Coroutine

Oliver Kowalke

Table of Contents

Overview

Boost.Coroutine provides templates for generalized subroutines which allow multiple entry points for 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.

This library is a follow-up on Boost.Coroutine by Giovanni P. Deretta.

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*.

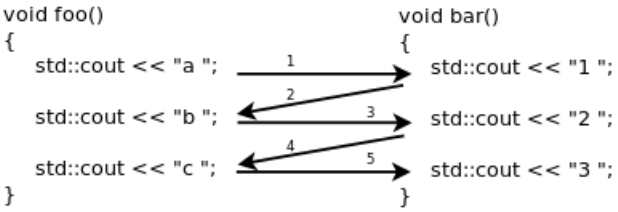
Introduction

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 would be called such as routines, the stack would grow with every call and will never be degraded. A jump into the middle of a coroutine would not be possible, because the return address would have been 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 coroutine's control-block. The registers of the newly activated coroutine must be restored from its associated control-block before it can continue with their work.

The context switch requires no system privileges and provides cooperative multitasking on the level of language. Coroutines provide quasi parallelism. When a program is supposed to do several things at the same time, coroutines help to do this much simpler and more elegant than with only a single flow of control. 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:

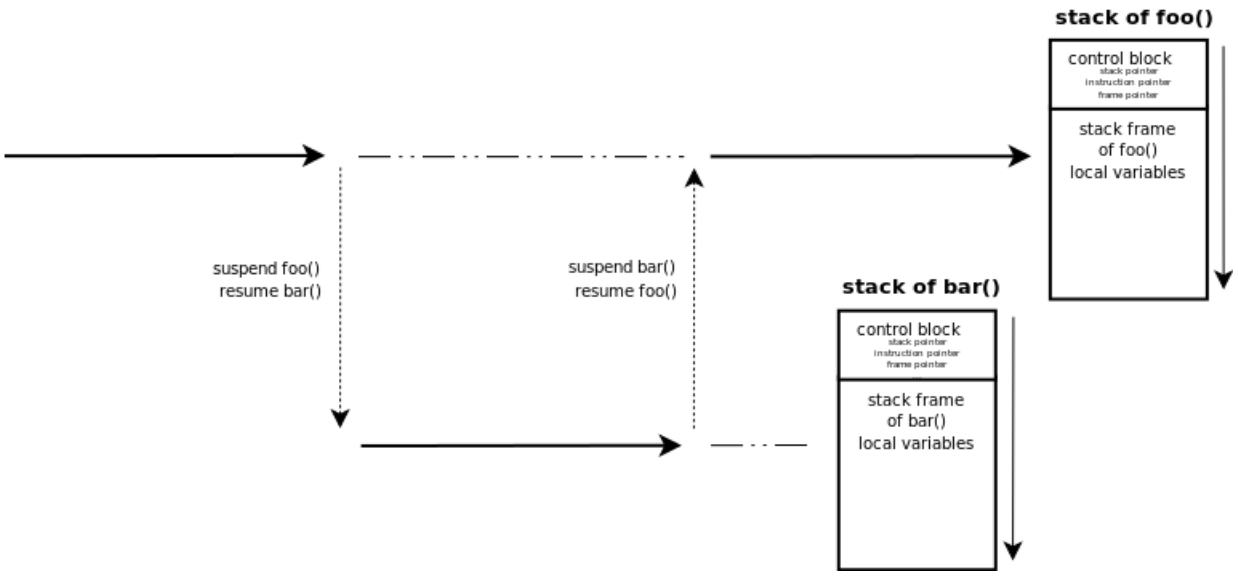
- values of local data persist between successive calls (context switches)
- execution is suspended as control leaves coroutine and resumed at certain time later
- symmetric or asymmetric control-transfer mechanism
- first-class object (can be passed as argument, returned by procedures, stored in a data 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, Articale 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. A symmetric coroutine transfers the execution control only via one operation. The target coroutine must be explicitly specified in the transfer operation. Asymmetric coroutines provide two transfer operations: the *suspend*-operation returns to the invoker by preserving the execution context and the *resume*-operation restores the execution context, e.g. re-enters the coroutine at the same point as it was suspended before.



Both concepts are equivalent and a coroutine library can provide either symmetric or asymmetric coroutines.

stackfulness

In contrast to a stackless coroutine a stackful coroutine allows to suspend from nested stackframes. The execution resumes at exact the same point in the code as 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 *coroutine<>::push_type* and *coroutine<>::pull_type* 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.

'same fringe' problem

The advantages 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 \cpp{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 *coroutine<>::push_type*. *coroutine<>::push_type* suspends the recursive computation and transfers the data value to the main execution context. *boost::coroutines::coroutine<>::pull_type::iterator*, created from *coroutine<>::pull_type*, steps over those data values and delivers them to *std::equal()* for comparison. Each increment of *boost::coroutines::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).

```
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::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::coroutine<std::string>::pull_type left_d_reader(
        [&]( boost::coroutines::coroutine<std::string>::push_type & out){
            traverse(left_d,out);
        });

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

    node::ptr_t right_x(create_right_tree_from("x"));
    boost::coroutines::coroutine<std::string>::pull_type right_x_reader(
        [&]( boost::coroutines::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
```

chaining coroutines

The following example demonstrates how coroutines could be chained.

```
typedef boost::coroutines::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);
    }
}
```

asynchronous operations with boost.asio

In the past the code using asio's *asynchronous operations* was scattered by callbacks. **Boost.Asio** provides with its new *asynchronous result* feature a new way to simplify the code and make it easier to read. *boost::asio::yield_context* uses internally **Boost.Coroutine**:

```
void echo(boost::asio::ip::tcp::socket& socket,boost::asio::yield_context yield){
    char data[128];
    // read asynchronous data from socket
    // execution context will be suspended until
    // some bytes are read from socket
    std::size_t n=socket.async_read_some(boost::asio::buffer(data),yield);
    // write some bytes asynchronously
    boost::asio::async_write(socket,boost::asio::buffer(data,n),yield);
}
```

Coroutine

Boost.Coroutine provides two interfaces - one with uni- and one with bidirectional data transfer (deprecated).

Unidirectional coroutine

Note

This is the default interface (macro BOOST_COROUTINES_UNIDIRECT).

Two coroutine types - *coroutine<>::push_type* and *coroutine<>::pull_type* - providing a unidirectional transfer of data.

coroutine<>::pull_type

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

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

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

```
boost::coroutines::coroutine<int>::pull_type source(
    [&](boost::coroutines::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 <<  " ";

std::cout << "\nDone" << std::endl;

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

In this example a *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 *coroutine<>::pull_type*. A *coroutine<>::push_type* is automatically generated by the runtime and passed as reference to the lambda function. Each time the lambda function calls *boost::coroutines::coroutine<>::push_type::operator()* with another Fibonacci number, *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 *coroutine<>::pull_type* provides input iterators and *boost::begin()*/*boost::end()* are overloaded, a *range-based for*-loop can be used to iterate over the generated Fibonacci numbers.

coroutine<>::push_type

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

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

coroutine<>::push_type provides output iterators (__push_coro__iterator) and *boost::begin()*/*boost::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::coroutine<std::string>::push_type writer(
    [&](boost::coroutines::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 a *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 *coroutine<>::push_type*. The main execution context passes the strings to the *coroutine-function* by calling *boost::coroutines::coroutine<>::push_type::operator()*. A *coroutine<>::pull_type* is automatically generated by the runtime and passed as reference to the lambda function. The *coroutine-function* accesses the strings passed from the main execution context by calling

boost::coroutines::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 *coroutine<>::push_type* provides output iterators and *boost::begin()*/*boost::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.

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 behavior.

clean-up

On coroutine destruction the associated stack will be unwound.

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

segmented stack

coroutine<>::push_type and *coroutine<>::pull_type* does support segmented stacks (growing on demand).

It is not always possible to estimated 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)\cite{gccsplit} 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.

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

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 *boost::coroutines::coroutine<>::push_type::operator()* and *boost::coroutines::coroutine<>::pull_type::operator()*.

Warning

Calling *boost::coroutines::coroutine<>::push_type::operator()*/*boost::coroutines::coroutine<>::pull_type::operator()* from inside the same coroutine results in undefined behaviour.

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 *coroutine<>::pull_type* the *coroutine-function* is entered at *coroutine<>::pull_type* construction. For *coroutine<>::push_type* the *coroutine-function* is not entered at *coroutine<>::push_type* construction but entered by the first invocation of *boost::coroutines::coroutine<>::push_type::operator()*. After execution control is returned from *coroutine-function* the state of the coroutine can be checked via *boost::coroutines::coroutine<>::pull_type::operator bool* returning true if the coroutine is still valid (*coroutine-function* has not terminated). Unless T 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 a *coroutine<>::pull_type* to the main-context the framework synthesizes a *coroutine<>::push_type* associated with the *coroutine<>::pull_type* instance in the main-context. The synthesized *coroutine<>::push_type* is passed as argument to *coroutine-function*.\ The *coroutine-function* must call this *boost::coroutines::coroutine<>::push_type::operator()* in order to transfer each data value back to the main-context. In the main-context, the *boost::coroutines::coroutine<>::pull_type::operator bool* determines whether the coroutine is still valid and a data value is available or *coroutine-function* has terminated (*coroutine<>::pull_type* is invalid; no data value available). Access to the transferred data value is given by *boost::coroutines::coroutine<>::pull_type::get()*.

```
boost::coroutines::coroutine<int>::pull_type source( // constructor enters coroutine-function
    [&](boost::coroutines::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 a *coroutine<>::push_type* from the main-context the framework synthesizes a *coroutine<>::pull_type* associated with the *coroutine<>::push_type* instance in the main-context. The synthesized *coroutine<>::pull_type* is passed as argument to *coroutine-function*. The main-context must call this *boost::coroutines::coroutine<>::push_type::operator()* in order to transfer each data value into the *coroutine-function*. Access to the transferred data value is given by *boost::coroutines::coroutine<>::pull_type::get()*.

```
boost::coroutines::coroutine<int>::push_type sink( // constructor does NOT enter coroutine-
function
    [&](boost::coroutines::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 *boost::coroutines::coroutine<>::pull_type::get()*.

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

```
boost::coroutines::coroutine<boost::tuple<int,int>>::push_type sink(
    [&](boost::coroutines::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 a *coroutine<>::pull_type*'s *coroutine-function* before its first call to *boost::coroutines::coroutine<>::push_type::operator()* will be re-thrown by the *coroutine<>::pull_type* constructor. After a *coroutine<>::pull_type*'s *coroutine-function*'s first call to *boost::coroutines::coroutine<>::push_type::operator()*, any subsequent exception inside that *coroutine-function* will be re-thrown by *boost::coroutines::coroutine<>::pull_type::operator()*. *boost::coroutines::coroutine<>::pull_type::get()* does not throw.

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

Important

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

```
try {
    // code that might throw
} catch(const 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 third argument of the coroutine constructor, `do_unwind`, 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::coroutine<void>::push_type sink(
        [&](boost::coroutines::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<<"c is complete: "<<std::boolalpha<<c.is_complete()<<"\n";
}

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

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

Range iterators

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

```
int number=2,exponent=8;
boost::coroutines::coroutine< int >::pull_type source(
    [&]( boost::coroutines::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 <<  " ";

std::cout << "\nDone" << std::endl;

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

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

```
boost::coroutines::coroutine<int>::push_type sink(
    [&](boost::coroutines::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));
```

Exit a *coroutine-function*

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

Important

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

Class `coroutine<>::pull_type`

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

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

    template<
        typename Fn,
        typename StackAllocator = stack_allocator,
        typename Allocator = std::allocator< coroutine >
    >
    pull_type( Fn fn, attributes const& attr = attributes(),
               StackAllocator const& stack_alloc = StackAllocator(),
               Allocator const& alloc = Allocator() );

    template<
        typename Fn,
        typename StackAllocator = stack_allocator,
        typename Allocator = std::allocator< coroutine >
    >
    pull_type( Fn && fn, attributes const& attr = attributes(),
               StackAllocator stack_alloc = StackAllocator(),
               Allocator const& alloc = Allocator() );

    pull_type( pull_type && other);

    pull_type & operator=( pull_type && other);

    operator unspecified-bool-type() const;

    bool operator!() const;

    void swap( pull_type & other);

    bool empty() const;

    pull_type & operator()();

    bool has_result() const;

    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, typename StackAllocator, typename Allocator > pull_type( Fn fn, attributes const& attr, StackAllocator const& stack_alloc, Allocator const& alloc)`

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

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 and internal data are allocated by Allocator.

Throws: Exceptions thrown inside *coroutine-function*.

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

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

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 and internal data are allocated by Allocator.

Throws: Exceptions thrown inside *coroutine-function*.

`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.

`bool empty()`

Returns: If `*this` refers to *not-a-coroutine*, 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 are passed to the coroutine-function).

Throws: Exceptions thrown inside *coroutine-function*.

`bool has_result()`

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

Returns: If `*this` has a, the function returns true. Otherwise false.

Throws: Nothing.

`R get()()`

```
R    coroutine<R>::pull_type::get();
R&   coroutine<R&>::pull_type::get();
void coroutine<void>pull_type::get()=delete;
```

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

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

Throws: Nothing.

`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 a end range-iterator (input-iterator).

Class `coroutine<>::push_type`

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

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

    template<
        typename Fn,
        typename StackAllocator = stack_allocator,
        typename Allocator = std::allocator< coroutine >
    >
    push_type( Fn fn, attributes const& attr = attributes(),
               StackAllocator const& stack_alloc = StackAllocator(),
               Allocator const& alloc = Allocator() );

    template<
        typename Fn,
        typename StackAllocator = stack_allocator,
        typename Allocator = std::allocator< coroutine >
    >
    push_type( Fn && fn, attributes const& attr = attributes(),
               StackAllocator stack_alloc = StackAllocator(),
               Allocator const& alloc = Allocator() );

    push_type( push_type && other);

    push_type & operator=( push_type && other);

    operator unspecified-bool-type() const;

    bool operator!() const;

    void swap( push_type & other);

    bool empty() const;

    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, typename StackAllocator, typename Allocator > push_type( Fn fn, attributes const& attr, StackAllocator const& stack_alloc, Allocator const& alloc)`

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

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 and internal data are allocated by Allocator.

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

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

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 and internal data are allocated by Allocator.

`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.

`bool empty()`

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

Throws: Nothing.

`push_type<> & operator()(Arg&& arg)`

```
push_type& coroutine<Arg>::push_type::operator()(const Arg&);
push_type& coroutine<Arg>::push_type::operator()(Arg&&);
push_type& coroutine<Arg&>::push_type::operator()(Arg&);
push_type& 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.

`T caller_type::operator()( R)`

Effects: Gives execution control back to calling context by returning a value of type R. The return type of this function is a *boost::tuple<>* containing the arguments passed to *boost::coroutines::coroutine<>::operator()*.

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).

Attributes

Class `attributes` is used to transfers parameters required to setup a coroutines's context.

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

    attributes() BOOST_NOEXCEPT;

    explicit attributes( std::size_t size_) BOOST_NOEXCEPT;

    explicit attributes( flag_unwind_t do_unwind_) BOOST_NOEXCEPT;

    explicit attributes( flag_fpu_t preserve_fpu_) BOOST_NOEXCEPT;

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

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

    explicit attributes( flag_unwind_t do_unwind_, flag_fpu_t preserve_fpu_) BOOST_NOEXCEPT;
};
```

`attributes()`

Effects: Default constructor using `ctx::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 inner `context`. 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 inner `context` 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 if a `context` switches. THe default stacksize is used for the inner `context` and the 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 preserved during 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 `ctx::default_stacksize()`.

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 both values 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 pointer not returned 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()` returns an address from the top of the stack (growing downwards) or the bottom of the stack (growing upwards).

Class *stack_allocator*

Boost.Coroutine provides the class *boost::coroutines::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.

Note

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

```
class stack_allocator
{
    static bool is_stack_unbound();

    static std::size_t maximum_stacksize();

    static std::size_t default_stacksize();

    static std::size_t minimum_stacksize();

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

    void deallocate( stack_context &);
}
```

`static bool is_stack_unbound()`

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

`static std::size_t maximum_stacksize()`

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

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

`static std::size_t default_stacksize()`

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

`static std::size_t minimum_stacksize()`

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

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

Preconditions: `minimum_stacksize() > size` and `! is_stack_unbound() && ( maximum_stacksize() < size)`.

Effects: Allocates memory of at least `size` Bytes and stores a pointer to the stack and its actual size in `sctx`.

Returns: Returns pointer to the start address of the new stack. Depending on the architecture the stack grows downwards/upwards the returned address is the highest/lowest address of the stack.

`void deallocate( stack_context & sctx)`

Preconditions: `sctx.sp` is valid, `minimum_stacksize() > sctx.size` and `! is_stack_unbound() && ( maximum_stacksize() < size)`.

Effects: Deallocates the stack space.

Class *stack_context*

Boost.Coroutine provides the class *boost::coroutines::stack_context* which will contain the stack pointer and the size of the stack. In case of a *segemented-stack boost::coroutines::stack_context* contains some extra controll structures.

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

    // might contain addition controll 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.

Segmented stacks

Boost.Coroutine supports usage of a *segemented-stack*, e. g. the size of the stack grows on demand. The coroutine is created with an minimal stack size and will be increased as required.

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

Performance

Performance of **Boost.Coroutine** was measured on the platforms shown in the following table. Performance measurements were taken using `rdtsc` and `::clock_gettime()`, with overhead corrections, on x86 platforms. In each case, stack protection was active, cache warm-up was accounted for, and the one running thread was pinned to a single CPU. The code was compiled using the build options, 'variant = release cxxflags = -DBOOST_DISABLE_ASSERTS'.

The numbers in the table are the number of cycles per iteration, based upon an average computed over 10 iterations.

Table 1. Perfomance of coroutine switch

Platform	CPU cycles	nanoseconds
AMD Athlon 64 DualCore 4400+ (32bit Linux)	58	65
Intel Core2 Q6700 (64bit Linux)	80	28

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.