await/yield: C++ coroutines

...

Zbigniew Skowron
30 November 2016

Agenda

Current status

Overview and motivation

Stackful vs. stackless

Coroutines as generators

Coroutines instead of callbacks

Awaitable types vs. coroutine return types

Gotchas

Many proposals

I will talk about Microsoft's proposal:

- stackless coroutines
- similar to C# await

There are other proposals too:

- stackful coroutines (fibers)
- resumable expressions

Current status

Microsoft's proposal:

- the proposal changed a lot during the last 2 years - it's not obvious it will end up in standard as it is
- Clang implementation being worked on by Gor Nishanov - the same guy who wrote Microsoft's implementation
- production ready as of Visual Studio 2015 Update 2

C++ Standard:

- Microsoft's proposal will probably end up as a Coroutine TS next year.
- But not fixed yet. Maybe we'll end up with something entirely different...

Introduction

What it’s all about?

Coroutines are functions that can be:

- suspended
- continued at a later time

How does it look like?

```
generator<int> tenInts()
{
    for (int i = 0; i < 10; ++i)
    {
        cout << "Next: " << i;
        co_yield i;
    }
}
```

Suspension point.

```
// Downloads url to cache and
// returns cache file path.
future<path> cacheUrl(string url)
{
    cout << "Downloading url.";
    string text = co_await downloadAsync(url);

    cout << "Saving in cache.";
    path p = randomFileName();

    co_await saveInCacheAsync(p, text);

    co_return p;
}
```

Suspension point.

Suspension point.

What can we use coroutines for?

- implementing generators
- implementing asynchronous functions, while avoiding callbacks

Difference between generator and async coroutine

What is the fundamental difference between a coroutine that is a generator and coroutine that is an asynchronous function?

Generators are resumed by the user on demand.

Asynchronous coroutines are resumed in background, by worker threads.

Stackless vs. stackful

Stackless vs. stackful

Stackless	Stackful
only local variables are available (one stack frame preserved)	all stack frames above are also preserved
only top-level coroutine function can be suspended	can suspend from helper functions
no special stack handling is needed	stack must be allocated on the side

pseudocode

Stackless vs. stackful example

Stackless	Stackful

	`void helper()`
	`{`
	`    suspend`
	`}`

`void stackless(const int& a)`
`{`
`    ...`
`    suspend`
`    ...`
`    a = 5;`
`}`
	`void stackful(const int& a)`
	`{`
	`    ...`
	`    helper()`
	`    ...`
	`    a = 5;`
	`}`

Very likely a bug.

Stackless workaround

pseudocode

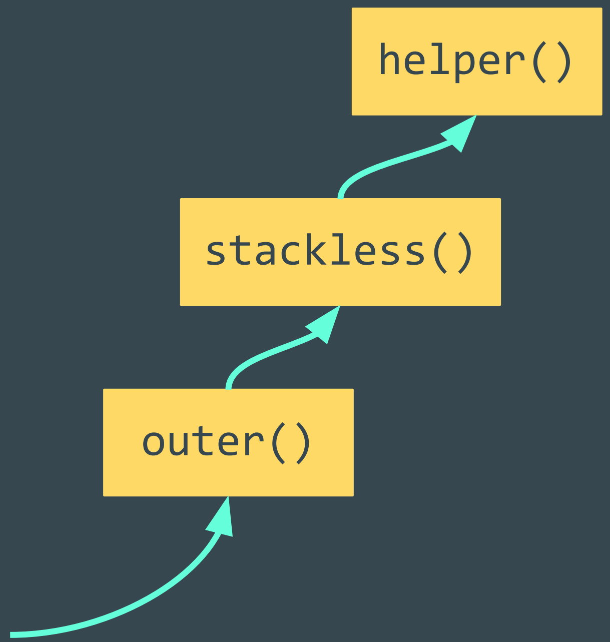
Stackless	Stackful

<pre>void helper()
{
    suspend
}
 	
void stackless(const int& a)
{
    ...
    await helper()
    ...
    a = 5;
}</pre>
	<pre>void helper()
	{
	    suspend
	}
	
	void stackful(const int& a)
	{
	    ...
	    helper()
	    ...
	    a = 5;
	}</pre>

Very likely a bug.

Stackless vs. stackful: how "stack" looks

Stackless: each "stack frame" is dynamically allocated



Stackful: one dynamically allocated stack

Motivation

TCP reader - synchronous

We want to create an asynchronous version of this simple TCP reader.

```
int tcp_reader(int total)
{
    char buf[4 * 1024];
    auto conn = Tcp::Connect("127.0.0.1", 1337);
    while (true)
    {
        auto bytesRead = conn.Read(buf, sizeof(buf));
        total -= bytesRead;
        if (total <= 0 || bytesRead == 0) return total;
    }
}
```

Example from Gor Nishanov's CppCon 2015 talk: "C++ Coroutines"

TCP reader - futures

That's how it looks using futures with .then():

```
future<int> tcp_reader(int64_t total) {
    struct State {
        char buf[4 * 1024];
        int64_t total;
        Tcp::Connection conn;
        explicit State(int64_t total) : total(total) {}
    };
    auto state = make_shared<State>(total);
    return Tcp::Connect("127.0.0.1", 1337).then(
        [state](future<Tcp::Connection> conn) {
        state->conn = std::move(conn.get());
        return do_while([state]()->future<bool> {
            if (state->total <= 0) return make_ready_future(false);
            return state->conn.read(state->buf, sizeof(state->buf)).then(
                [state](future<int> nBytesFut) {
                auto nBytes = nBytesFut.get()
                    if (nBytes == 0) return make_ready_future(false);
                state->total -= nBytes;
                return make_ready_future(true);
            }); // read
        }); // do_while
    }).then([state](future<void>) {return make_ready_future(state->total)});
}
```

Incomprehensible stuff.

Example from Gor Nishanov's CppCon 2015 talk: "C++ Coroutines"

TCP reader - synchronous again

```
int tcp_reader(int total)
{
    char buf[4 * 1024];
    auto conn = Tcp::Connect("127.0.0.1", 1337);
    while (true)
    {
        auto bytesRead = conn.Read(buf, sizeof(buf));
        total -= bytesRead;
        if (total <= 0 || bytesRead == 0) return total;
    }
}
```

Example from Gor Nishanov's CppCon 2015 talk: "C++ Coroutines"

TCP reader - async using coroutines

```
future<int> tcp_reader(int total)
{
    char buf[4 * 1024];
    auto conn = co_await Tcp::Connect("127.0.0.1", 1337);
    while (true)
    {
        auto bytesRead = co_await conn.Read(buf, sizeof(buf));
        total -= bytesRead;
        if (total <= 0 || bytesRead == 0) co_return total;
    }
}
```

Example from Gor Nishanov's CppCon 2015 talk: "C++ Coroutines"

Motivation:
Writing games in C++ easily, without scripting languages

I want to be able to write this game AI code

```
future<void> think(Guy& guy)
{
    guy.talk("I'm thinking.");
    co_await wait_some_time(1s);
    guy.talk.clear();

    co_await walkTo(guy, guy.position +random(-100.0f, 100.0f));

    guy.talk( thingsToSay[random(thingsToSay.size())] );
    co_await wait_some_time(1s);
    guy.talk.clear();
}
```

Demo: SfmlGame

Coroutines Overview

Generators

Generators are functions that generate a sequence of values.

```
generator<int> tenInts()
{
    for (int i = 0; i < 10; ++i)
    {
        cout << "Next: " << i;
        co_yield i;
    }
}
```

We can iterate over the result as if it was a collection.

```
int main()
{
    for (auto i : tenInts())
        std::cout << i;
}
```

Each value is computed on demand.

Generators - execution

```cpp
int main()
{
    for (auto i : tenInts())
    {
        std::cout << i;
        ...
        ...
        ...
        ...
        ...
        ...
        ...
        ...
    }
}
```

```cpp
generator<int> tenInts()
{
    cout << "Start";

    for (int i = 0; i < 10; ++i)
    {
        cout << "Next: " << i;
        co_yield i;
    }

    cout << "End";
}
```

Generators - execution

First iteration.

```
int main()
{
    for (auto i : tenInts())
    {
        std::cout << i;
        ...
        ...
        ...
        ...
        ...
        ...
        ...
        ...
    }
}
```

Execute loop with i == 0.

```
generator<int> tenInts()
{
    cout << "Start";

    for (int i = 0; i < 10; ++i)
    {
        cout << "Next: " << i;
        co_yield i;
    }

    cout << "End";
}
```

Suspend and return 0.

Generators - execution

Second iteration.

```
int main()
{
    for (auto i : tenInts())
    {
        std::cout << i;
        ...
        ...
        ...
        ...
        ...
        ...
        ...
        ...
    }
}
```

Resume.

```
generator<int> tenInts()
{
    cout << "Start";

    for (int i = 0; i < 10; ++i)
    {
        cout << "Next: " << i;
        co_yield i;
    }

    cout << "End";
}
```

Suspend and return 1.

Execute loop with i == 1.

Generators - execution

Last iteration.

```
int main()
{
    for (auto i : tenInts())
    {
        std::cout << i;
        ...
        ...
        ...
        ...
        ...
        ...
        ...
        ...
    }
}
```

End loop.

Resume.

```
generator<int> tenInts()
{
    cout << "Start";

    for (int i = 0; i < 10; ++i)
    {
        cout << "Next: " << i;
        co_yield i;
    }

    cout << "End";
}
```

End loop.

TenInts Demo

Avoiding callbacks

```
void writeAndVerifySerial(uint8_t* buf)
{
    auto file = new File("file.txt");
    file.write(buf);
    file.verify();
    std::cout << "File ok!";
}

future<void> writeAndVerifyCoro(uint8_t* buf)
{
    auto file = new File("file.txt");
    co_await file.asyncWrite(buf);
    co_await file.asyncVerify();
    std::cout << "File ok!";
}
```

Asynchronous code written as one function.

```
void writeAndVerifyAsync(uint8_t* buf)
{
    auto file = new File("file.txt");
    file.cbWrite(buf, writeDone, file);
}

int writeDone(void* userData)
{
    auto file = (File*)userData;
    file.cbVerify(verifyDone, file);
}

void verifyDone(void* userData)
{
    std::cout << "File ok!";
}
```

Conceptually one function executed in three parts.

Callbacks - execution

MAIN

```
int main()
{
    auto f = writeAndVerify(buf);
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    f.wait();
}
```

```
future<void> writeAndVerify(uint8_t* buf)
{
    auto file = new File("file.txt");

    co_await file.asyncWrite(buf);



    std::cout << "Written!";
}
```

File's IO thread

```
future<void> File::asyncWrite(uint8_t* buf)
{
    ...
}
```

Callbacks - execution

MAIN

Call function.

```
int main()
{
    auto f = writeAndVerify(buf);
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    f.wait();
}
```

Wait for result.

Wait done.

Schedule IO work, then suspend and return future.

```
future
{
    auto

    co_await file.asyncWrite(buf);



    std::cout << "Written!";
}
```

Done.

File's IO thread

Start IO work.

```
future<void> File::asyncWrite(uint8_t* buf)
{
    ...
}
```

Resume coroutine.

Callbacks - execution

MAIN

```
int main()
{
    auto f = writeAndVerify(buf);
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    f.wait();
}
```

```
future<void> writeAndVerify(uint8_t* buf)
{
    auto file = new File("file.txt");

    co_await file.asyncWrite(buf);


    std::cout << "Written!";

    co_await file.asyncVerify();


    std::cout << "Verified!";
}
```

File’s IO thread

```
future<void> File::asyncWrite(uint8_t* buf)
{
    ...
}
```

```
future<void> File::asyncVerify()
{
    ...
}
```

Callbacks - execution

MAIN

Call function.

```
int main()
{
    auto f = writeAndVerify(buf);
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    f.wait();
}
```

Schedule IO work, then suspend and return future.

```
future
{
    au
    co_await file.asyncWrite(buf);
    ritten!";
    co_await file.asyncVerify();
    std::cout << "Verified!";
}
```

Schedule IO work, then suspend and return future.

Wait for result.

Wait done.

Done.

File's IO thread

Start IO work.

```
future<void> File::asyncWrite(uint8_t* buf)
{
    ...
}
```

Resume coroutine.

Start IO work.

```
future<void> File::asyncVerify()
{
    ...
}
```

Resume coroutine.

Callbacks - execution

MAIN

```
int main()
{
    auto f = writeAndVerify(buf);
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    f.wait();
}
```

```
future<void> writeAndVerify(uint8_t* buf)
{
    auto file = new File("file.txt");

    co_await file.asyncWrite(buf);


    std::cout << "Written!";

    co_await file.asyncVerify();


    std::cout << "Verified!";
}
```

File's IO thread

```
future<void> File::asyncWrite(uint8_t* buf)
{
    ...
}

future<void> File::asyncVerify()
{
    ...
}
```

This code executes in parallel.

Callbacks - execution

MAIN

```
int main()
{
    auto f = writeAndVerify(buf);
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    f.wait();
}
```

```
future<void> writeAndVerify(uint8_t* buf)
{
    auto file = new File("file.txt");

    co_await file.asyncWrite(buf);

    std::cout << "Written!";

    co_await file.asyncVerify();


    std::cout << "Verified!";
}
```

File’s IO thread

```
future<void> File::asyncWrite(uint8_t* buf)
{
    ...
}
```

```
future<void> File::asyncVerify()
{
    ...
}
```

Coroutine can start on one thread, and continue on another.

Callbacks - execution

MAIN

```
int main()
{
    auto f = writeAndVerify(buf);
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    f.wait();
}
```

```
future<void> writeAndVerify(uint8_t* buf)
{
    auto file = new File("file.txt");

    co_await file.asyncWrite(buf);

    std::cout << "Written!";

    co_await file.asyncVerify();

    std::cout << "Verified!";
}
```

File’s IO thread

```
future<void> File::asyncWrite(uint8_t* buf)
{
    ...
}
```

File’s Verify thread

```
future<void> File::asyncVerify()
{
    ...
}
```

Coroutine can start on one thread,
and continue on another, and another…

AsyncWrite Demo

Awaitable types
&
Coroutine return types

Awaitable types and coro return types are not the same

Often coroutines return types that we can await on:

future<int> can be both awaited on and returned from a coroutine.

generator<int> can be returned from a coroutine, but cannot be awaited on!

So we can see, that types can be:

- awaitable
- returnable from a coroutine
- both
- or none...

What makes a type awaitable?

MyType, to be awaitable, must provide three functions:

```
bool await_ready(MyType& val);

void await_suspend(MyType& val, coroutine_handle<> ch);

T await_resume(MyType& val);
```

Should be named:
is_ready()
schedule_continuation()
retrieve_value()

Those methods define how to await for the value of MyType:

- whether the value is immediately ready, and suspension is not needed
- how and when coroutine that is waiting for will be resumed
- how to retrieve value from MyType.

We can define those functions in three ways:

- by providing free functions like in example above
- by providing all needed methods in MyType itself
- by providing operator co_await that will return proxy type with required methods.

Making std::future<T> awaitable

Those methods define how to await for the value of std::future<T>:

- whether the value is immediately ready, and suspension is not needed

```
bool await_ready(future<T>& f)
{
    return f.is_ready();
}
```

- how and when coroutine that is waiting for will be resumed

```
void await_suspend(future<T>& f, std::experimental::coroutine_handle<> coro)
{
    f.then([coro](const future<T>& f) { coro.resume(); });
}
```

- how to retrieve value from std::future<T>

```
T await_resume(const future<T>& f)
{
    return f.get();
}
```

suspend_never and suspend_always

suspend_never and suspend_always are simple helper awaitable types, used when nothing special is required in some customisation point.

Awaiting suspend_never will never cause a coroutine to suspend.

Awaiting suspend_always will always cause a coroutine to suspend.

No-op.

```
co_await suspend_never;

co_await suspend_always;
```

Will suspend coroutine. Someone will have to resume it later! (or destroy)

Demo: <resumable> suspend_never suspend_always

What makes a type returnable from coroutine?

To be able to return MyType from a coroutine we must define coroutine promise for it.

Coroutine promise specifies how the coroutine will behave, in particular it specifies the meaning of those things for your type:

- co_return
- co_yield

We can define coroutine promise for MyType in two ways:

- by providing type MyType::promise_type
- by specializing struct coroutine_traits<MyType> with promise_type inside

Promise for MyType explained

```
struct promise_type
{
    MyType get_return_object()
    {
        return MyType(coroutine_handle<promise_type>::from_promise(*this));
    }

    auto initial_suspend()
    {
        return std::experimental::suspend_never();
    }

    auto final_suspend()
    {
        return std::experimental::suspend_always();
    }

    void return_value(int v);

    void return_void();

    void set_exception(std::exception_ptr exc);

    void yield_value(int i);

    WrapperAwaitable await_transform(Awaitable a)
    {
        return WrapperAwaitable(a);
    }
};
```

Promise creates return value for the coroutine: MyType.

Promise defines whether coroutine will suspend before executing it's body

...or suspend after executing it's body

What co_return does.

How to propagate exceptions (optional).

What co_yield does.

Extension point for co_await.

Controlling coroutines - coroutine_handle

Promise type can get access to the coroutine_handle, which provides a way of controlling the execution and lifetime of the coroutine.

```
template <typename Promise>
struct coroutine_handle
{
    void resume() const;

    void destroy();

    bool done() const;

    Promise& promise();

    static coroutine_handle from_promise(Promise& promise);
};
```

Resume, destroy, check if completed.

Get promise from coroutine_handle or coroutine_handle from promise.

Promise type can pass this handle to the return object, if return object needs to control the coroutine:

```
generator<int> get_return_object()
{
    return generator<int>(coroutine_handle<promise_type>::from_promise(*this));
}
```

Making std::future<T> returnable from coroutine

```
template<typename T>
struct coroutine_traits<future<T>>
{
    struct promise_type
    {
        promise<T> promise;

        future<T> get_return_object()
        {
            return promise.get_future();
        }

        auto initial_suspend()
        {
            return std::experimental::suspend_never();
        }

        auto final_suspend()
        {
            return std::experimental::suspend_never();
        }

        void return_value(T v)
        {
            promise.set_value(v);
        }

        void set_exception(std::exception_ptr exc)
        {
            promise.set_exception(exc);
        }
    };
};
```

Coroutine will store a promise.

And will return a future of this promise.

We don’t need to suspend before or after the function body.

Return value by setting it in the promise.

Propagate exception by setting it in the promise.

Demo: <future>

Anatomy of a coroutine

Coroutine is a normal function

Whether a function is a coroutine or not is not visible on the outside:

```
future<path> cacheUrl(string url);
```

Coroutine or not?

The only difference is the special syntax inside, that instructs compiler to split the function into three parts:

- initial part
- resumable part
- cleanup

pseudocode

Coroutine split

```
generator<int> tenInts()
{
    cout << "Start";

    for (int i = 0; i < 10; ++i)
    {
        cout << "Next: " << i;
        co_yield i;
    }

    cout << "End";
}

generator<int> tenInts()
{
    auto context = new TenIntsContext();
    auto& promise = context->get_promise();

    _return = promise.get_return_object();

    tenIntsStateMachine();

    return _return;
}
```

```
void tenIntsStateMachine()
{
    if (context.resumePoint == 2)
        goto label2;

    cout << "Start";

    for (int i = 0; i < 10; ++i)
    {
        cout << "Next: " << i;
        promise.yield_value(i);
        context.resumePoint = 2;
        return;
    label2:;
    }

    cout << "End";
}


void tenIntsCleanup()
{
    delete context;
}
```

Coroutine parts explained

Initial part is a normal C++ function, that does the following:

- creates a coroutine frame
- creates the return object
- runs the coroutine up to the first suspension point.

Resumable part is a compiler generated state machine, with all the suspension / resume points.

Cleanup part destroys the coroutine frame. It is called:

- when coroutine is explicitly destroyed using destroy() method,
- when coroutine is done executing (and not stopped on final suspend),

whichever happens first.

ThreeParts Step Into Demo

What is a coroutine

Coroutine is a function with one or more of the following special keywords:

- co_await — Suspends coroutine
- co_return — Returns result from coroutine
- co_yield — Returns one value and suspends

Coroutine anatomy - step by step

```
future<int> compute()
{
    ...
}
```

↓

```
future<int> compute()
{
    auto context = new ComputeContext();
    auto& promise = context->get_promise();

    __return = promise.get_return_object();

    co_await promise.initial_suspend();

    ...

final_suspend:

    co_await promise.final_suspend();
}
```

Coroutine context, containing a promise

At the very beginning promise provides return object of the coroutine.

Customisation points

Coroutine anatomy - co_return

```
future<int> compute()
{
    co_return 5;
}
```

```
future<int> compute()
{
    ...
    __return = promise.get_return_object();

    co_await promise.initial_suspend();

    promise.return_value(5);
    goto final_suspend;

final_suspend:

    co_await promise.final_suspend();
}
```

Promise is a connection between coroutine and returned object.

Promise defines how to return a value.

Coroutine anatomy - co_await

```
future<int> compute()
{
    int v = co_await think();
    ...
}
```

```
future<int> compute()
{
    ...
    auto&& a = think();
    if (!await_ready(a))
    {
        await_suspend(a, coroutine_handle);

        suspend
    }
    int v = await_resume(a);

    ...
}
```

Call the function and get awaitable object.

If the result is not ready…

…suspend and wait for it.

Retrieve the value.

Coroutine anatomy - co_await

```
future<int> compute()
{
    int v = co_await think();
    ...
}
```

```
future<int> compute()
{
    ...

    auto&& a = think();

    if (!await_ready(a))
    {

        await_suspend(a, coroutine_handle);

        suspend                  resume
    }

    int v = await_resume(a);

    ...
}
```

Coroutine anatomy - co_yield

```
generator<int> compute()
{
    co_yield 5;
}
```

```
generator<int> compute()
{
    ...
    __return = promise.get_return_object();

    co_await promise.yield_value(5);

    ...
}
```

Promise is a connection between coroutine and returned object.

Promise defines how to yield a value.

Coroutine anatomy - await_transform

```
future<int> compute()
{
    int v = co_await think();
    ...
}
```

```
future<int> compute()
{
    ...

    int v = co_await promise.await_transform(think());

    ...
}
```

Promise can redefine what it means to await for an object.

One use could be cancellation support.

Writing your own generator class

What we want to achieve

```
template<typename T>
struct my_generator
{
    const T* getNext();
};

my_generator<int> years()
{
    for (int i = 2010; i <= 2015; ++i)
        co_yield i;
}

int main()
{
    auto gen = years();
    while (auto i = gen.getNext())
        std::cout << *i << std::endl;
}
```

Simple interface.
Returns nullptr when done.

Returnable from coroutines.

What the generator needs:

- coroutine_handle to resume coroutine and get next value
- coroutine promise that will pass values from coroutine to my_generator

Coroutine promise

```
struct promise_type
{
    my_generator get_return_object()
    {
        return my_generator(coroutine_handle<promise_type>::from_promise(*this));
    }

    const T* value;

    void yield_value(const T& v)
    {
        value = &v;
    }

    auto initial_suspend()
    {
        return std::experimental::suspend_always();
    }

    auto final_suspend()
    {
        return std::experimental::suspend_always();
    }
};
```

Create generator, and pass a coroutine_handle to it.

Store pointer to the generated value.

Don't generate first value until asked to.

Don't destroy coroutine frame immediately.
We want to check if coroutine was done or not.

my_generator

```
template<typename T>
struct my_generator
{
    struct promise_type { ... };

    coroutine_handle<promise_type> m_coroutine = nullptr;

    explicit my_generator(coroutine_handle<promise_type> coroutine)
        : m_coroutine(coroutine)
    {}

    ~my_generator()
    {
        if (m_coroutine)
            m_coroutine.destroy();
    }

    const T* getNext()
    {
        m_coroutine.resume();
        if (m_coroutine.done())
        {
            return nullptr;
        }

        return m_coroutine.promise().value;
    }
};
```

Generator needs to control the coroutine.

Destroy the coroutine when we die.

Resume.

If we're done, return null.

Otherwise return generated value,

Not so obvious details

Why generators are not copyable?

We might want to copy a generator to create a copy of the sequence.

If the promise type used and all local variables of the coroutine are copyable, it should be possible, right?

```
generator<int> produce()
{
    vector<int> vec{ 1, 2, 3 };
    for (auto i : vec)
        co_yield i;
}
```

Copied iterators would point to the original vector!

When copying generator we would copy:

- vector vec
- variable i
- interators inside for loop.

Why std::generator<T> is not recursive?

```
generator<int> years()
{
    for (int i = 2010; i <= 2015; ++i)
        co_yield i;
}

generator<int> more_years()
{
    co_yield 2009;

    co_yield years();

    co_yield 2016;
}
```

Support for yielding generators would make generator<T> larger and slower.

But if you need it it’s easy to write your own type that supports that.

Why can't I await in generator?

```
generator<int> years()
{
    int start = co_await get_year();
    for (int i = start; i <= 2025; ++i)
        co_yield i;
}
```

This is explicitly blocked, because it doesn't make sense.
What would return generator when suspended on await?

```
generator<int> years()
{
     int start = get_year().get();
     for (int i = start; i <= 2025; ++i)
         co_yield i;
}
```

Waiting makes sense.

```
template <typename _Uty>
_Uty && await_transform(_Uty &&_Whatever)
{
    static_assert(_Always_false<_Uty>::value,
      "co_await is not supported in coroutines of type std::experiemental::generator");
    return _STD forward<_Uty>(_Whatever);
}
```

async_generator and for co_await

It turns out, that doing awaiting in generators actually make sense.
We get something called async generators this way.

```
async_generator<int> years()
{
    int start = co_await get_year();
    for (int i = start; i <= 2025; ++i)
        co_yield i;
}
```

Available in RxCpp

for co_await is like range for, but awaits for begin() and ++it.
We can use it to process async_generators.

```
async_generator<int> squares(async_generator<int> gen)
{
    for co_await (auto v : gen)
        co_yield v * v;
}
```

Changing threads mid-function

Coroutine machinery can be used for many things, for example to delegate work to proper thread.

```
future<void> doUIWork()
{
    button.Text = "Working.";

    co_await resume_on_thread_pool {};

    doHeavyComputations();

    co_await resume_on_ui_thread {};

    button.Text = "Done!";
}
```

Executed on UI thread.

Executed on background thread.

Executed on UI thread.

CppCon 2016: Kenny Kerr & James McNellis “Putting Coroutines to Work with the Windows Runtime"

await_transform and cancellation

await_transform can be used for extra customisation, for example for supporting coroutines that can be cancelled.

```
struct promise_type
{
    //...

    bool is_cancelled;

    //...

    template<typename T>
    T& await_transform(T& awaitable)
    {
        if (is_cancelled)
            throw CancelledException();

        return awaitable;
    }
};
```

Every time we await on something, we first check our cancelled flag.

Extra promise customisation

For each coroutine return type we must define a promise type.

But in fact we can define different promise types for different signatures of coroutines.

Main template: parametrized with signature.

```
template <typename Ret, typename... Args>
struct coroutine_traits
{
    using promise_type = ...
};
```

Specializations for different signatures.

```
template<>
struct coroutine_traits<MyType>
{
    using promise_type = Promise0;
};

template<>
struct coroutine_traits<MyType, int, string>
{
    using promise_type = PromiseIS;
};
```

```
MyType coro();
```

Uses Promise0

```
MyType coro(int, string);
```

Uses PromiseIS

operator co_await

Allows to write:
co_await 3s;

```
auto operator co_await(std::chrono::system_clock::duration duration)
{
    class awaiter
    {
        std::chrono::system_clock::duration duration;

    public:

        explicit awaiter(std::chrono::system_clock::duration d) : duration(d) {}

        bool await_ready() const
        {
            return false;
        }

        void await_suspend(std::experimental::coroutine_handle<> coro)
        {
            SetTimer(d, [coro] { coro.resume(); });
        }

        void await_resume() {}
    };

    return awaiter{ duration };
}
```

Problem with awaiting for a future

future.then() creates a thread, so awaiting for a future<T> will create a thread!

This is super weak…

Solution: use your own types instead of std::future…

And BTW, future.then() blocks!
At least in Boost implementation. It's not clear whether it must block or not.
So the code shown for awaiting futures won't actually work!

The end, finally!

Resources

Presentations about C++ coroutines:

CppCon 2014: Gor Nishanov "await 2.0: Stackless Resumable Functions" (intro)
CppCon 2015: Gor Nishanov “C++ Coroutines - a negative overhead abstraction" (intro)
CppCon 2016: Gor Nishanov “C++ Coroutines: Under the covers" (optimization)
2016 LLVM Developers’ Meeting: Gor Nishanov “LLVM Coroutines” (optimization)

Meeting C++ 2015: James McNellis “An Introduction to C++ Coroutines" (intro)
CppCon 2016: James McNellis “Introduction to C++ Coroutines" (intro)

CppCon 2016: Kenny Kerr & James McNellis “Putting Coroutines to Work with the Windows Runtime" (usage)

Kirk Shoop “Reactive programming in C++" (async_generators)

ISO papers:

N4402: Resumable Functions (revision 4)
P0054R00: Coroutines: Reports from the field