Document number: N3708
Date: 2013-03-04
Project: ProgrammingLanguageC++LibraryWorking Group
Reply-to: Oliver Kowalke(oliver dot kowalke at gmail dot com) Nat Goodspeed( nat at lindenlab dot com)

# A proposal to add coroutines to the C++ standard library

Introduction . . . 1
Motivation . . . 2
Impact on the Standard . . . 5
Design Decisions . . . 5
Technical Specification . . . 10
References . . . 17

## Introduction

This proposal suggests adding two first-class continuations to the C++ standard library:
`std::coroutine<T>::pull_type` and `std::coroutine<T>::push_type` .

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).

**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

Several programming languages adopted particular features (C# yield, Python generators, ...).

| | | | | | |
|---|---|---|---|---|---|
| BCPL | Erlang | Go | Lua | PHP | Ruby |
| C# | F# | Haskell | Modula-2 | Prolog | Sather |
| D | Factor | Icon | Perl | Python | Scheme |

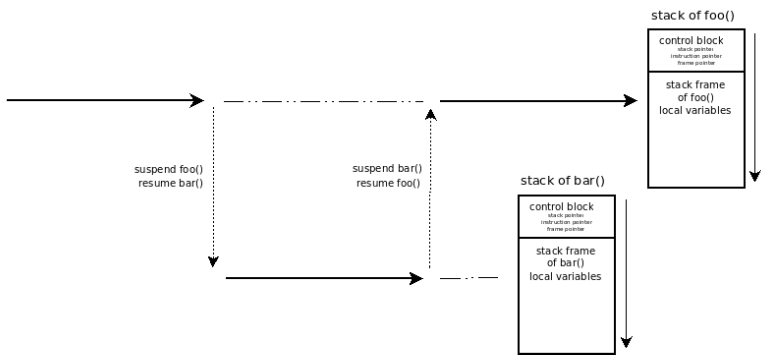
Table 1: some programming languages with native support of coroutines[14]

Coroutines are useful in simulation, artificial intelligence, concurrent programming, text processing and data manipulation,[3] 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: control re-enters the coroutine at the point at which it was suspended.



Both concepts are equivalent and a coroutine library can provide either symmetric or asymmetric coroutines.[3]

**stackfulness:** In contrast to a stackless coroutine a stackful coroutine allows to suspend from nested stackframes. The execution resumes at the exact 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

This proposal refers to boost.coroutine[6] as reference implementation - providing a test suite and examples (some are described in this section).

In order to support a broad range of execution control behaviour `std::coroutine<T>::push_type` and `std::coroutine<T>::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 even if the tree structure is different.

The same fringe problem could be solved using coroutines by iterating over the leaf nodes and comparing this sequence via `std::equal()` . The range of data values is generated by function `traverse()` which recursively traverses the tree and passes each node's data value to its `std::coroutine<T>::push_type` .
`std::coroutine<T>::push_type` suspends the recursive computation and transfers the data value to the main execution context.
`std::coroutine<T>::pull_type::iterator` , created from `std::coroutine<T>::pull_type` , steps over those data values and delivers them to `std::equal()` for comparison. Each increment of `std::coroutine<T>::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 std::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(std::begin(left_d_reader),
                            std::end(left_d_reader),
                            std::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(std::begin(left_d_reader),
                            std::end(left_d_reader),
                            std::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
```

### asynchronous operations with boost.asio

In the past the code using asio's *asynchronous-operations* was scattered by callbacks. boost.asio[5] provides with its new *asynchronous-result* feature a new way to simplify the code and make it easier to read.
yield_context[15] uses internally boost.coroutine[6]:

```
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);
}
```

### *C# await*

*C#* contains the two keywords *async* and *await*. *async* introduces a control flow that involves awaiting asynchronous operations. The compiler reorganizes the code into a continuation-passing style. *await* wraps the rest of the function after calling *await* into a continuation if the asynchronous operation has not yet completed.
The project await_emu[8] uses boost.coroutine[6] for a proof-of-concept demonstrating the implementation of a full emulation of *C# await* as a library extension. Because of stackful coroutines *await* is **not limited** by "one level" as in *C#*.
Evgeny Panasyuk describes the advantages of boost.coroutine[6] over *await* at Channel 9 - 'The Future of C++'[8].

```
int bar(int i) {
    // await is not limited by "one level" as in C#
    auto result = await async([i]{ return reschedule(), i*100; });
    return result + i*10;
}

int foo(int i) {
    cout << i << ":\tbegin" << endl;
    cout << await async([i]{ return reschedule(), i*10; }) << ":\tbody" << endl;
    cout << bar(i) << ":\tend" << endl;
    return i*1000;
}

void async_user_handler() {
    vector<future<int>> fs;

    // instead of 'async' at function signature, 'asynchronous' should be
    // used at the call place:
    for(auto i=0; i!=5; ++i)
        fs.push_back( asynchronous([i]{ return foo(i+1); }) );

    for(auto &&f : fs)
        cout << await f << ":\tafter␣end" << endl;
}
```

## Impact on the Standard

This proposal is a library extension. It does not require changes to any standard classes, functions or headers. It can be implemented in C++03 and C++11 and requires no core language changes.

## Design Decisions

### Proposed Design

The design suggests two coroutine types - `std::coroutine<T>::push_type` and `std::coroutine<T>::pull_type` - providing a unidirectional transfer of data.

**std::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 `std::coroutine<T>::pull_type` takes a function (*coroutine-function*) accepting a reference to a `std::coroutine<T>::push_type` as argument. Instantiating a `std::coroutine<T>::pull_type` passes the control of execution to *coroutine-function* and a complementary `std::coroutine<T>::push_type` is synthesized by the runtime and passed as reference to *coroutine-function*.

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

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

```
std::coroutine<int>::pull_type source(
    [&](std::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 `std::coroutine<T>::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 `std::coroutine<T>::pull_type`.
A `std::coroutine<T>::push_type` is automatically generated by the runtime and passed as reference to the lambda function. Each time the lambda function calls `std::coroutine<T>::push_type::operator()(Arg&&)` with another Fibonacci number, `std::coroutine<T>::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 `std::coroutine<T>::pull_type` provides input iterators and `std::begin()` / `std::end()` are overloaded, a *range-based for*-loop can be used to iterate over the generated Fibonacci numbers.

**std::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 `std::coroutine<T>::push_type` takes a function (*coroutine-function*) accepting a reference to a `std::coroutine<T>::pull_type` as argument. In contrast to `std::coroutine<T>::pull_type`, instantiating a `std::coroutine<T>::push_type` does not pass the control of execution to *coroutine-function* - instead the first call of `std::coroutine<T>::push_type::operator()(Arg&&)` synthesizes a complementary `std::coroutine<T>::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.

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

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

const int num=5, width=15;
std::coroutine<std::string>::push_type writer(
    [&](std::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(std::begin(words),std::end(words),std::begin(writer));

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

In this example a `std::coroutine<T>::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 `std::coroutine<T>::push_type`.
The main execution context passes the strings to the *coroutine-function* by calling `std::coroutine<T>::push_type::operator()(Arg&&)`.
A `std::coroutine<T>::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 `std::coroutine<T>::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 `std::coroutine<T>::push_type` provides output iterators and `std::begin()` / `std::end()` are overloaded, the `std::copy` algorithm can be used to iterate over the vector containing the strings and pass them one by one to the coroutine.

**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 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 implementer is free to deallocate the stack or cache it for future usage (for coroutines created later).

**segmented stack:** `std::coroutine<T>::push_type` and `std::coroutine<T>::pull_type` must 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)[12] is known to support segmented stacks. With version 1.54 boost.coroutine[6] 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.

This also includes the floating point environment as required by the ABI. The implementer can omit preserving the floating point env if he can predict that it's safe.

On POSIX systems, a coroutine context switch must not preserve signal masks for performance reasons.

A context switch is done via `std::coroutine<T>::push_type::operator()(Arg&&)` and `std::coroutine<T>::pull_type::operator()()` .

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

```
std::coroutine<int>::pull_type source( // constructor enters coroutine-function
    [&](std::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 `std::coroutine<T>::push_type` from the main-context the framework synthesizes a `std::coroutine<T>::pull_type` associated with the `std::coroutine<T>::push_type` instance in the main-context. The synthesized `std::coroutine<T>::pull_type` is passed as argument to *coroutine-function*.

The main-context must call this `std::coroutine<T>::push_type::operator()(Arg&&)` in order to transfer each data value into the *coroutine-function*.
Access to the transferred data value is given by `std::coroutine<T>::pull_type::get()`.

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

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

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

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

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

An exception thrown inside a `std::coroutine<T>::push_type`'s *coroutine-function* will be re-thrown by `std::coroutine<T>::push_type::operator()(Arg&&)`.

## Other libraries

**boost.coroutine from 'Google Summer of Code 2006':** boost.coroutine[6] is a follow-up to boost.coroutine (Google Summer of Code 2006)[9] which is unfinished and not part of the official boost release: development of this library was stopped.
During the boost review process the interface of boost.coroutine[6] was changed; it differs fairly significantly from boost.coroutine (Google Summer of Code 2006)[9].

**Mordor:** Mordor[10] is another C++ library implementing cooperative multitasking in order to achieve high I/O performance. The difference from this design is that this proposal focuses on enhanced control flow, while Mordor[10] abstracts on the level of tasking: providing a cooperatively scheduled fiber engine.

**AT&T Task Library:** Another design of a task library was published by AT&T[11] describing a tasking system with non-preemptive scheduling.
`std::coroutine<T>::push_type` / `std::coroutine<T>::pull_type` does not provide scheduling logic but could be used as the basic mechanism for such a tasking abstraction.

**C++ proposal: resumable functions (N3328[4]):** This proposal is a library superset of N3328: the *resumable function* can be implemented on top of coroutines. The proposed coroutine library does not require memory allocation for the future on a context switch and does not require language changes (no keywords like *resumable* and *await*).
As described in N3328 section 3.2.5 'Function Prolog' - the body of a *resumable function* is transformed into a switch statement. This is similar to the stackless coroutines of Python and C#.
A proof-of-concept how *await* could be built upon boost.coroutine[6] has already been implemented in await_emu[8].

Without stackfulness and first-class semantics, some useful execution control flows cannot be supported (for instance cooperative multitasking, checkpointing) and recursive problems such as the 'same fringe' example become much more difficult.

## Technical Specification

### std::coroutine<>::pull_type

Defined in header `<coroutine>` .

| |
|---|
| `template<class T> class coroutine<T>::pull_type;` |
| `template<class T> class coroutine<T&>::pull_type;` |
| `template<> class coroutine<void>::pull_type;` |

The class `std::coroutine<T>::pull_type` provides a mechanism to receive data values from another execution context.

### member types

| | | |
|---|---|---|
| `iterator` | std::input_iterator | (not defined for coroutine<void>::pull_type template specialization) |

### member functions

### (constructor) constructs new coroutine

| | |
|---|---|
| `pull_type();` | (1) |
| `pull_type(Function&& fn);` | (2) |
| `pull_type(pull_type&& other);` | (3) |
| `pull_type(const pull_type& other)=delete;` | (4) |

**1)** creates a `std::coroutine<T>::pull_type` which does not represent a context of execution

**2)** creates a `std::coroutine<T>::pull_type` object and associates it with a execution context

**3)** move constructor, constructs a `std::coroutine<T>::pull_type` object to represent a context of execution that was represented by *other*, after this call *other* no longer represents a coroutine

**4)** copy constructor is deleted; coroutines are not copyable

**Notes**
Return values from the *coroutine-function* are accessible via `std::coroutine<T>::pull_type::get()` .
If the *coroutine-function* throws an exception, this exception is re-thrown when the caller returns from `std::coroutine<T>::pull_type::operator()()` .

**Parameters**

**other** another coroutine object with which to construct this coroutine object

**fn** function to execute in the new coroutine

**Exceptions**

**1), 3)** noexcept specification: `noexcept`

**2)** `std::system_error` if the coroutine could not be started - the exception may represent a implementation-specific error condition; re-throw user defined exceptions from *coroutine-function*

**Example**

```
std::coroutine<int>::pull_type source(
    [&](std::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
```

## (destructor) destructs a coroutine

| | |
|---|---|
| `~pull_type();` | (1) |

**1)** destroys a `std::coroutine<T>::pull_type`. If that `std::coroutine<T>::pull_type` is associated with a context of execution, then the context of execution is destroyed too. Specifically, its stack is unwound.

## operator= moves the coroutine object

| | |
|---|---|
| `pull_type & operator=(pull_type&& other);` | (1) |
| `pull_type & operator=(const pull_type& other)=delete;` | (2) |

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

**2)** copy assignment is deleted; coroutines are not copyable

**Parameters**

**other** another coroutine object to assign to this coroutine object

**Return value**

***this**

**Exceptions**

**1)** noexcept specification: `noexcept`

## operator bool indicates whether context of execution is still valid and a return value can be retrieved, or *coroutine-function* has finished

| | |
|---|---|
| `operator bool();` | (1) |

**1)** evaluates to true if coroutine is not complete (*coroutine-function* has not terminated)

**Exceptions**

**1)** noexcept specification: `noexcept`

### operator() jump context of execution

| | |
|---|---|
| `pull_type & operator()();` | (1) |

**1)** transfer control of execution to *coroutine-function*

**Notes**
It is important that the coroutine is still valid ( `operator bool()` returns `true` ) before calling this function, otherwise it results in undefined behaviour.

**Return value**

***this**

**Exceptions**

**1)** `std::system_error` if control of execution could not be transferred to other execution context - the exception may represent a implementation-specific error condition; re-throw user-defined exceptions from *coroutine-function*

### get accesses the current value from *coroutine-function*

| | | |
|---|---|---|
| `R get();` | (1) | (member of generic template) |
| `R& get();` | (2) | (member of generic template) |
| `void get()=delete;` | (3) | (only for coroutine<void>::pull_type template specialization) |

**1)** access values returned from *coroutine-function* (if move-assignable, the value is moved, otherwise copied)

**2)** access reference returned from *coroutine-function*

**Notes**
It is important that the coroutine is still valid ( `operator bool()` returns `true` ) before calling this function, otherwise it results in undefined behaviour.
If type T is move-assignable, it will be returned using move semantics. With such a type, if you call `get()` a second time before calling `operator()()` , `get()` will throw an exception – see below.

**Return value**

**R** return type is defined by coroutine's template argument

**void** coroutine does not support `get()`

**Exceptions**

**1)** Once a particular move-assignable value has already been retrieved by `get()` , any subsequent `get()` call throws `std::coroutine_error` with an error-code `std::coroutine_errc::no_data` until `operator()()` is called.

### swap swaps two coroutine objects

| | |
|---|---|
| `void swap(pull_type& other);` | (1) |

**1)** exchanges the underlying context of execution of two coroutine objects

**Exceptions**

**1)** noexcept specification: `noexcept`

## non-member functions

**std::swap** Specializes `std::swap` for `std::coroutine<T>::pull_type` and swaps the underlying context of lhs and rhs.

| | |
|---|---|
| `void swap(pull_type& lhs,pull_type& rhs);` | (1) |

**1)** exchanges the underlying context of execution of two coroutine objects by calling `lhs.swap(rhs)` .

**Exceptions**

**1)** noexcept specification: `noexcept`

**std::begin** Specializes `std::begin` for `std::coroutine<T>::pull_type` .

| | |
|---|---|
| `template<class R> coroutine<R>::pull_type::iterator begin(coroutine<R>::pull_type& c);` | (1) |

**1)** creates and returns a `std::input_iterator`

**std::end** Specializes `std::end` for `std::coroutine<T>::pull_type` .

| | |
|---|---|
| `template<class R> coroutine<R>::pull_type::iterator end(coroutine<R>::pull_type& c);` | (1) |

**1)** creates and returns a `std::input_iterator` indicating the termination of the *coroutine-function*

Incrementing the iterator switches the execution context.
When a main-context calls `iterator::operator++()` on an iterator obtained from an explicitly-instantiated `std::coroutine<T>::pull_type` , it must compare the incremented value with the iterator returned by `std::end()` . If they are unequal, the *coroutine-function* has passed a new data value, which can be accessed via `iterator::operator*()` . Otherwise the *coroutine-function* has terminated and the incremented iterator has become invalid.
When a `std::coroutine<T>::push_type` 's *coroutine-function* calls `iterator::operator++()` on an iterator obtained from the `std::coroutine<T>::pull_type` passed by the library, control is transferred back to the main-context. The main-context may never pass another data value. From the *coroutine-function*'s point of view, the `iterator::operator++()` call may never return. If it does return, the main-context has passed a new data value, which can be accessed via `iterator::operator*()` .
A function written to compare the incremented iterator with the iterator returned by `std::end()` can be used in either situation.
If the return-type is move-assignable the first call to `iterator::operator*()` moves the value. After that, any subsequent call to `iterator::operator*()` throws an exception ( `std::coroutine_error` ) until `iterator::operator++()` is called.
The iterator is forward-only.

**Example**

```
int j=10;
std::coroutine<int>::pull_type source(
    [&](std::coroutine<int>::push_type& sink){
        for(int i=0;i<j;++i){
            sink(i);
        }
    });

auto e(std::end(source));
for(auto i(std::begin(source));i!=e;++i){
    std::cout << *i << "␣";
}
```

## std::coroutine<>::push_type

Defined in header `<coroutine>`.

| |
|---|
| `template<class T> class coroutine<T>::push_type;` |
| `template<class T> class coroutine<T&>::push_type;` |
| `template<> class coroutine<void>::push_type;` |

The class `std::coroutine<T>::push_type` provides a mechanism to send a data value from one execution context to another.

## member types

| | | |
|---|---|---|
| `iterator` | std::output_iterator | (not defined for coroutine<void>::push_type template specialization) |

## member functions

### (constructor)   constructs new coroutine

| | |
|---|---|
| `push_type();` | (1) |
| `push_type(Function&& fn);` | (2) |
| `push_type(push_type&& other);` | (3) |
| `push_type(const push_type& other)=delete;` | (4) |

**1)** creates a `std::coroutine<T>::push_type` which does not represent a context of execution

**2)** creates a `std::coroutine<T>::push_type` object and associates it with a execution context

**3)** move constructor, constructs a `std::coroutine<T>::push_type` object to represent a context of execution that was represented by *other*, after this call *other* no longer represents a coroutine

**4)** copy constructor is deleted; coroutines are not copyable

**Parameters**

**other** another coroutine object with which to construct this coroutine object

**fn** function to execute in the new coroutine

**Exceptions**

**1), 3)** noexcept specification: `noexcept`

**2)** `std::system_error` if the coroutine could not be started - the exception may represent a implementation-specific error condition

**Notes**

If the *coroutine-function* throws an exception, this exception is re-thrown when the caller returns from `std::coroutine<T>::push_type::operator()(Arg&&)`.

**Example**

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

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

**(destructor)** destructs a coroutine

| | |
|---|---|
| `~push_type();` | (1) |

**1)** destroys a `std::coroutine<T>::push_type` . If that `std::coroutine<T>::push_type` is associated with a context of execution, then the context of execution is destroyed too. Specifically, its stack is unwound.

**operator=** moves the coroutine object

| | |
|---|---|
| `push_type & operator=(push_type&& other);` | (1) |
| `push_type & operator=(const push_type& other)=delete;` | (2) |

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

**2)** copy assignment operator is deleted; coroutines are not copyable

**Parameters**

**other** another coroutine object to assign to this coroutine object

**Return value**

***this**

**Exceptions**

**1)** noexcept specification: `noexcept`

**operator bool** indicates if context of execution is still valid, that is, *coroutine-function* has not finished

| | |
|---|---|
| `operator bool();` | (1) |

**1)** evaluates to true if coroutine is not complete (*coroutine-function* has not terminated)

**Exceptions**

**1)** noexcept specification: `noexcept`

**operator()** jump context of execution

| | | |
|---|---|---|
| `push_type & operator()(const Arg& arg);` | (1) | (member of generic template) |
| `push_type & operator()(Arg&& arg);` | (2) | (member of generic template) |
| `push_type & operator()(Arg& arg);` | (3) | (member only of coroutine<Arg&>::push_type template specialization) |
| `push_type & operator()();` | (4) | (member only of coroutine<void>::push_type template specialization) |

**1),2)** If *Arg* is move-assignable, it will be passed using move semantics. Otherwise it will be copied.

Switches the context of execution, transferring *arg* to *coroutine-function*.

**Note**
It is important that the coroutine is still valid ( `operator bool()` returns `true` ) before calling this function, otherwise it results in undefined behaviour.

**Parameters**

**arg** argument to pass to the *coroutine-function*

**Return value**

***this**

**Exceptions**

**1)** `std::system_error` if control of execution could not be transferred to other execution context - the exception may represent a implementation-specific error condition; re-throw user-defined exceptions from *coroutine-function*

### swap  swaps two coroutine objects

```
void swap(push_type& other);    (1)
```

**1)** exchanges the underlying context of execution of two coroutine objects

**Exceptions**

**1)** noexcept specification: `noexcept`

## non-member functions

### std::swap  Specializes `std::swap` for `std::coroutine<T>::push_type` and swaps the underlying context of lhs and rhs.

```
void swap(push_type& lhs,push_type& rhs);    (1)
```

**1)** exchanges the underlying context of execution of two coroutine objects by calling `lhs.swap(rhs)` .

**Exceptions**

**1)** noexcept specification: `noexcept`

### std::begin  Specializes `std::begin` for `std::coroutine<T>::push_type` .

```
template<class R> coroutine<R>::push_type::iterator begin(coroutine<R>::push_type& c);    (1)
```

**1)** creates and returns a `std::output_iterator`

### std::end  Specializes `std::end` for `std::coroutine<T>::push_type` .

```
template<class R> coroutine<R>::push_type::iterator end(coroutine<R>::push_type& c);    (1)
```

**1)** creates and returns a `std::output_iterator` indicating the termination of the coroutine

Calling `iterator::operator*(Arg&&)` switches the execution context and transfers the given data value.
`iterator::operator*(Arg&&)` returns if other context has transferred control of execution back.
The iterator is forward-only.

**Example**

```
std::coroutine<int>::push_type sink(
    [&](std::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(std::begin(v),std::end(v),std::begin(sink));
```

## std::coroutine_errc

Defined in header `<coroutine>`.

```
enum class coroutine_errc { no_data };
```

Enumeration `std::coroutine_errc` defines the error codes reported by `std::coroutine<T>::pull_type` in `std::coroutine_error` exception object.

### member constants
Determines error code.

| | |
|---|---|
| `no_data` | `std::coroutine<T>::pull_type` has no valid data (maybe moved by prior access) |

## std::coroutine_error

Defined in header `<coroutine>`.

```
class coroutine_error;
```

The class `std::coroutine_error` defines an exception class that is derived from `std::logic_error`.

### member functions

### (constructor)
constructs new coroutine error object.

| | |
|---|---|
| `coroutine_error( std::error_code ec);` | (1) |

**1)** creates a `std::coroutine_error` error object from an error-code.

**Parameters**

**ec** error-code

### code
Returns the error-code.

| | |
|---|---|
| `const std::error_code& code() const;` | (1) |

**1)** returns the stored error code.

**Return value**

**std::error_code** stored error code

**Exceptions**

**1)** noexcept specification: `noexcept`

### what
Returns a error-description.

| | |
|---|---|
| `virtual const char* what() const;` | (1) |

**1)** returns a description of the error.

**Return value**

**char*** null-terminated string with error description

**Exceptions**

**1)** noexcept specification: `noexcept`

## References

[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.). Addison-Wesley. Section 1.4.2: Coroutines

[3] Moura, Ana Lúcia De and Ierusalimschy, Roberto. "Revisiting coroutines". ACM Trans. Program. Lang. Syst., Volume 31 Issue 2, February 2009, Article No. 6

[4] N3328: Resumable Functions

[5] boost.asio

[6] boost.coroutine (proposed interface will be available in boost-1.55)

[7] boost.context

[8] await_emu by Evgeny Panasyuk

[9] boost.coroutine by Giovanni P. Deretta, 'Google Summer of Code 2006'

[10] Mordor high performance I/O library

[11] AT&T Task Library

[12] Split Stacks in GCC

[13] channel9 - 'The Future of C++'

[14] Wikipedia - 'Coroutine'

[15] boost::asio::yield_context