutine can be suspended from within a nested stackframe. Execution resumes at exactly the same point in the code where it was suspended before. With a stackless coroutine, only the top-level routine may be suspended. Any routine called by that top-level routine may not itself suspend. This prohibits providing suspend/resume operations in routines within a general-purpose library.

first-class continuation

A first-class continuation can be passed as an argument, returned by a function and stored in a data structure to be used later. In some implementations (for instance C# *yield*) the continuation can not be directly accessed or directly manipulated.

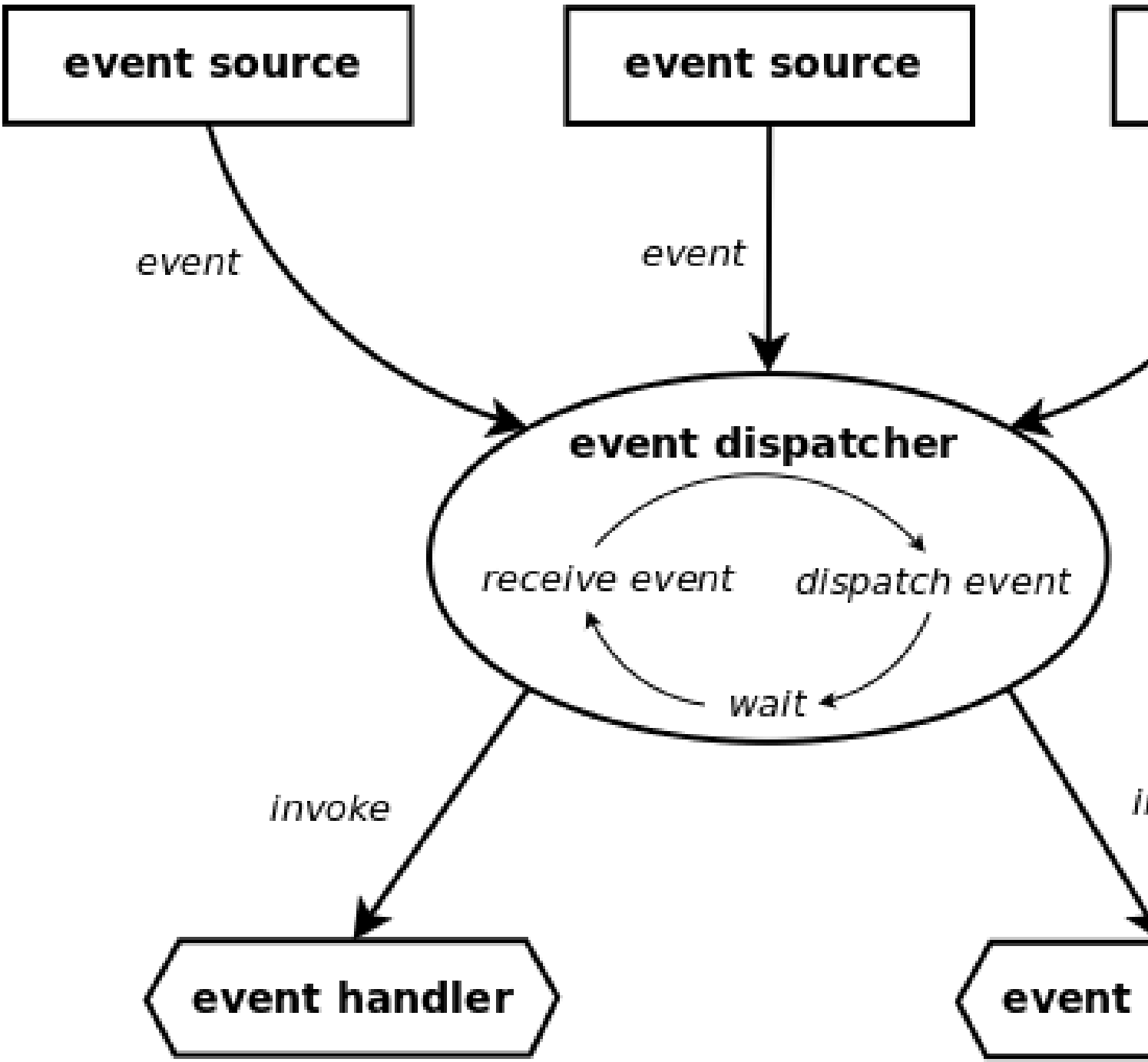
Without stackfulness and first-class semantics, some useful execution control flows cannot be supported (for instance cooperative multitasking or checkpointing).

Motivation

In order to support a broad range of execution control behaviour the coroutine types of *symmetric_coroutine<>* and *asymmetric_coroutine<>* can be used to *escape-and-reenter* loops, to *escape-and-reenter* recursive computations and for *cooperative* multitasking helping to solve problems in a much simpler and more elegant way than with only a single flow of control.

event-driven model

The event-driven model is a programming paradigm where the flow of a program is determined by events. The events are generated by multiple independent sources and an event-dispatcher, waiting on all external sources, triggers callback functions (event-handlers) whenever one of those events is detected (event-loop). The application is divided into event selection (detection) and event handling.



The resulting applications are highly scalable, flexible, have high responsiveness and the components are loosely coupled. This makes the event-driven model suitable for user interface applications, rule-based productions systems or applications dealing with asynchronous I/O (for instance network servers).

event-based asynchronous paradigm

A classic synchronous console program issues an I/O request (e.g. for user input or filesystem data) and blocks until the request is complete.

In contrast, an asynchronous I/O function initiates the physical operation but immediately returns to its caller, even though the operation is not yet complete. A program written to leverage this functionality does not block: it can proceed with other work (including other I/O requests in parallel) while the original operation is still pending. When the operation completes, the program is notified. Because asynchronous applications spend less overall time waiting for operations, they can outperform synchronous programs.

Events are one of the paradigms for asynchronous execution, but not all asynchronous systems use events. Although asynchronous programming can be done using threads, they come with their own costs:

- hard to program (traps for the unwary)
- memory requirements are high
- large overhead with creation and maintenance of thread state
- expensive context switching between threads

The event-based asynchronous model avoids those issues:

- simpler because of the single stream of instructions
- much less expensive context switches

The downside of this paradigm consists in a sub-optimal program structure. An event-driven program is required to split its code into multiple small callback functions, i.e. the code is organized in a sequence of small steps that execute intermittently. An algorithm that would usually be expressed as a hierarchy of functions and loops must be transformed into callbacks. The complete state has to be stored into a data structure while the control flow returns to the event-loop. As a consequence, event-driven applications are often tedious and confusing to write. Each callback introduces a new scope, error callback etc. The sequential nature of the algorithm is split into multiple callstacks, making the application hard to debug. Exception handlers are restricted to local handlers: it is impossible to wrap a sequence of events into a single try-catch block. The use of local variables, while/for loops, recursions etc. together with the event-loop is not possible. The code becomes less expressive.

In the past, code using asio's *asynchronous operations* was convoluted by callback functions.

```
class session
{
public:
    session(boost::asio::io_service& io_service) :
          socket_(io_service) // construct a TCP-socket from io_service
    {}

    tcp::socket& socket(){
        return socket_;
    }

    void start(){
        // initiate asynchronous read; handle_read() is callback-function
        socket_.async_read_some(boost::asio::buffer(data_,max_length),
            boost::bind(&session::handle_read,this,
                boost::asio::placeholders::error,
                boost::asio::placeholders::bytes_transferred));
    }

private:
    void handle_read(const boost::system::error_code& error,
                     size_t bytes_transferred){
        if (!error)
            // initiate asynchronous write; handle_write() is callback-function
            boost::asio::async_write(socket_,
                boost::asio::buffer(data_,bytes_transferred),
                boost::bind(&session::handle_write,this,
                    boost::asio::placeholders::error));
        else
            delete this;
    }

    void handle_write(const boost::system::error_code& error){
        if (!error)
            // initiate asynchronous read; handle_read() is callback-function
            socket_.async_read_some(boost::asio::buffer(data_,max_length),
                boost::bind(&session::handle_read,this,
                    boost::asio::placeholders::error,
                    boost::asio::placeholders::bytes_transferred));
        else
            delete this;
    }

    boost::asio::ip::tcp::socket socket_;
    enum { max_length=1024 };
    char data_[max_length];
};
```

In this example, a simple echo server, the logic is split into three member functions - local state (such as data buffer) is moved to member variables.

Boost.Asio provides with its new *asynchronous result* feature a new framework combining event-driven model and coroutines, hiding the complexity of event-driven programming and permitting the style of classic sequential code. The application is not required to pass callback functions to asynchronous operations and local state is kept as local variables. Therefore the code is much easier to read and understand. [4]. *boost::asio::yield_context* internally uses **Boost.Coroutine**:

[4] Christopher Kohlhoff, N3964 - Library Foundations for Asynchronous Operations, Revision 1

```
void session(boost::asio::io_service& io_service){
    // construct TCP-socket from io_service
    boost::asio::ip::tcp::socket socket(io_service);

    try{
        for(;;){
            // local data-buffer
            char data[max_length];

            boost::system::error_code ec;

            // read asynchronous data from socket
            // execution context will be suspended until
            // some bytes are read from socket
            std::size_t length=socket.async_read_some(
                    boost::asio::buffer(data),
                    boost::asio::yield[ec]);
            if (ec==boost::asio::error::eof)
                break; //connection closed cleanly by peer
            else if(ec)
                throw boost::system::system_error(ec); //some other error

            // write some bytes asynchronously
            boost::asio::async_write(
                    socket,
                    boost::asio::buffer(data,length),
                    boost::asio::yield[ec]);
            if (ec==boost::asio::error::eof)
                break; //connection closed cleanly by peer
            else if(ec)
                throw boost::system::system_error(ec); //some other error
        }
    } catch(std::exception const& e){
        std::cerr<<"Exception: "<<e.what()<<"\n";
    }
}
```

In contrast to the previous example this one gives the impression of sequential code and local data (*data*) while using asynchronous operations (*async_read()*, *async_write()*). The algorithm is implemented in one function and error handling is done by one try-catch block.

recursive SAX parsing

To someone who knows SAX, the phrase "recursive SAX parsing" might sound nonsensical. You get callbacks from SAX; you have to manage the element stack yourself. If you want recursive XML processing, you must first read the entire DOM into memory, then walk the tree.

But coroutines let you invert the flow of control so you can ask for SAX events. Once you can do that, you can process them recursively.

```
// Represent a subset of interesting SAX events
struct BaseEvent{
   BaseEvent(const BaseEvent&)=delete;
   BaseEvent& operator=(const BaseEvent&)=delete;
};

// End of document or element
struct CloseEvent: public BaseEvent{
   // CloseEvent binds (without copying) the TagType reference.
   CloseEvent(const xml::sax::Parser::TagType& name):
       mName(name)
   {}

   const xml::sax::Parser::TagType& mName;
};

// Start of document or element
struct OpenEvent: public CloseEvent{
   // In addition to CloseEvent's TagType, OpenEvent binds AttributeIterator.
   OpenEvent(const xml::sax::Parser::TagType& name,
             xml::sax::AttributeIterator& attrs):
       CloseEvent(name),
       mAttrs(attrs)
   {}

   xml::sax::AttributeIterator& mAttrs;
};

// text within an element
struct TextEvent: public BaseEvent{
   // TextEvent binds the CharIterator.
   TextEvent(xml::sax::CharIterator& text):
       mText(text)
   {}

   xml::sax::CharIterator& mText;
};

// The parsing coroutine instantiates BaseEvent subclass instances and
// successively shows them to the main program. It passes a reference so we
// don't slice the BaseEvent subclass.
typedef boost::coroutines::asymmetric_coroutine<const BaseEvent&> coro_t;

void parser(coro_t::push_type& sink,std::istream& in){
   xml::sax::Parser xparser;
   // startDocument() will send OpenEvent
   xparser.startDocument([&sink](const xml::sax::Parser::TagType& name,
                                 xml::sax::AttributeIterator& attrs)
                         {
                             sink(OpenEvent(name,attrs));
                         });
   // startTag() will likewise send OpenEvent
   xparser.startTag([&sink](const xml::sax::Parser::TagType& name,
                            xml::sax::AttributeIterator& attrs)
                    {
                        sink(OpenEvent(name,attrs));
                    });
   // endTag() will send CloseEvent
   xparser.endTag([&sink](const xml::sax::Parser::TagType& name)
                  {
                      sink(CloseEvent(name));
                  });
   // endDocument() will likewise send CloseEvent
```

```
    xparser.endDocument([&sink](const xml::sax::Parser::TagType& name)
                        {
                            sink(CloseEvent(name));
                        });
    // characters() will send TextEvent
    xparser.characters([&sink](xml::sax::CharIterator& text)
                       {
                           sink(TextEvent(text));
                       });
    try
    {
        // parse the document, firing all the above
        xparser.parse(in);
    }
    catch (xml::Exception e)
    {
        // xml::sax::Parser throws xml::Exception. Helpfully translate the
        // name and provide it as the what() string.
        throw std::runtime_error(exception_name(e));
    }
}

// Recursively traverse the incoming XML document on the fly, pulling
// BaseEvent& references from 'events'.
// 'indent' illustrates the level of recursion.
// Each time we're called, we've just retrieved an OpenEvent from 'events';
// accept that as a param.
// Return the CloseEvent that ends this element.
const CloseEvent& process(coro_t::pull_type& events,const OpenEvent& context,
                          const std::string& indent=""){
    // Capture OpenEvent's tag name: as soon as we advance the parser, the
    // TagType& reference bound in this OpenEvent will be invalidated.
    xml::sax::Parser::TagType tagName = context.mName;
    // Since the OpenEvent is still the current value from 'events', pass
    // control back to 'events' until the next event. Of course, each time we
    // come back we must check for the end of the results stream.
    while(events()){
        // Another event is pending; retrieve it.
        const BaseEvent& event=events.get();
        const OpenEvent* oe;
        const CloseEvent* ce;
        const TextEvent* te;
        if((oe=dynamic_cast<const OpenEvent*>(&event))){
            // When we see OpenEvent, recursively process it.
            process(events,*oe,indent+"    ");
        }
        else if((ce=dynamic_cast<const CloseEvent*>(&event))){
            // When we see CloseEvent, validate its tag name and then return
            // it. (This assert is really a check on xml::sax::Parser, since
            // it already validates matching open/close tags.)
            assert(ce->mName == tagName);
            return *ce;
        }
        else if((te=dynamic_cast<const TextEvent*>(&event))){
            // When we see TextEvent, just report its text, along with
            // indentation indicating recursion level.
            std::cout<<indent<<"text: '"<<te->mText.getText()<<"'\n";
        }
    }
}

// pretend we have an XML file of arbitrary size
std::istringstream in(doc);
```

```
try
{
   coro_t::pull_type events(std::bind(parser,_1,std::ref(in)));
   // We fully expect at least ONE event.
   assert(events);
   // This dynamic_cast<&> is itself an assertion that the first event is an
   // OpenEvent.
   const OpenEvent& context=dynamic_cast<const OpenEvent&>(events.get());
   process(events, context);
}
catch (std::exception& e)
{
   std::cout << "Parsing error: " << e.what() << '\n';
}
```

This problem does not map at all well to communicating between independent threads. It makes no sense for either side to proceed independently of the other. You want them to pass control back and forth.

The solution involves a small polymorphic class event hierarchy, to which we're passing references. The actual instances are temporaries on the coroutine's stack; the coroutine passes each reference in turn to the main logic. Copying them as base-class values would slice them.

If we were trying to let the SAX parser proceed independently of the consuming logic, one could imagine allocating event-subclass instances on the heap, passing them along on a thread-safe queue of pointers. But that doesn't work either, because these event classes bind references passed by the SAX parser. The moment the parser moves on, those references become invalid.

Instead of binding a *TagType&* reference, we could store a copy of the *TagType* in *CloseEvent*. But that doesn't solve the whole problem. For attributes, we get an *AttributeIterator&*; for text we get a *CharIterator&*. Storing a copy of those iterators is pointless: once the parser moves on, those iterators are invalidated. You must process the attribute iterator (or character iterator) during the SAX callback for that event.

Naturally we could retrieve and store a copy of every attribute and its value; we could store a copy of every chunk of text. That would effectively be all the text in the document -- a heavy price to pay, if the reason we're using SAX is concern about fitting the entire DOM into memory.

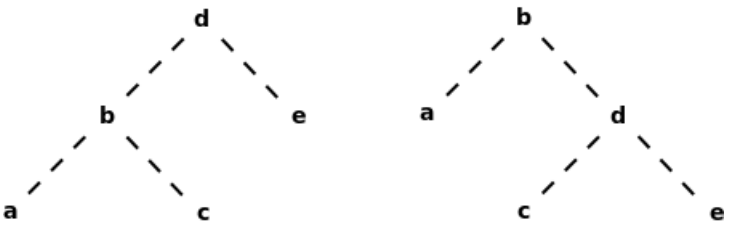
There's yet another advantage to using coroutines. This SAX parser throws an exception when parsing fails. With a coroutine implementation, you need only wrap the calling code in try/catch.

With communicating threads, you would have to arrange to catch the exception and pass along the exception pointer on the same queue you're using to deliver the other events. You would then have to rethrow the exception to unwind the recursive document processing.

The coroutine solution maps very naturally to the problem space.

'same fringe' problem

The advantages of suspending at an arbitrary call depth can be seen particularly clearly with the use of a recursive function, such as traversal of trees. If traversing two different trees in the same deterministic order produces the same list of leaf nodes, then both trees have the same fringe.



Both trees in the picture have the same fringe even though the structure of the trees is different.

The same fringe problem could be solved using coroutines by iterating over the leaf nodes and comparing this sequence via *std::equal()*. The range of data values is generated by function *traverse()* which recursively traverses the tree and passes each node's data value to its *asymmetric_coroutine<>::push_type*. *asymmetric_coroutine<>::push_type* suspends the recursive computation and transfers the data value to the main execution context. *asymmetric_coroutine<>::pull_type::iterator*, created from *asymmetric_coroutine<>::pull_type*, steps over those data values and delivers them to *std::equal()* for comparison. Each increment of *asymmetric_coroutine<>::pull_type::iterator* resumes *traverse()*. Upon return from *iterator::operator++()*, either a new data value is available, or tree traversal is finished (iterator is invalidated).

In effect, the coroutine iterator presents a flattened view of the recursive data structure.

```
struct node{
    typedef boost::shared_ptr<node> ptr_t;

    // Each tree node has an optional left subtree,
    // an optional right subtree and a value of its own.
    // The value is considered to be between the left
    // subtree and the right.
    ptr_t       left,right;
    std::string value;

    // construct leaf
    node(const std::string& v):
        left(),right(),value(v)
    {}
    // construct nonleaf
    node(ptr_t l,const std::string& v,ptr_t r):
        left(l),right(r),value(v)
    {}

    static ptr_t create(const std::string& v){
        return ptr_t(new node(v));
    }

    static ptr_t create(ptr_t l,const std::string& v,ptr_t r){
        return ptr_t(new node(l,v,r));
    }
};

node::ptr_t create_left_tree_from(const std::string& root){
    /* --------
         root
         / \
        b   e
       / \
      a   c
     -------- */
    return node::create(
            node::create(
                node::create("a"),
                "b",
                node::create("c")),
            root,
            node::create("e"));
}

node::ptr_t create_right_tree_from(const std::string& root){
    /* --------
         root
         / \
        a   d
           / \
          c   e
```

```
       -------- */
    return node::create(
            node::create("a"),
            root,
            node::create(
                node::create("c"),
                "d",
                node::create("e")));
}

// recursively walk the tree, delivering values in order
void traverse(node::ptr_t n,
              boost::coroutines::asymmetric_coroutine<std::string>::push_type& out){
    if(n->left) traverse(n->left,out);
    out(n->value);
    if(n->right) traverse(n->right,out);
}

// evaluation
{
    node::ptr_t left_d(create_left_tree_from("d"));
    boost::coroutines::asymmetric_coroutine<std::string>::pull_type left_d_reader(
        [&]( boost::coroutines::asymmetric_coroutine<std::string>::push_type & out){
            traverse(left_d,out);
        });

    node::ptr_t right_b(create_right_tree_from("b"));
    boost::coroutines::asymmetric_coroutine<std::string>::pull_type right_b_reader(
        [&]( boost::coroutines::asymmetric_coroutine<std::string>::push_type & out){
            traverse(right_b,out);
        });

    std::cout << "left tree from d == right tree from b? "
              << std::boolalpha
              << std::equal(boost::begin(left_d_reader),
                            boost::end(left_d_reader),
                            boost::begin(right_b_reader))
              << std::endl;
}
{
    node::ptr_t left_d(create_left_tree_from("d"));
    boost::coroutines::asymmetric_coroutine<std::string>::pull_type left_d_reader(
        [&]( boost::coroutines::asymmetric_coroutine<std::string>::push_type & out){
            traverse(left_d,out);
        });

    node::ptr_t right_x(create_right_tree_from("x"));
    boost::coroutines::asymmetric_coroutine<std::string>::pull_type right_x_reader(
        [&]( boost::coroutines::asymmetric_coroutine<std::string>::push_type & out){
            traverse(right_x,out);
        });

    std::cout << "left tree from d == right tree from x? "
              << std::boolalpha
              << std::equal(boost::begin(left_d_reader),
                            boost::end(left_d_reader),
                            boost::begin(right_x_reader))
              << std::endl;
}
```

```
std::cout << "Done" << std::endl;

output:
left tree from d == right tree from b? true
left tree from d == right tree from x? false
Done
```

merging two sorted arrays

This example demonstrates how symmetric coroutines merge two sorted arrays.

```
std::vector<int> merge(const std::vector<int>& a,const std::vector<int>& b){
    std::vector<int> c;
    std::size_t idx_a=0,idx_b=0;
    boost::coroutines::symmetric_coroutine<void>::call_type *other_a=0,*other_b=0;

    boost::coroutines::symmetric_coroutine<void>::call_type coro_a(
        [&](boost::coroutines::symmetric_coroutine<void>::yield_type& yield){
            while(idx_a<a.size()){
                if(b[idx_b]<a[idx_a])    // test if element in array b is less than in array a
                    yield(*other_b);     // yield to coroutine coro_b
                c.push_back(a[idx_a++]); // add element to final array
            }
            // add remaining elements of array b
            while(idx_b<b.size())
                c.push_back(b[idx_b++]);
        });

    boost::coroutines::symmetric_coroutine<void>::call_type coro_b(
        [&](boost::coroutines::symmetric_coroutine<void>::yield_type& yield){
            while(idx_b<b.size()){
                if(a[idx_a]<b[idx_b])    // test if element in array a is less than in array b
                    yield(*other_a);     // yield to coroutine coro_a
                c.push_back(b[idx_b++]); // add element to final array
            }
            // add remaining elements of array a
            while(idx_a<a.size())
                c.push_back(a[idx_a++]);
        });


    other_a=&coro_a;
    other_b=&coro_b;

    coro_a(); // enter coroutine-fn of coro_a

    return c;
}
```

chaining coroutines

This code shows how coroutines could be chained.

```
typedef boost::coroutines::asymmetric_coroutine<std::string> coro_t;

// deliver each line of input stream to sink as a separate string
void readlines(coro_t::push_type& sink,std::istream& in){
    std::string line;
    while(std::getline(in,line))
        sink(line);
}

void tokenize(coro_t::push_type& sink, coro_t::pull_type& source){
    // This tokenizer doesn't happen to be stateful: you could reasonably
    // implement it with a single call to push each new token downstream. But
    // I've worked with stateful tokenizers, in which the meaning of input
    // characters depends in part on their position within the input line.
    BOOST_FOREACH(std::string line,source){
        std::string::size_type pos=0;
        while(pos<line.length()){
            if(line[pos]=='"'){
                std::string token;
                ++pos;              // skip open quote
                while(pos<line.length()&&line[pos]!='"')
                    token+=line[pos++];
                ++pos;              // skip close quote
                sink(token);        // pass token downstream
            } else if (std::isspace(line[pos])){
                ++pos;              // outside quotes, ignore whitespace
            } else if (std::isalpha(line[pos])){
                std::string token;
                while (pos < line.length() && std::isalpha(line[pos]))
                    token += line[pos++];
                sink(token);        // pass token downstream
            } else {                // punctuation
                sink(std::string(1,line[pos++]));
            }
        }
    }
}

void only_words(coro_t::push_type& sink,coro_t::pull_type& source){
    BOOST_FOREACH(std::string token,source){
        if (!token.empty() && std::isalpha(token[0]))
            sink(token);
    }
}

void trace(coro_t::push_type& sink, coro_t::pull_type& source){
    BOOST_FOREACH(std::string token,source){
        std::cout << "trace: '" << token << "'\n";
        sink(token);
    }
}

struct FinalEOL{
    ~FinalEOL(){
        std::cout << std::endl;
    }
};

void layout(coro_t::pull_type& source,int num,int width){
    // Finish the last line when we leave by whatever means
    FinalEOL eol;

    // Pull values from upstream, lay them out 'num' to a line
```

```
    for (;;){
        for (int i = 0; i < num; ++i){
            // when we exhaust the input, stop
            if (!source) return;

            std::cout << std::setw(width) << source.get();
            // now that we've handled this item, advance to next
            source();
        }
        // after 'num' items, line break
        std::cout << std::endl;
    }
}

// For example purposes, instead of having a separate text file in the
// local filesystem, construct an istringstream to read.
std::string data(
    "This is the first line.\n"
    "This, the second.\n"
    "The third has \"a phrase\"!\n"
    );

{
    std::cout << "\nfilter:\n";
    std::istringstream infile(data);
    coro_t::pull_type reader(boost::bind(readlines, _1, boost::ref(infile)));
    coro_t::pull_type tokenizer(boost::bind(tokenize, _1, boost::ref(reader)));
    coro_t::pull_type filter(boost::bind(only_words, _1, boost::ref(tokenizer)));
    coro_t::pull_type tracer(boost::bind(trace, _1, boost::ref(filter)));
    BOOST_FOREACH(std::string token,tracer){
        // just iterate, we're already pulling through tracer
    }
}

{
    std::cout << "\nlayout() as coroutine::push_type:\n";
    std::istringstream infile(data);
    coro_t::pull_type reader(boost::bind(readlines, _1, boost::ref(infile)));
    coro_t::pull_type tokenizer(boost::bind(tokenize, _1, boost::ref(reader)));
    coro_t::pull_type filter(boost::bind(only_words, _1, boost::ref(tokenizer)));
    coro_t::push_type writer(boost::bind(layout, _1, 5, 15));
    BOOST_FOREACH(std::string token,filter){
        writer(token);
    }
}

{
    std::cout << "\nfiltering output:\n";
    std::istringstream infile(data);
    coro_t::pull_type reader(boost::bind(readlines,_1,boost::ref(infile)));
    coro_t::pull_type tokenizer(boost::bind(tokenize,_1,boost::ref(reader)));
    coro_t::push_type writer(boost::bind(layout,_1,5,15));
    // Because of the symmetry of the API, we can use any of these
    // chaining functions in a push_type coroutine chain as well.
    coro_t::push_type filter(boost::bind(only_words,boost::ref(writer),_1));
    BOOST_FOREACH(std::string token,tokenizer){
        filter(token);
    }
}
```

Coroutine

Boost.Coroutine provides two implementations - asymmetric and symmetric coroutines.

Symmetric coroutines occur usually in the context of concurrent programming in order to represent independent units of execution. Implementations that produce sequences of values typically use asymmetric coroutines. [5]

stackful

Each instance of a coroutine has its own stack.

In contrast to stackless coroutines, stackful coroutines allow invoking the suspend operation out of arbitrary sub-stackframes, enabling escape-and-reenter recursive operations.

move-only

A coroutine is moveable-only.

If it were copyable, then its stack with all the objects allocated on it would be copied too. That would force undefined behaviour if some of these objects were RAII-classes (manage a resource via RAII pattern). When the first of the coroutine copies terminates (unwinds its stack), the RAII class destructors will release their managed resources. When the second copy terminates, the same destructors will try to doubly-release the same resources, leading to undefined behaviour.

clean-up

On coroutine destruction the associated stack will be unwound.

The constructor of coroutine allows you to pass a customized *stack-allocator*. *stack-allocator* is free to deallocate the stack or cache it for future usage (for coroutines created later).

segmented stack

symmetric_coroutine<>::call_type, *asymmetric_coroutine<>::push_type* and *asymmetric_coroutine<>::pull_type* support segmented stacks (growing on demand).

It is not always possible to accurately estimate the required stack size - in most cases too much memory is allocated (waste of virtual address-space).

At construction a coroutine starts with a default (minimal) stack size. This minimal stack size is the maximum of page size and the canonical size for signal stack (macro SIGSTKSZ on POSIX).

At this time of writing only GCC (4.7) [6] is known to support segmented stacks. With version 1.54 **Boost.Coroutine** provides support for segmented stacks.

The destructor releases the associated stack. The implementer is free to deallocate the stack or to cache it for later usage.

context switch

A coroutine saves and restores registers according to the underlying ABI on each context switch (using **Boost.Context**).

Some applications do not use floating-point registers and can disable preserving FPU registers for performance reasons.

[5] Moura, Ana Lucia De and Ierusalimschy, Roberto. "Revisiting coroutines". ACM Trans. Program. Lang. Syst., Volume 31 Issue 2, February 2009, Article No. 6
[6] Ian Lance Taylor, Split Stacks in GCC

Note

According to the calling convention the FPU registers are preserved by default.

On POSIX systems, the coroutine context switch does not preserve signal masks for performance reasons.

A context switch is done via *symmetric_coroutine<>::call_type::operator()*, *asymmetric_coroutine<>::push_type::operator()* and *asymmetric_coroutine<>::pull_type::operator()*.

Warning

Calling *symmetric_coroutine<>::call_type::operator()*, *asymmetric_coroutine<>::push_type::operator()* and *asymmetric_coroutine<>::pull_type::operator()* from inside the same coroutine results in undefined behaviour.

As an example, the code below will result in undefined behaviour:

```
boost::coroutines::symmetric_coroutine<void>::call_type coro(
    [&](boost::coroutines::symmetric_coroutine<void>::yield_type& yield){
        yield(coro); // yield to same symmetric_coroutine
});
coro();
```

Asymmetric coroutine

Two asymmetric coroutine types - *asymmetric_coroutine<>::push_type* and *asymmetric_coroutine<>::pull_type* - provide a uni-directional transfer of data.

asymmetric_coroutine<>::pull_type

asymmetric_coroutine<>::pull_type transfers data from another execution context (== pulled-from). The template parameter defines the transferred parameter type. The constructor of *asymmetric_coroutine<>::pull_type* takes a function (*coroutine-function*) accepting a reference to an *asymmetric_coroutine<>::push_type* as argument. Instantiating an *asymmetric_coroutine<>::pull_type* passes the control of execution to *coroutine-function* and a complementary *asymmetric_coroutine<>::push_type* is synthesized by the library and passed as reference to *coroutine-function*.

This kind of coroutine provides *asymmetric_coroutine<>::pull_type::operator()*. This method only switches context; it transfers no data.

asymmetric_coroutine<>::pull_type provides input iterators (*asymmetric_coroutine<>::pull_type::iterator*) and *std::begin()*/*std::end()* are overloaded. The increment-operation switches the context and transfers data.

```
boost::coroutines::asymmetric_coroutine<int>::pull_type source(
    [&](boost::coroutines::asymmetric_coroutine<int>::push_type& sink){
        int first=1,second=1;
        sink(first);
        sink(second);
        for(int i=0;i<8;++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
```

In this example an *asymmetric_coroutine<>::pull_type* is created in the main execution context taking a lambda function (== *coroutine-function*) which calculates Fibonacci numbers in a simple *for*-loop. The *coroutine-function* is executed in a newly created execution context which is managed by the instance of *asymmetric_coroutine<>::pull_type*. An *asymmetric_coroutine<>::push_type* is automatically generated by the library and passed as reference to the lambda function. Each time the lambda function calls *asymmetric_coroutine<>::push_type::operator()* with another Fibonacci number, *asymmetric_coroutine<>::push_type* transfers it back to the main execution context. The local state of *coroutine-function* is preserved and will be restored upon transferring execution control back to *coroutine-function* to calculate the next Fibonacci number. Because *asymmetric_coroutine<>::pull_type* provides input iterators and *std::begin()*/*std::end()* are overloaded, a *range-based for*-loop can be used to iterate over the generated Fibonacci numbers.

asymmetric_coroutine<>::push_type

asymmetric_coroutine<>::push_type transfers data to the other execution context (== pushed-to). The template parameter defines the transferred parameter type. The constructor of *asymmetric_coroutine<>::push_type* takes a function (*coroutine-function*) accepting a reference to an *asymmetric_coroutine<>::pull_type* as argument. In contrast to *asymmetric_coroutine<>::pull_type*, instantiating an *asymmetric_coroutine<>::push_type* does not pass the control of execution to *coroutine-function* - instead the first call of *asymmetric_coroutine<>::push_type::operator()* synthesizes a complementary *asymmetric_coroutine<>::pull_type* and passes it as reference to *coroutine-function*.

The *asymmetric_coroutine<>::push_type* interface does not contain a *get()*-function: you can not retrieve values from another execution context with this kind of coroutine.

asymmetric_coroutine<>::push_type provides output iterators (*asymmetric_coroutine<>::push_type::iterator*) and *std::begin()*/*std::end()* are overloaded. The increment-operation switches the context and transfers data.

```
struct FinalEOL{
    ~FinalEOL(){
        std::cout << std::endl;
    }
};

const int num=5, width=15;
boost::coroutines::asymmetric_coroutine<std::string>::push_type writer(
    [&](boost::coroutines::asymmetric_coroutine<std::string>::pull_type& in){
        // finish the last line when we leave by whatever means
        FinalEOL eol;
        // pull values from upstream, lay them out 'num' to a line
        for (;;){
            for(int i=0;i<num;++i){
                // when we exhaust the input, stop
                if(!in) return;
                std::cout << std::setw(width) << in.get();
                // now that we've handled this item, advance to next
                in();
            }
            // after 'num' items, line break
            std::cout << std::endl;
        }
    });

std::vector<std::string> words{
    "peas", "porridge", "hot", "peas",
    "porridge", "cold", "peas", "porridge",
    "in", "the", "pot", "nine",
    "days", "old" };

std::copy(boost::begin(words),boost::end(words),boost::begin(writer));

output:
           peas       porridge            hot           peas       porridge
           cold           peas       porridge             in            the
            pot           nine           days            old
```

In this example an *asymmetric_coroutine<>::push_type* is created in the main execution context accepting a lambda function (== *coroutine-function*) which requests strings and lays out 'num' of them on each line. This demonstrates the inversion of control permitted by coroutines. Without coroutines, a utility function to perform the same job would necessarily accept each new value as a function parameter, returning after processing that single value. That function would depend on a static state variable. A *coroutine-function*, however, can request each new value as if by calling a function -- even though its caller also passes values as if by calling a function. The *coroutine-function* is executed in a newly created execution context which is managed by the instance of *asymmetric_coroutine<>::push_type*. The main execution context passes the strings to the *coroutine-function* by calling *asymmetric_coroutine<>::push_type::operator()*. An *asymmetric_coroutine<>::pull_type* instance is automatically generated by the library and passed as reference to the lambda function. The *coroutine-function* accesses the strings passed from the main execution context by calling *asymmetric_coroutine<>::pull_type::get()* and lays those strings out on *std::cout* according the parameters 'num' and 'width'. The local state of *coroutine-function* is preserved and will be restored after transferring execution control back to *coroutine-function*. Because *asymmetric_coroutine<>::push_type* provides output iterators and *std::begin()*/*std::end()* are overloaded, the *std::copy* algorithm can be used to iterate over the vector containing the strings and pass them one by one to the coroutine.

coroutine-function

The *coroutine-function* returns *void* and takes its counterpart-coroutine as argument, so that using the coroutine passed as argument to *coroutine-function* is the only way to transfer data and execution control back to the caller. Both coroutine types take the same template argument. For *asymmetric_coroutine<>::pull_type* the *coroutine-function* is entered at *asymmetric_coroutine<>::pull_type* construction. For *asymmetric_coroutine<>::push_type* the *coroutine-function* is not entered at *asymmetric_coroutine<>::push_type* construction but entered by the first invocation of *asymmetric_coroutine<>::push_type::operator()*. After execution control is returned from *coroutine-function* the state of the coroutine can be checked via *asymmetric_coroutine<>::pull_type::operator bool* returning

`true` if the coroutine is still valid (*coroutine-function* has not terminated). Unless the first template parameter is `void`, `true` also implies that a data value is available.

passing data from a pull-coroutine to main-context

In order to transfer data from an *asymmetric_coroutine<>::pull_type* to the main-context the framework synthesizes an *asymmetric_coroutine<>::push_type* associated with the *asymmetric_coroutine<>::pull_type* instance in the main-context. The synthesized *asymmetric_coroutine<>::push_type* is passed as argument to *coroutine-function*. The *coroutine-function* must call this *asymmetric_coroutine<>::push_type::operator()* in order to transfer each data value back to the main-context. In the main-context, the *asymmetric_coroutine<>::pull_type::operator bool* determines whether the coroutine is still valid and a data value is available or *coroutine-function* has terminated (*asymmetric_coroutine<>::pull_type* is invalid; no data value available). Access to the transferred data value is given by *asymmetric_coroutine<>::pull_type::get()*.

```
boost::coroutines::asymmetric_coroutine<int>::pull_type source( // constructor enters coroutine-function
    [&](boost::coroutines::asymmetric_coroutine<int>::push_type& sink){
        sink(1); // push {1} back to main-context
        sink(1); // push {1} back to main-context
        sink(2); // push {2} back to main-context
        sink(3); // push {3} back to main-context
        sink(5); // push {5} back to main-context
        sink(8); // push {8} back to main-context
    });

while(source){            // test if pull-coroutine is valid
    int ret=source.get(); // access data value
    source();             // context-switch to coroutine-function
}
```

passing data from main-context to a push-coroutine

In order to transfer data to an *asymmetric_coroutine<>::push_type* from the main-context the framework synthesizes an *asymmetric_coroutine<>::pull_type* associated with the *asymmetric_coroutine<>::push_type* instance in the main-context. The synthesized *asymmetric_coroutine<>::pull_type* is passed as argument to *coroutine-function*. The main-context must call this *asymmetric_coroutine<>::push_type::operator()* in order to transfer each data value into the *coroutine-function*. Access to the transferred data value is given by *asymmetric_coroutine<>::pull_type::get()*.

```
boost::coroutines::asymmetric_coroutine<int>::push_type sink( // constructor does NOT enter ↵
coroutine-function
    [&](boost::coroutines::asymmetric_coroutine<int>::pull_type& source){
        for (int i:source) {
            std::cout << i <<  " ";
        }
    });

std::vector<int> v{1,1,2,3,5,8,13,21,34,55};
for( int i:v){
    sink(i); // push {i} to coroutine-function
}
```

accessing parameters

Parameters returned from or transferred to the *coroutine-function* can be accessed with *asymmetric_coroutine<>::pull_type::get()*.

Splitting-up the access of parameters from context switch function enables to check if *asymmetric_coroutine<>::pull_type* is valid after return from *asymmetric_coroutine<>::pull_type::operator()*, e.g. *asymmetric_coroutine<>::pull_type* has values and *coroutine-function* has not terminated.

```
boost::coroutines::asymmetric_coroutine<boost::tuple<int,int>>::push_type sink(
    [&](boost::coroutines::asymmetric_coroutine<boost::tuple<int,int>>::pull_type& source){
        // access tuple {7,11}; x==7 y==1
        int x,y;
        boost::tie(x,y)=source.get();
    });

sink(boost::make_tuple(7,11));
```

exceptions

An exception thrown inside an *asymmetric_coroutine<>::pull_type*'s *coroutine-function* before its first call to *asymmetric_coroutine<>::push_type::operator()* will be re-thrown by the *asymmetric_coroutine<>::pull_type* constructor. After an *asymmetric_coroutine<>::pull_type*'s *coroutine-function*'s first call to *asymmetric_coroutine<>::push_type::operator()*, any subsequent exception inside that *coroutine-function* will be re-thrown by *asymmetric_coroutine<>::pull_type::operator()*. *asymmetric_coroutine<>::pull_type::get()* does not throw.

An exception thrown inside an *asymmetric_coroutine<>::push_type*'s *coroutine-function* will be re-thrown by *asymmetric_coroutine<>::push_type::operator()*.

Important

Code executed by *coroutine-function* must not prevent the propagation of the *detail::forced_unwind* exception. Absorbing that exception will cause stack unwinding to fail. Thus, any code that catches all exceptions must re-throw any pending *detail::forced_unwind* exception.

```
try {
    // code that might throw
} catch(const boost::coroutines::detail::forced_unwind&) {
    throw;
} catch(...) {
    // possibly not re-throw pending exception
}
```

Stack unwinding

Sometimes it is necessary to unwind the stack of an unfinished coroutine to destroy local stack variables so they can release allocated resources (RAII pattern). The `attributes` argument of the coroutine constructor indicates whether the destructor should unwind the stack (stack is unwound by default).

Stack unwinding assumes the following preconditions:

- The coroutine is not *not-a-coroutine*
- The coroutine is not complete
- The coroutine is not running
- The coroutine owns a stack

After unwinding, a *coroutine* is complete.

```
struct X {
    X(){
        std::cout<<"X()"<<std::endl;
    }

    ~X(){
        std::cout<<"~X()"<<std::endl;
    }
};

{
    boost::coroutines::asymmetric_coroutine<void>::push_type sink(
        [&](boost::coroutines::asymmetric_coroutine<void>::pull_type& source){
            X x;
            for(int=0;;++i){
                std::cout<<"fn(): "<<i<<std::endl;
                // transfer execution control back to main()
                source();
            }
        });

    sink();
    sink();
    sink();
    sink();
    sink();

    std::cout<<"sink is complete: "<<std::boolalpha<<!sink<<"\n";
}

output:
    X()
    fn(): 0
    fn(): 1
    fn(): 2
    fn(): 3
    fn(): 4
    fn(): 5
    sink is complete: false
    ~X()
```

Range iterators

Boost.Coroutine provides output- and input-iterators using **Boost.Range**. *asymmetric_coroutine<>::pull_type* can be used via input-iterators using *std::begin()* and *std::end()*.

```
int number=2,exponent=8;
boost::coroutines::asymmetric_coroutine< int >::pull_type source(
    [&]( boost::coroutines::asymmetric_coroutine< int >::push_type & sink){
        int counter=0,result=1;
        while(counter++<exponent){
            result=result*number;
            sink(result);
        }
    });

for (auto i:source)
    std::cout << i <<  " ";

output:
    2 4 8 16 32 64 128 256
```

asymmetric_coroutine<>::pull_type::iterator::operator++() corresponds to *asymmetric_coroutine<>::pull_type::operator()*; *asymmetric_coroutine<>::pull_type::iterator::operator*()* roughly corresponds to *asymmetric_coroutine<>::pull_type::get()*. An iterator originally obtained from *std::begin()* of an *asymmetric_coroutine<>::pull_type* compares equal to an iterator obtained from *std::end()* of that same *asymmetric_coroutine<>::pull_type* instance when its *asymmetric_coroutine<>::pull_type::operator bool* would return `false`].

Note

If `T` is a move-only type, then *asymmetric_coroutine<T>::pull_type::iterator* may only be dereferenced once before it is incremented again.

Output-iterators can be created from *asymmetric_coroutine<>::push_type*.

```
boost::coroutines::asymmetric_coroutine<int>::push_type sink(
    [&](boost::coroutines::asymmetric_coroutine<int>::pull_type& source){
        while(source){
            std::cout << source.get() <<  " ";
            source();
        }
    });

std::vector<int> v{1,1,2,3,5,8,13,21,34,55};
std::copy(boost::begin(v),boost::end(v),boost::begin(sink));
```

asymmetric_coroutine<>::push_type::iterator::operator()* roughly corresponds to *asymmetric_coroutine<>::push_type::operator()*. An iterator originally obtained from *std::begin()* of an *asymmetric_coroutine<>::push_type* compares equal to an iterator obtained from *std::end()* of that same *asymmetric_coroutine<>::push_type* instance when its *asymmetric_coroutine<>::push_type::operator bool* would return `false`.

Exit a *coroutine-function*

coroutine-function is exited with a simple return statement jumping back to the calling routine. The *asymmetric_coroutine<>::pull_type*, *asymmetric_coroutine<>::push_type* becomes complete, e.g. *asymmetric_coroutine<>::pull_type::operator bool*, *asymmetric_coroutine<>::push_type::operator bool* will return `false`.

Important

After returning from *coroutine-function* the *coroutine* is complete (can not resumed with *asymmetric_coroutine<>::push_type::operator()*, *asymmetric_coroutine<>::pull_type::operator()*).

Class asymmetric_coroutine<>::pull_type

```
#include <boost/coroutine/asymmetric_coroutine.hpp>

template< typename R >
class asymmetric_coroutine<>::pull_type
{
public:
    pull_type() noexcept;

    template< typename Fn >
    pull_type( Fn && fn, attributes const& attr = attributes() );

    template< typename Fn, typename StackAllocator >
    pull_type( Fn && fn, attributes const& attr, StackAllocator stack_alloc);

    pull_type( pull_type const& other)=delete;

    pull_type & operator=( pull_type const& other)=delete;

    ~pull_type();

    pull_type( pull_type && other) noexcept;

    pull_type & operator=( pull_type && other) noexcept;

    operator unspecified-bool-type() const noexcept;

    bool operator!() const noexcept;

    void swap( pull_type & other) noexcept;

    pull_type & operator()();

    R get() const;
};

template< typename R >
void swap( pull_type< R > & l, pull_type< R > & r);

template< typename R >
range_iterator< pull_type< R > >::type begin( pull_type< R > &);

template< typename R >
range_iterator< pull_type< R > >::type end( pull_type< R > &);
```

pull_type()

Effects: Creates a coroutine representing *not-a-coroutine*.

Throws: Nothing.

template< typename Fn > pull_type(Fn && fn, attributes const& attr)

Preconditions: `size` >= minimum_stacksize(), `size` <= maximum_stacksize() when ! is_stack_unbounded().

Effects: Creates a coroutine which will execute `fn`, and enters it. Argument `attr` determines stack clean-up and preserving floating-point registers.

Throws: Exceptions thrown inside *coroutine-function*.

`template< typename Fn, typename StackAllocator > pull_type( Fn && fn, attributes const& attr, StackAllocator const& stack_alloc)`

Preconditions: `size` >= minimum_stacksize(), `size` <= maximum_stacksize() when ! is_stack_unbounded().

Effects: Creates a coroutine which will execute `fn`. Argument `attr` determines stack clean-up and preserving floating-point registers. For allocating/deallocating the stack `stack_alloc` is used.

Throws: Exceptions thrown inside *coroutine-function*.

`~pull_type()`

Effects: Destroys the context and deallocates the stack.

`pull_type( pull_type && other)`

Effects: Moves the internal data of `other` to `*this`. `other` becomes *not-a-coroutine*.

Throws: Nothing.

`pull_type & operator=( pull_type && other)`

Effects: Destroys the internal data of `*this` and moves the internal data of `other` to `*this`. `other` becomes *not-a-coroutine*.

Throws: Nothing.

`operator unspecified-bool-type() const`

Returns: If `*this` refers to *not-a-coroutine* or the coroutine-function has returned (completed), the function returns `false`. Otherwise `true`.

Throws: Nothing.

`bool operator!() const`

Returns: If `*this` refers to *not-a-coroutine* or the coroutine-function has returned (completed), the function returns `true`. Otherwise `false`.

Throws: Nothing.

`pull_type<> & operator()()`

Preconditions: `*this` is not a *not-a-coroutine*.

Effects: Execution control is transferred to *coroutine-function* (no parameter is passed to the coroutine-function).

Throws: Exceptions thrown inside *coroutine-function*.

`R get()`

```
R    asymmetric_coroutine<R,StackAllocator>::pull_type::get();
R&   asymmetric_coroutine<R&,StackAllocator>::pull_type::get();
void asymmetric_coroutine<void,StackAllocator>::pull_type::get()=delete;
```

Preconditions: `*this` is not a *not-a-coroutine*.

Returns: Returns data transferred from coroutine-function via *asymmetric_coroutine<>::push_type::operator()*.

Throws: `invalid_result`

Note: If `R` is a move-only type, you may only call `get()` once before the next *asymmetric_coroutine<>::pull_type::operator()* call.

`void swap( pull_type & other)`

Effects: Swaps the internal data from `*this` with the values of `other`.

Throws: Nothing.

Non-member function `swap()`

```
template< typename R >
void swap( pull_type< R > & l, pull_type< R > & r);
```

Effects: As if 'l.swap(r)'.

Non-member function `begin( pull_type< R > &)`

```
template< typename R >
range_iterator< pull_type< R > >::type begin( pull_type< R > &);
```

Returns: Returns a range-iterator (input-iterator).

Non-member function `end( pull_type< R > &)`

```
template< typename R >
range_iterator< pull_type< R > >::type end( pull_type< R > &);
```

Returns: Returns an end range-iterator (input-iterator).

Note: When first obtained from `begin( pull_type< R > &)`, or after some number of increment operations, an iterator will compare equal to the iterator returned by `end( pull_type< R > &)` when the corresponding *asymmetric_coroutine<>::pull_type::operator bool* would return `false`.

Class asymmetric_coroutine<>::push_type

```
#include <boost/coroutine/asymmetric_coroutine.hpp>

template< typename Arg >
class asymmetric_coroutine<>::push_type
{
public:
    push_type() noexcept;

    template< typename Fn >
    push_type( Fn && fn, attributes const& attr = attributes() );

    template< typename Fn, typename StackAllocator >
    push_type( Fn && fn, attributes const& attr, StackAllocator stack_alloc);

    push_type( push_type const& other)=delete;

    push_type & operator=( push_type const& other)=delete;

    ~push_type();

    push_type( push_type && other) noexcept;

    push_type & operator=( push_type && other) noexcept;

    operator unspecified-bool-type() const noexcept;

    bool operator!() const noexcept;

    void swap( push_type & other) noexcept;

    push_type & operator()( Arg arg);
};

template< typename Arg >
void swap( push_type< Arg > & l, push_type< Arg > & r);

template< typename Arg >
range_iterator< push_type< Arg > >::type begin( push_type< Arg > &);

template< typename Arg >
range_iterator< push_type< Arg > >::type end( push_type< Arg > &);
```

push_type()

Effects: Creates a coroutine representing *not-a-coroutine*.

Throws: Nothing.

template< typename Fn > push_type(Fn && fn, attributes const& attr)

Preconditions: `size` >= minimum_stacksize(), `size` <= maximum_stacksize() when ! is_stack_unbounded().

Effects: Creates a coroutine which will execute `fn`. Argument `attr` determines stack clean-up and preserving floating-point registers.

template< typename Fn, typename StackAllocator > push_type(Fn && fn, attributes const& attr, StackAllocator const& stack_alloc)

Preconditions: `size` >= minimum_stacksize(), `size` <= maximum_stacksize() when ! is_stack_unbounded().

Effects: Creates a coroutine which will execute `fn`. Argument `attr` determines stack clean-up and preserving floating-point registers. For allocating/deallocating the stack `stack_alloc` is used.

`~push_type()`

Effects: Destroys the context and deallocates the stack.

`push_type( push_type && other)`

Effects: Moves the internal data of `other` to `*this`. `other` becomes *not-a-coroutine*.

Throws: Nothing.

`push_type & operator=( push_type && other)`

Effects: Destroys the internal data of `*this` and moves the internal data of `other` to `*this`. `other` becomes *not-a-coroutine*.

Throws: Nothing.

`operator unspecified-bool-type() const`

Returns: If `*this` refers to *not-a-coroutine* or the coroutine-function has returned (completed), the function returns `false`. Otherwise `true`.

Throws: Nothing.

`bool operator!() const`

Returns: If `*this` refers to *not-a-coroutine* or the coroutine-function has returned (completed), the function returns `true`. Otherwise `false`.

Throws: Nothing.

`push_type & operator()(Arg arg)`

```
push_type& asymmetric_coroutine<Arg>::push_type::operator()(Arg);
push_type& asymmetric_coroutine<Arg&>::push_type::operator()(Arg&);
push_type& asymmetric_coroutine<void>::push_type::operator()();
```

Preconditions: operator unspecified-bool-type() returns `true` for `*this`.

Effects: Execution control is transferred to *coroutine-function* and the argument `arg` is passed to the coroutine-function.

Throws: Exceptions thrown inside *coroutine-function*.

`void swap( push_type & other)`

Effects: Swaps the internal data from `*this` with the values of `other`.

Throws: Nothing.

Non-member function `swap()`

```
template< typename Arg >
void swap( push_type< Arg > & l, push_type< Arg > & r);
```

Effects: As if 'l.swap(r)'.

Non-member function `begin( push_type< Arg > &)`

```
template< typename Arg >
range_iterator< push_type< Arg > >::type begin( push_type< Arg > &);
```

Returns: Returns a range-iterator (output-iterator).

Non-member function `end( push_type< Arg > &)`

```
template< typename Arg >
range_iterator< push_type< Arg > >::type end( push_type< Arg > &);
```

Returns: Returns a end range-iterator (output-iterator).

Note: When first obtained from `begin( push_type< R > &)`, or after some number of increment operations, an iterator will compare equal to the iterator returned by `end( push_type< R > &)` when the corresponding *asymmetric_coroutine<>::push_type::operator bool* would return `false`.

Symmetric coroutine

In contrast to asymmetric coroutines, where the relationship between caller and callee is fixed, symmetric coroutines are able to transfer execution control to any other (symmetric) coroutine. E.g. a symmetric coroutine is not required to return to its direct caller.

symmetric_coroutine<>::call_type

symmetric_coroutine<>::call_type starts a symmetric coroutine and transfers its parameter to its *coroutine-function*. The template parameter defines the transferred parameter type. The constructor of *symmetric_coroutine<>::call_type* takes a function (*coroutine-function*) accepting a reference to a *symmetric_coroutine<>::yield_type* as argument. Instantiating a *symmetric_coroutine<>::call_type* does not pass the control of execution to *coroutine-function* - instead the first call of *symmetric_coroutine<>::call_type::operator()* synthesizes a *symmetric_coroutine<>::yield_type* and passes it as reference to *coroutine-function*.

The *symmetric_coroutine<>::call_type* interface does not contain a *get()*-function: you can not retrieve values from another execution context with this kind of coroutine object.

symmetric_coroutine<>::yield_type

symmetric_coroutine<>::yield_type::operator() is used to transfer data and execution control to another context by calling *symmetric_coroutine<>::yield_type::operator()* with another *symmetric_coroutine<>::call_type* as first argument. Alternatively, you may transfer control back to the code that called *symmetric_coroutine<>::call_type::operator()* by calling *symmetric_coroutine<>::yield_type::operator()* without a *symmetric_coroutine<>::call_type* argument.

The class has only one template parameter defining the transferred parameter type. Data transferred to the coroutine are accessed through *symmetric_coroutine<>::yield_type::get()*.

Important

symmetric_coroutine<>::yield_type can only be created by the framework.

```
std::vector<int> merge(const std::vector<int>& a,const std::vector<int>& b)
{
    std::vector<int> c;
    std::size_t idx_a=0,idx_b=0;
    boost::coroutines::symmetric_coroutine<void>::call_type* other_a=0,* other_b=0;

    boost::coroutines::symmetric_coroutine<void>::call_type coro_a(
        [&](boost::coroutines::symmetric_coroutine<void>::yield_type& yield) {
            while(idx_a<a.size())
            {
                if(b[idx_b]<a[idx_a])    // test if element in array b is less than in array a
                    yield(*other_b);     // yield to coroutine coro_b
                c.push_back(a[idx_a++]); // add element to final array
            }
            // add remaining elements of array b
            while ( idx_b < b.size())
                c.push_back( b[idx_b++]);
        });

    boost::coroutines::symmetric_coroutine<void>::call_type coro_b(
        [&](boost::coroutines::symmetric_coroutine<void>::yield_type& yield) {
            while(idx_b<b.size())
            {
                if (a[idx_a]<b[idx_b])   // test if element in array a is less than in array b
                    yield(*other_a);     // yield to coroutine coro_a
                c.push_back(b[idx_b++]); // add element to final array
            }
            // add remaining elements of array a
            while ( idx_a < a.size())
                c.push_back( a[idx_a++]);
        });


    other_a = & coro_a;
    other_b = & coro_b;

    coro_a(); // enter coroutine-fn of coro_a

    return c;
}

std::vector< int > a = {1,5,6,10};
std::vector< int > b = {2,4,7,8,9,13};
std::vector< int > c = merge(a,b);
print(a);
print(b);
print(c);

output:
    a : 1 5 6 10
    b : 2 4 7 8 9 13
    c : 1 2 4 5 6 7 8 9 10 13
```

In this example two *symmetric_coroutine<>::call_type* are created in the main execution context accepting a lambda function (== *coroutine-function*) which merges elements of two sorted arrays into a third array. `coro_a()` enters the *coroutine-function* of `coro_a` cycling through the array and testing if the actual element in the other array is less than the element in the local one. If so, the coroutine yields to the other coroutine `coro_b` using `yield(*other_b)`. If the current element of the local array is less than the element of the other array, it is put to the third array. Because the coroutine jumps back to `coro_a()` (returning from this method) after leaving the *coroutine-function*, the elements of the other array will appended at the end of the third array if all element of the local array are processed.

coroutine-function

The *coroutine-function* returns *void* and takes *symmetric_coroutine<>::yield_type*, providing coroutine functionality inside the *coroutine-function*, as argument. Using this instance is the only way to transfer data and execution control. *symmetric_coroutine<>::call_type* does not enter the *coroutine-function* at *symmetric_coroutine<>::call_type* construction but at the first invocation of *symmetric_coroutine<>::call_type::operator()*.

Unless the template parameter is `void`, the *coroutine-function* of a *symmetric_coroutine<>::call_type* can assume that (a) upon initial entry and (b) after every *symmetric_coroutine<>::yield_type::operator()* call, its *symmetric_coroutine<>::yield_type::get()* has a new value available.

However, if the template parameter is a move-only type, *symmetric_coroutine<>::yield_type::get()* may only be called once before the next *symmetric_coroutine<>::yield_type::operator()* call.

passing data from main-context to a symmetric-coroutine

In order to transfer data to a *symmetric_coroutine<>::call_type* from the main-context the framework synthesizes a *symmetric_coroutine<>::yield_type* associated with the *symmetric_coroutine<>::call_type* instance. The synthesized *symmetric_coroutine<>::yield_type* is passed as argument to *coroutine-function*. The main-context must call *symmetric_coroutine<>::call_type::operator()* in order to transfer each data value into the *coroutine-function*. Access to the transferred data value is given by *symmetric_coroutine<>::yield_type::get()*.

```
boost::coroutines::symmetric_coroutine<int>::call_type coro( // constructor does NOT enter ↵
coroutine-function
    [&](boost::coroutines::symmetric_coroutine<int>::yield_type& yield){
        for (;;) {
            std::cout << yield.get() <<  " ";
            yield(); // jump back to starting context
         }
    });

coro(1); // transfer {1} to coroutine-function
coro(2); // transfer {2} to coroutine-function
coro(3); // transfer {3} to coroutine-function
coro(4); // transfer {4} to coroutine-function
coro(5); // transfer {5} to coroutine-function
```

exceptions

An uncaught exception inside a *symmetric_coroutine<>::call_type*'s *coroutine-function* will call *std::terminate()*.

Important

Code executed by coroutine must not prevent the propagation of the *detail::forced_unwind* exception. Absorbing that exception will cause stack unwinding to fail. Thus, any code that catches all exceptions must re-throw any pending *detail::forced_unwind* exception.

```
try {
    // code that might throw
} catch(const boost::coroutines::detail::forced_unwind&) {
    throw;
} catch(...) {
    // possibly not re-throw pending exception
}
```

Stack unwinding

Sometimes it is necessary to unwind the stack of an unfinished coroutine to destroy local stack variables so they can release allocated resources (RAII pattern). The `attributes` argument of the coroutine constructor indicates whether the destructor should unwind the stack (stack is unwound by default).

Stack unwinding assumes the following preconditions:

- The coroutine is not *not-a-coroutine*
- The coroutine is not complete
- The coroutine is not running
- The coroutine owns a stack

After unwinding, a *coroutine* is complete.

```
struct X {
    X(){
        std::cout<<"X()"<<std::endl;
    }

    ~X(){
        std::cout<<"~X()"<<std::endl;
    }
};

boost::coroutines::symmetric_coroutine<int>::call_type other_coro(...);

{
    boost::coroutines::symmetric_coroutine<void>::call_type coro(
        [&](boost::coroutines::symmetric_coroutine<void>::yield_type& yield){
            X x;
            std::cout<<"fn()"<<std::endl;
            // transfer execution control to other coroutine
            yield( other_coro, 7);
        });

    coro();

    std::cout<<"coro is complete: "<<std::boolalpha<<!coro<<"\n";
}

output:
    X()
    fn()
    coro is complete: false
    ~X()
```

Exit a *coroutine-function*

coroutine-function is exited with a simple return statement. This jumps back to the calling *symmetric_coroutine<>::call_type::operator()* at the start of symmetric coroutine chain. That is, symmetric coroutines do not have a strong, fixed relationship to the caller as do asymmetric coroutines. The *symmetric_coroutine<>::call_type* becomes complete, e.g. *symmetric_coroutine<>::call_type::operator bool* will return `false`.

Important

After returning from *coroutine-function* the *coroutine* is complete (can not be resumed with *symmetric_coroutine<>::call_type::operator()*).

Class `symmetric_coroutine<>::call_type`

```
#include <boost/coroutine/symmetric_coroutine.hpp>

template< typename Arg >
class symmetric_coroutine<>::call_type
{
public:
    call_type() noexcept;

    template< typename Fn >
    call_type( Fn && fn, attributes const& attr = attributes() );

    template< typename Fn, typename StackAllocator >
    call_type( Fn && fn, attributes const& attr, StackAllocator stack_alloc);

    ~call_type();

    call_type( call_type const& other)=delete;

    call_type & operator=( call_type const& other)=delete;

    call_type( call_type && other) noexcept;

    call_type & operator=( call_type && other) noexcept;

    operator unspecified-bool-type() const;

    bool operator!() const noexcept;

    void swap( call_type & other) noexcept;

    call_type & operator()( Arg arg) noexcept;
};

template< typename Arg >
void swap( symmetric_coroutine< Arg >::call_type & l, symmetric_coroutine< Arg >::call_type & r);
```

`call_type()`

Effects: Creates a coroutine representing *not-a-coroutine*.

Throws: Nothing.

`template< typename Fn > call_type( Fn fn, attributes const& attr)`

Preconditions: `size` >= minimum_stacksize(), `size` <= maximum_stacksize() when ! is_stack_unbounded().

Effects: Creates a coroutine which will execute `fn`. Argument `attr` determines stack clean-up and preserving floating-point registers. For allocating/deallocating the stack `stack_alloc` is used.

`template< typename Fn, typename StackAllocator > call_type( Fn && fn, attributes const& attr, StackAllocator const& stack_alloc)`

Preconditions: `size` >= minimum_stacksize(), `size` <= maximum_stacksize() when ! is_stack_unbounded().

Effects: Creates a coroutine which will execute `fn`. Argument `attr` determines stack clean-up and preserving floating-point registers. For allocating/deallocating the stack `stack_alloc` is used.

`~call_type()`

Effects: Destroys the context and deallocates the stack.

`call_type( call_type && other)`

Effects: Moves the internal data of `other` to `*this`. `other` becomes *not-a-coroutine*.

Throws: Nothing.

`call_type & operator=( call_type && other)`

Effects: Destroys the internal data of `*this` and moves the internal data of `other` to `*this`. `other` becomes *not-a-coroutine*.

Throws: Nothing.

`operator unspecified-bool-type() const`

Returns: If `*this` refers to *not-a-coroutine* or the coroutine-function has returned (completed), the function returns `false`. Otherwise `true`.

Throws: Nothing.

`bool operator!() const`

Returns: If `*this` refers to *not-a-coroutine* or the coroutine-function has returned (completed), the function returns `true`. Otherwise `false`.

Throws: Nothing.

`void swap( call_type & other)`

Effects: Swaps the internal data from `*this` with the values of `other`.

Throws: Nothing.

`call_type & operator()(Arg arg)`

```
symmetric_coroutine::call_type& coroutine<Arg,StackAllocator>::call_type::operator()(Arg);
symmetric_coroutine::call_type& coroutine<Arg&,StackAllocator>::call_type::operator()(Arg&);
symmetric_coroutine::call_type& coroutine<void,StackAllocator>::call_type::operator()();
```

Preconditions: operator unspecified-bool-type() returns `true` for `*this`.

Effects: Execution control is transferred to *coroutine-function* and the argument `arg` is passed to the coroutine-function.

Throws: Nothing.

Non-member function `swap()`

```
template< typename Arg >
void swap( symmetric_coroutine< Arg >::call_type & l, symmetric_coroutine< Arg >::call_type & r);
```

Effects: As if 'l.swap(r)'.

Class `symmetric_coroutine<>::yield_type`

```
#include <boost/coroutine/symmetric_coroutine.hpp>

template< typename R >
class symmetric_coroutine<>::yield_type
{
public:
    yield_type() noexcept;

    yield_type( yield_type const& other)=delete;

    yield_type & operator=( yield_type const& other)=delete;

    yield_type( yield_type && other) noexcept;

    yield_type & operator=( yield_type && other) noexcept;

    void swap( yield_type & other) noexcept;

    operator unspecified-bool-type() const;

    bool operator!() const noexcept;

    yield_type & operator()();

    template< typename X >
    yield_type & operator()( symmetric_coroutine< X >::call_type & other, X & x);

    template< typename X >
    yield_type & operator()( symmetric_coroutine< X >::call_type & other);

    R get() const;
};
```

`operator unspecified-bool-type() const`

Returns: If `*this` refers to *not-a-coroutine*, the function returns `false`. Otherwise `true`.

Throws: Nothing.

`bool operator!() const`

Returns: If `*this` refers to *not-a-coroutine*, the function returns `true`. Otherwise `false`.

Throws: Nothing.

yield_type & operator()()

```
yield_type & operator()();
template< typename X >
yield_type & operator()( symmetric_coroutine< X >::call_type & other, X & x);
template<>
yield_type & operator()( symmetric_coroutine< void >::call_type & other);
```

Preconditions: `*this` is not a *not-a-coroutine*.

Effects: The first function transfers execution control back to the starting point, e.g. invocation of *symmetric_coroutine<>::call_type::operator()*. The last two functions transfer the execution control to another symmetric coroutine. Parameter `x` is passed as value into `other`'s context.

Throws: *detail::forced_unwind*

R get()

```
R    symmetric_coroutine<R>::yield_type::get();
R&   symmetric_coroutine<R&>::yield_type::get();
void symmetric_coroutine<void>yield_type::get()=delete;
```

Preconditions: `*this` is not a *not-a-coroutine*.

Returns: Returns data transferred from coroutine-function via *asymmetric_coroutine<>::push_type::operator()*.

Throws: `invalid_result`

Attributes

Class `attributes` is used to specify parameters required to setup a coroutine's context.

```
enum flag_unwind_t
{
    stack_unwind,
    no_stack_unwind
};

enum flag_fpu_t
{
    fpu_preserved,
    fpu_not_preserved
};

struct attributes
{
    std::size_t     size;
    flag_unwind_t   do_unwind;
    flag_fpu_t      preserve_fpu;

    attributes() noexcept;

    explicit attributes( std::size_t size_) noexcept;

    explicit attributes( flag_unwind_t do_unwind_) noexcept;

    explicit attributes( flag_fpu_t preserve_fpu_) noexcept;

    explicit attributes( std::size_t size_, flag_unwind_t do_unwind_) noexcept;

    explicit attributes( std::size_t size_, flag_fpu_t preserve_fpu_) noexcept;

    explicit attributes( flag_unwind_t do_unwind_, flag_fpu_t preserve_fpu_) noexcept;

   explicit attributes( std::size_t size_, flag_unwind_t do_unwind_, flag_fpu_t preserve_fpu_) no↵
except;
};
```

`attributes()`

Effects: Default constructor using `boost::context::default_stacksize()`, does unwind the stack after coroutine/generator is complete and preserves FPU registers.

Throws: Nothing.

`attributes( std::size_t size)`

Effects: Argument `size` defines stack size of the new coroutine. Stack unwinding after termination and preserving FPU registers is set by default.

Throws: Nothing.

`attributes( flag_unwind_t do_unwind)`

Effects: Argument `do_unwind` determines if stack will be unwound after termination or not. The default stacksize is used for the new coroutine and FPU registers are preserved.

Throws: Nothing.

`attributes( flag_fpu_t preserve_fpu)`

Effects: Argument `preserve_fpu` determines if FPU register have to be preserved across context switches. The default stacksize is used for the new coroutine and its stack will be unwound after termination.

Throws: Nothing.

`attributes( std::size_t size, flag_unwind_t do_unwind)`

Effects: Arguments `size` and `do_unwind` are given by the user. FPU registers are preserved across each context switch.

Throws: Nothing.

`attributes( std::size_t size, flag_fpu_t preserve_fpu)`

Effects: Arguments `size` and `preserve_fpu` are given by the user. The stack is automatically unwound after coroutine/generator terminates.

Throws: Nothing.

`attributes( flag_unwind_t do_unwind, flag_fpu_t preserve_fpu)`

Effects: Arguments `do_unwind` and `preserve_fpu` are given by the user. The stack gets a default value of `boost::context::default_stacksize()`.

Throws: Nothing.

`attributes( std::size_t size, flag_unwind_t do_unwind, flag_fpu_t preserve_fpu)`

Effects: Arguments `size`, `do_unwind` and `preserve_fpu` are given by the user.

Throws: Nothing.

Stack allocation

A *coroutine* uses internally a *context* which manages a set of registers and a stack. The memory used by the stack is allocated/deallocated via a *stack-allocator* which is required to model a *stack-allocator concept*.

stack-allocator concept

A *stack-allocator* must satisfy the *stack-allocator concept* requirements shown in the following table, in which `a` is an object of a *stack-allocator* type, `sctx` is a `stack_context`, and `size` is a `std::size_t`:

expression	return type	notes
`a.allocate( sctx, size)`	`void`	creates a stack of at least `size` bytes and stores its pointer and length in `sctx`
`a.deallocate( sctx)`	`void`	deallocates the stack created by `a.allocate()`

Important

The implementation of `allocate()` might include logic to protect against exceeding the context's available stack size rather than leaving it as undefined behaviour.

Important

Calling `deallocate()` with a `stack_context` not set by `allocate()` results in undefined behaviour.

Note

The stack is not required to be aligned; alignment takes place inside *coroutine*.

Note

Depending on the architecture `allocate()` stores an address from the top of the stack (growing downwards) or the bottom of the stack (growing upwards).

class *stack_allocator* is a typedef of *standard_stack_allocator*.

Class *protected_stack_allocator*

Boost.Coroutine provides the class *protected_stack_allocator* which models the *stack-allocator concept*. It appends a guard page at the end of each stack to protect against exceeding the stack. If the guard page is accessed (read or write operation) a segmentation fault/access violation is generated by the operating system.

Important

Using *protected_stack_allocator* is expensive. That is, launching a new coroutine with a new stack is expensive; the allocated stack is just as efficient to use as any other stack.

Note

The appended `guard page` is **not** mapped to physical memory, only virtual addresses are used.

```
#include <boost/coroutine/protected_stack_allocator.hpp>

template< typename traitsT >
struct basic_protected_stack_allocator
{
    typedef traitT  traits_type;

    void allocate( stack_context &, std::size_t size);

    void deallocate( stack_context &);
}

typedef basic_protected_stack_allocator< stack_traits > protected_stack_allocator
```

`void allocate( stack_context & sctx, std::size_t size)`

Preconditions: `traits_type::minimum:size() <= size` and `! traits_type::is_unbounded() && ( traits_type::maximum:size() >= size)`.

Effects: Allocates memory of at least `size` Bytes and stores a pointer to the stack and its actual size in `sctx`. Depending on the architecture (the stack grows downwards/upwards) the stored address is the highest/lowest address of the stack.

`void deallocate( stack_context & sctx)`

Preconditions: `sctx.sp` is valid, `traits_type::minimum:size() <= sctx.size` and `! traits_type::is_unbounded() && ( traits_type::maximum:size() >= sctx.size)`.

Effects: Deallocates the stack space.

Class *standard_stack_allocator*

Boost.Coroutine provides the class *standard_stack_allocator* which models the *stack-allocator concept*. In contrast to *protected_stack_allocator* it does not append a guard page at the end of each stack. The memory is simply managed by `std::malloc()` and `std::free()`.

Note

The *standard_stack_allocator* is the default stack allocator.

```
#include <boost/coroutine/standard_stack_allocator.hpp>

template< typename traitsT >
struct standard_stack_allocator
{
    typedef traitT  traits_type;

    void allocate( stack_context &, std::size_t size);

    void deallocate( stack_context &);
}

typedef basic_standard_stack_allocator< stack_traits > standard_stack_allocator
```

`void allocate( stack_context & sctx, std::size_t size)`

Preconditions: `traits_type::minimum:size() <= size` and `! traits_type::is_unbounded() && ( traits_type::maximum:size() >= size)`.

Effects: Allocates memory of at least `size` Bytes and stores a pointer to the stack and its actual size in `sctx`. Depending on the architecture (the stack grows downwards/upwards) the stored address is the highest/lowest address of the stack.

`void deallocate( stack_context & sctx)`

Preconditions: `sctx.sp` is valid, `traits_type::minimum:size() <= sctx.size` and `! traits_type::is_unbounded() && ( traits_type::maximum:size() >= sctx.size)`.

Effects: Deallocates the stack space.

Class *segmented_stack_allocator*

Boost.Coroutine supports usage of a *segmented-stack*, e. g. the size of the stack grows on demand. The coroutine is created with a minimal stack size and will be increased as required. Class *segmented_stack_allocator* models the *stack-allocator concept*. In contrast to *protected_stack_allocator* and *standard_stack_allocator* it creates a stack which grows on demand.

Note

Segmented stacks are currently only supported by **gcc** from version **4.7 clang** from version **3.4** onwards. In order to use a *segmented-stack* **Boost.Coroutine** must be built with **toolset=gcc segmented-stacks=on** at b2/bjam command-line. Applications must be compiled with compiler-flags **-fsplit-stack -DBOOST_USE_SEGMENTED_STACKS**.

```
#include <boost/coroutine/segmented_stack_allocator.hpp>

template< typename traitsT >
struct basic_segmented_stack_allocator
{
    typedef traitT  traits_type;

    void allocate( stack_context &, std::size_t size);

    void deallocate( stack_context &);
}

typedef basic_segmented_stack_allocator< stack_traits > segmented_stack_allocator;
```

`void allocate( stack_context & sctx, std::size_t size)`

Preconditions: `traits_type::minimum:size() <= size` and `! traits_type::is_unbounded() && ( traits_type::maximum:size() >= size)`.

Effects: Allocates memory of at least `size` Bytes and stores a pointer to the stack and its actual size in `sctx`. Depending on the architecture (the stack grows downwards/upwards) the stored address is the highest/lowest address of the stack.

`void deallocate( stack_context & sctx)`

Preconditions: `sctx.sp` is valid, `traits_type::minimum:size() <= sctx.size` and `! traits_type::is_unbounded() && ( traits_type::maximum:size() >= sctx.size)`.

Effects: Deallocates the stack space.

Class *stack_traits*

stack_traits models a *stack-traits* providing a way to access certain properites defined by the enironment. Stack allocators use *stack-traits* to allocate stacks.

```
#include <boost/coroutine/stack_traits.hpp>

struct stack_traits
{
    static bool is_unbounded() noexcept;

    static std::size_t page_size() noexcept;

    static std::size_t default_size() noexcept;

    static std::size_t minimum_size() noexcept;

    static std::size_t maximum_size() noexcept;
}
```

`static bool is_unbounded()`

Returns: Returns `true` if the environment defines no limit for the size of a stack.

Throws: Nothing.

`static std::size_t page_size()`

Returns: Returns the page size in bytes.

Throws: Nothing.

`static std::size_t default_size()`

Returns: Returns a default stack size, which may be platform specific. If the stack is unbounded then the present implementation returns the maximum of `64 kB` and `minimum_size()`.

Throws: Nothing.

`static std::size_t minimum_size()`

Returns: Returns the minimum size in bytes of stack defined by the environment (Win32 4kB/Win64 8kB, defined by rlimit on POSIX).

Throws: Nothing.

`static std::size_t maximum_size()`

Preconditions: `is_unbounded()` returns `false`.

Returns: Returns the maximum size in bytes of stack defined by the environment.

Throws: Nothing.

Class *stack_context*

Boost.Coroutine provides the class *stack_context* which will contain the stack pointer and the size of the stack. In case of a *segmented-stack*, *stack_context* contains some extra control structures.

```
struct stack_context
{
    void    *   sp;
    std::size_t size;

    // might contain additional control structures
    // for instance for segmented stacks
}
```

`void * sp`

Value: Pointer to the beginning of the stack.

`std::size_t size`

Value: Actual size of the stack.

Performance

Performance of **Boost.Coroutine** was measured on the platforms shown in the following table. Performance measurements were taken using `rdtsc` and `boost::chrono::high_resolution_clock`, with overhead corrections, on x86 platforms. In each case, cache warm-up was accounted for, and the one running thread was pinned to a single CPU.

Table 1. Performance of asymmetric coroutines

Platform	switch	construction (protected stack-allocator)	construction (preallocated stack-allocator)	construction (standard stack-allocator)
i386 (AMD Athlon 64 DualCore 4400+, Linux 32bit)	49 ns / 50 cycles	51 µs / 51407 cycles	14 µs / 15231 cycles	14 µs / 15216 cycles
x86_64 (Intel Core2 Q6700, Linux 64bit)	12 ns / 39 cycles	16 µs / 41802 cycles	6 µs / 10350 cycles	6 µs / 18817 cycles

Table 2. Performance of symmetric coroutines

Platform	switch	construction (protected stack-allocator)	construction (preallocated stack-allocator)	construction (standard stack-allocator)
i386 (AMD Athlon 64 DualCore 4400+, Linux 32bit)	47 ns / 49 cycles	27 µs / 28002 cycles	98 ns / 116 cycles	319 ns / 328 cycles
x86_64 (Intel Core2 Q6700, Linux 64bit)	10 ns / 33 cycles	10 µs / 22828 cycles	42 ns / 710 cycles	135 ns / 362 cycles

Architectures

Boost.Coroutine depends on **Boost.Context** which supports these architectures.

Acknowledgments

I'd like to thank Alex Hagen-Zanker, Christopher Kormanyos, Conrad Poelman, Eugene Yakubovich, Giovanni Piero Deretta, Hartmut Kaiser, Jeffrey Lee Hellrung, **Nat Goodspeed**, Robert Stewart, Vicente J. Botet Escriba and Yuriy Krasnoschek.

Especially Eugene Yakubovich, Giovanni Piero Deretta and Vicente J. Botet Escriba contributed many good ideas during the review.