Exploring the C++ Coroutine

Approach, Compiler, and Issues

Park Dong Ha

C++ Korea Facebook Group

Researcher at Alchera Inc.

github.com/luncliff

Reference List: **Proposal**

- N4736: C++ Extension for Coroutines (Working Draft)
  - N4723
- N4402: Resumable Functions (Rev 4)
  - N4134
  - N3977
  - N3858

All documents after N4800 are skipped because
the presenter was lack of time for review ☹ (Sorry !)

Reference List: **Video (A lot!)**

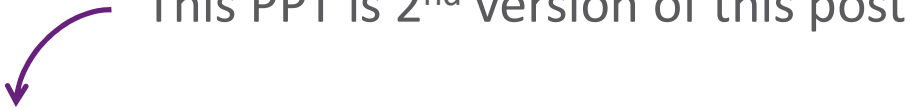
- CppCon 2018 : Gor Nishanov “Nano-coroutines to the Rescue!”
- CppCon 2017 : Toby Allsopp “Coroutines: what can’t they do?”
- CppCon 2017 : Gor Nishanov “Naked coroutines live(with networking)”
- CppCon 2016 : Gor Nishanov “C++ Coroutines: Under the covers”
- CppCon 2016 : James McNellis “Introduction to C++ Coroutines”
- CppCon 2016 : Kenny Kerr & James McNellis “Putting Coroutines to Work with the Windows Runtime”
- CppCon 2016 : John Bandela “Channels - An alternative to callbacks and futures”
- CppCon 2015 : Gor Nishanov “C++ Coroutines - a negative overhead abstraction”
- Meeting C++ 2015 : James McNellis “An Introduction to C++ Coroutines”
- Meeting C++ 2015 : Grigory Demchenko “Asynchrony and Coroutines”
- CppCon 2014 : Gor Nishanov “await 2.0: Stackless Resumable Functions”

Reference List: **Code**

- https://github.com/lewissbaker/cppcoro
- https://github.com/kirkshoop/await
- https://github.com/toby-allsopp/coroutine_monad
- https://github.com/jbandela/stackless_coroutine
- https://github.com/luncliff/coroutine

Reference List: **The others**

- https://github.com/GorNishanov/await
- http://cpp.mimuw.edu.pl/files/await-yield-c++-coroutines.pdf
- Coroutines in Visual Studio 2015 – Update 1
- https://llvm.org/docs/Coroutines.html
- https://luncliff.github.io/posts/Exploring-MSVC-Coroutine.html
- (no more space ☹)



Topics

Background knowledge for coroutine beginners

Understanding components of the C++ Coroutine

- Operators & Awaitable Type
- Promise
- Coroutine Handle

Difference between MSVC & Clang

Some short examples (If we have time…)

Mostly covered in CppCon

(Maybe) First in this talk…

Let’s start with some concepts …

Forward declaration for this session

Function: Sequence of statements

Function Code

```
int mul(int a, int b);

int mul(int a, int b) {
    return a * b;
}
```

Routine

```
int mul(int,int) PROC
        mov     DWORD PTR [rsp+16], edx
        mov     DWORD PTR [rsp+8], ecx
        mov     eax, DWORD PTR a$[rsp]
        imul    eax, DWORD PTR b$[rsp]
        ret     0
int mul(int,int) ENDP
```

https://godbolt.org/

Routine == Instruction[]

```
int mul(int,int) PROC
        mov     DWORD PTR [rsp+16], edx
        mov     DWORD PTR [rsp+8], ecx
        mov     eax, DWORD PTR a$[rsp]
        imul    eax, DWORD PTR b$[rsp]
        ret     0
int mul(int,int) ENDP
```

Routine:

- An ordered group of instructions

Instruction:

- Abstraction of machine behavior.
- Transition between machine states

Invocation

Jump(goto) to the start of the routine

Activation

Jump into a point of the routine

Suspension

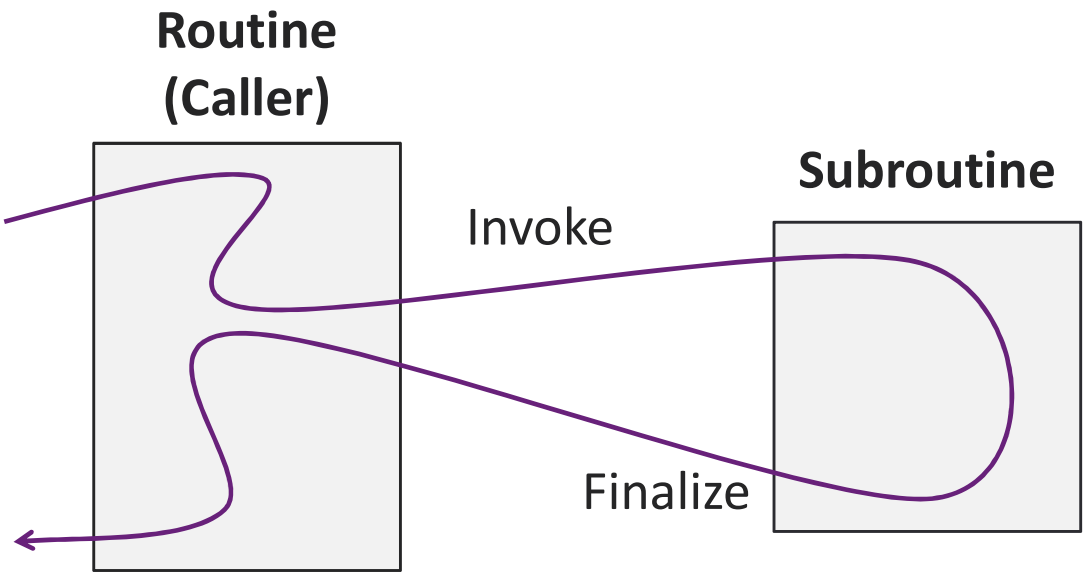
Jump to another routine’s point without finalization

Finalization

Destruction(Clean up) of routine state (reaching the end of epilogue)

Subroutine

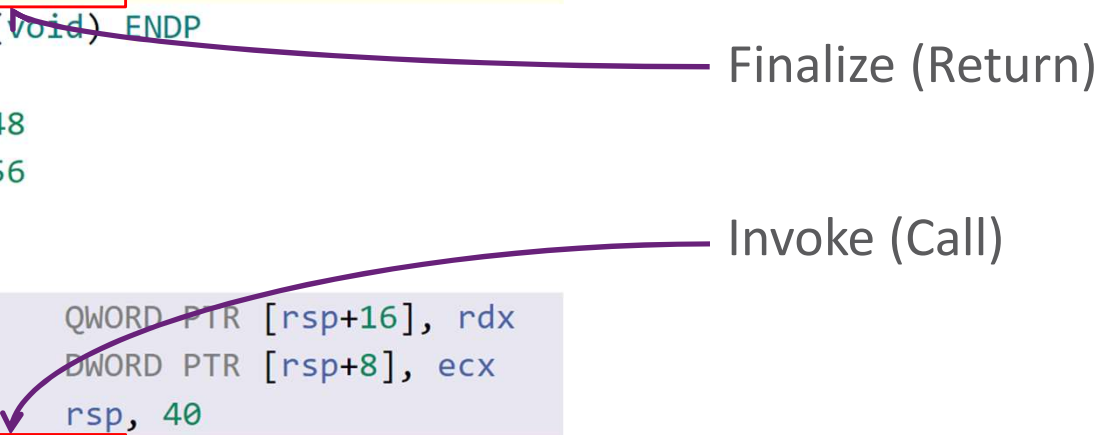
A routine that supports 2 operations: **Invoke/Finalize**



Subroutine

A routine that supports **Invoke/Finalize**

```
int get_zero(void) PROC
        xor     eax, eax
        ret     0
int get_zero(void) ENDP

__formal$ = 48
__formal$ = 56
main    PROC
$LN3:
        mov     QWORD PTR [rsp+16], rdx
        mov     DWORD PTR [rsp+8], ecx
        sub     rsp, 40
        call    int get_zero(void)
        add     rsp, 40
        ret     0
main    ENDP
```



Process

The way to run a program over the O/S.

Thread

The abstraction of a control flow in a process

Coroutine

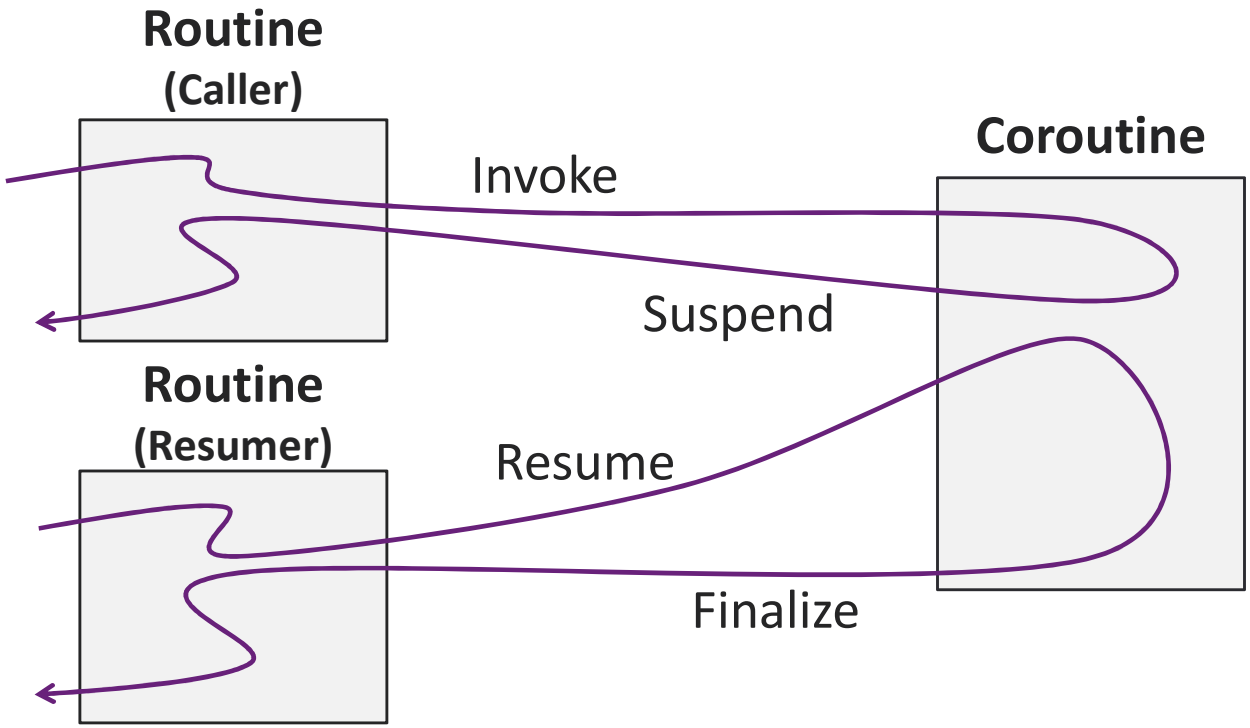
"Subroutines are special case of … coroutines" – Donald Knuth

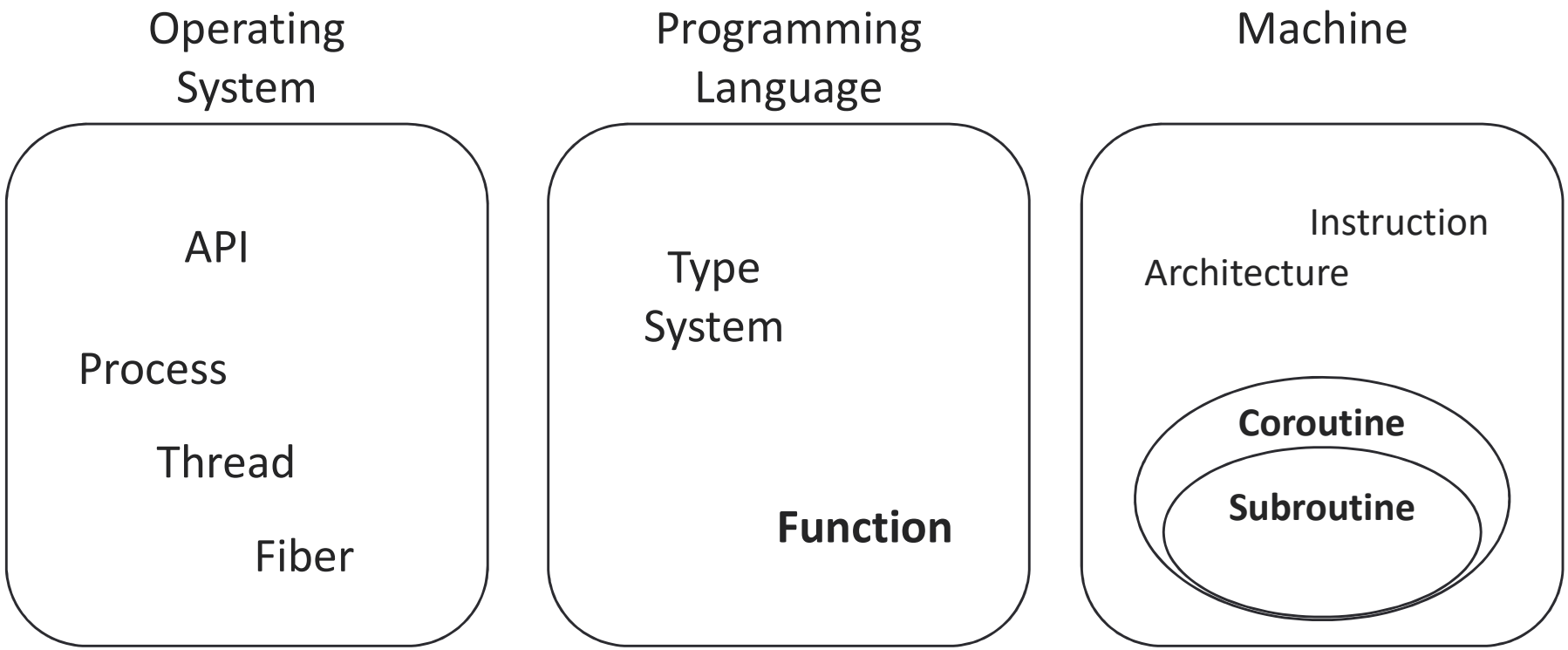
Operation	Subroutine	Coroutine	
Invoke	O	O	Goto start of a procedure(**call**)
Finalize	O	O	Cleanup and **return**
Suspend	X	O	Yield current control flow
Resume	X	O	Goto the suspended point in the procedure

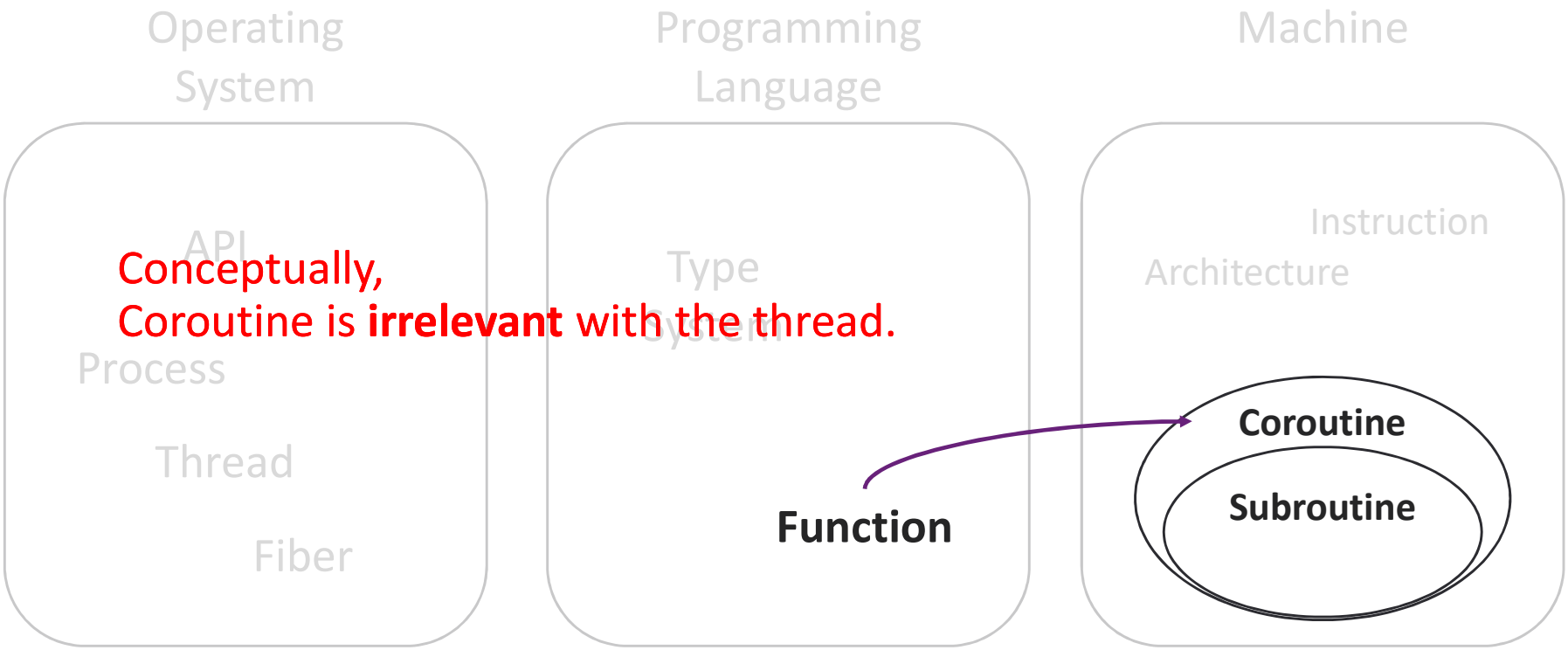
Don't take it so hard. You are already using it!

Coroutine

A routine that supports 4 operations: Invoke/Finalize/**Suspend/Resume**




Operating System
API
Process
Thread
Fiber
Programming Language
Type System
Function
Machine
Instruction Architecture
Coroutine
Subroutine



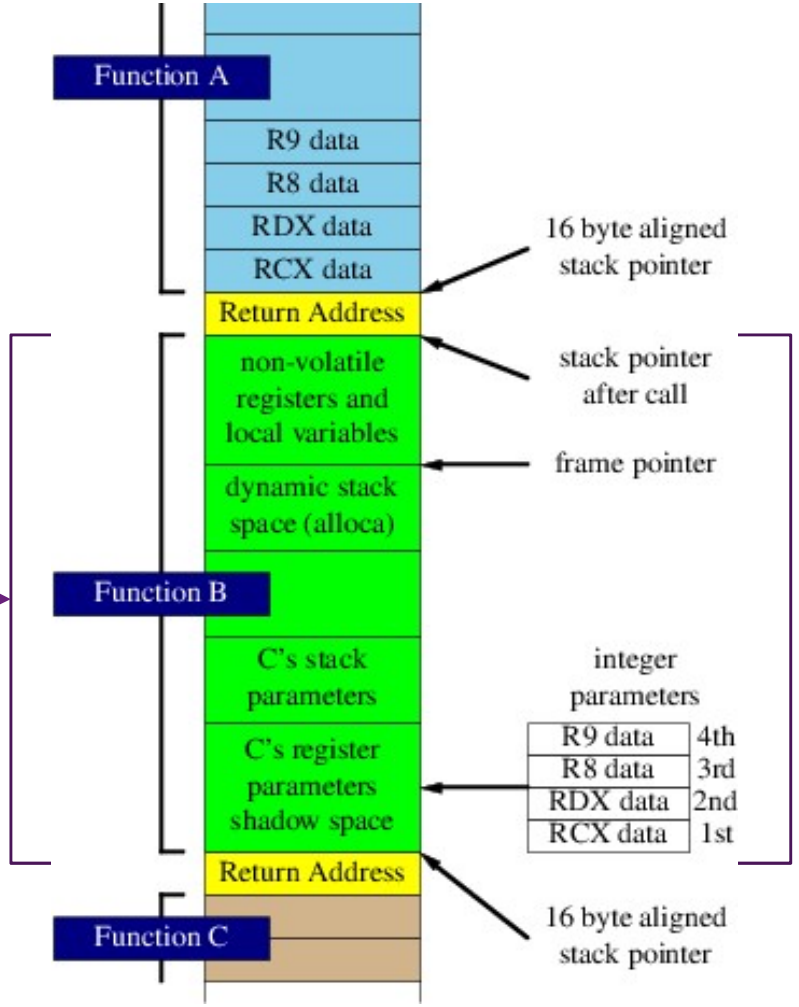
Be aware of the domain !

Routine has a state?

State == Memory

Function Frame

A memory object to hold status of the routine

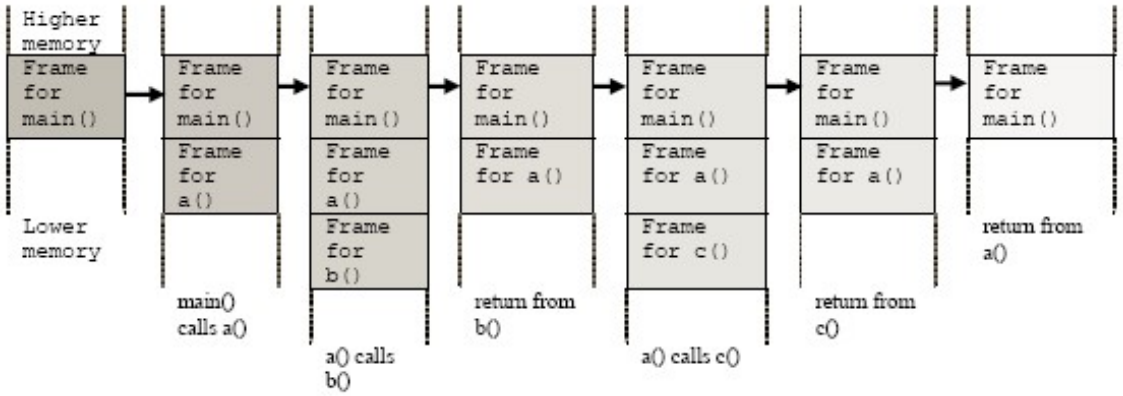


http://www.tortall.net/projects/yasm/manual/ru/html/objfmt-win64-exception.html

Call Stack

A way to manage function frame.

- Invoke: **Push** a function frame
- Return: **Pop** a function frame



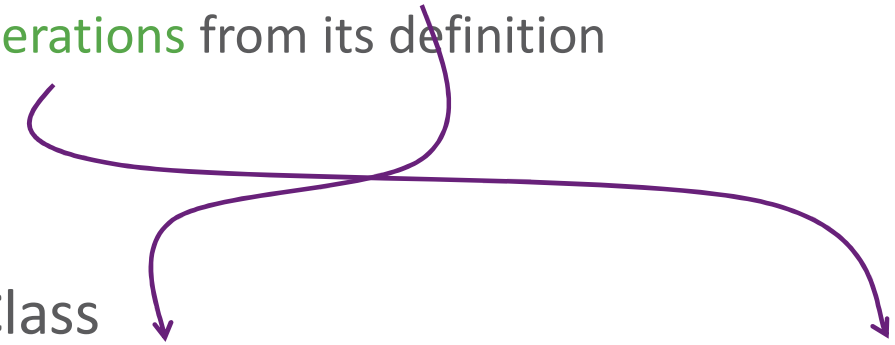
Efficient for subroutine.

In the **C** language,
all functions are subroutine !

https://manybutfinite.com/post/journey-to-the-stack/

Coroutine

- Routine that holds its state (Function Frame)
- 4 Operations from its definition



Task Class

- An organized data structure with its member functions

There is no much difference !

Coroutine over call stack?

Invoke/Return is OK. But what about Suspend/Resume?

Function frame's creation/**destruction**

To return to the suspended point, the frame must be **alive**!

Concern: The function frame's **life-cycle**

Stackful & Stackless

Stackful Coroutine
- Allocate coroutine's frame in stack

Stackless Coroutine
- Allocate coroutine's frame in free store (dynamically allocate)

How can we write coroutine
without harming subroutine?

C++ Extension for Coroutines

C++ will do like this!

Concept	In C++ Coroutine
Invoke	**No change**
Finalize	`co_return`
Suspend	`co_await`, `co_yield` `// unary operator`
Resume	`coro.resume()` `// coroutine_handle<P>::resume()`

At a glance...

How can we **define** the C++ Coroutine?

If one of the following exists in the function's body…

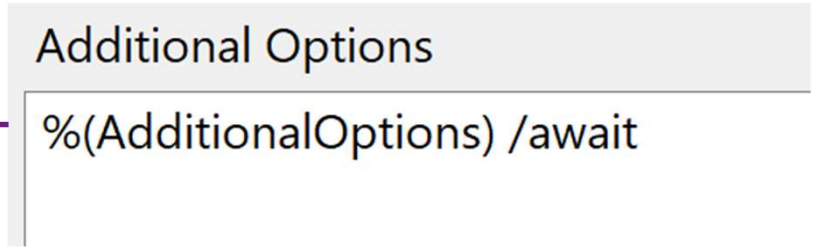
- `co_await` expression
- `co_yield` expression
- `co_return` statement
- `for co_await` statement

How can we **compile** the C++ Coroutine?

MSVC

- Visual Studio 2015 or later
- `/await`

vcxproj property > C/C++



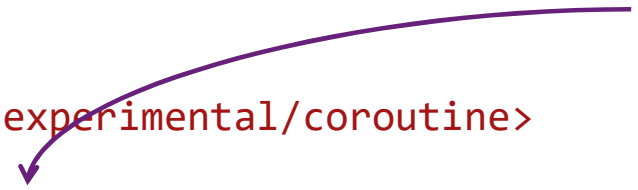
Clang Family

- 5.0 or later
- `-fcoroutines-ts -stdlib=libc++ -std=c++2a`

GCC

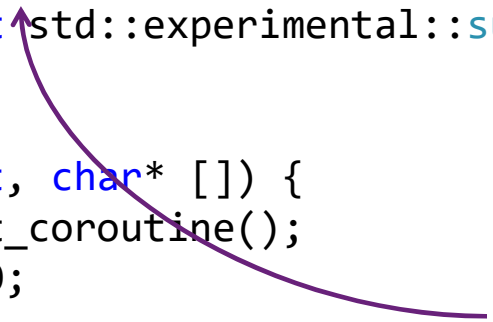
- Not yet …

C3783: ‘main’ cannot be a coroutine



```
#include <experimental/coroutine>

int main(int, char*[]) {
    co_await std::experimental::suspend_never{};
    return 0;
}
```

```
#include <experimental/coroutine>

auto my_first_coroutine() {
    co_await std::experimental::suspend_never{};
}

int main(int, char* []) {
    my_first_coroutine();
    return 0;
}
```



E0135:
class "std::experimental::coroutine_traits<<error-type>>"
has no member "promise_type"

promise_type ??

Coroutine Promise Requirement

A special type for compiler

- Helper type to generate coroutine code
- Frame allocation/deallocation
- Provide access to `coroutine_handle<P>`

https://isocpp.org/files/papers/N4402.pdf
https://lewissbaker.github.io/2018/09/05/understanding-the-promise-type

Coroutine Promise Requirement (N4402)

Expression	Note
P{}	Promise must be default constructible
p.get_return_object()	The return value of funtion. It can be future<T>, or so fined type.
p.return_value(v)	co_return statement. Pass the value v and th be consumed later.
p.return_value()	co_return statement. Pass void. Ca when the coroutine returns. And calling this can be the more value".
p.set_exceptio(e)	Pass the exceptio It will th e resumer activates the function with this context.
p.yield_value(v)	co_yield pression. Similar to return_value(v).
p.initial_suspend()	If return true, suspends at initial suspend point.
p.final_suspend()	If return true, suspends at final suspend point.

Let me explain these later

https://isocpp.org/files/papers/N4402.pdf
https://luncliff.github.io/posts/Exploring-MSVC-Coroutine.html

Coroutine Promise Requirement (N4402)

Expression	Note
P{}	Promise must be default constructible
p.get_return_object()	The return value of funtion. It can be future<T>, or some user-defined type.
p.return_value(v)	co_return statement. Pass the value v and the value will be consumed later.
p.return_value()	co_return statement. Pass void. Can be invoked when the coroutine returns. And calling this can be thought as "No more value".
p.set_exception(e)	Pass the exception. It will throw when the resumer activates the function with this context.
p.yield_value(v)	co_yield expression. Similar to return_value(v).
p.initial_suspend()	If return true, suspends at initial suspend point.
p.final_suspend()	If return true, suspends at final suspend point.

Programmer have to fill out the functions

https://isocpp.org/files/papers/N4402.pdf
https://luncliff.github.io/posts/Exploring-MSVC-Coroutine.html

We will cover that later…

The point is that…

Coroutine(stack-less) frame is managed **by Type System**(`promise_type`)

Awaitable Type and `co_await` Operator

How can we suspend the coroutine?

```cpp
#include <iostream>

using namespace std;
namespace coro = std::experimental;

auto example() -> return_ignore {
    puts("step 1");
    co_await coro::suspend_always{};
    puts("step 2");
}

int main(int, char*[]) {
    example();
    puts("step 3");
    return 0;
}
```

Expected output?

```cpp
#include <iostream>

using namespace std;
namespace coro = std::experimental;

auto example() -> return_ignore {
    puts("step 1");
    co_await coro::suspend_always{};
    puts("step 2");
}

int main(int, char*[]) {
    example();
    puts("step 3");
    return 0;
}
```

Output

```
step 1
step 3
```

https://wandbox.org/permlink/fRebS2VGQHRdGepp

```
#include <iostream>

using namespace std;
namespace coro = std::experimental;

auto example() -> return_ignore {
    puts("step 1");
    co_await coro::suspend_always{};
    puts("step 2");
}

int main(int, char*[]) {
    example();
    puts("step 3");
    return 0;
}
```

Where is this line?

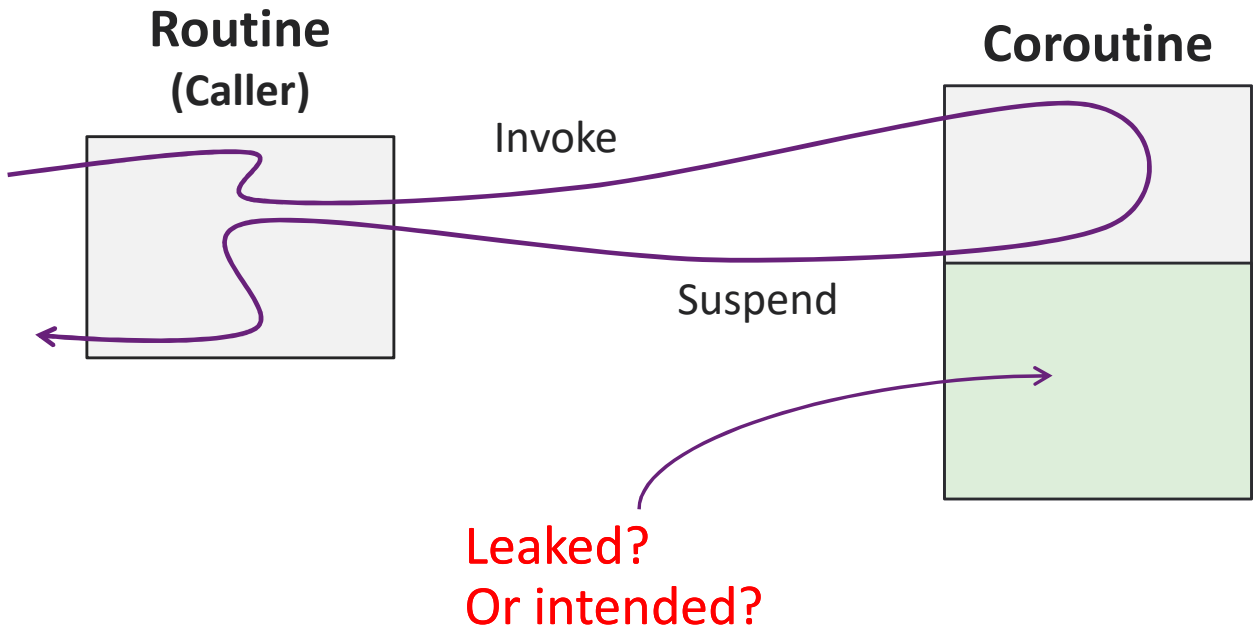
Output

```
step 1
step 3
```

Coverage Leak?

When the coroutine is suspended, the other routine have to resume it.

If not, the code after suspend point can be leaked…



```cpp
#include <iostream>

using namespace std;
namespace coro = std::experimental;

auto example() -> return_ignore {
    puts("step 1");
    co_await coro::suspend_never{};
    puts("step 2");
}

int main(int, char*[]) {
    example();
    puts("step 3");
    return 0;
}
```

Output

```
step 1
step 2
step 3
```

https://wandbox.org/permlink/PoX9rQzx0u1rTAx6

```cpp
#include <experimental/coroutine>
#include <future>

auto async_get_zero() -> std::future<int> {
    co_await std::experimental::suspend_always{};
    co_return 0;
}

int main(int, char*[]) {
    auto fz = async_get_zero();
    return fz.get();
}
```

Coroutine: suspend and wait for resume

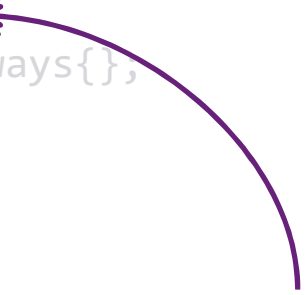
Subroutine: wait for coroutine's return

Some Deadlock (for VC++)

```cpp
#include <experimental/coroutine>
#include <future>

auto async_get_zero() -> std::future<int> {
    co_await std::experimental::suspend_always{};
    co_return 0;
}
```

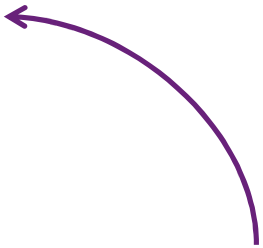
Problem of this code?

```
#include <experimental/coroutine>
#include <future>

auto async_get_zero() -> std::future<int> {
    co_await std::experimental::suspend_always{};
    co_return 0;
}
```



Future expects return.

```
#include <experimental/coroutine>
#include <future>

auto async_get_zero() -> std::future<int> {
    co_await std::experimental::suspend_always{};
    co_return 0;
}
```



But the coroutine might not guarantee `co_return`.

Be cautious with the interface !

```
using namespace std::experimental;
using awaitable = suspend_always;

auto routine_with_await(awaitable& aw) -> return_ignore
{
    co_await aw; // unary operator
}
```

co_await expression

```
using namespace std::experimental;
using awaitable = suspend_always;

auto routine_with_await(awaitable& aw) -> return_ignore
{
    using promise_type = return_ignore::promise_type;
    promise_type *p;

    if (aw.await_ready() == false) {
        auto rh = coroutine_handle<promise_type>::from_promise(*p);
        aw.await_suspend(rh);
        // ... return ...
    }
__suspend_point_n:
    aw.await_resume();
}
```

co_await: compiler's view

```
using namespace std::experimental;
using awaitable = suspend_always;

auto routine_with_await(awaitable& aw) -> return_ignore
{
    using promise_type = return_ignore::promise_type;
    promise_type *p;

    if (aw.await_ready() == false) {
        auto rh = coroutine_handle<promise_type>::from_promise(*p);
        aw.await_suspend(rh);
        // ... return ...
    }
__suspend_point_n:
    aw.await_resume();
}
```

For each `co_await`,
suspend point is generated

```
using namespace std::experimental;
using awaitable = suspend_always;

auto routine_with_await(awaitable& aw) -> return_ignore
{
    using promise_type = return_ignore::promise_type;
    promise_type *p;

    if (aw.await_ready() == false) {
        auto rh = coroutine_handle<promise_type>::from_promise(*p);
        aw.await_suspend(rh);
        // ... return ...
    }
__suspend_point_n:
    aw.await_resume();
}
```

Current coroutine's frame

await_suspend & coroutine_handle<P>

```
using namespace std::experimental;
using awaitable = suspend_always;

auto routine_with_await(awaitable& aw) -> return_ignore
{
    using promise_type = return_ignore::promise_type;
    promise_type *p;

    if (aw.await_ready() == false) {
        auto rh = coroutine_handle<promise_type>::from_promise(*p);
        aw.await_suspend(rh);
        // ... return ...
    }
__suspend_point_n:
    aw.await_resume();
}
```

await_suspend & coroutine_handle<P>

```
// <experimental/coroutine> // namespace std::experimental
class suspend_never
{
  public:
    bool await_ready() {
        return true;
    }
    void await_suspend(coroutine_handle<void>){}
    void await_resume(){}
};

class suspend_always
{
  public:
    bool await_ready() {
        return false;
    }
    void await_suspend(coroutine_handle<void>){}
    void await_resume(){}
};
```

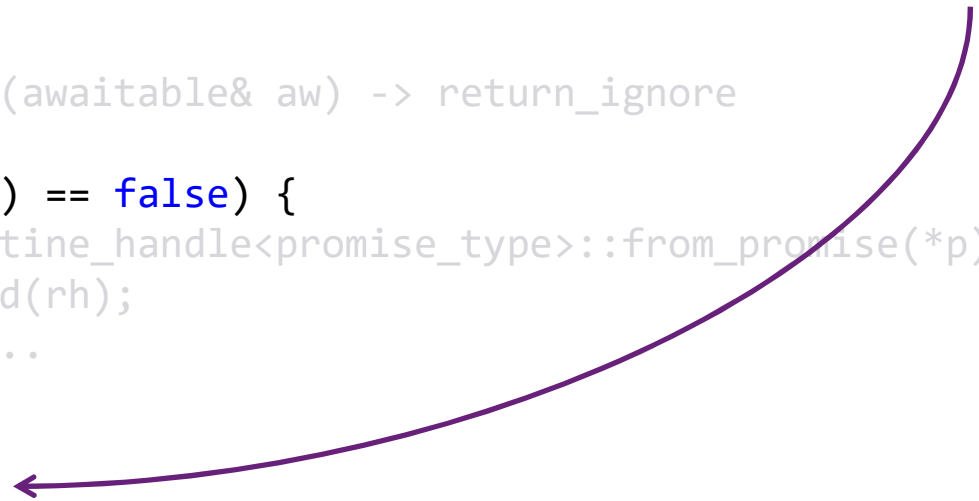
Pre-defined Awaitable Types

```
class suspend_never
{
  public:
    bool await_ready() {
        return true;
    }
    void await_suspend(coroutine_handle<void>){}
    void await_resume(){}
};
```

await_ready() == true

```
class suspend_never
{
  public:
    bool await_ready() {
        return true;
    }
    void await_suspend(coroutine_handle<void>){}
    void await_resume(){}
};

auto routine_with_await(awaitable& aw) -> return_ignore
{
    if (aw.await_ready() == false) {
        auto rh = coroutine_handle<promise_type>::from_promise(*p);
        aw.await_suspend(rh);
        // ... Return ...
    }
__suspend_point_n:
    aw.await_resume();
}
```

If `true`, bypass to `await_resume`



Ready - Resume

```
class suspend_always
{
  public:
    bool await_ready() {
        return false;
    }
    void await_suspend(coroutine_handle<void>){}
    void await_resume(){}
};
```

await_ready() == false

```
class suspend_always
{
  public:
    bool await_ready() {
        return false;
    }
    void await_suspend(coroutine_handle<void>){}
    void await_resume(){}
};
```

If `false`, call `await_suspend`
and **yield control flow** to activator routine

```
auto routine_with_await(awaitable& aw) -> return_ignore
{
    if (aw.await_ready() == false) {
        auto rh = coroutine_handle<promise_type>::from_promise(*p);
        aw.await_suspend(rh);
        // ... Return ...
    }
__suspend_point_n:
    aw.await_resume();
}
```

Ready – Suspend - Resume

```
struct wait_for_tuple
{
    bool await_ready();
    void await_suspend(coroutine_handle<void>);
    auto await_resume() -> std::tuple<int, bool>;
};
```

What about non-void?

```
struct wait_for_tuple
{
    bool await_ready();
    void await_suspend(coroutine_handle<void>);
    auto await_resume() -> std::tuple<int, bool>;
};

auto routine_with_await(wait_for_tuple& aw) -> return_ignore
{
    auto t = co_await aw; // t == std::tuple<int, bool>
}
```

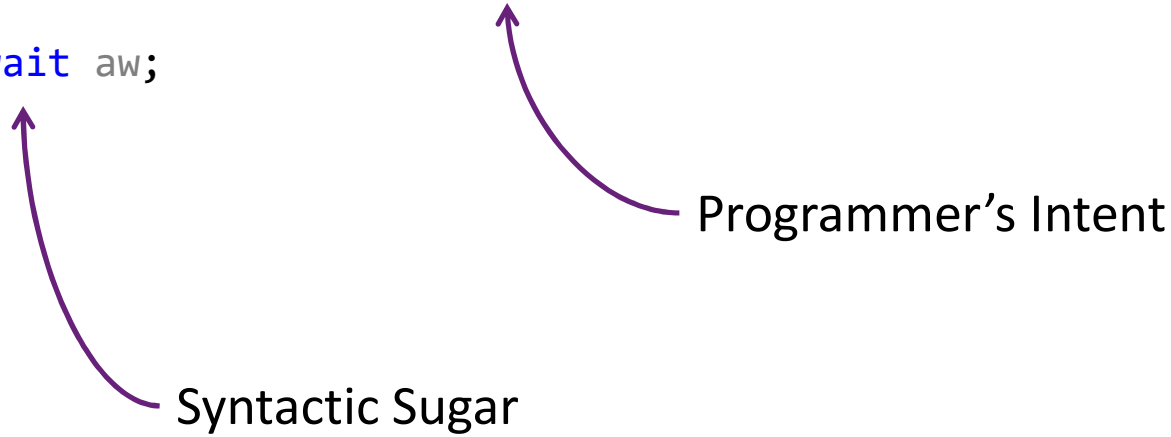
Nothing special…

```
struct wait_for_tuple
{
    bool await_ready();
    void await_suspend(coroutine_handle<void>);
    auto await_resume() -> std::tuple<int, bool>;
};

auto routine_with_await(wait_for_tuple& aw) -> return_ignore
{
    using promise_type = return_ignore::promise_type;
    promise_type *p;
    if (aw.await_ready() == false) {
        auto rh = coroutine_handle<promise_type>::from_promise(*p);
        aw.await_suspend(rh);
        // ... Return ...
    }
__suspend_point_n:
    auto t = aw.await_resume(); // t == std::tuple<int, bool>
}
```

```
struct wait_for_tuple
{
    bool await_ready();
    void await_suspend(coroutine_handle<void>);
    auto await_resume() -> std::tuple<int, bool>;
};


auto routine_with_await(wait_for_tuple& aw) -> return_ignore
{
    using promise_type = return_ignore::promise_type;
    promise_type *p;
    if (aw.await_ready() == false) {
        auto rh = coroutine_handle<promise_type>::from_promise(*p);
        aw.await_suspend(rh);
        // ... Return ...
    }
__suspend_point_n:
    auto t = aw.await_resume(); // t == std::tuple<int, bool>
}
```

```
using namespace std::experimental;
using awaitable = suspend_always;

auto routine_with_await(awaitable& aw) -> return_ignore
{
    auto v = co_await aw;
}
```

```
using namespace std::experimental;
using awaitable = suspend_always;

auto routine_with_await(awaitable& aw) -> return_ignore
{
    co_await aw;
}
```



`co_await` expression

Awaitable Type's Role

It's an interface for `co_await`

- Operator `co_await` requires multiple function
  - `await_ready`
  - `await_suspend`
  - `await_resume`

By using `co_await`…

- Compiler can generates suspend point at the line.
- Programmer can manage coroutine's control flow with the suspension

Coroutine Promise Requirement (N4736)

What is Promise Type? What should be in it?

```
using namespace std::experimental;
using awaitable = suspend_always;

auto routine_with_await(awaitable& aw) -> return_ignore
{
    using promise_type = return_ignore::promise_type;
    promise_type *p;

    if (aw.await_ready() == false) {
        auto rh = coroutine_handle<promise_type>::from_promise(*p);
        aw.await_suspend(rh);
        // ... return ...
    }
__suspend_point_n:
    aw.await_resume();
}
```

…promise_type?

Promise Type's Role

Allow compile time check (Type System)

- `coroutine_traits<T...>`

Coroutine frame construction/destruction

- Operator `new`/`delete`
- ctor/dtor
- `get_return_object`, `get_return_object_on_allocation_failure`

Receive return from the coroutine

- `co_return`: `return_value`, `return_void`
- `co_yield` : `yield_value`

```cpp
#include <experimental/coroutine>
namespace coro = std::experimental;

auto example(int a, double b, char *c) -> return_type
{
    co_await coro::suspend_never{};
}
```

2018/n4736.pdf

When compiler meets a coroutine...

```cpp
#include <experimental/coroutine>
namespace coro = std::experimental;

auto example(int a, double b, char *c) -> return_type
{
    // coroutine_traits<R, P1, ..., Pn>
    using T = coro::coroutine_traits<return_type, int, double, char *>;
    using promise_type = T::promise_type;

    co_await coro::suspend_never{};
}
```

Test the function with `coroutine_traits<T…>`

```cpp
#include <experimental/coroutine>
namespace coro = std::experimental;

auto example(int a, double b, char *c) -> return_type
{
    // coroutine_traits<R, P1, ..., Pn>
    using T = coro::coroutine_traits<return_type, int, double, char *>;
    using promise_type = T::promise_type;

    co_await coro::suspend_never{};
}
```

Return Type + Function Parameter Types

```
#include <experimental/coroutine>
namespace coro = std::experimental;

auto example(int a, double b, char *c) -> return_type
{
    // coroutine_traits<R, P1, ..., Pn>
    using T = coro::coroutine_traits<return_type, int, double, char *>;
    using promise_type = T::promise_type;

    co_await coro::suspend_never{};
}
```

```
template <class>
struct __void_t { typedef void type; };

template <class _Tp, class = void>
struct __coroutine_traits_sfinae {};

template <class _Tp>
struct __coroutine_traits_sfinae<_Tp,
                                 typename __void_t<typename _Tp::promise_type>::type>
{
    using promise_type = typename _Tp::promise_type;
};

template <typename _Ret, typename... _Args>
struct coroutine_traits
    : public __coroutine_traits_sfinae<_Ret>
{
};
```

llvm/libcxx/release_70/include/experimental/coroutine#L75

Traits in libc++ 7.0

```
template <class>
struct __void_t { typedef void type; };

template <class _Tp, class = void>
struct __coroutine_traits_sfinae {};

template <class _Tp>
struct __coroutine_traits_sfinae<_Tp,
                                 typename __void_t<typename _Tp::promise_type>::type>
{
    using promise_type = typename _Tp::promise_type;
};

template <typename _Ret, typename... _Args>
struct coroutine_traits
    : public __coroutine_traits_sfinae<_Ret>
{
};
```

Ignore SFINAE …

```
template <class>
struct __void_t { typedef void type; };

template <class _Tp, class = void>
struct __coroutine_traits_sfinae {};

template <class _Tp>
struct __coroutine_traits_sfinae<_Tp,
                                 typename __void_t<typename _Tp::promise_type>::type>
{
    using promise_type = typename _Tp::promise_type;
};

template <typename _Ret, typename... _Args>
struct coroutine_traits
    : public __coroutine_traits_sfinae<_Ret>
{
};
```

The core

```
#include <experimental/coroutine>
namespace coro = std::experimental;

auto example(int a, double b, char *c) -> return_type
{
    // coroutine_traits<R, P1, ..., Pn>
    using T = coro::coroutine_traits<return_type, int, double, char *>;
    using promise_type = T::promise_type;

    co_await coro::suspend_never{};
}
```

Does the `return_type` has `promise_type`?

`coroutine_traits<T...>`

Return Type Extension using `coroutine_traits<>`

Even though `return_type` doesn't contain `promise_type`,

Programmer can provide **template specialization** of `coroutine_traits<T...>`

to use the type as a return type of C++ coroutine

```
auto example() -> return_type
{
    // coroutine_traits<R, P1, ..., Pn>
    using T = coro::coroutine_traits<return_type>;
    using promise_type = T::promise_type;

    co_await coro::suspend_never{};
}
```

It doesn't need to be
the `return_type::promise_type`

```
#include <experimental/coroutine>

auto my_first_coroutine() {
    co_await std::experimental::suspend_never{};
}

int main(int, char* []) {
    my_first_coroutine();
    return 0;
}
```

E0135:
class "std::experimental::coroutine_traits<<error-type>>"
has no member "**promise_type**"

The reason of E0135

What does compiler do with the Coroutine Promise Requirement?

```
using namespace std::experimental;
auto example(int a, double b, char *c) -> return_type
{
    // ... programmer's code ...
}
```

If there is a coroutine...

```
using namespace std::experimental;
auto example(int a, double b, char *c) -> return_type
{
    using T = coroutine_traits<return_type, int, double, char *>;
    using promise_type = T::promise_type;

    // ... programmer's code ...
}
```

If test was successful…

```
using namespace std::experimental;
auto example(int a, double b, char *c) -> return_type
{
    using T = coroutine_traits<return_type, int, double, char *>;
    using promise_type = T::promise_type;

    // return_type * __return_object = ...
    promise_type p{ a,b,c };

    *__return_object = { p.get_return_object() };
    co_await p.initial_suspend();
    try {
        // ... programmer's code ...
    }
    catch (...) {
        p.unhandled_exception();
    }
__final_suspend_point:
    co_await p.final_suspend();
}
```

Code Generation from Promise

```
using namespace std::experimental;
auto example(int a, double b, char *c) -> return_type
{
    using T = coroutine_traits<return_type, int, double, char *>;
    using promise_type = T::promise_type;

    // return_type * __return_object = ...
    promise_type p{ a,b,c };

    *__return_object = { p.get_return_object() };
    co_await p.initial_suspend();
    try {
        // ... programmer's code ...
    }
    catch (...) {
        p.unhandled_exception();
    }
__final_suspend_point:
    co_await p.final_suspend();
}
```

Where is my code?

```
using namespace std::experimental;
auto example(int a, double b, char *c) -> return_type
{
    using T = coroutine_traits<return_type, int, double, char *>;
    using promise_type = T::promise_type;

    // return_type * __return_object = ...
    promise_type p{ a,b,c };

    *__return_object = { p.get_return_object() };
    co_await p.initial_suspend();
    try {
        // ... programmer's code ...
    }
    catch (...) {
        p.unhandled_exception();
    }
__final_suspend_point:
    co_await p.final_suspend();
}
```

Most of operations come from `promise_type`

```
using namespace std::experimental;
auto example(int a, double b, char *c) -> return_type
{
    using T = coroutine_traits<return_type, int, double, char *>;
    using promise_type = T::promise_type;

    // return_type * __return_object = ...
    promise_type p{ a,b,c };

    *__return_object = { p.get_return_object() };
    co_await p.initial_suspend();
    try {
        // ... programmer's code ...
    }
    catch (...) {
        p.unhandled_exception();
    }
__final_suspend_point:
    co_await p.final_suspend();
}
```

Promise: Construction

```
using namespace std::experimental;
auto example() -> return_type
{
    using T = coroutine_traits<return_type>;
    using promise_type = T::promise_type;

    // return_type * __return_object = ...
    promise_type p{};

    *__return_object = { p.get_return_object() };
    co_await p.initial_suspend();
    try {
        // ... programmer's code ...
    }
    catch (...) {
        p.unhandled_exception();
    }
__final_suspend_point:
    co_await p.final_suspend();
}
```

For non-matching argument,
use default constructor

Promise: Construction

```
#include <experimental/coroutine>

struct return_sample
{
    struct promise_type
    {
        promise_type();
        ~promise_type();

        promise_type(int, double, char *);
    };
};
using return_type = return_sample;
```

For general case

For special case

Promise: Ctor/Dtor

```
using namespace std::experimental;
auto example() -> return_type
{
    using T = coroutine_traits<return_type>;
    using promise_type = T::promise_type;

    // return_type * __return_object = ...
    promise_type p{};

    *__return_object = { p.get_return_object() };
    co_await p.initial_suspend();
    try {
        // ... programmer's code ...
    }
    catch (...) {
        p.unhandled_exception();
    }
__final_suspend_point:
    co_await p.final_suspend();
}
```

Code generation from Promise

```
using namespace std::experimental;
auto example() -> return_type
{
    using T = coroutine_traits<return_type>;
    using promise_type = T::promise_type;

    // return_type * __return_object = ...
    promise_type p{};

    *__return_object = { p.get_return_object() };
    co_await p.initial_suspend();
    try {
        // ... programmer's code ...
    }
    catch (...) {
        p.unhandled_exception();
    }
__final_suspend_point:
    co_await p.final_suspend();
}
```

Code generation from Promise

```
using return_type = return_sample;
auto example() -> return_type
{
    // return_type * __return_object = ...
    promise_type p{};
    *__return_object = { p.get_return_object() };
}
```

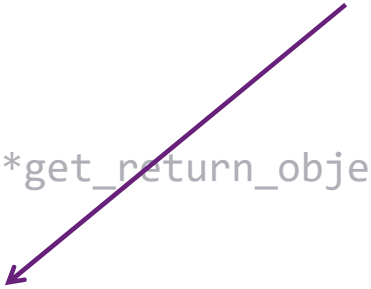
Promise: return object

```
struct return_sample
{
    struct promise_type
    {
        auto get_return_object() -> promise_type*
        {
            return this;
        }
        static promise_type *get_return_object_on_allocation_failure() noexcept;
    };

    return_sample(const promise_type *) noexcept;
};

using return_type = return_sample;
auto example() -> return_type
{
    // return_type * __return_object = ...
    promise_type p{};
    *__return_object = { p.get_return_object() };
}
```

```
struct return_sample
{
    struct promise_type
    {
        auto get_return_object() -> promise_type*
        {
            return this;
        }
        static promise_type *get_return_object_on_allocation_failure() noexcept;
    };

    return_sample(const promise_type *) noexcept;
};

using return_type = return_sample;
auto example() -> return_type
{
    // return_type * __return_object = ...
    promise_type p{};
    *__return_object = { p.get_return_object() };
}
```

It doesn't have to be `promise_type`

```
struct return_sample
{
    struct promise_type
    {
        auto get_return_object() -> promise_type*
        {
            return this;
        }
        static promise_type *get_return_object_on_allocation_failure() noexcept;
    };

    return_sample(const promise_type *) noexcept;
};

using return_type = return_sample;
auto example() -> return_type
{
    // return_type * __return_object = ...
    promise_type p{};
    *__return_object = { p.get_return_object() };
}
```

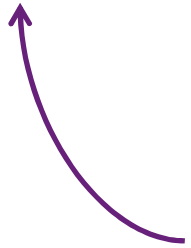


```
struct return_sample
{
    struct promise_type
    {
        auto get_return_object() -> promise_type*
        {
            return this;
        }
        static promise_type *get_return_object_on_allocation_failure() noexcept;
    };

    return_sample(const promise_type *) noexcept;
};

using return_type = return_sample;
auto example() -> return_type
{
    // return_type * __return_object = ...
    promise_type p{};
    *__return_object = { p.get_return_object() };
}
```

```
struct return_sample
{
    struct promise_type
    {
        auto get_return_object() -> promise_type*
        {
            return this;
        }
        static promise_type *get_return_object_on_allocation_failure() noexcept;
    };

    return_sample(const promise_type *) noexcept;
};

using return_type = return_sample;
auto example() -> return_type
{
    // return_type * __return_object = ...
    promise_type p{};
    *__return_object = { p.get_return_object() };
}
```

Actually, This is a dynamic allocation with `operator new`

```
struct return_sample
{
    struct promise_type
    {
        auto get_return_object() -> promise_type*
        {
            return this;
        }
        static promise_type *get_return_object_on_allocation_failure() noexcept;
    };

    return_sample(const promise_type *) noexcept;
};

using return_type = return_sample;
auto example() -> return_type
{
    // return_type * __return_object = ...
    promise_type p{};
    *__return_object = { p.get_return_object() };
}
```



Used when the allocation failed

```
using namespace std::experimental;
auto example() -> return_type
{
    using T = coroutine_traits<return_type>;
    using promise_type = T::promise_type;

    // return_type * __return_object = ...
    promise_type p{};

    *__return_object = { p.get_return_object() };
    co_await p.initial_suspend();
    try {
        // ... programmer's code ...
    }
    catch (...) {
        p.unhandled_exception();
    }
__final_suspend_point:
    co_await p.final_suspend();
}
```

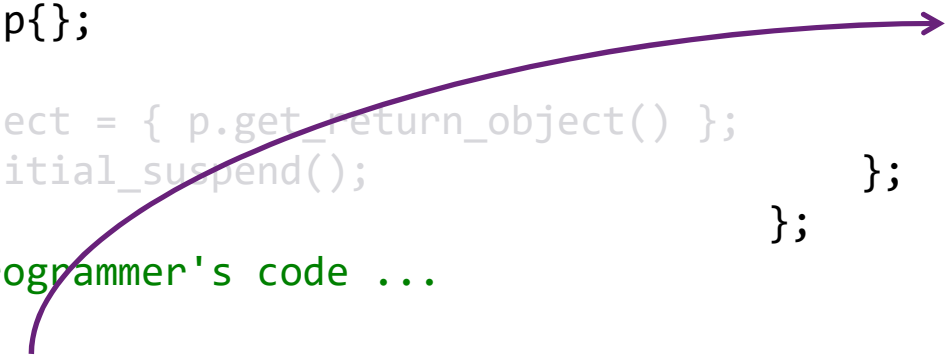
What about the exception handling?

```
using namespace std::experimental;
auto example() -> return_type
{
    using T = coroutine_traits<return_type>;
    using promise_type = T::promise_type;

    // return_type * __return_object = ...
    promise_type p{};

    *__return_object = { p.get_return_object() };
    co_await p.initial_suspend();
    try {
        // ... programmer's code ...
    }
    catch (...) {
        p.unhandled_exception();
    }
__final_suspend_point:
    co_await p.final_suspend();
}
```

The last exception handler by the compiler

```
using namespace std::experimental;
auto example() -> return_type
{
    using T = coroutine_traits<return_type>;
    using promise_type = T::promise_type;

    // return_type * __return_object = ...
    promise_type p{};

    *__return_object = { p.get_return_object() };
    co_await p.initial_suspend();
    try {
        // ... programmer's code ...
    }
    catch (...) {
        p.unhandled_exception();
    }
__final_suspend_point:
    co_await p.final_suspend();
}
```

```
struct return_sample
{
    struct promise_type
    {
        void unhandled_exception()
        {
            // std::current_exception();
            std::terminate();
        }
    };
};
```



Promise: Unhandled Exception

```
using namespace std::experimental;
auto example() -> return_type
{
    using T = coroutine_traits<return_type>;
    using promise_type = T::promise_type;

    // return_type * __return_object = ...
    promise_type p{};

    *__return_object = { p.get_return_object() };
    co_await p.initial_suspend();
    try {
        // ... programmer's code ...
    }
    catch (...) {
        p.unhandled_exception();
    }
__final_suspend_point:
    co_await p.final_suspend();
}
```

Promise: initial/final suspend

```
using namespace std::experimental;
auto example() -> return_type
{
    using T = coroutine_traits<return_type>;
    using promise_type = T::promise_type;

    // return_type * __return_object = ...
    promise_type p{};

    *__return_object = { p.get_return_object() };
    co_await p.initial_suspend();
    try {
        // ... programmer's code ...
    }
    catch (...) {
        p.unhandled_exception();
    }
__final_suspend_point:
    co_await p.final_suspend();
}
```

```
struct return_sample
{
    struct promise_type
    {
        auto initial_suspend()
        {
            return suspend_never{};
        }
        auto final_suspend()
        {
            return suspend_never{};
        }
    };
};
```

Expect Awaitable Return Type

```
using namespace std::experimental;
auto example() -> return_type
{
    using T = coroutine_traits<return_type>;
    using promise_type = T::promise_type;

    // return_type * __return_object = ...
    promise_type p{};

    *__return_object = { p.get_return_object() };
    co_await p.initial_suspend();
    try {
        // ... programmer's code ...
    }
    catch (...) {
        p.unhandled_exception();
    }
__final_suspend_point:
    co_await p.final_suspend();
}
```

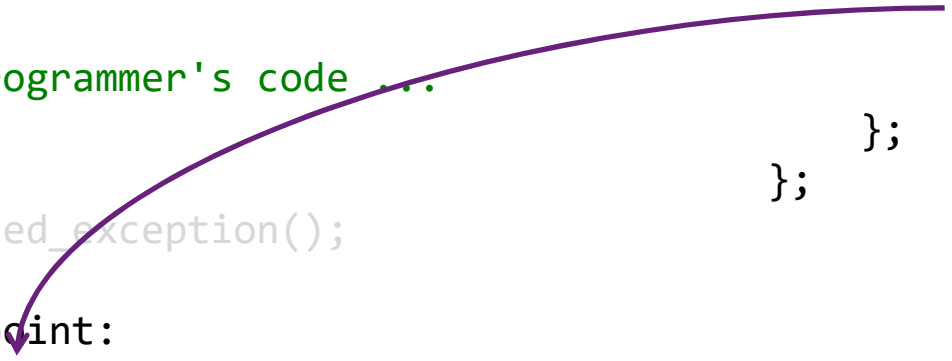
```
struct return_sample
{
    struct promise_type
    {
        auto initial_suspend()
        {
            return suspend_never{};
        }
        auto final_suspend()
        {
            return suspend_never{};
        }
    };
};
```

Initial Suspend

Enter programmer's code immediately?

```
using namespace std::experimental;
auto example() -> return_type
{
    using T = coroutine_traits<return_type>;
    using promise_type = T::promise_type;

    // return_type * __return_object = ...
    promise_type p{};

    *__return_object = { p.get_return_object() };
    co_await p.initial_suspend();
    try {
        // ... programmer's code ...
    }
    catch (...) {
        p.unhandled_exception();
    }
__final_suspend_point:
    co_await p.final_suspend();
}
```

```
struct return_sample
{
    struct promise_type
    {
        auto initial_suspend()
        {
            return suspend_never{};
        }
        auto final_suspend()
        {
            return suspend_never{};
        }
    };
};
```



Final Suspend

Delete coroutine frame after `co_return`?

```
#include <experimental/coroutine>
using namespace std::experimental;

auto example() -> pack<int> {
    co_await suspend_never{};
    co_return 0;
}
```

Let’s try a simple return type…

```
#include <experimental/coroutine>
using namespace std::experimental;

auto example() -> pack<int> {
    co_await suspend_never{};
    co_return 0;
}

template <typename Item>
struct pack
{
    promise_type* prom;

    pack(promise_type* p) :prom{ p } {};

    auto get() -> Item& {
        Item* ptr = prom->ptr;
        return *ptr;
    }
};
```

We will read data via promise

Definition of the return type

```cpp
#include <experimental/coroutine>
using namespace std::experimental;

auto example() -> pack<int> {
    co_await suspend_never{};
    co_return 0;
}

template <typename Item>
struct pack
{
    promise_type* prom;

    pack(promise_type* p) :prom{ p } {};

    auto get() -> Item& {
        Item* ptr = prom->ptr;
        return *ptr;
    }
};
```

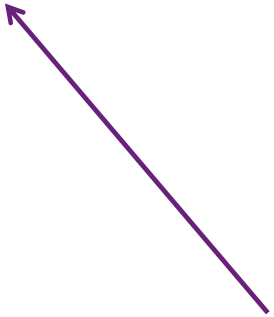
```cpp
struct promise_type
{
    Item* ptr = nullptr;

    suspend_never initial_suspend(){ return{}; }
    suspend_never final_suspend(){ return{}; }
    auto get_return_object() {
        return this;
    }
};
```

E2665: "pack<int>::promise_type" has no member "return_value"

+ with Promise

```cpp
#include <experimental/coroutine>
using namespace std::experimental;

auto example() -> pack<int> {
    co_await suspend_never{};
    co_return 0;
}

template <typename Item>
struct pack
{
    promise_type* prom;

    pack(promise_type* p) :prom{ p } {};

    auto get() -> Item& {
        Item* ptr = prom->ptr;
        return *ptr;
    }
};
```

```cpp
struct promise_type
{
    Item* ptr = nullptr;

    suspend_never initial_suspend(){ return{}; }
    suspend_never final_suspend(){ return{}; }
    auto get_return_object() {
        return this;
    }
    // for co_return with value
    void return_value(Item& ref) {
        ptr = std::addressof(ref);
    }
};
```



co_return requires return_value for value return

```cpp
#include <experimental/coroutine>
using namespace std::experimental;

auto example() -> pack<int> {
    co_await suspend_never{};
    co_return;
}
```

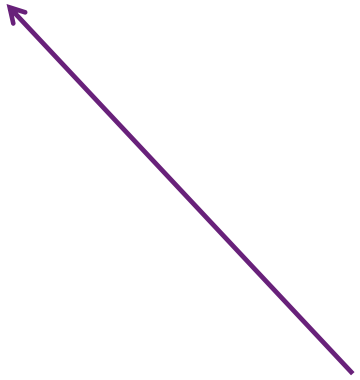
```cpp
struct promise_type
{
    Item* ptr = nullptr;

    suspend_never initial_suspend(){ return{}; }
    suspend_never final_suspend(){ return{}; }
    auto get_return_object() {
        return this;
    }
};
```

E2665: "pack<int>::promise_type" has no member "return_void"

Just `co_return` ?

```cpp
#include <experimental/coroutine>
using namespace std::experimental;

auto example() -> pack<int> {
    co_await suspend_never{};
    co_return;
}
```

```cpp
struct promise_type
{
    Item* ptr = nullptr;

    suspend_never initial_suspend(){ return{}; }
    suspend_never final_suspend(){ return{}; }
    auto get_return_object() {
        return this;
    }
    // for empty co_return
    void return_void() {}
};
```



`co_return` requires `return_void` for empty return

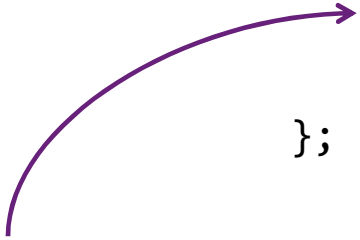
```
#include <experimental/coroutine>
using namespace std::experimental;

auto example() -> pack<int> {
    co_await suspend_never{};
}
```

```
struct promise_type
{
    Item* ptr = nullptr;

    suspend_never initial_suspend(){ return{}; }
    suspend_never final_suspend(){ return{}; }
    auto get_return_object() {
        return this;
    }

    void return_value(Item& ref) {
        ptr = std::addressof(ref);
    }
    void return_void() {}
};
```

Can we have both?

```cpp
#include <experimental/coroutine>
using namespace std::experimental;

auto example() -> pack<int> {
    co_await suspend_never{};
}
```

```cpp
struct promise_type
{
    Item* ptr = nullptr;

    suspend_never initial_suspend(){ return{}; }
    suspend_never final_suspend(){ return{}; }
    auto get_return_object() {
        return this;
    }

    void return_value(Item& ref) {
        ptr = std::addressof(ref);
    }
    void return_void() {}
};
```



C3782: pack<int>::promise_type: a coroutine's promise cannot contain both return_value and return_void

```
#include <experimental/coroutine>

auto example() -> pack<int> {
    co_await suspend_never{};
    co_return 0;
}
```

Let’s dig in `co_return` expression

```
#include <experimental/coroutine>

auto example() -> pack<int> {
    using promise_type = pack<int>::promise_type;
    promise_type *p;

    try {
    co_return 0; // programmer's code
    }
    catch (...) {
        p->unhandled_exception();
    }
__final_suspend_point:
    co_await p->final_suspend();
__destroy_point:
    delete p;
}
```

co_return: Compiler's View

```
#include <experimental/coroutine>

auto example() -> pack<int> {
    using promise_type = pack<int>::promise_type;
    promise_type *p;

    try {
    int _t1 = 0;
    p->return_value(_t1);
    goto __final_suspend_point;
    }
    catch (...) {
        p->unhandled_exception();
    }
__final_suspend_point:
    co_await p->final_suspend();
__destroy_point:
    delete p;
}
```

Generated code from `co_return 0;`

Promise Type's Role

Allow compile time check (Type System)

- `coroutine_traits<T...>`

Coroutine frame construction/destruction

- Operator `new`/`delete`
- ctor/dtor
- `get_return_object`, `get_return_object_on_allocation_failure`

Receive return from the coroutine

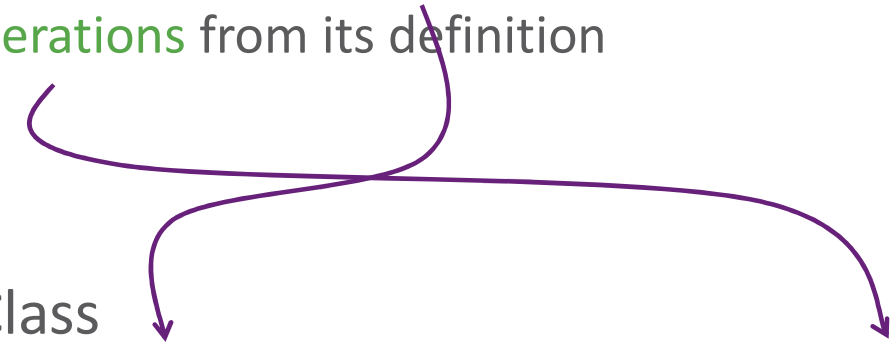
- `co_return`: `return_value`, `return_void`
- `co_yield` : `yield_value`

Coroutine Handle

Safe manner to resume/destroy a coroutine object

Coroutine

- Routine that holds its state (Function Frame)
- 4 Operations from its definition



Task Class

- An organized data structure with its member functions

Coroutine frames are treated like a normal object

```
template <typename PromiseType = void>
class coroutine_handle;

template <>
class coroutine_handle<void>
{
  protected:
    prefix_t prefix;
    static_assert(sizeof(prefix_t) == sizeof(void*));

  public:
    operator bool() const;
    void resume();
    void destroy();
    bool done() const;

    void* address() const;
    static coroutine_handle from_address(void*);
};
```

<experimental/resumable> in VC++
github.com/llvm-mirror/libcxx/release_70/include/experimental/coroutine

Coroutine Handle Type

```
template <typename PromiseType>
class coroutine_handle : public coroutine_handle<void>
{
  public:
    using promise_type = PromiseType;

  public:
    using coroutine_handle<void>::coroutine_handle;

  public:
    auto promise() -> promise_type&;
    static coroutine_handle from_promise(promise_type& prom);
};
```

<experimental/resumable> in VC++
github.com/llvm-mirror/libcxx/release_70/include/experimental/coroutine

Promise-Aware handle

```
bool operator==(const coroutine_handle<void>, const coroutine_handle<void>);
bool operator!=(const coroutine_handle<void>, const coroutine_handle<void>);
bool operator< (const coroutine_handle<void>, const coroutine_handle<void>);
bool operator> (const coroutine_handle<void>, const coroutine_handle<void>);
bool operator<=(const coroutine_handle<void>, const coroutine_handle<void>);
bool operator>=(const coroutine_handle<void>, const coroutine_handle<void>);
```

<experimental/resumable> in VC++
github.com/llvm-mirror/libcxx/release_70/include/experimental/coroutine

And some helpers

C++ Coroutine over Type System

Programmer guides the compiler using types

- Promise Type
- Awaitable Type

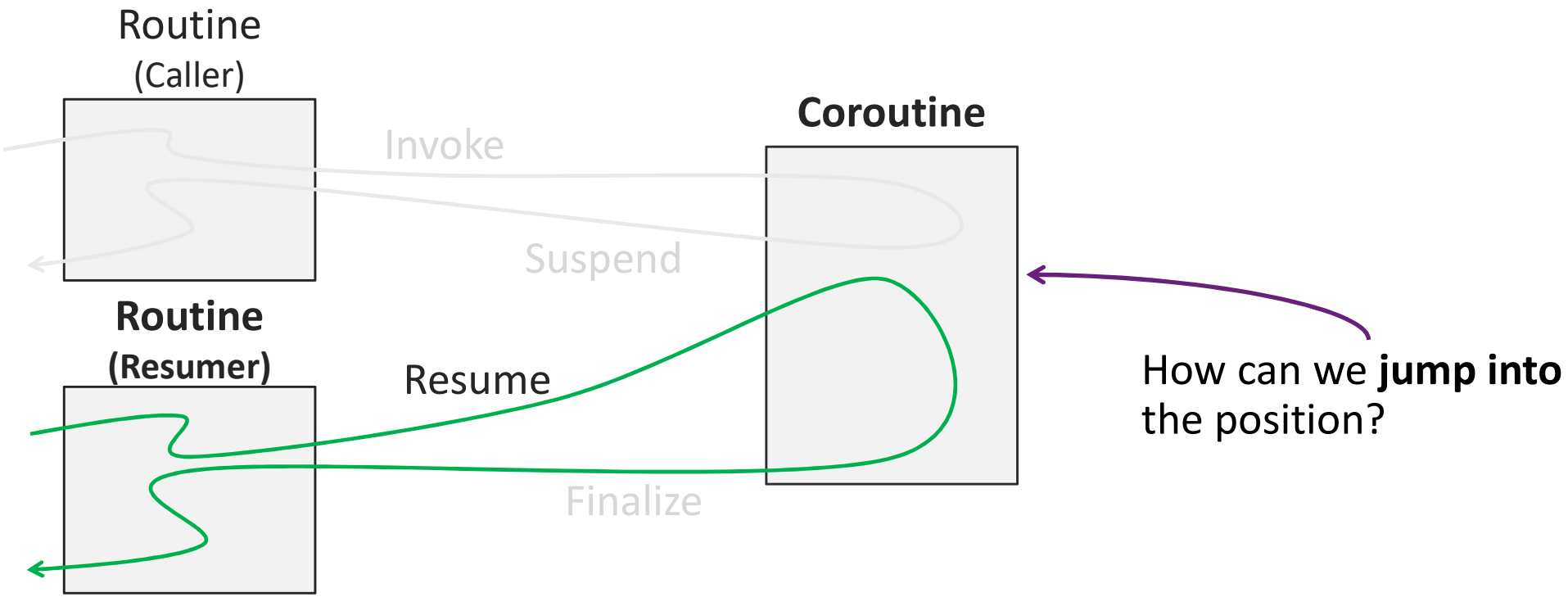
Insert operators to suspend/return

- `co_await`, `co_yield`
- `co_return`

But how can we control **resume operation**?

Coroutine

The code is generated by compiler. If so…



```
template <typename PromiseType = void>
class coroutine_handle;

template <>
class coroutine_handle<void>
{
  protected:
    prefix_t prefix;
    static_assert(sizeof(prefix_t) == sizeof(void*));

  public:
    operator bool() const;
    void resume();
    void destroy();
    bool done() const;

    void* address() const;
    static coroutine_handle from_address(void*);
};
```

→ Compiler specific memory layout

→ Compiler Intrinsic

It’s about the compiler

Compiler Intrinsic for C++ Coroutine

Intrinsic: Built in functions of compiler.

Both MSVC and Clang exports intrinsic for `coroutine_handle<void>` implementation.

GCC? Not yet…

Compiler Intrinsic for C++ Coroutine

MSVC

- `size_t _coro_done(void *)`
- `size_t _coro_resume(void *)`
- `void _coro_destroy(void *)`
- `...`

Clang

- `__builtin_coro_done`
- `__builtin_coro_resume`
- `__builtin_coro_destroy`
- `__builtin_coro_promise`
- `...`

There are more, but their usage are not visible.

`<experimental/resumable>` in VC++
github.com/llvm-mirror/libcxx/release_70/include/experimental/coroutine
https://clang.llvm.org/docs/LanguageExtensions.html#c-coroutines-support-builtins

Coroutine Intrinsic: MSVC

```
explicit operator bool() const {
    return _Ptr != nullptr;
}

void resume() const {
    _coro_resume(_Ptr);
}

void destroy(){
    _coro_destroy(_Ptr);
}

bool done() const {
    // REVISIT: should return _coro_done() == 0; when intrinsic is
    // hooked up
    return (_Ptr->_Index == 0);
}
```

Coroutine Intrinsic: Clang

```
explicit operator bool() const {
    return __handle_;
}

void resume() {
    __builtin_coro_resume(__handle_);
}

void destroy() {
    __builtin_coro_destroy(__handle_);
}

bool done() const {
    return __builtin_coro_done(__handle_);
}
```

```
template <typename PromiseType = void>
class coroutine_handle;

template <>
class coroutine_handle<void>
{
  protected:
    prefix_t prefix;
    static_assert(sizeof(prefix_t) == sizeof(void*));

  public:
    operator bool() const;
    void resume();
    void destroy();
    bool done() const;

    void* address() const;
    static coroutine_handle from_address(void*);
};
```

What about this?

Structure of Coroutine Frame?

What are in the Coroutine Frame?

Frame == Routine's state

Similar to subroutine's frame, but some additional comes with it…

CppCon 2016: "Introduction to C++ Coroutines"

What are in the Coroutine Frame?

- Local variables
  - Function arguments
  - Temporary variables (+ Awaitable)
  - Return value
- Coroutine Frame's Prefix (for coroutine_handle<void>)
- Promise object
- Compiler specific storage (maybe)

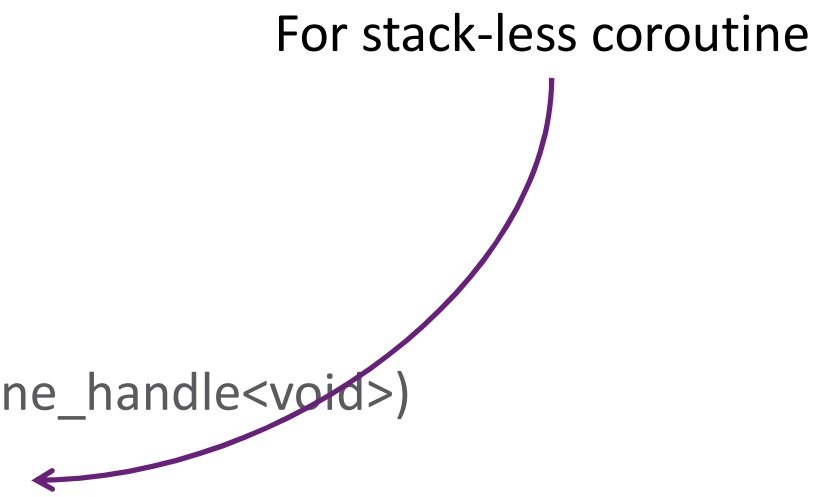
CppCon 2016: "Introduction to C++ Coroutines"

Same with subroutine

- Local variables
  - Function arguments
  - Temporary variables (+ Awaitable)
  - Return value
- Coroutine Frame's Prefix (for coroutine_handle<void>)
- Promise object
- Compiler specific storage (maybe)

CppCon 2016: "Introduction to C++ Coroutines"

- Local variables
  - Function arguments
  - Temporary variables (+ Awaitable)
  - Return value
- Coroutine Frame's Prefix (for coroutine_handle<void>)
- Promise object
- Compiler specific storage (maybe)



CppCon 2016: "Introduction to C++ Coroutines"

How they are allocated?

Allocation of the frame: via Promise Type

N4736, 11.4.4

… The allocation function's name is looked up in the scope of P.
If this lookup fails, the allocation function's name is looked up in the global scope. …

… The deallocation function's name is looked up in the scope of P.
If this lookup fails, the deallocation function's name is looked up in the global scope …

2018/n4736.pdf

```
class return_type {
public:
  struct promise_type {
    auto operator new(size_t sz) -> void *;
    void operator delete(void *ptr, size_t sz);
  };
};

auto example(Args... args) -> return_type {
  using T = coroutine_traits<return_type, Args...>;
  using promise_type = T::promise_type;
  using frame_type = tuple<frame_prefix, promise_type, Args...>;

  auto *frame = (frame_type *)promise_type::operator new(sizeof(frame_type));
  auto *p = addressof(get<1>(*frame)); // promise_type
  // ... coroutine code generation ...

__destroy_point:
  promise_type::operator delete(frame, sizeof(frame_type));
}
```

Allocation of the Frame

```
class return_type {
public:
  struct promise_type {
    auto operator new(size_t sz) -> void *;
    void operator delete(void *ptr, size_t sz);
  };
};

auto example(Args... args) -> return_type {
  using T = coroutine_traits<return_type, Args...>;
  using promise_type = T::promise_type;
  using frame_type = tuple<frame_prefix, promise_type, Args...>;

  auto *frame = (frame_type *)promise_type::operator new(sizeof(frame_type));
  auto *p = addressof(get<1>(*frame)); // promise_type
  // ... coroutine code generation ...

__destroy_point:
  promise_type::operator delete(frame, sizeof(frame_type));
}
```

Compiler will generate something like this…

Frame type for the function

```
class return_type {
public:
  struct promise_type {
    auto operator new(size_t sz) -> void *;
    void operator delete(void *ptr, size_t sz);
  };
};

auto example(Args... args) -> return_type {
  using T = coroutine_traits<return_type, Args...>;
  using promise_type = T::promise_type;
  using frame_type = tuple<frame_prefix, promise_type, Args...>;

  auto *frame = (frame_type *)promise_type::operator new(sizeof(frame_type));
  auto *p = addressof(get<1>(*frame)); // promise_type
  // ... coroutine code generation ...

__destroy_point:
  promise_type::operator delete(frame, sizeof(frame_type));
}
```

Managed via `promise_type`

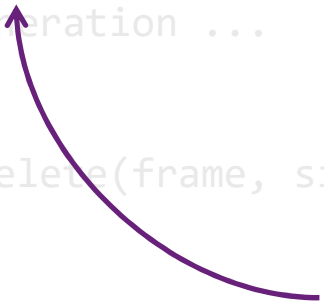
Look up in the scope of P

```
class return_type {
public:
  struct promise_type {
    auto operator new(size_t sz) -> void *;
    void operator delete(void *ptr, size_t sz);
  };
};

auto example(Args... args) -> return_type {
  using T = coroutine_traits<return_type, Args...>;
  using promise_type = T::promise_type;
  using frame_type = tuple<frame_prefix, promise_type, Args...>;

  auto *frame = (frame_type *)promise_type::operator new(sizeof(frame_type));
  auto *p = addressof(get<1>(*frame)); // promise_type
  // ... coroutine code generation ...

__destroy_point:
  promise_type::operator delete(frame, sizeof(frame_type));
}
```

If there is no definition,
Use global allocation/deallocation

Look up in the global scope

```
class return_type {
public:
  struct promise_type {
    auto operator new(size_t sz) -> void *;
    void operator delete(void *ptr, size_t sz);
  };
};

auto example(Args... args) -> return_type {
  using T = coroutine_traits<return_type, Args...>;
  using promise_type = T::promise_type;
  using frame_type = tuple<frame_prefix, promise_type, Args...>;

  auto *frame = (frame_type *)promise_type::operator new(sizeof(frame_type));
  auto *p = addressof(get<1>(*frame)); // promise_type
  // ... coroutine code generation ...

__destroy_point:
  promise_type::operator delete(frame, sizeof(frame_type));
}
```



Frame includes the promise object

Lifetime & Copy/Move elision

N4736, 11.4.4

When a coroutine is invoked, a copy is created for each coroutine parameter …
*… **The lifetime of parameter** copies ends immediately after the lifetime of the coroutine promise object ends. …*

N4736, 15.8.3

*in a coroutine, a copy of a coroutine parameter can be omitted and references to **that copy replaced with references to the corresponding parameters** if the meaning of the program will be unchanged …*

2018/n4736.pdf

```
auto example(Args... args) -> return_type {
  using T = coroutine_traits<return_type, Args...>;
  using promise_type = T::promise_type;
  using frame_type = tuple<frame_prefix, promise_type, Args...>;

  auto *frame = (frame_type *)promise_type::operator new(sizeof(frame_type));
  auto *p = addressof(get<1>(*frame)); // promise_type
  // ... coroutine code generation ...

__destroy_point:
  promise_type::operator delete(frame, sizeof(frame_type));
}
```

Let's talk about this …

Frame Prefix?

```
template <>
struct coroutine_handle<void> {
    struct _Resumable_frame_prefix {
        typedef void(__cdecl *_Resume_fn)(void *);
        _Resume_fn _Fn;
        uint16_t _Index;
        uint16_t _Flags;
    };
  protected:
    _Resumable_frame_prefix *_Ptr = nullptr;
};
```

<experimental/resumable>

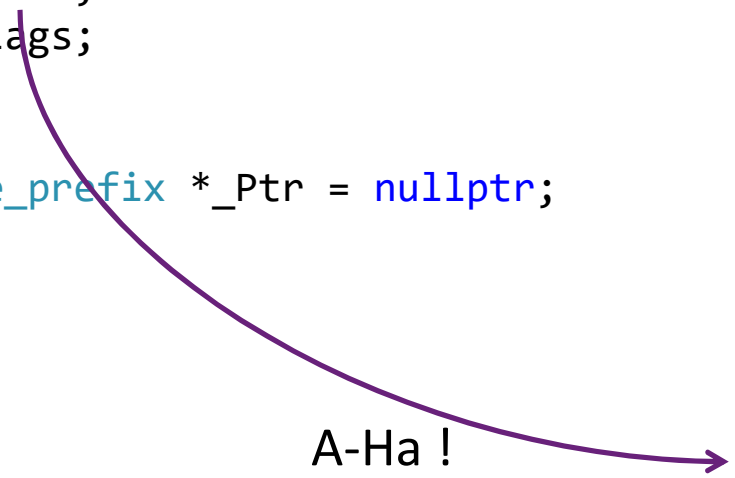
Definition in VC++

```
template <>
struct coroutine_handle<void> {
    struct _Resumable_frame_prefix {
        typedef void(__cdecl *_Resume_fn)(void *);
        _Resume_fn _Fn;
        uint16_t _Index;
        uint16_t _Flags;
    };
  protected:
    _Resumable_frame_prefix *_Ptr = nullptr;
};
```

Index for the suspend points?

<experimental/resumable>

```
template <>
struct coroutine_handle<void> {
    struct _Resumable_frame_prefix {
        typedef void(__cdecl *_Resume_fn)(void *);
        _Resume_fn _Fn;
        uint16_t _Index;
        uint16_t _Flags;
    };
  protected:
    _Resumable_frame_prefix *_Ptr = nullptr;
};
```




```
switch (frame->index) {
  case 2: // initial suspended
    goto __suspend_point_1;
  case 4: // suspended in point_1
    goto __suspend_point_2;

  // the other case ...

  case 0: // final suspended
    // resume is error !
}
```

<experimental/resumable>

```
template <>
struct coroutine_handle<void> {
    struct _Resumable_frame_prefix {
        typedef void(__cdecl *_Resume_fn)(void *);
        _Resume_fn _Fn;
        uint16_t _Index;
        uint16_t _Flags;
    };
  protected:
    _Resumable_frame_prefix *_Ptr = nullptr;
};
```

cdecl + void(void*) ?

<experimental/resumable>

Calling Convention: `__cdecl`

Stack clean-up by caller(calling function)

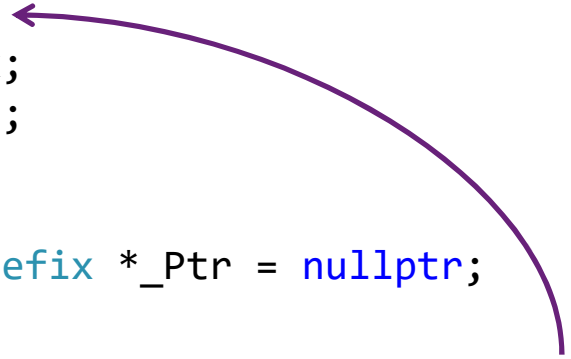
== For `void` return, no clean-up is required

== coroutine's frame is never touched by caller after `_Resume_fn`.

Since those variables are in placed in (stack-less) coroutine frame,
this is understandable and necessary!

https://docs.microsoft.com/ko-kr/cpp/cpp/cdecl?view=vs-2017

```
template <>
struct coroutine_handle<void> {
    struct _Resumable_frame_prefix {
        typedef void(__cdecl *_Resume_fn)(void *);
        _Resume_fn _Fn;
        uint16_t _Index;
        uint16_t _Flags;
    };
  protected:
    _Resumable_frame_prefix *_Ptr = nullptr;
};
```



So this is almost equivalent to goto

<experimental/resumable>

```
template <>
class coroutine_handle<void> {
  private:
    template <class _PromiseT> friend class coroutine_handle;
    void* __handle_;
};
```

Zero information :(

But I was sure that each compiler' had different layout because **clang-cl Compiler + VC++ Header** made a crash…

[libcxx/release_70/include/experimental/coroutine]

Definition in libc++ 7.0

```
template <>
class coroutine_handle<void> {
  private:
    template <class _PromiseT> friend class coroutine_handle;
    void* __handle_;
};

using procedure_t = void(__cdecl*)(void*);

struct clang_frame_prefix final
{
    procedure_t factivate;
    procedure_t fdestroy;
};
static_assert(aligned_size_v<clang_frame_prefix> == 16);
```

Gor Nishanov “C++ Coroutines: Under the covers”

https://github.com/luncliff/coroutine/blob/1.4/interface/coroutine/frame.h

Not that complex !

```
template <>
class coroutine_handle<void> {
  private:
    template <class _PromiseT> friend class coroutine_handle;
    void* __handle_;
};

using procedure_t = void(__cdecl*)(void*);

struct clang_frame_prefix final
{
    procedure_t factivate;
    procedure_t fdestroy;
};
static_assert(aligned_size_v<clang_frame_prefix> == 16);
```

Resume function comes first.
When the coroutine is final suspended, the pointer becomes `nullptr`

Gor Nishanov “C++ Coroutines: Under the covers”

```
template <>
class coroutine_handle<void> {
  private:
    template <class _PromiseT> friend class coroutine_handle;
    void* __handle_;
};

using procedure_t = void(__cdecl*)(void*);

struct clang_frame_prefix final
{
    procedure_t factivate;
    procedure_t fdestroy;
};
static_assert(aligned_size_v<clang_frame_prefix> == 16);
```

Destroy function invokes destructors of all variables in the coroutine frame.

[Gor Nishanov “C++ Coroutines: Under the covers”]

That’s enough for `coroutine_handle<void>`

Next: `coroutine_handle<promise_type>`

```
static coroutine_handle from_promise(_Promise& __promise) _NOEXCEPT {
    typedef typename remove_cv<_Promise>::type _RawPromise;
    coroutine_handle __tmp;
    __tmp.__handle_ = __builtin_coro_promise(
        _VSTD::addressof(const_cast<_RawPromise&>(__promise)),
        __alignof(_Promise), true);
    return __tmp;
}
```

libcxx/release_70/include/experimental/coroutine#L252

A strange calculation

```
static coroutine_handle from_promise(_Promise& __promise) _NOEXCEPT {
    typedef typename remove_cv<_Promise>::type _RawPromise;
    coroutine_handle __tmp;
    __tmp.__handle_ = __builtin_coro_promise(
        _VSTD::addressof(const_cast<_RawPromise&>(__promise)),
        __alignof(_Promise), true);
    return __tmp;
}
```

`__alignof` returns 16 * N

```
__handle_ = __builtin_coro_promise(addressof(__promise), __alignof(_Promise), true);
```

Address & Integer argument?
The result must be a address !

libcxx/release_70/include/experimental/coroutine#L252

```
static const size_t _ALIGN_REQ = sizeof(void *) * 2;
static const size_t _ALIGNED_SIZE =
    is_empty_v<_PromiseT>
    ? 0
    : ((sizeof(_PromiseT) + _ALIGN_REQ - 1) & ~(_ALIGN_REQ - 1));

_PromiseT &promise() const noexcept {
    return *const_cast<_PromiseT *>(reinterpret_cast<_PromiseT const *>(
        reinterpret_cast<char const *>(_Ptr) - _ALIGNED_SIZE));
}
static coroutine_handle from_promise(_PromiseT &_Prom) noexcept {
    auto _FramePtr = reinterpret_cast<char *>(_STD addressof(_Prom)) + _ALIGNED_SIZE;
    coroutine_handle<_PromiseT> _Result;
    _Result._Ptr = reinterpret_cast<_Resumable_frame_prefix *>(_FramePtr);
    return _Result;
}
```

<experimental/resumable>

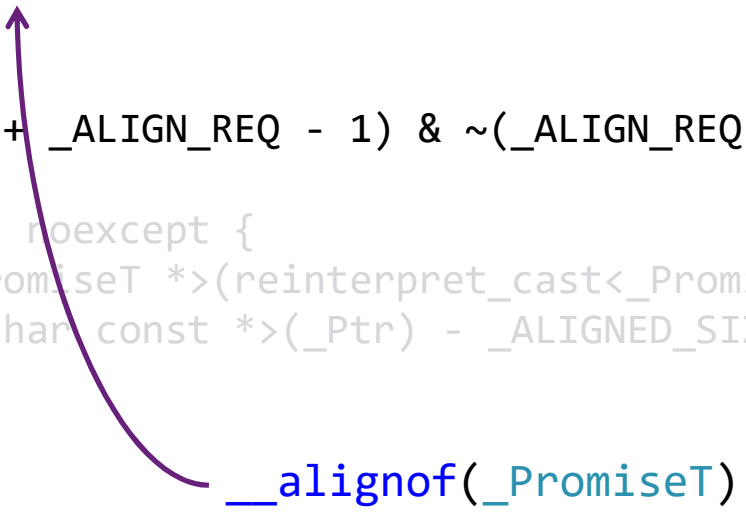
In VC++...

```
static const size_t _ALIGN_REQ = sizeof(void *) * 2;
static const size_t _ALIGNED_SIZE =
    is_empty_v<_PromiseT>
    ? 0
    : ((sizeof(_PromiseT) + _ALIGN_REQ - 1) & ~(_ALIGN_REQ - 1));

_PromiseT &promise() const noexcept {
    return *const_cast<_PromiseT *>(reinterpret_cast<_PromiseT const *>(
        reinterpret_cast<char const *>(_Ptr) - _ALIGNED_SIZE));
}
```

Complex, but it enforces 16 * N
(Same with `__alignof` in clang)

Align size

```
static const size_t _ALIGN_REQ = sizeof(void *) * 2;
static const size_t _ALIGNED_SIZE =
    is_empty_v<_PromiseT>
    ? 0
    : ((sizeof(_PromiseT) + _ALIGN_REQ - 1) & ~(_ALIGN_REQ - 1));

_PromiseT &promise() const noexcept {
    return *const_cast<_PromiseT *>(reinterpret_cast<_PromiseT const *>(
        reinterpret_cast<char const *>(_Ptr) - _ALIGNED_SIZE));
}
```



<experimental/resumable>

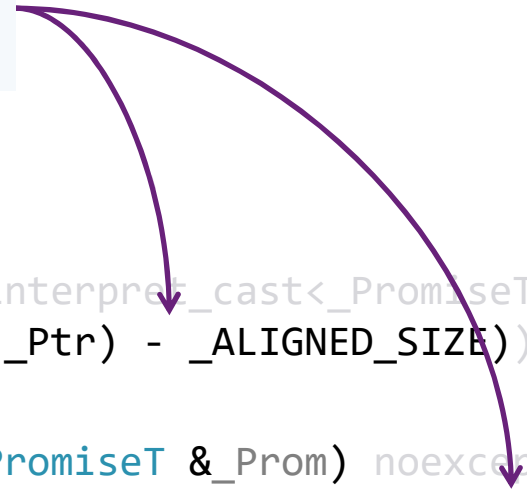
Align size of Promise Type

```
_PromiseT &promise() const noexcept {
    return *const_cast<_PromiseT *>(reinterpret_cast<_PromiseT const *>(
        reinterpret_cast<char const *>(_Ptr) - _ALIGNED_SIZE));
}
static coroutine_handle from_promise(_PromiseT &_Prom) noexcept {
    auto _FramePtr = reinterpret_cast<char *>(_STD addressof(_Prom)) + _ALIGNED_SIZE;
    coroutine_handle<_PromiseT> _Result;
    _Result._Ptr = reinterpret_cast<_Resumable_frame_prefix *>(_FramePtr);
    return _Result;
}
```

<experimental/resumable>

Core part of 2 functions

```
| Promise | Frame Prefix | Local variables |
          \
          resumable_handle<void>
```



```
_PromiseT &promise() const noexcept {
    return *const_cast<_PromiseT *>(reinterpret_cast<_PromiseT const *>(
        reinterpret_cast<char const *>(_Ptr) - _ALIGNED_SIZE));
}
static coroutine_handle from_promise(_PromiseT &_Prom) noexcept {
    auto _FramePtr = reinterpret_cast<char *>(_STD addressof(_Prom)) + _ALIGNED_SIZE;
    coroutine_handle<_PromiseT> _Result;
    _Result._Ptr = reinterpret_cast<_Resumable_frame_prefix *>(_FramePtr);
    return _Result;
}
```

<experimental/resumable>

The layout of MSVC

```
__handle_ = __builtin_coro_promise(addressof(__promise), __alignof(_Promise), true);
```

Clang's Frame

After some analysis…

```
| Frame Prefix | Promise | ? | Local variables |
 \
 resumable_handle<void>
```

MSVC's Frame `| Promise | Frame Prefix | Local variables |`

Clang's Frame `| Frame Prefix | Promise | ? | Local variables |`

The placement of Promise Type and Frame Prefix was
the reason of crash for **clang-cl compiler & VC++ header** combination.

`?` part includes index (When it's modified, `resume()` leads to crash.)

In the meantime,
how can we acquire the `coroutine_handle<void>` object?

Way to acquire `coroutine_handle<void>`

- Promise Type

  From invoke operation(mostly, promise_type&)

- `void*`

  Simple conversion

- Awaitable Type

  From suspend operation(await_suspend)

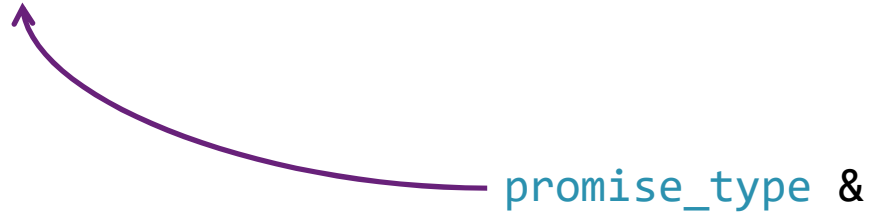
```
promise_type &_prom;
```

When we know a promise …

```
promise_type &_prom;

auto coro = coroutine_handle<promise_type>::from_promise(_prom);
```

Coroutine Handle from Promise

```
promise_type &_prom;

auto coro = coroutine_handle<promise_type>::from_promise(_prom);

auto& promise = coro.promise();
```



Promise from Coroutine Handle

```
void *ptr;
```

When we have a pointer …

```
void *ptr;

auto coro = coroutine_handle<void>::from_address(ptr);
```

Coroutine Handle from `void*`

```
void *ptr;

auto coro = coroutine_handle<void>::from_address(ptr);

auto *addr = coro.address();
```

`void*` from Coroutine Handle

```
struct suspend_never
{
    bool await_ready() {  return true;  }
    void await_suspend(coroutine_handle<void>){}
    void await_resume(){}
};
```

Coroutine Handle from Awaitable

`coroutine_handle<P>`'s Role

Indirect(safe) use of compiler intrinsic (Type System)

- `done, resume, destroy`

Coroutine frame destruction

- `destroy`

Address calculation for the compiler specific frame layout

- Address of the coroutine frame
- Address calculation between promise object and frame prefix

Summary of the C++ Coroutine components

Awaitable, Promise, and Handle

Awaitable

Operand for the `co_await`

- `await_ready`
- `await_suspend`, `await_resume`

Control of suspension (== programmer's Intent)

Promise

Coroutine Code Generation

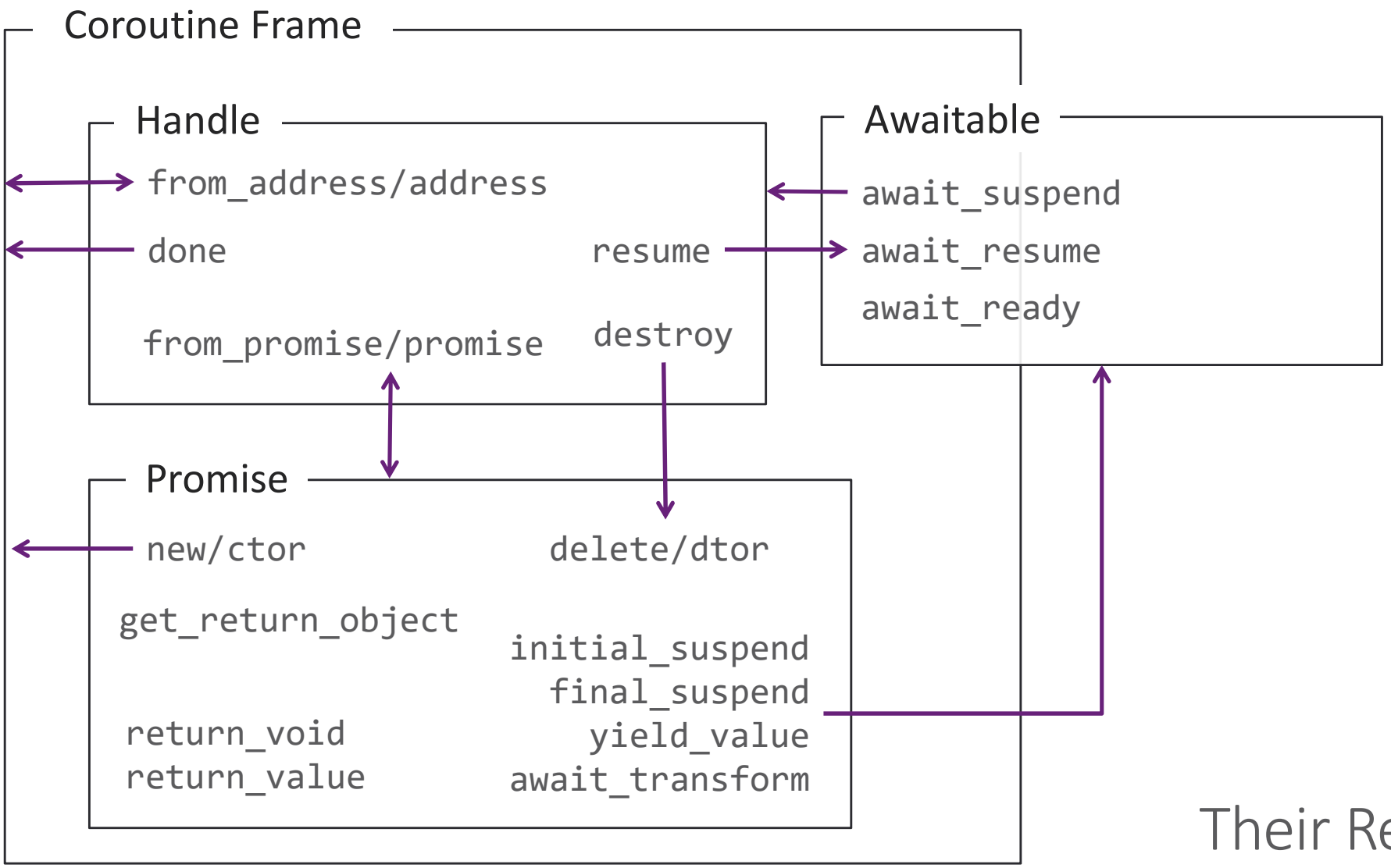
- Frame Lifecycle
  - Alloc/Dealloc
  - Initial / Final suspend
- Return/Exception Handling

Handle

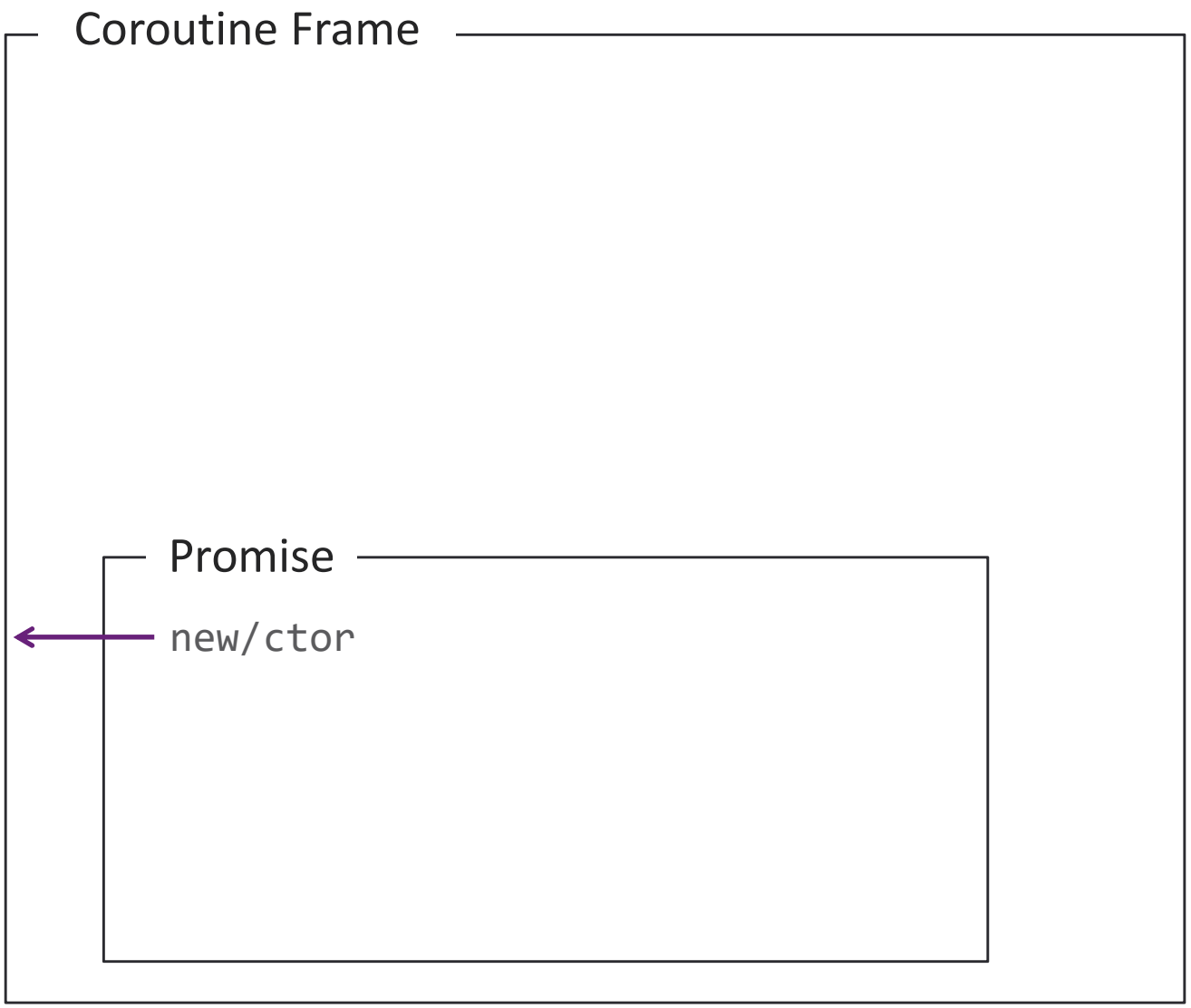
Interface for compiler generated struct and intrinsic

- Suspend
- Resume
- Destroy

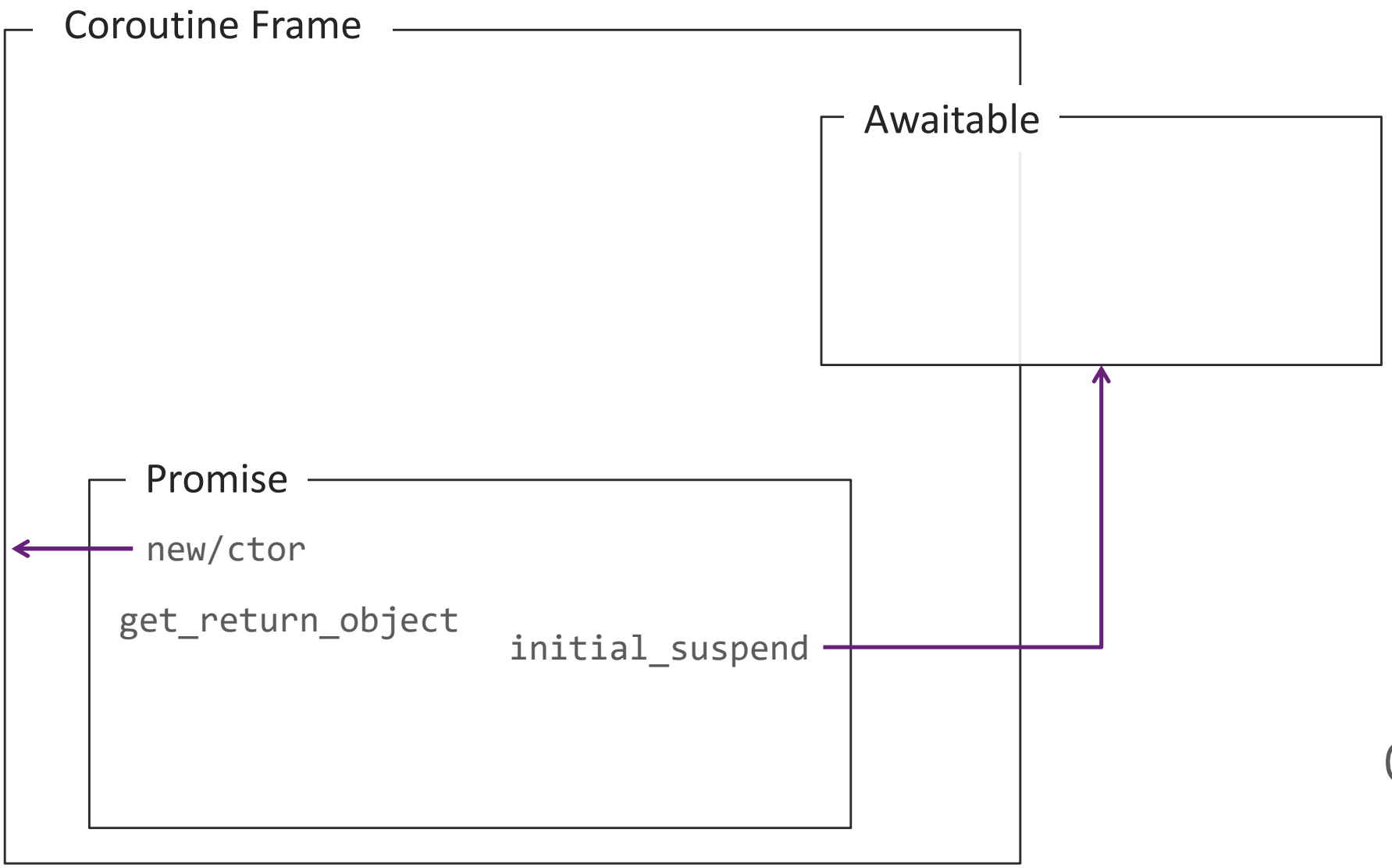
Their Role



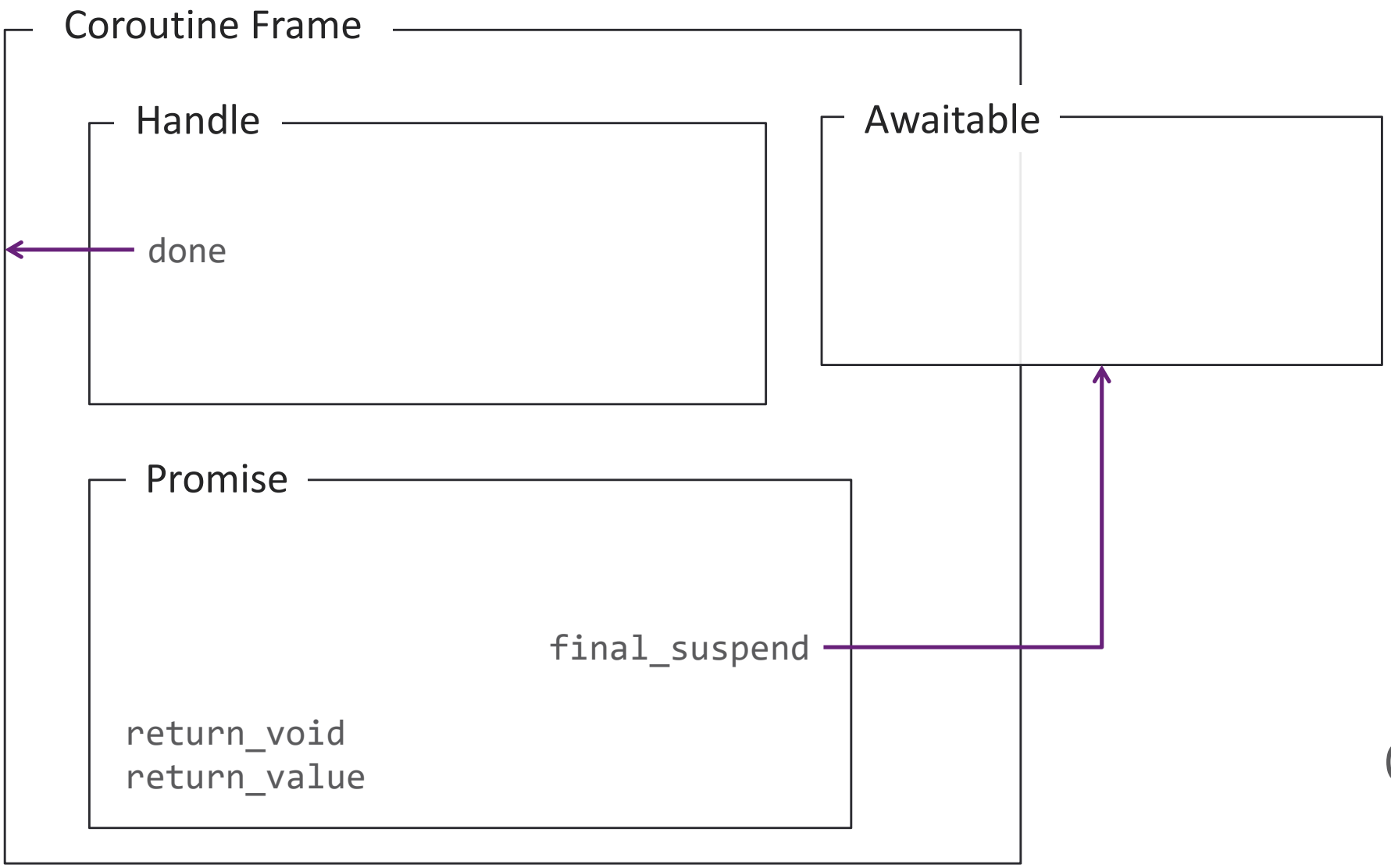
Their Relationship



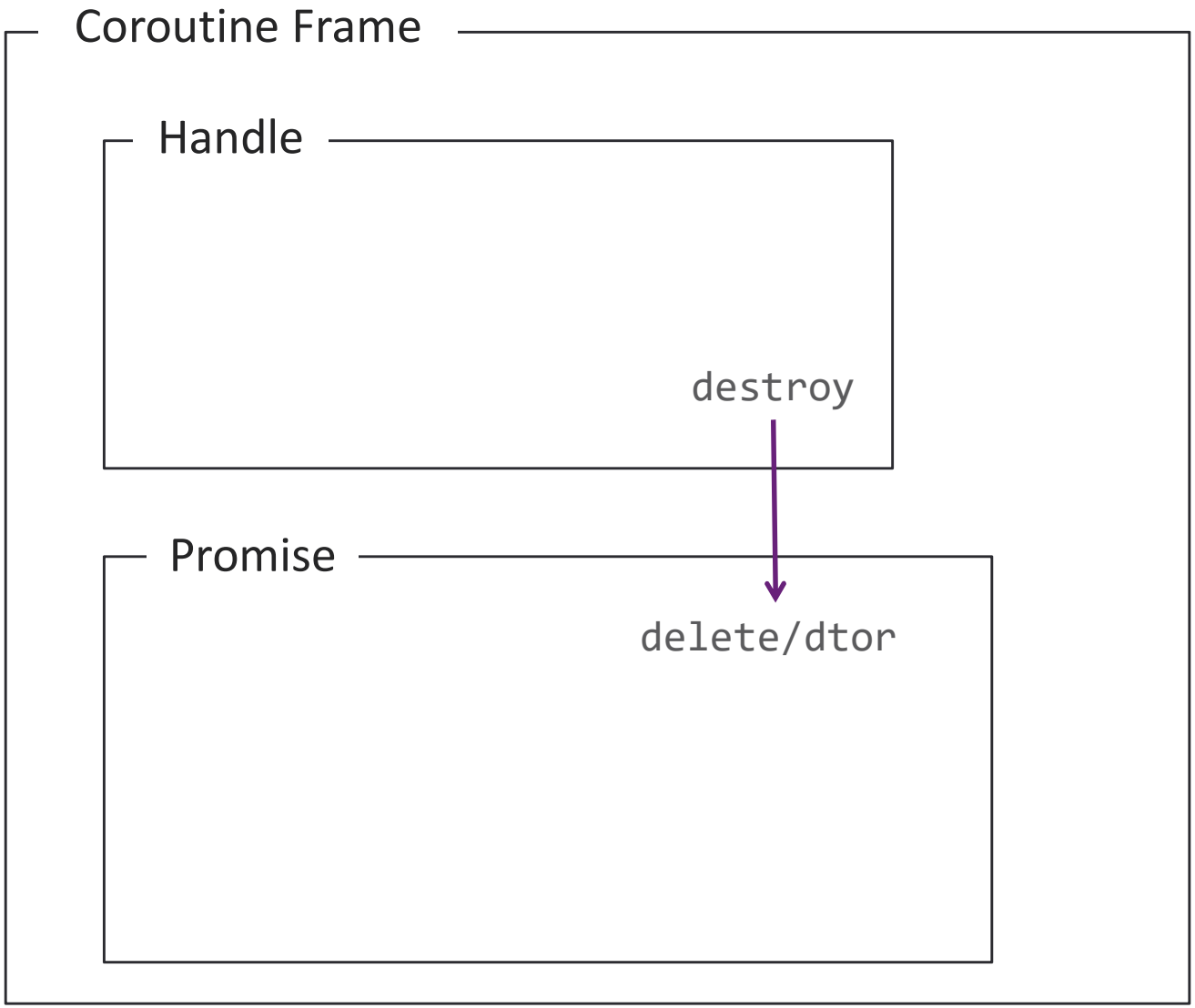
Frame Life Cycle: Create



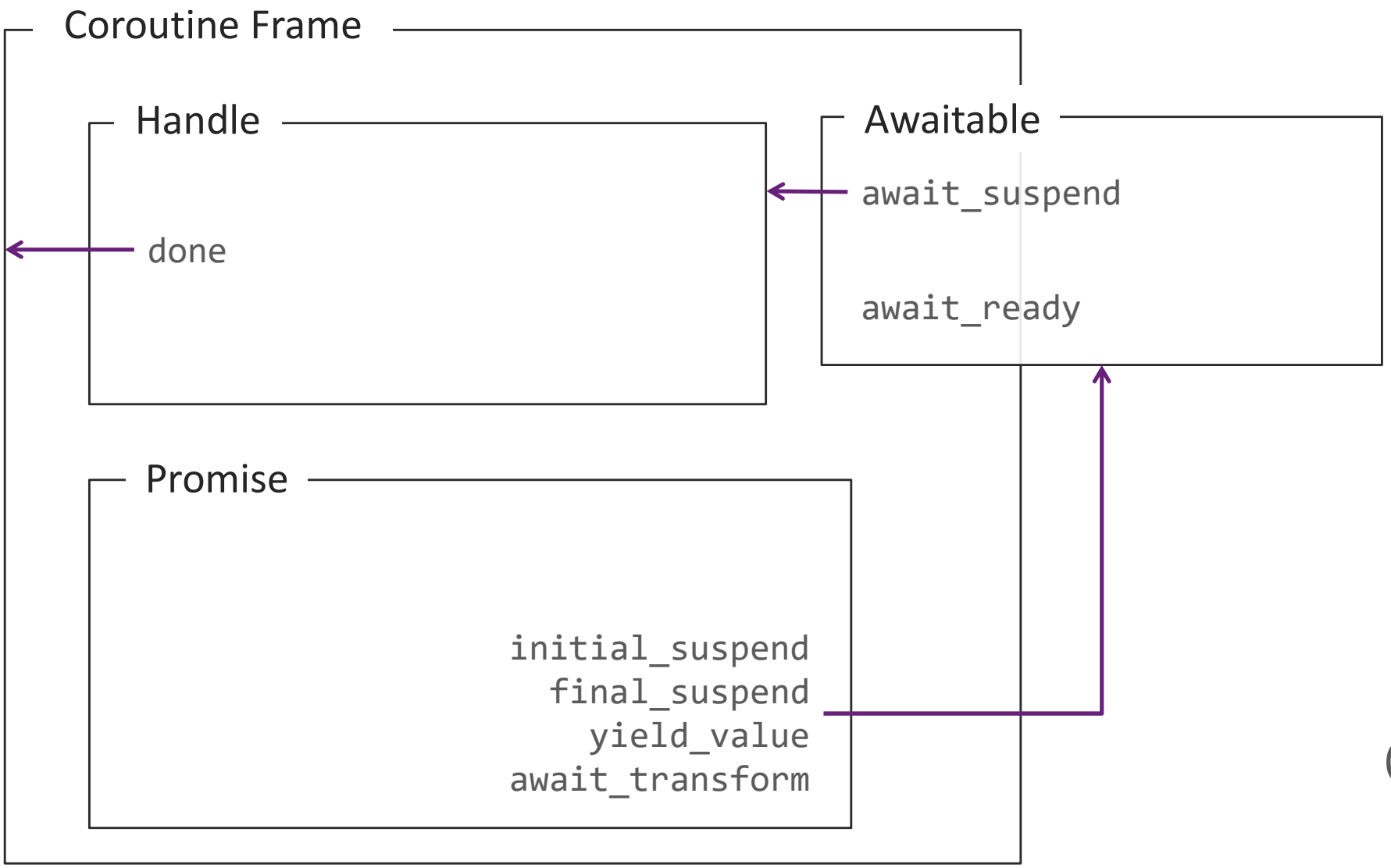
Operation: Invoke



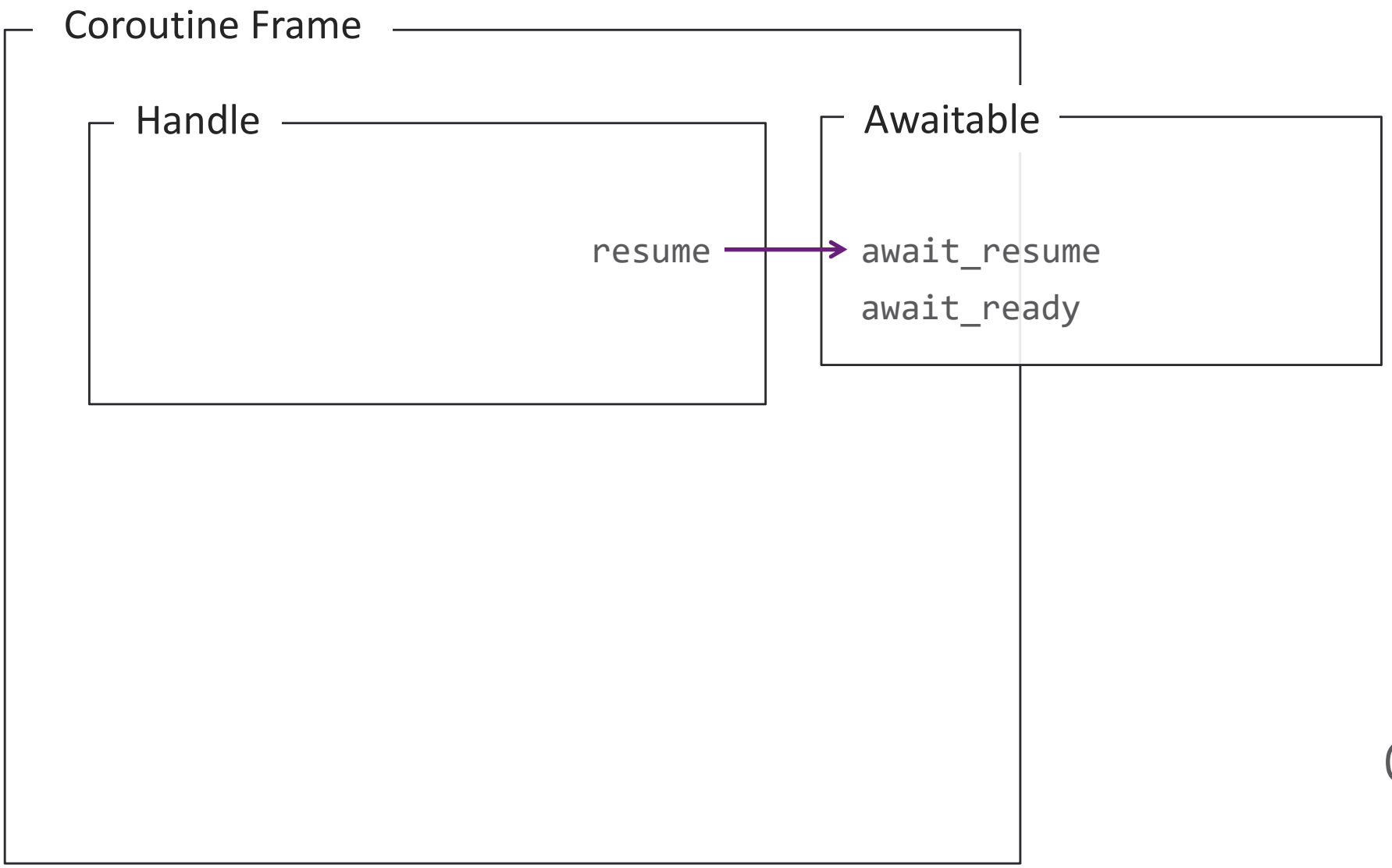
Operation: Return



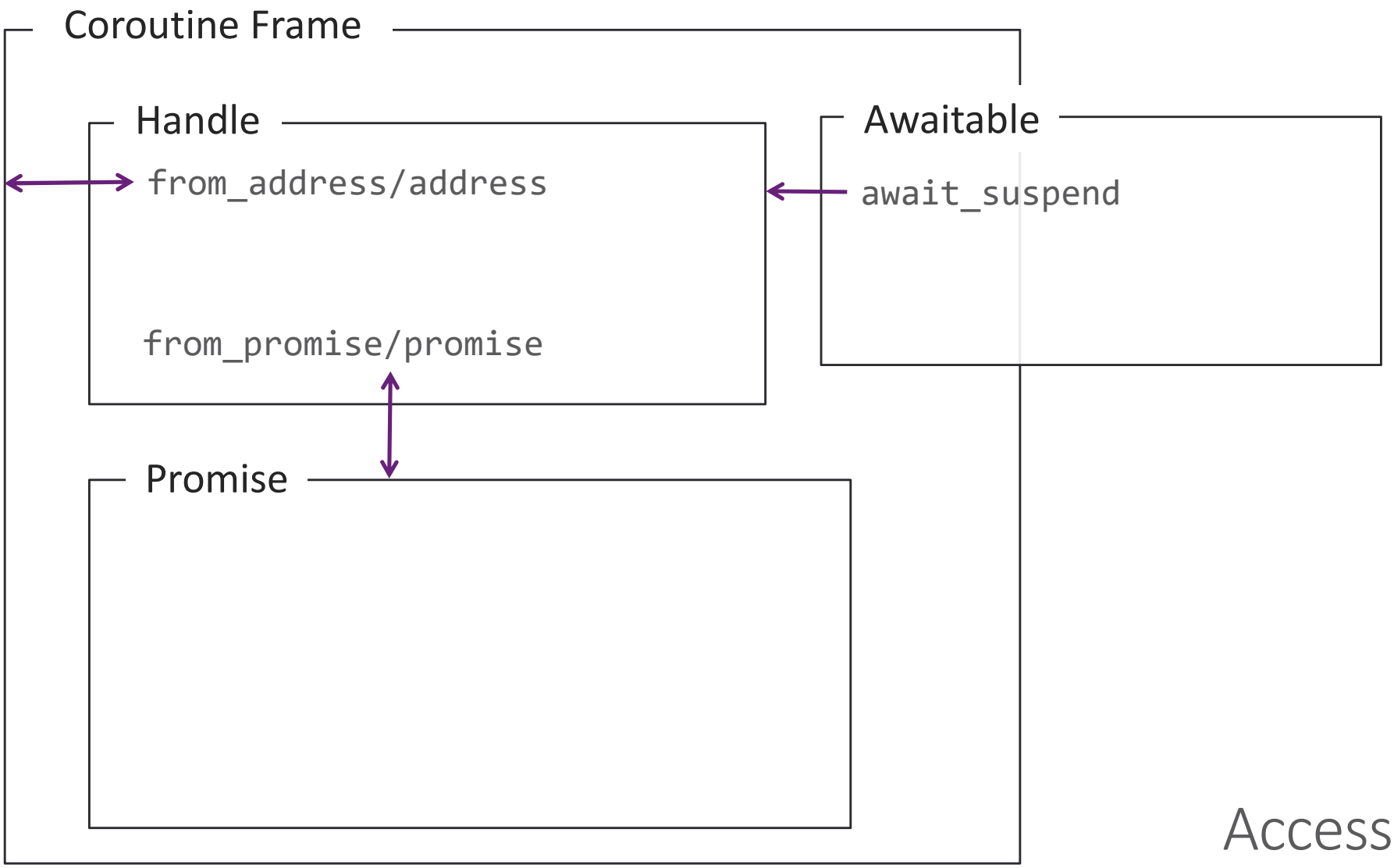
Frame Life Cycle: Destroy



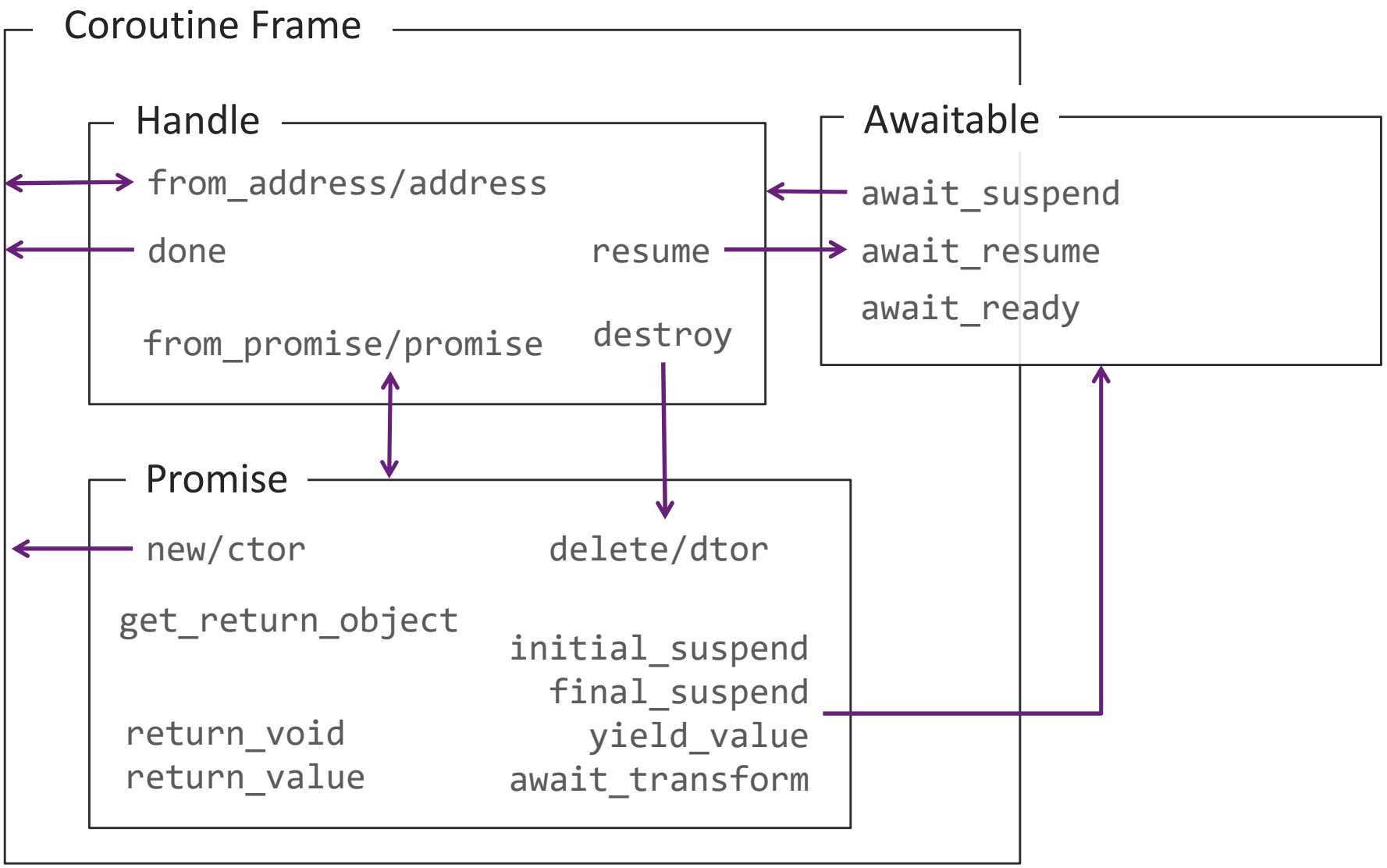
Operation: Suspend



Operation: Resume



Access to Handle


Coroutine Frame
Handle
from_address/address
done
resume
from_promise/promise
destroy
Awaitable
await_suspend
await_resume
await_ready
Promise
new/ctor
delete/dtor
get_return_object
initial_suspend
final_suspend
yield_value
return_void
return_value
await_transform

All in One

Thank You!

For question & error, please send mail to luncliff@gmail.com

Coroutine Generator

Understanding `co_yield`

`co_yield` Operator

Similar to `co_return`, but the name implies **suspension** rather than **return**

```
auto subroutine(uint32_t sum = 0) -> uint32_t
{
    for (uint32_t v : example())
        sum += v;

    return sum;
}

auto example() -> generator<uint32_t>
{

    uint32_t item{};

    co_yield item = 1;
}
```

2018/n4736.pdf

When there is a `co_yield` expression…

```
auto subroutine(uint32_t sum = 0) -> uint32_t
{
    for (uint32_t v : example())
        sum += v;

    return sum;
}

auto example() -> generator<uint32_t>
{
    promise_type p{};
    uint32_t item{};

    co_await p.yield_value(item = 1);
}
```

Programmer's code is forwarded to `promise_type::yield_value`

2018/n4736.pdf

Semantics of the `co_yield`

```
auto subroutine(uint32_t sum = 0) -> uint32_t
{
    for (uint32_t v : example())
        sum += v;

    return sum;
}

auto example() -> generator<uint32_t>
{
    promise_type p{};
    uint32_t item{};

    p.yield_value(item);
    co_await suspend_always{}; // this is not return!
}
```

Also, the generated code can be separated.

<experimental/generator> in VC++

```
auto subroutine(uint32_t sum = 0) -> uint32_t
{
    for (uint32_t v : example())
        sum += v;

    return sum;
}
```

Generator: Usage

```
auto subroutine(uint32_t sum = 0) -> uint32_t
{
    {
        auto g = example();
        auto it = g.begin();
        auto e = g.end();
        for (; it != e; ++it)
        {
            auto v = *it;
            sum += v;
        }
    }
    // g is destroyed
    return sum;
}
```

Just that of input iterator
(directional iteration)

Semantics of the generator

```
template <typename _Ty, typename _Alloc = allocator<char>>
struct generator
{
    struct promise_type;
    struct iterator;

    _NODISCARD iterator begin();
    _NODISCARD iterator end();

    explicit generator(promise_type &_Prom);
    ~generator();

    generator(generator const &) = delete;
    generator &operator=(generator const &) = delete;
    generator(generator &&_Right);
    generator &operator=(generator &&_Right);
  private:
    coroutine_handle<promise_type> _Coro = nullptr;
};
```

Not copyable, but movable

<experimental/generator> in VC++

Generator: Overview

```
template <typename _Ty, typename _Alloc = allocator<char>>
struct generator
{
    struct promise_type;

    explicit generator(promise_type &_Prom)
        : _Coro(coroutine_handle<promise_type>::from_promise(_Prom))
    {}

    ~generator(){
        if (_Coro)
            _Coro.destroy();
    }

  private:
    coroutine_handle<promise_type> _Coro = nullptr;
};
```

Delete frame with destructor

<experimental/generator> in VC++

Generator: Ctor/Dtor

```
template <typename _Ty, typename _Alloc = allocator<char>>
struct generator
{
    struct iterator {
        using iterator_category = input_iterator_tag;
        using difference_type = ptrdiff_t;
        using value_type = _Ty;
        using reference = _Ty const &;
        using pointer = _Ty const *;

        coroutine_handle<promise_type> _Coro = nullptr;

        iterator() = default;
        iterator(nullptr_t) : _Coro(nullptr){}
        iterator(coroutine_handle<promise_type> _CoroArg) : _Coro(_CoroArg){}
    };

    _NODISCARD iterator begin();
    _NODISCARD iterator end();


};
```

iterator tag

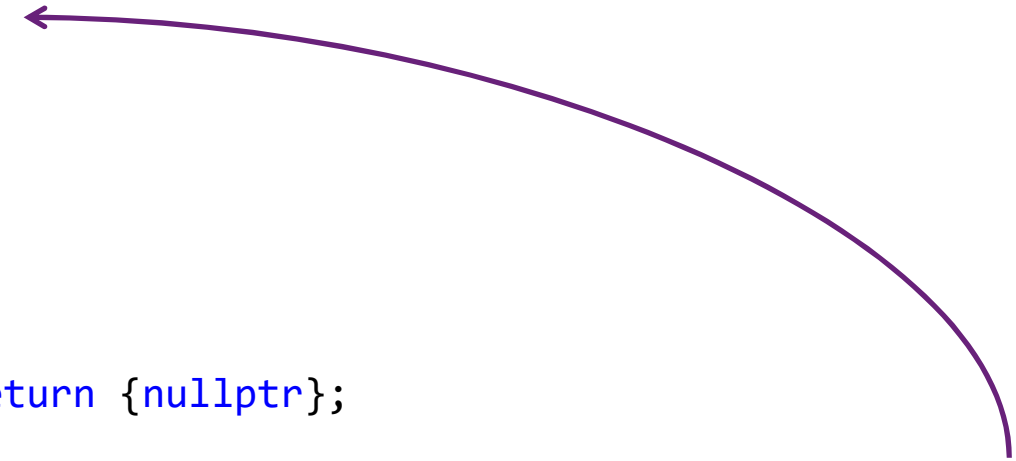
Basically, this is a pointer.
So the object is really light

Generator: Iterator

```
template <typename _Ty, typename _Alloc = allocator<char>>
struct generator
{
    struct iterator {
        using iterator_category = input_iterator_tag;

        coroutine_handle<promise_type> _Coro = nullptr;

        _NODISCARD bool operator==(iterator const &_Right) const{
            return _Coro == _Right._Coro;
        }
        _NODISCARD bool operator!=(iterator const &_Right) const;

        _NODISCARD reference operator*() const{
            return *_Coro.promise()._CurrentValue;
        }
        _NODISCARD pointer operator->() const{
            return _Coro.promise()._CurrentValue;
        }
    };
    // ...
};
```

Access to value through the promise object

```
template <typename _Ty, typename _Alloc = allocator<char>>
struct generator
{
    struct iterator {
        coroutine_handle<promise_type> _Coro = nullptr;

        iterator &operator++(){
            _Coro.resume();
            if (_Coro.done())
                _Coro = nullptr;
            return *this;
        }
    };
    _NODISCARD iterator begin(){
        if (_Coro) {
            _Coro.resume();
            if (_Coro.done())  return {nullptr};
        }
        return {_Coro};
    }
    _NODISCARD iterator end(){ return {nullptr}; }
};
```



Advance ==
Resume Operation

```
template <typename _Ty, typename _Alloc = allocator<char>>
struct generator
{
    struct promise_type {
        _Ty const *_CurrentValue;

        promise_type &get_return_object(){
            return *this;
        }
        bool initial_suspend(){ return (true); }
        bool final_suspend(){   return (true); }
        void yield_value(_Ty const &_Value){
            _CurrentValue = _STD addressof(_Value);
        }
    };

    explicit generator(promise_type &_Prom)
        : _Coro(coroutine_handle<promise_type>::from_promise(_Prom))
    {}
  private:
    coroutine_handle<promise_type> _Coro = nullptr;
};
```

Just address calculation

Generator: Promise

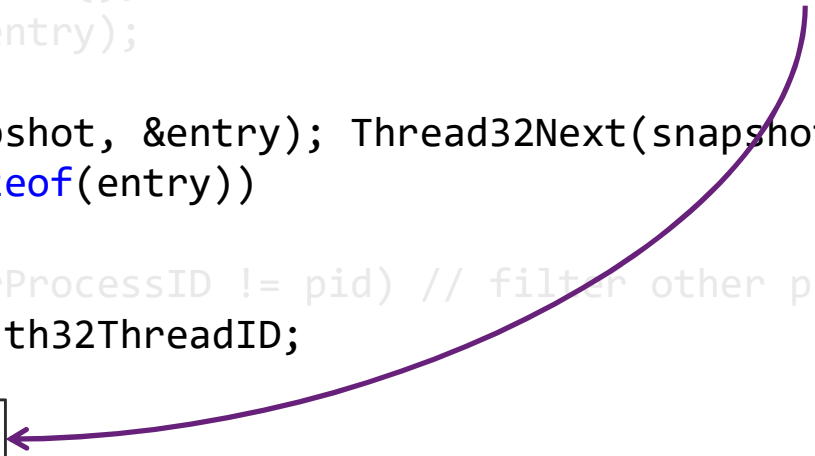
Is this type really safe?

```
auto current_threads() -> generator<DWORD>
{
    auto pid = GetCurrentProcessId();

    auto snapshot = CreateToolhelp32Snapshot(TH32CS_SNAPTHREAD, 0);
    if (snapshot == INVALID_HANDLE_VALUE)
        throw system_error{GetLastError(),system_category()};

    auto entry = THREADENTRY32{};
    entry.dwSize = sizeof(entry);

    for (Thread32First(snapshot, &entry); Thread32Next(snapshot, &entry);
         entry.dwSize = sizeof(entry))

        if (entry.th32OwnerProcessID != pid) // filter other process threads
            co_yield entry.th32ThreadID;

    CloseHandle(snapshot);
}
```

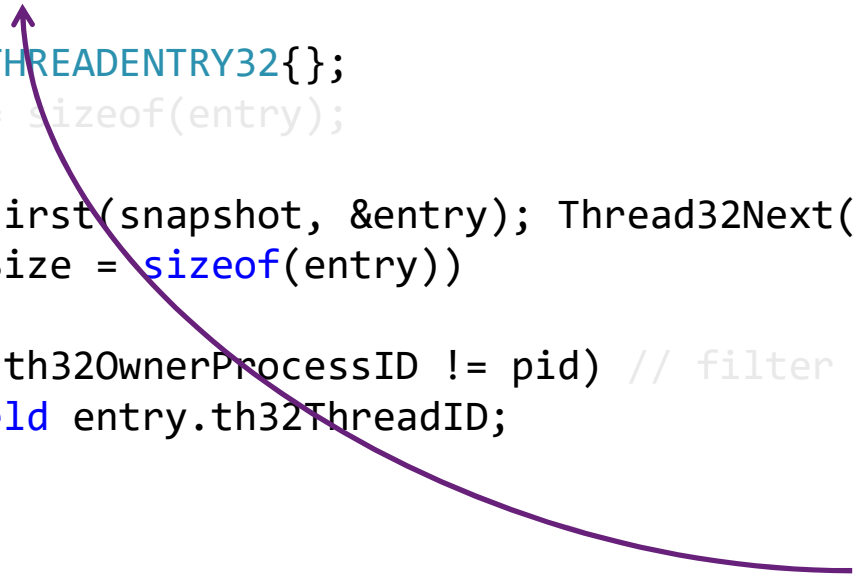
Problem of the code?

```
auto current_threads() -> generator<DWORD>
{
    auto pid = GetCurrentProcessId();

    auto snapshot = CreateToolhelp32Snapshot(TH32CS_SNAPTHREAD, 0);
    if (snapshot == INVALID_HANDLE_VALUE)
        throw system_error{

    auto entry = THREADENTRY32{};
    entry.dwSize = sizeof(entry);

    for (Thread32First(snapshot, &entry); Thread32Next(snapshot, &entry);
         entry.dwSize = sizeof(entry))

        if (entry.th32OwnerProcessID != pid) // filter other process threads
            co_yield entry.th32ThreadID;

    CloseHandle(snapshot);
}
```

If caller break the iterator loop, the line will be skipped
(+ this frame will be deleted)



```
auto current_threads() -> generator<DWORD>
{
    auto pid = GetCurrentProcessId();

    auto snapshot = CreateToolhelp32Snapshot(TH32CS_SNAPTHREAD, 0);
    if (snapshot == INVALID_HANDLE_VALUE)
        throw system_error{GetLastError(),system_category()};

    auto h = gsl::finally([=]() noexcept { CloseHandle(snapshot); });

    auto entry = THREADENTRY32{};
    entry.dwSize = sizeof(entry);

    for (Thread32First(snapshot, &entry); Thread32Next(snapshot, &entry);
         entry.dwSize = sizeof(entry))

        if (entry.th32OwnerProcessID != pid) // filter other process threads
            co_yield entry.th32ThreadID;
}
```



Leak Prevention with RAII

Switching Thread

Coroutine + Message Queue

```
struct coro_queue
{
    virtual ~coro_queue() noexcept = default;
    virtual void push(coroutine_handle<void> rh) = 0;
    virtual bool try_pop(coroutine_handle<void>& rh) = 0;
};

auto make_queue() -> std::unique_ptr<coro_queue>;
```

https://wandbox.org/permlink/6FGKZjuzjNYoSmI1
https://godbolt.org/z/M4atrm

Let’s assume there is a queue…

```
auto program(coro_queue& fq, coro_queue& bq) -> return_ignore;

void coro_worker(coro_queue* q); // worker thread function

void main_subroutine()
{
    auto fg = make_queue(); // for foreground
    auto bg = make_queue(); // for background

    // launch background worker
    auto fb = std::async(std::launch::async,
                         coro_worker, bg.get());

    program(*fg, *bg);     // start the program
    coro_worker(fg.get()); // run as foreground worker
    fb.get();              // clean-up or join background thread
}
```

Main subroutine with 2 queue

```
auto program(coro_queue& foreground, //
             coro_queue& background) -> return_ignore
{
    using namespace std;
    print_thread_id("invoke");

    auto repeat = 3;
    while (repeat--)
    {
        co_await foreground;
        print_thread_id("front");

        co_await background;
        print_thread_id("back");
    }
    print_thread_id("return");
    co_return;
}
```

```
void print_thread_id(const char* label)
{
    cout << label
      << "\t" << this_thread::get_id()
      << endl;
}
```

Our coroutine

```
auto program(coro_queue& foreground, //
             coro_queue& background) -> return_ignore
{
    using namespace std;
    print_thread_id("invoke");

    auto repeat = 3;
    while (repeat--)
    {
        co_await foreground;
        print_thread_id("front");

        co_await background;
        print_thread_id("back");
    }
    print_thread_id("return");
    co_return;
}
```

Expression:
Function selects its thread

Semantics:
Send a handle through Message Queue

```
auto program(coro_queue& fq, coro_queue& bq) -> return_ignore;

void coro_worker(coro_queue* q); // worker thread function
{
    auto coro = coroutine_handle<void>{};
    auto repeat = 10;
PopNext:
    if (q->try_pop(coro) == false)
        std::this_thread::sleep_for(10ms);
    else
    {
        if (coro.done())
            coro.destroy();
        else
            coro.resume();
    }
    if (repeat--) // for some condition ...
        goto PopNext; // continue
}
```

The worker thread function

```
auto program(coro_queue& fq, coro_queue& bq) -> return_ignore;

void coro_worker(coro_queue* q)
{
    auto coro = coroutine_handle<void>{};
    auto repeat = 10;
PopNext:
    if (q->try_pop(coro) == false)
        std::this_thread::sleep_for(10ms);
    else
    {
        if (coro.done())
            coro.destroy();
        else
            coro.resume();
    }
    if (repeat--) // for some condition ...
        goto PopNext; // continue
}
```

Pop & Resume/Destroy

await_transform

Providing type conversion for the `co_await`

```
struct return_ignore; // ... we already covered this type ...

auto example() -> return_ignore {
  co_await true;
  co_await false;
}
```

https://godbolt.org/z/EnNBrL
https://godbolt.org/z/eCVc6I

Can we use `bool` for `co_await` ?

```
struct return_ignore; // ... we already covered this type ...

auto example() -> return_ignore {
  co_await true;
  co_await false;
}
```

E2660: this co_await expression requires a suitable "await_ready" function and none was found

```
struct return_ignore;

auto example() -> return_ignore {
  co_await true;
  co_await false;
}
class suspend_with_condition {
  bool cond;
public:
  suspend_with_condition(bool _cond) : cond{_cond} {}

  bool await_ready() { return cond; }
  void await_suspend(coroutine_handle<void>) { /* ... */  }
  void await_resume() { /* ... */  }
};
```

Simple awaitable type.
The code is from `suspend_if` in VC++

https://godbolt.org/z/EnNBrL
https://godbolt.org/z/eCVc6I

```
struct return_ignore;

auto example() -> return_ignore {
  co_await true;
  co_await false;
}

class suspend_with_condition;

struct return_ignore {
  struct promise_type {
    // ...
    auto await_transform(bool cond) {
       // return an awaitable
       // that is came from its argument
       return suspend_with_condition{cond};
    }
  };
  // ...
};
```

If there is `await_transform`,
it is applied before `co_await` operator

```
struct return_ignore;

auto example() -> return_ignore {
  co_await true;
  co_await false;
}

class suspend_with_condition;

struct return_ignore {
  struct promise_type {
    // ...
    auto await_transform(bool cond) {
      // return an awaitable
      // that is came from its argument
      return suspend_with_condition{cond};
    }
  };
  // ...
};
```

```
auto example() -> return_ignore {
  promise_type *p;

  auto aw = p->await_transform(true);
  co_await aw;
}
```