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*.

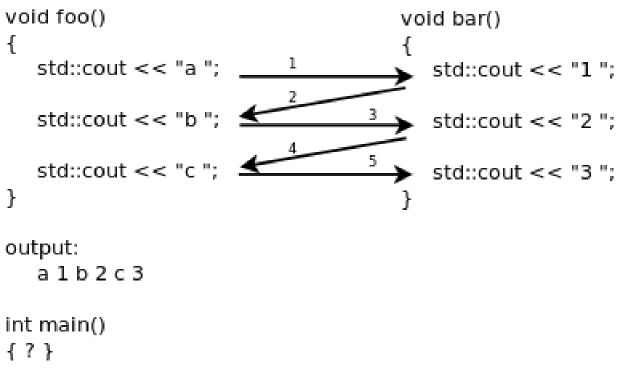
Introduction

Definition

In computer science routines are defined as a sequence of operations. The execution of routines form a parent-child relationship and the child terminates always before the parent. Coroutines are a generalization of routines. The principal difference between coroutines and routines is that a coroutine enables explicit suspend and resume of their progress via additional operations by preserving local state, e.g. a coroutine is a kind of continuation. A continuation is a object representing a suspended execution (registers, stack). Each coroutine has its own stack and local variables, sub-routine calls etc. In this sense coroutines are (actually) a language concept.

How it works

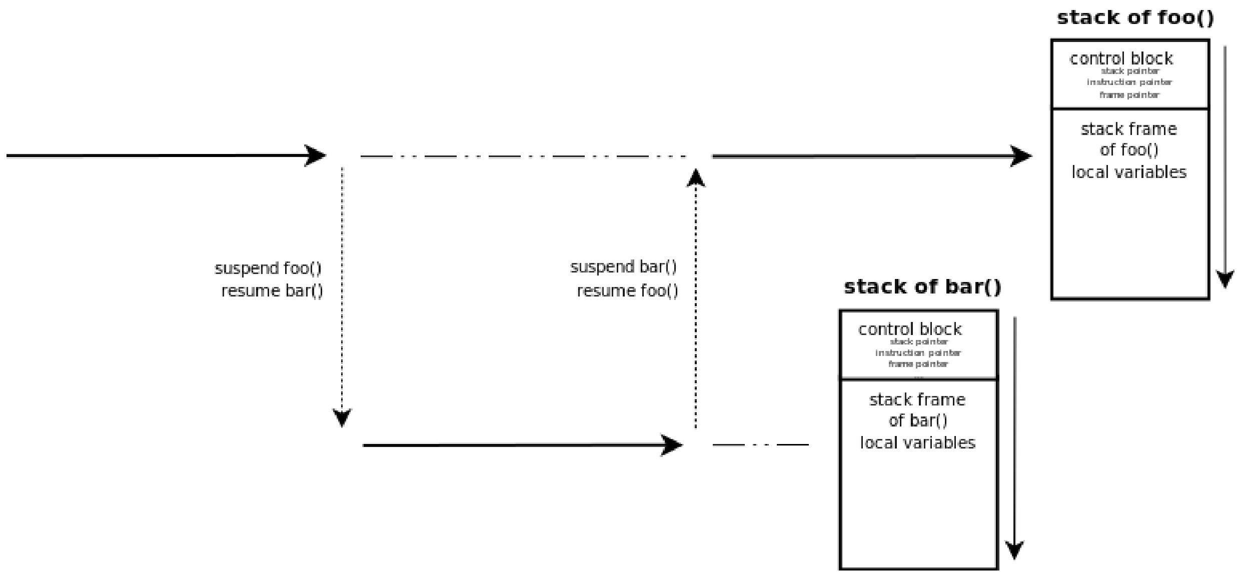
Functions foo() and bar() are supposed to alternate their execution (leave and enter function body).



```
void foo()                              void bar()
{                                       {
    std::cout << "a ";                      std::cout << "1 ";

    std::cout << "b ";                      std::cout << "2 ";

    std::cout << "c ";                      std::cout << "3 ";
}                                       }

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').

Example: asio::io_stream with std::stream

This section demonstrates how stackfull coroutines help to use standard C++ IO-streams together with IO-demultiplexer like *boost::asio::io_sevice* (using non-blocking IO).

```
int main( int argc, char * argv[])
{
    ...
    {
        boost::asio::io_service io_service;
        io_service.post(
            boost::bind(
                & server::start,
                server::create(
                    io_service, port) ) );
        io_service.run();
    }
    ...
}
```

server accepts connection-requests made by clients, creates for each new connection an instance of type *session* and invokes *session::start()* on it.

```
class server : public boost::enable_shared_from_this< server >
{
private:
    boost::asio::io_service         &   io_service_;
    boost::asio::ip::tcp::acceptor      acceptor_;

    void handle_accept_( session * new_session, boost::system::error_code const& error)
    {
        if ( ! error)
        {
            // start asynchronous read
            new_session->start();

            // start asynchronous accept
            start();
        }
    }

    server( boost::asio::io_service & io_service, short port) :
        io_service_( io_service),
        acceptor_(
            io_service_,
            boost::asio::ip::tcp::endpoint( boost::asio::ip::tcp::v4(), port) )
    {}

public:
    typedef boost::shared_ptr< server > ptr_t;

    static ptr_t create( boost::asio::io_service & io_service, short port)
    { return ptr_t( new server( io_service, port) ); }

    void start()
    {
        // create new session which gets started if asynchronous
        // accept completes
        session * new_session( new session( io_service_) );
        acceptor_.async_accept(
            new_session->socket(),
            boost::bind( & server::handle_accept_, this->shared_from_this(),
                new_session, boost::asio::placeholders::error) );
    }
};
```

Each *session* communicates with the connected client and handles the requests. The application protocol in this example uses TCP-sockets as channel and 'newline' to separate the messages in the byte stream. An application protocol is a set of rules for the order in which messages are exchanged. *std::istream* is used to extract the messages from the character stream . Message 'exit' terminates the session.

```
class session : private boost::noncopyable
{
private:
    void handle_read_( coro_t::caller_type & self)
    {
        // create stream-buffer reading from socket
        inbuf buf( socket_);
        std::istream s( & buf);

        // messages are separated by 'newline'
        std::string msg;
        std::getline( s, msg);
        std::cout << msg << std::endl;

        // terminate session for message 'exit'
        // else do asynchronous read
        if ( "exit" == msg)
            io_service_.post(
                boost::bind(
                    & session::destroy_, this) );
        else
            start();
    }

    void destroy_()
    { delete this; }

    boost::asio::io_service     &   io_service_;
    boost::asio::ip::tcp::socket    socket_;

public:
    session( boost::asio::io_service & io_service) :
        io_service_( io_service),
        socket_( io_service_)
    { std::cout << "service(): " << socket_.remote_endpoint() << std::endl; }

    ~session()
    { std::cout << "~service(): " << socket_.remote_endpoint() << std::endl; }

    boost::asio::ip::tcp::socket & socket()
    { return socket_; }

    void start()
    {
        // register on io_service for asynchronous read
        io_service_.async_read(
            socket_,
            boost::bind(
                & session::handle_read_, this->shared_from_this(), _1, _2) );
    }
};
```

Function *std::getline()* returns only if a 'newline' was read from the socket. Therefore the application will block until 'newline' is received by the socket. The stream-buffer used by the stream maintains an internal buffer which gets (re-)filled by its function *stream_buf::underflow()*. *stream_buf::underflow()* does the read-operation on the socket. The C++ IO-streams framework does not provide an easy way to create an continuation which represents reading bytes from the socket.

Coroutines help in this case to make the application non-blocking even if no 'newline' was received. Class *session* creates a coroutine which uses *session::handle_read()* as *coroutine-function*. On a new created *session start()* called starting the coroutine. In the *coroutine-function session::handle_read()* the messages are received via *std::getline()* in a loop until 'exit' is delivered.

```
class session : private boost::noncopyable
{
private:
    void handle_read_( coro_t::caller_type & ca)
    {
        // create stream-buffer with coroutine
        inbuf buf( socket_, coro_, ca);
        std::istream s( & buf);

        std::string msg;
        do
        {
            // read message
            // we not block if no newline was received yet
            std::getline( s, msg);
            std::cout << msg << std::endl;
        } while ( msg != "exit");
        io_service_.post(
                boost::bind(
                    & session::destroy_, this) );
    }

    void destroy_()
    { delete this; }

    coro_t                          coro_;
    boost::asio::io_service     &   io_service_;
    boost::asio::ip::tcp::socket    socket_;

public:
    session( boost::asio::io_service & io_service) :
        coro_(),
        io_service_( io_service),
        socket_( io_service_)
    { std::cout << "service(): " << socket_.remote_endpoint() << std::endl; }

    ~session()
    { std::cout << "~service(): " << socket_.remote_endpoint() << std::endl; }

    boost::asio::ip::tcp::socket & socket()
    { return socket_; }

    void start()
    {
        // create and start a coroutine
        // handle_read_() is used as coroutine-function
        coro_ = coro_t( boost::bind( & session::handle_read_, this, _1) );
    }
};
```

The stream-buffer is created with *boost::coroutines::coroutine<>::caller_type* and handles suspend/resume of this code path depending on if bytes can be read from the socket.

```
class inbuf : public std::streambuf,
              private boost::noncopyable
{
private:
    static const std::streamsize        pb_size;

    enum
    { bf_size = 16 };

    int fetch_()
    {
        std::streamsize num = std::min(
            static_cast< std::streamsize >( gptr() - eback() ), pb_size);

        std::memmove(
            buffer_ + ( pb_size - num),
            gptr() - num, num);

        // read bytes from the socket into internal buffer 'buffer_'
        // make coro_t::operator() as callback, invoked if some
        // bytes are read into 'buffer_'
        s_.async_read_some(
                boost::asio::buffer( buffer_ + pb_size, bf_size - pb_size),
                boost::bind( & coro_t::operator(), & coro_, _1, _2) );
        // suspend this coroutine
        ca_();

        // coroutine was resumed by boost::asio::io_sevice
        boost::system::error_code ec;
        std::size_t n = 0;

        // check arguments
        boost::tie( ec, n) = ca_.get();

        // check if an error occurred
        if ( ec)
        {
            setg( 0, 0, 0);
            return -1;
        }

        setg( buffer_ + pb_size - num, buffer_ + pb_size, buffer_ + pb_size + n);
        return n;
    }

    boost::asio::ip::tcp::socket      &   s_;
    coro_t                            &   coro_;
    coro_t::caller_type               &   ca_;
    char                                  buffer_[bf_size];

protected:
    virtual int underflow()
    {
        if ( gptr() < egptr() )
            return traits_type::to_int_type( * gptr() );

        if ( 0 > fetch_() )
            return traits_type::eof();
        else
            return traits_type::to_int_type( * gptr() );
    }

public:
    inbuf(
            boost::asio::ip::tcp::socket & s,
            coro_t & coro,
            coro_t::caller_type & ca) :
        s_( s), coro_( coro), ca_( ca), buffer_()
    { setg( buffer_ + 4, buffer_ + 4, buffer_ + 4); }
};
const std::streamsize inbuf::pb_size = 4;
```

inbuf::fetch() uses *boost::coroutines::coroutine<>::operator()* as callback for the asynchronous read-operation on the socket and suspends itself (*ca_()* jumps back to *session::start()*). If some bytes are available in the socket receive buffer *boost::asio::io_sevice* copies the bytes to the internal buffer *buffer_* and invokes the callback which resumes the coroutine (*ca_()* returns).

Coroutine

Each instance of *coroutine* has its own context of execution (CPU registers and stack space) or represents *not-a-coroutine* (similar to *boost::thread*). Objects of type *coroutine* are moveable but not copyable and can be returned by a function.

```
boost::coroutines::coroutine< void() > f();

void g()
{
    boost::coroutines::coroutine< void() > c( f() );
    c();
}
```

Note

Boost.Move is used to emulate rvalue references.

Creating a coroutine

A new *coroutine* is created from a *coroutine-function* (function or functor) which will be executed in a new *context* (CPU registers and stack space).

Note

coroutine-function is required to return *void* and accept a reference of type *boost::coroutines::coroutine<>::caller_type*.

The template argument *Signature* determines the data-types transferred to *coroutine-function* and from *coroutine-function* by calling *boost::coroutines::coroutine<>::operator()* and *boost::coroutines::coroutine<>::get()*.

```
typedef boost::coroutines::coroutine< int( std::string const&) > coro_t;

// void f( boost::coroutine< std::string const&( int) > & ca)
void f( coro_t::caller_type & ca)
{
    ...
    // access argument
    std::string str( ca.get() );
    ...
    ca( 7);
    ...
}

std::string str;
...
coro_t c( f);
// pass argument
c( str);
// returned value
int res = c.get();
```

The *coroutine-function* is started at *coroutine* construction (similar to *boost::thread*) in a newly created *coroutine* complete with registers, flags, stack and instruction pointer. If *coroutine-function* requires some arguments (types defined by *Signature*) on start-up those parameters must be applied to the *coroutine* constructor. A single arguments can be passed as it is:

```
typedef boost::coroutines::coroutine< int( std::string const&) > coro_t;

// void f( boost::coroutine< std::string const&( int) > & ca)
void f( coro_t::caller_type & ca);

std::string str("abc");
coro_t c( f, str);
```

For multiple arguments *boost::coroutines::coroutine<>::arguments* must be used (it is a typedef of *boost::tuple<>*):

```
typedef boost::coroutines::coroutine< int( int, std::string const&) > coro_t;

// void f( boost::coroutine< boost::tuple< int, std::string const& >( int) > & ca)
void f( coro_t::caller_type & ca);

std::string str("abc");
coro_t c( f, coro_t::arguments( 7, str) );
```

Note

The maximum number of arguments is limited to 10 (limit defined by **Boost.Tuple**).

Note

Parameters bound with *boost::bind()* to *coroutine-function* will not be part of the *boost::coroutines::coroutine<>::operator()* signature.

boost::coroutines::attributes, an additional constructor argument of *coroutine*, defines the stack size, stack unwinding and floating-point preserving behaviour used for *context* construction.

The *coroutine* constructor uses the *stack-allocator concept* to allocate an associated stack, and the destructor uses the same *stack-allocator concept* to deallocate the stack. The default *stack-allocator concept* is *stack-allocator*, but a custom stack-allocator can be passed to the constructor.

Calling a coroutine

The execution control is transferred to *coroutine* at construction (*coroutine-function* entered) - when control should be returned to the original calling routine, invoke *boost::coroutines::coroutine<>::operator()* on the first argument of type *boost::coroutines::coroutine<>::caller_type* inside *coroutine-function*. *boost::coroutines::coroutine<>::caller_type* is a typedef of *coroutine* with an inverted *Signature*. Inverted *Signature* means that the return type becomes an argument and vice versa. Multiple arguments are wrapped into *boost::tuple<>*.

```
void f( boost::coroutines::coroutine< std::string const&( int) & ca);
boost::coroutines::coroutine< int( std::string const&) > c1( f);

void g( boost::coroutines::coroutine< boost::tuple< X, Y >( int) & ca);
boost::coroutines::coroutine< int( X, X) > c2( g);
```

The current coroutine information (registers, flags, and stack and instruction pointer) is saved and the original context information is restored. Calling *boost::coroutines::coroutine<>::operator()* resumes execution in the coroutine after saving the new state of the original routine.

```
typedef boost::coroutines::coroutine< void() > coro_t;

// void fn( boost::coroutines::coroutine< void() > & ca, int j)
void fn( coro_t::caller_type & ca, int j)
{
    for( int i = 0; i < j; ++i)
    {
        std::cout << "fn(): local variable i == " << i << std::endl;

        // save current coroutine
        // value of local variable is preserved
        // transfer execution control back to main()
        ca();

        // coroutine<>::operator()() was called
        // execution control transferred back from main()
    }
}

int main( int argc, char * argv[])
{
    // bind parameter '7' to coroutine-fn
    coro_t c( boost::bind( fn, _1, 7) );

    std::cout << "main() starts coroutine c" << std::endl;

    while ( c)
    {
        std::cout << "main() calls coroutine c" << std::endl;
        // execution control is transferred to c
        c();
    }

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

    return EXIT_SUCCESS;
}

output:
    main() starts coroutine c
    fn(): local variable i == 0
    main() calls coroutine c
    fn(): local variable i == 1
    main() calls coroutine c
    fn(): local variable i == 2
    main() calls coroutine c
    fn(): local variable i == 3
    main() calls coroutine c
    fn(): local variable i == 4
    main() calls coroutine c
    fn(): local variable i == 5
    main() calls coroutine c
    fn(): local variable i == 6
    main() calls coroutine c
    Done
```

Warning

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

Transfer of data

Signature, the template argument of *coroutine*, defines the types transferred to and returned from the *coroutine-function*, e.g. it determines the signature of *boost::coroutines::coroutine<>::operator()* and the return-type of *boost::coroutines::coroutine<>::get()*.

Note

boost::coroutines::coroutine<>::caller_type is not part of *Signature* and *coroutine-function* is required to return void and accept *boost::coroutines::coroutine<>::caller_type* as argument.

boost::coroutines::coroutine<>::operator() accepts arguments as defined in *Signature* and returns a reference to *coroutine*. The arguments passed to *boost::coroutines::coroutine<>::operator()*, in one coroutine, is returned (as a *boost::tuple<>*) by *boost::coroutines::coroutine<>::get()* in the other coroutine. If *coroutine* is constructed and arguments are passed to the constructor, the *coroutine-function* will be entered and the arguments are accessed thorough *boost::coroutines::coroutine<>::get()* in *coroutine-function* on entry.

The value given to *boost::coroutines::coroutine<>::operator()* of *boost::coroutines::coroutine<>::caller_type*, in one coroutine, is returned by *boost::coroutines::coroutine<>::get()* in the other routine.

```
typedef boost::coroutines::coroutine< int( int) >    coro_t;

// void fn( boost::coroutines::coroutine< int( int) > & ca)
void fn( coro_t::caller_type & ca)
{
    // access the integer argument given to coroutine ctor
    int i = ca.get();
    std::cout << "fn(): local variable i == " << i << std::endl;

    // save current coroutine context and
    // transfer execution control back to caller
    // pass content of variable 'i' to caller
    // after execution control is returned back coroutine<>::operator()
    // returns and the transferred integer s accessed via coroutine<>::get()
    i = ca( i).get();

    // i == 10 because c( 10) in main()
    std::cout << "fn(): local variable i == " << i << std::endl;
    ca( i);
}

int main( int argc, char * argv[])
{
    std::cout << "main(): call coroutine c" << std::endl;
    coro_t c( fn, 7);

    int x = c.get();
    std::cout << "main(): transferred value: " << x << std::endl;

    x = c( 10).get();
    std::cout << "main(): transferred value: " << x << std::endl;

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

    return EXIT_SUCCESS;
}

output:
    main(): call coroutine c
    fn(): local variable i == 7
    main(): transferred value: 7
    fn(): local variable i == 10
    main(): transferred value: 10
    Done
```

coroutine-function with multiple arguments

If *coroutine-function* has more than one argument *boost::coroutines::coroutine<>::operator()* has the same size of arguments and *boost::coroutines::coroutine<>::get()* from *boost::coroutines::coroutine<>::caller_type* returns a *boost::tuple<>* corresponding to the arguments of *Signature*. *boost::tie* helps to access the values stored in the *boost::tuple<>* returned by *boost::coroutines::coroutine<>::get()*.

```
typedef boost::coroutines::coroutine< int(int,int) > coro_t;

// void fn( boost::coroutines::coroutine< boost::tuple< int, int >( int) > & ca)
void fn( coro_t::caller_type & ca)
{
    int a, b;
    boost::tie( a, b) = ca.get();
    boost::tie( a, b) = ca( a + b).get();
    ca( a + b);
}

int main( int argc, char * argv[])
{
    std::cout << "main(): call coroutine c" << std::endl;
    coro_t coro( fn, coro_t::arguments( 3, 7) );

    int res = coro.get();
    std::cout << "main(): 3 + 7 == " << res << std::endl;

    res = coro( 5, 7).get();
    std::cout << "main(): 5 + 7 == " << res << std::endl;

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

    return EXIT_SUCCESS;
}

output:
    main(): call coroutine c
    main(): 3 + 7 == 10
    main(): 5 + 7 == 12
    Done
```

Transfer of pointers and references

You can transfer references and pointers from and to coroutines but as usual you must take care (scope, no re-assignment of const references etc.). In the following code `x` points to `local` which is allocated on stack of `c`. When `c` goes out of scope the stack becomes deallocated. Using `x` after `c` is gone will fail!

```
struct X
{
    void g();
};

typedef boost::coroutines::coroutine< X*() >    coro_t;

// void fn( boost::coroutines::coroutine< void( X*) > & ca)
void fn( coro_t::caller_t & ca) {
    X local;
    ca( & local);
}

int main() {
    X * x = 0;
    {
        coro_t c( fn);
        x = c.get(); // let x point to X on stack owned by c
        // stack gets unwound -> X will be destructed
    }
    x->g(); // segmentation fault!
    return EXIT_SUCCESS;
}
```

Range iterators

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

```
typedef boost::coroutines::coroutine< int() >       coro_t;
typedef boost::range_iterator< coro_t >::type       iterator_t;

// void power( boost::coroutines::coroutine< void( int) > & ca, int number, int exponent)
void power( coro_t::caller_type & ca, int number, int exponent)
{
    int counter = 0;
    int result = 1;
    while ( counter++ < exponent)
    {
            result = result * number;
            ca( result);
    }
}

int main()
{
    coro_t c( boost::bind( power, _1, 2, 8) );
    iterator_t e( boost::end( c) );
    for ( iterator_t i( boost::begin( c) ); i != e; ++i)
        std::cout << * i <<  " ";

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

    return EXIT_SUCCESS;
}

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

`BOOST_FOREACH` can be used to iterate over the coroutine range too.

```
typedef boost::coroutines::coroutine< int() >       coro_t;
typedef boost::range_iterator< coro_t >::type       iterator_t;

// void power( boost::coroutines::coroutine< void( int) > & ca, int number, int exponent)
void power( coro_t::caller_type & ca, int number, int exponent)
{
    int counter = 0;
    int result = 1;
    while ( counter++ < exponent)
    {
            result = result * number;
            ca( result);
    }
}

int main()
{
    coro_t c( boost::bind( power, _1, 2, 8) );
    BOOST_FOREACH( int i, c)
    { std::cout << i <<  " "; }

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

    return EXIT_SUCCESS;
}

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

Input iterators are created from coroutines of type `coroutine< void( T) >`.

Exit a *coroutine-function*

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

```
typedef boost::coroutines::coroutine< int(int,int) > coro_t;

// void power( boost::coroutines::coroutine< boost::tuple< int, int >( int) > & ca, int number, ↵
int exponent)
void fn( coro_t::caller_type & ca)
{
    int a, b;
    boost::tie( a, b) = ca.get();
    boost::tie( a, b) = ca( a + b).get();
    ca( a + b);
}

int main( int argc, char * argv[])
{
    std::cout << "main(): call coroutine c" << std::endl;
    coro_t coro( fn, coro_t::arguments( 3, 7) );

    BOOST_ASSERT( coro);
    int res = coro.get();
    std::cout << "main(): 3 + 7 == " << res << std::endl;

    res = coro( 5, 7).get();
    BOOST_ASSERT( ! coro);
    std::cout << "main(): 5 + 7 == " << res << std::endl;

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

    return EXIT_SUCCESS;
}

output:
    main(): call coroutine c
    main(): 3 + 7 == 10
    main(): 5 + 7 == 12
    Done
```

Important

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

Exceptions in *coroutine-function*

An exception thrown inside *coroutine-function* will transferred via exception-pointer (see **Boost.Exception** for details) and re-thrown by constructor or *boost::coroutines::coroutine<>::operator()*.

```
typedef boost::coroutines::coroutine< void() >    coro_t;

// void fn( boost::coroutines::coroutine< void() > & ca)
void fn( coro_t::caller_type & ca)
{
    ca();
    throw std::runtime_error("abc");
}

int main( int argc, char * argv[])
{
    coro_t c( f);
    try
    {
        c();
    }
    catch ( std::exception const& e)
    {
        std::cout << "exception catched:" << e.what() << std::endl;
        return EXIT_FAILURE;
    }

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

    return EXIT_SUCCESS;
}

output:
    exception catched: abc
```

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( 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.

```
typedef boost::coroutines::coroutine< void() >    coro_t;

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

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

// void fn( boost::coroutines::coroutine< void() > & ca)
void fn( coro_t::caller_type & ca)
{
    X x;

    for ( int  = 0;; ++i)
    {
        std::cout << "fn(): " << i << std::endl;
        // transfer execution control back to main()
        ca();
    }
}

int main( int argc, char * argv[])
{
    {
        coro_t c( fn,
                  boost::coroutines::attributes(
                    boost::ctx::default_stacksize(),
                    boost::coroutines::stack_unwind) );

        c();
        c();
        c();
        c();
        c();

        std::cout << "c is complete: " << std::boolalpha << c.is_complete() << "\n";
    }

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

    return EXIT_SUCCESS;
}

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

Important

You must not swallow *boost::coroutines::detail::forced_unwind* exceptions!

FPU preserving

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.

Class coroutine

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

template< typename Signature >
class coroutine;

template<
    typename R,
    typename ArgTypes...
>
class coroutine< R ( ArgTypes...) >
{
public:
    typedef unspec-type caller_type;
    typedef unspec-type arguments;

    coroutine();

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

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

    coroutine( coroutine && other);

    coroutine & operator=( coroutine && other);

    operator unspecified-bool-type() const;
    bool operator!() const;

    void swap( coroutine & other);

    bool empty() const;

    coroutine & operator()(A0 a0, ..., A9 a9);

    R get() const;
};

template< typename Signature >
void swap( coroutine< Signature > & l, coroutine< Signature > & r);

template< typename T >
range_iterator< coroutine< T() > >::type begin( coroutine< T() > &);
template< typename T >
range_iterator< coroutine< void(T) > >::type begin( coroutine< void(T) > &);

template< typename T >
range_iterator< coroutine< T() > >::type end( coroutine< T() > &);
template< typename T >
range_iterator< coroutine< void(T) > >::type end( coroutine< void(T) > &);
```

`coroutine()`

Effects:	Creates a coroutine representing *not-a-coroutine*.
Throws:	Nothing.

`template< typename Fn, typename StackAllocator, typename Allocator > coroutine( 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 > coroutine( 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.

`coroutine( coroutine && other)`

Effects:	Moves the internal data of `other` to `*this`. `other` becomes *not-a-coroutine*.
Throws:	Nothing.

`coroutine & operator=( coroutine && 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.

`coroutine<> & operator()(A0 a0, A9 a9)`

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

[Effects:

Execution control is transferred to *coroutine-function* and the arguments `a0`,..., are passed to the coroutine-function.

Throws: Exceptions thrown inside *coroutine-function*.

`R get()()`

Preconditions: `*this` is not a *not-a-coroutine*, `! is_complete()`.

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

Throws: Nothing.

`void swap( coroutine & 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 Signature >
void swap( coroutine< Signature > & l, coroutine< Signature > & r);
```

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

Non-member function `begin( coroutine< T() > &)`

```
template< typename T >
range_iterator< coroutine< T() > >::type begin( coroutine< T() > &);
```

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

Non-member function `begin( coroutine< void(T) > &)`

```
template< typename T >
range_iterator< coroutine< void(T) > >::type begin( coroutine< void(T) > &);
```

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

Non-member function `end( coroutine< T() > &)`

```
template< typename T >
range_iterator< coroutine< T() > >::type end( coroutine< T() > &);
```

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

Non-member function `end( coroutine< void(T) > &)`

```
template< typename T >
range_iterator< coroutine< void(T) > >::type end( coroutine< void(T) > &);
```

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;
    bool            preserve_fpu;

    attributes() BOOST_NOEXCEPT :
        size( ctx::default_stacksize() ),
        do_unwind( stack_unwind),
        preserve_fpu( true)
    {}

    explicit attributes( std::size_t size_) BOOST_NOEXCEPT :
        size( size_),
        do_unwind( stack_unwind),
        preserve_fpu( true)
    {}

    explicit attributes( flag_unwind_t do_unwind_) BOOST_NOEXCEPT :
        size( ctx::default_stacksize() ),
        do_unwind( do_unwind_),
        preserve_fpu( true)
    {}

    explicit attributes( bool preserve_fpu_) BOOST_NOEXCEPT :
        size( ctx::default_stacksize() ),
        do_unwind( stack_unwind),
        preserve_fpu( preserve_fpu_)
    {}

    explicit attributes(
            std::size_t size_,
            flag_unwind_t do_unwind_) BOOST_NOEXCEPT :
        size( size_),
        do_unwind( do_unwind_),
        preserve_fpu( true)
    {}

    explicit attributes(
            std::size_t size_,
            bool preserve_fpu_) BOOST_NOEXCEPT :
        size( size_),
        do_unwind( stack_unwind),
        preserve_fpu( preserve_fpu_)
    {}

    explicit attributes(
            flag_unwind_t do_unwind_,
            bool preserve_fpu_) BOOST_NOEXCEPT :
        size( ctx::default_stacksize() ),
        do_unwind( do_unwind_),
        preserve_fpu( preserve_fpu_)
    {}
};
```

`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( bool 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, bool 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, bool 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 of a coroutine grows on demand. The coroutine is created with a stack with an minimal stack and if the coroutine is execute the stack size is increased as required.

Segmented stack are currently only supported by **gcc** from version **4.7** onwards. n order to use *segemented-stack* compile **Boost.Coroutine** with **toolset=gcc segmented-stacks=on** at b2/bjam command-line. Application using **Boost.Coroutine** with enabled __segmented-stack__ 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.