C++ Coroutine 알아보기

접근법, 컴파일러, 그리고 이슈들

박 동하

C++ Korea Facebook Group

Alchera Inc. 전문연구원

github.com/luncliff

참고자료: **제안서**

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

N4800 이후의 문서는

발표자의 시간적 한계로 인해 생략되었습니다 ☹

참고자료: **영상 (많다!)**

- 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”

참고자료: **코드**

-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

참고자료: **나머지**

-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

-…

지금 보시는 자료는
이 글의 **두 번째** 버전입니다.

오늘 다룰 것들

Coroutine을 처음 접하는 분들을 위한 토막지식

C++ Coroutine의 구성요소Component들

- Operators & Awaitable Type
- Promise
- Coroutine Handle

MSVC와 Clang 컴파일러의 차이

(시간이 남는다면) 몇가지 예시들

CppCon에서 다뤄진 것들

(아마도) 이번이 처음…

먼저, 용어 정리부터…

이번 발표를 위한 전방 선언(Forward Declaration)

함수: 순서대로 배치된 구문statement들

Function

```
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/

루틴 == 명령 []

```
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
```

루틴Routine:

-명령들의 (순서있는) 집합

명령Instruction:

- 기계의 동작behavior을 추상화 한 것
- 기계 상태가 전이Transition

호출 Invocation

루틴의 **시작점**으로 Jump

활성화 Activation

루틴 **안의 임의 지점**으로 Jump

중단 Suspension

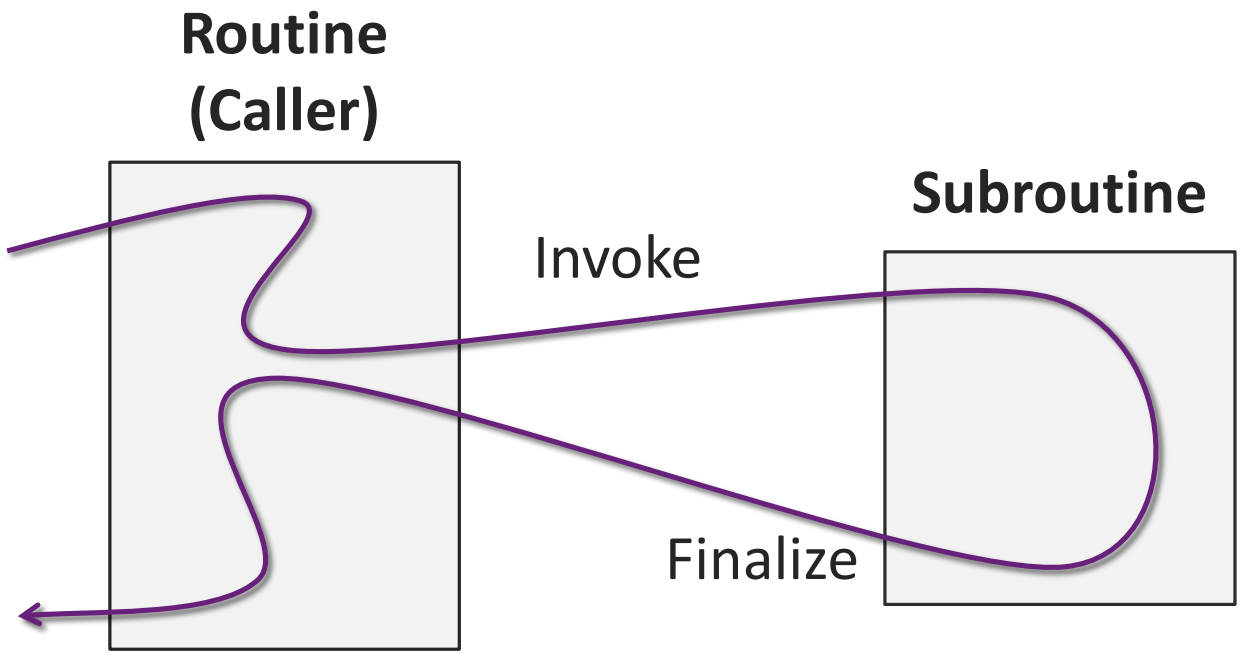
종결하지 않고 다른 루틴의 지점으로 Jump

종결 Finalization

루틴의 **끝**에 도달 한 후 루틴 상태의 소멸 및 정리

서브루틴Subroutine

호출/종결할 수 있는 루틴



서브루틴Subroutine

호출/종결할 수 있는 루틴

```
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
```

Finalize (Return)

Invoke (Call)

프로세스Process

OS (혹은 VM) 에서 프로그램을 실행하는 방법

루틴들의 집합체

스레드Thread

프로세스 내에서의 제어 흐름을 추상화한 것

프로세서 (CPU)

코루틴Coroutine

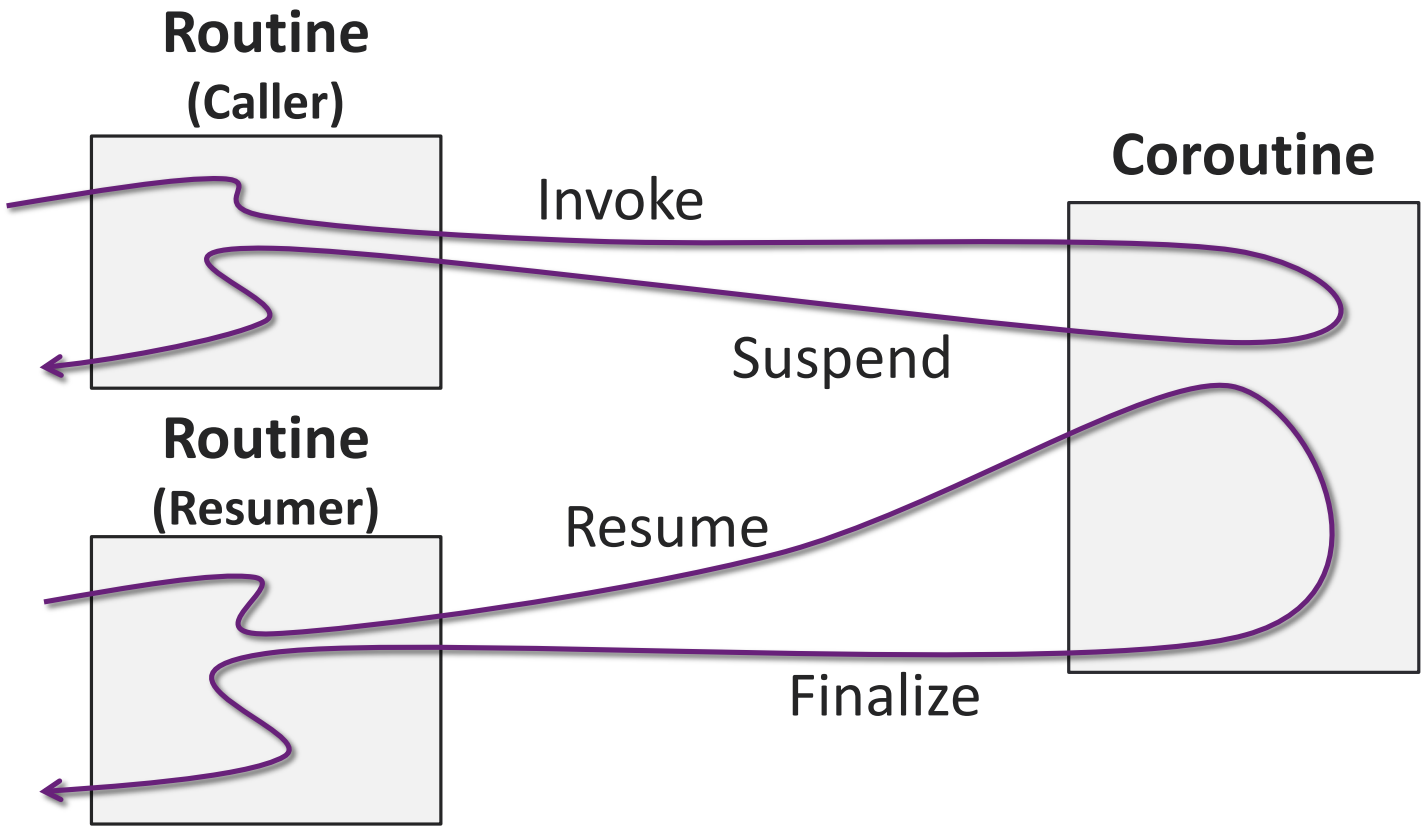
"Subroutines are special case of … coroutines"– Donald Knuth

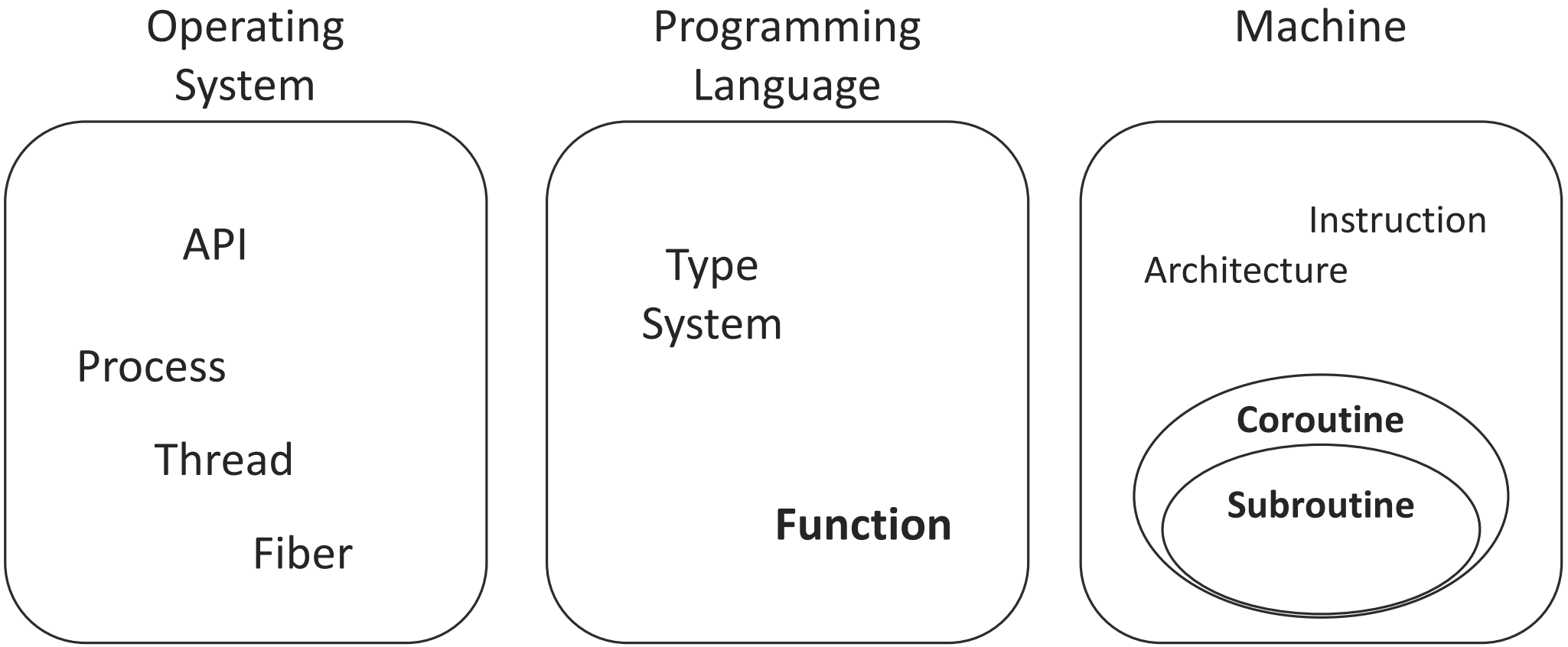
연산Operation	**서브루틴**	**코루틴**	
호출Invoke	O	O	Goto start of a procedure(**call**)
종결Finalize	O	O	Cleanup and **return**
중단Suspend	X	O	Yield current control flow
재개Resume	X	O	Goto the suspended point in the procedure

이미 코루틴들은 사용되고 있었다!

코루틴Coroutine

호출/종결/**중단/재개** 할 수 있는 루틴




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

Operating System

Programming Language

Machine

API

Type System

Instruction Architecture

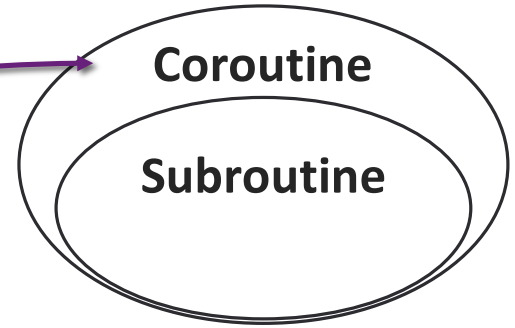
Process

Thread

Fiber

개념적으로,
코루틴은 스레드와 **무관하다.**

Function

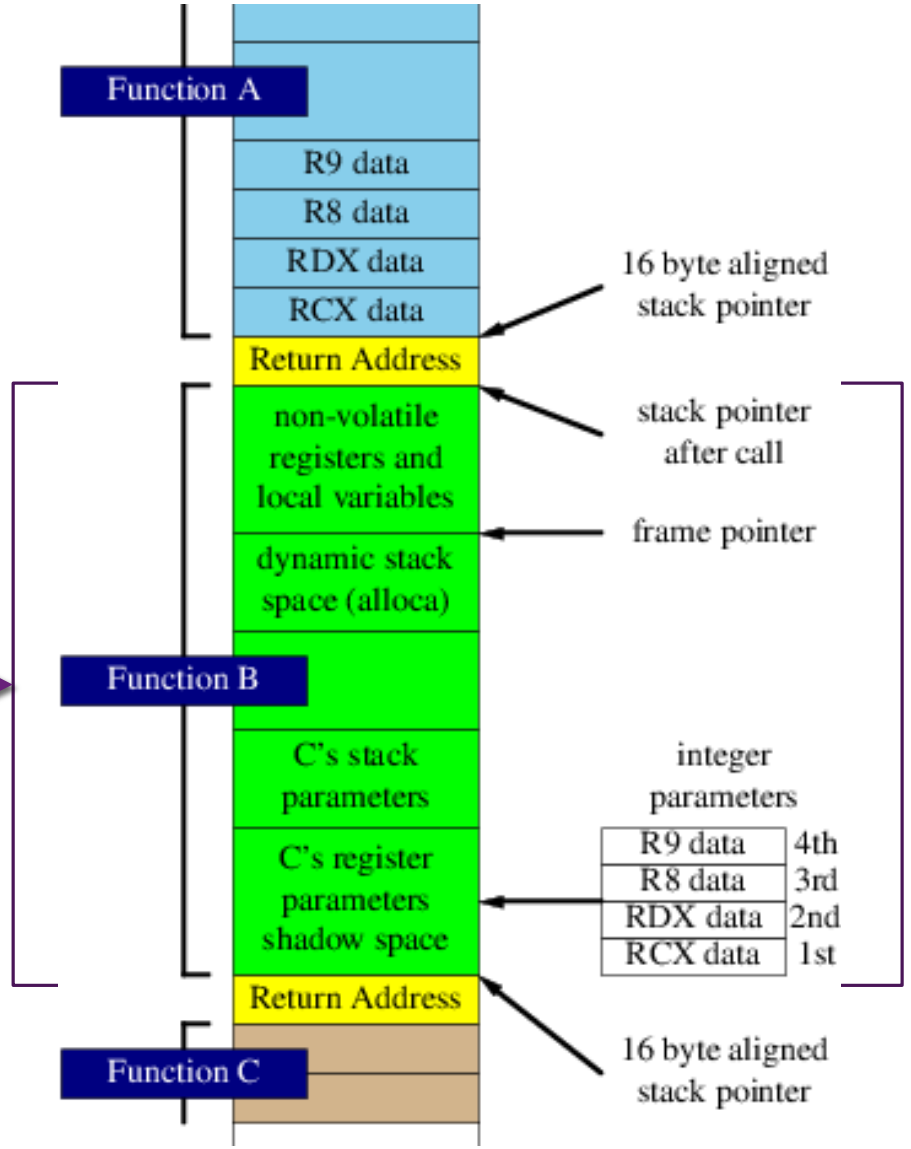


루틴이 상태를 가진다?

상태State == 메모리Memory

함수 프레임Function Frame

루틴의 상태를 저장한 메모리 개체memory object

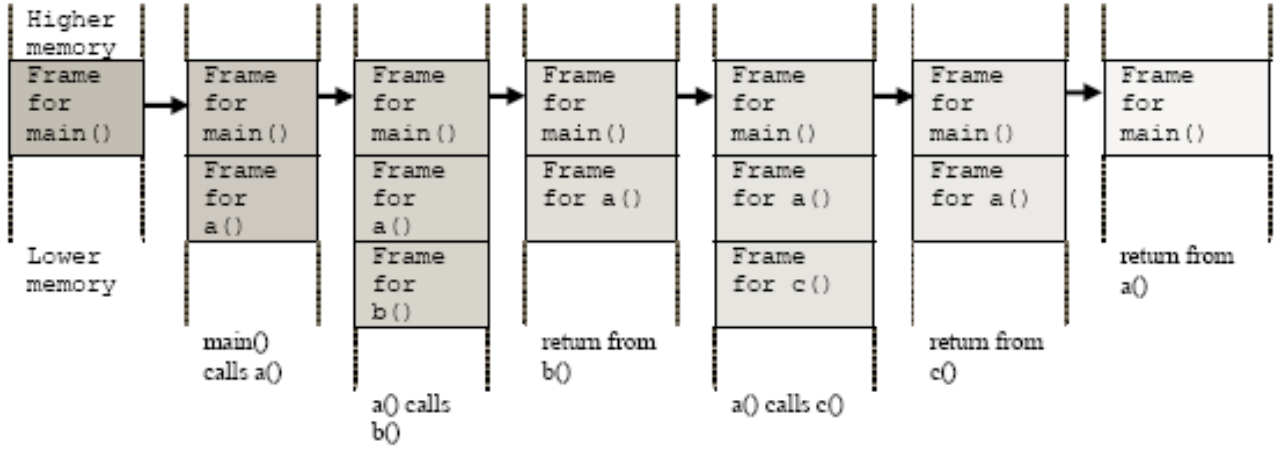


http://www.tortall.net/projects/yasm/manual/ru/html/objfmt-win64-exception.html

호출 스택Call Stack

함수 프레임을 관리하는 방법 중 하나.

- 호출 == 프레임 **Push**
- 반환 == 프레임 **Pop**



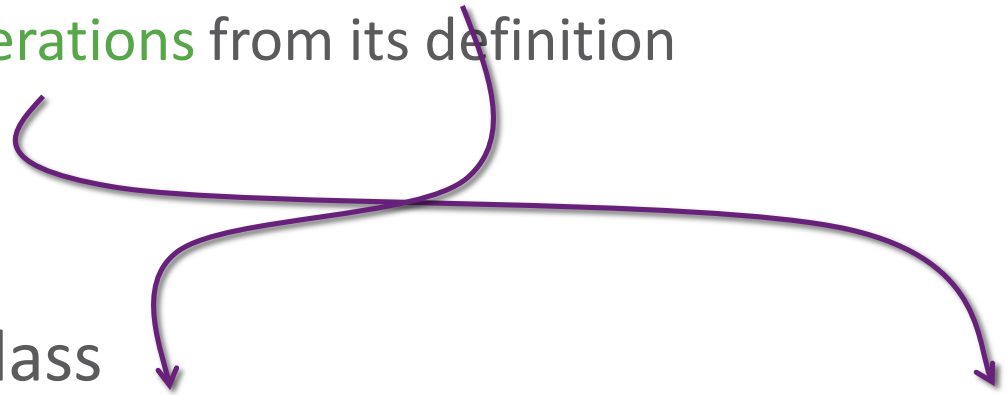
서브루틴에 매우 적합

C 언어의 모든 함수는 서브루틴!

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

실제적으로 다른 점이 없다...

호출 스택에서 코루틴을?

호출/반환은 OK.

함수 프레임의
생성/**소멸**

하지만 중단/재개는 어떻게?

중단한 지점으로 돌아가기 위해선
함수 프레임이 **유지**되어야 한다!

문제: 함수 프레임의 수명주기life-cycle

Stackful & Stackless

Stackful Coroutine
- 코루틴의 프레임을 스택에 할당

Stackless Coroutine
- 코루틴의 프레임을 스택 바깥에 (동적) 할당

서브루틴을 사용하면서
코루틴을 어떻게 구현할 것인가?

코루틴을 위한 C++ 확장C++ Extension for Coroutines

이것이 C++의 접근법이다!

개념	C++ Coroutine
호출	**변화 없음**
종결	`co_return`
중단	`co_await`, `co_yield` `// 1항 연산자unary operator`
재개	`coro.resume()` `// coroutine_handle<P>::resume()`

간단히 살펴보면…

C++ Coroutine 은 어떻게 **정의** 하는가?

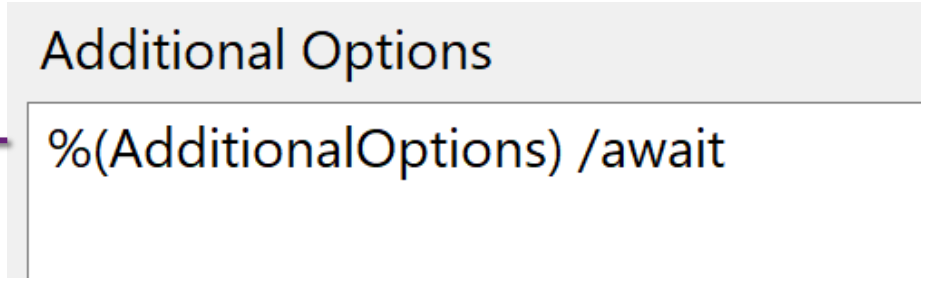
함수 안에 다음 중 하나가 존재하면, 그 함수는 코루틴으로 처리한다...

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

C++ Coroutine은 어떻게 **컴파일**하는가?

MSVC
- Visual Studio 2015 이후 버전
- /await

vcxproj 속성 > C/C++



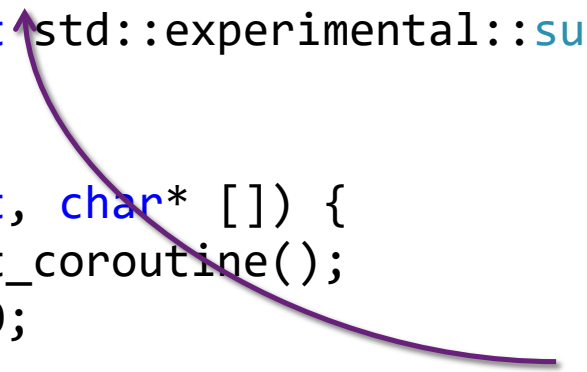
Clang Family
- 5.0 이후
- -fcoroutines-ts –stdlib=libc++ -std=c++2a

GCC
- 아직은 지원하지 않음…

C3783: ‘main’ cannot be a coroutine

```
#include <experimental/coroutine>

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

```cpp
#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

컴파일러를 위한 특별 타입(Promise Type)에 대한 요구사항

- 코루틴 코드 생성을 위한 도움 타입Helper Type
- 프레임의 할당/해제
- `coroutine_handle<P>` 로의 접근

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 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.

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.

프로그래머가 작성해야 하는 함수들

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

이 내용은 잠시 후에 다루고...

이를 통해 알 수 있는 것은...

Coroutine(stack-less) frame 을 **타입 시스템을 사용해서**(`promise_type`) 관리한다

Awaitable Type 과 `co_await` 연산자

어떻게 코루틴을 중단하는가

```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;
}
```

이 부분은 어디로?

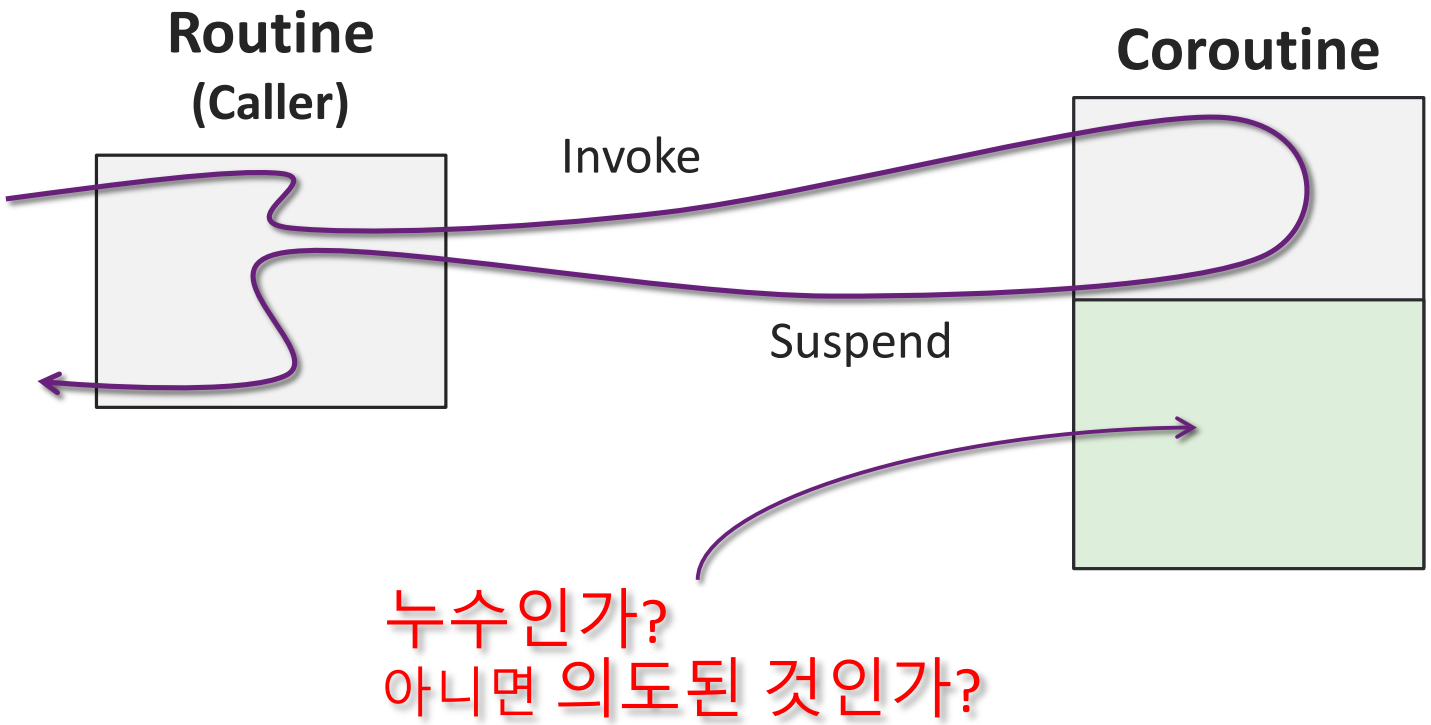
Output

```
step 1
step 3
```

실행되지 않고 넘어갔다Coverage Leak ?

코루틴이 중단되고 나면suspended, 다른 루틴이 재개resume해줘야 한다.

그렇지 않을 경우, 중단점 이후의 코드는 실행되지 않는다...



```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: 중단한 후 재개하길 기다린다

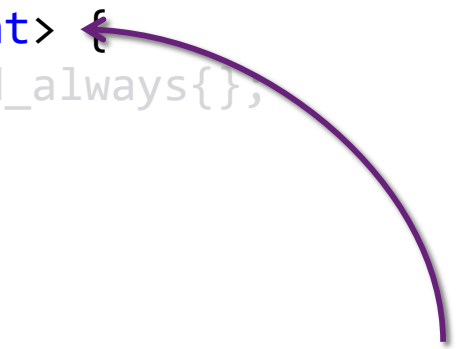
Subroutine: 코루틴이 반환하길 기다린다

(VC++ 에서) 어떤 Deadlock

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

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

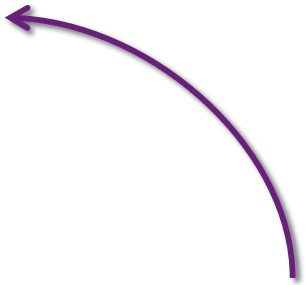
이 코드의 문제점?

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

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



Future는 반환return을 기대한다.

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

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



하지만 코루틴은 co_return 을 보장하지 않는다

인터페이스는 신중하게!!

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

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

co_await 표현식

```
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: 컴파일러가 보는 코드

```
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();
}
```

Awaitable의 멤버 함수 호출

```
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();
}
```

현재 코루틴의 프레임

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();
}
```

프레임을 전달받는 함수

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(){}
};
```

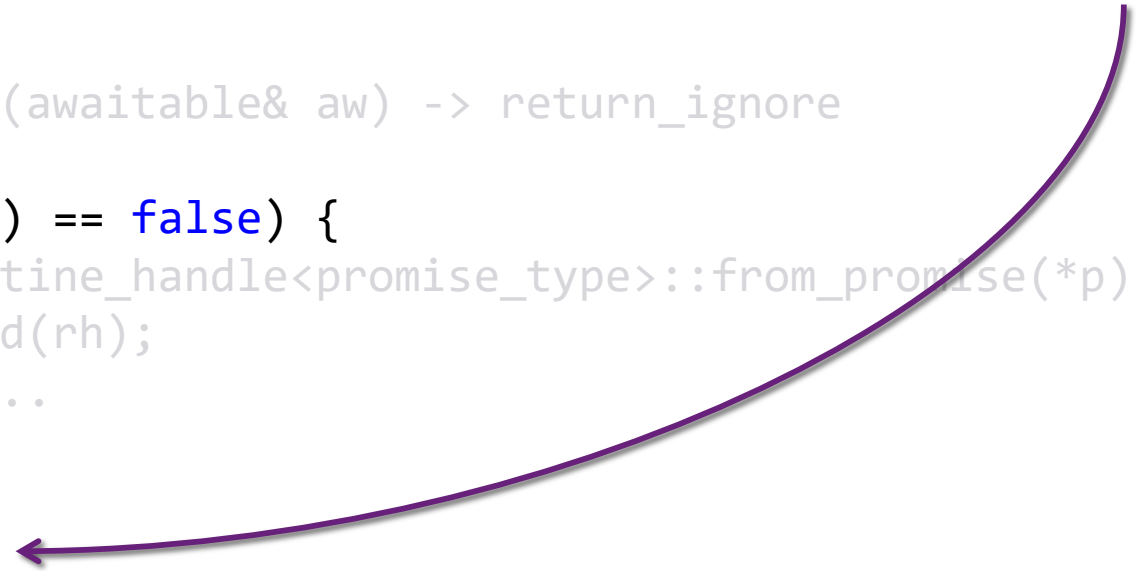
미리 정의된 Awaitable 타입들

```
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();
}
```

true인 경우, await_resume로 직행



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(){}
};

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();
}
```

`false` 인 경우, `await_suspend` 호출 후
이전 루틴으로 **제어흐름을 양도**Yield

Ready – Suspend - Resume

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

void 반환이 아니라면?

```
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>
}
```

딱히 다르지 않다...

```
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>
}
```

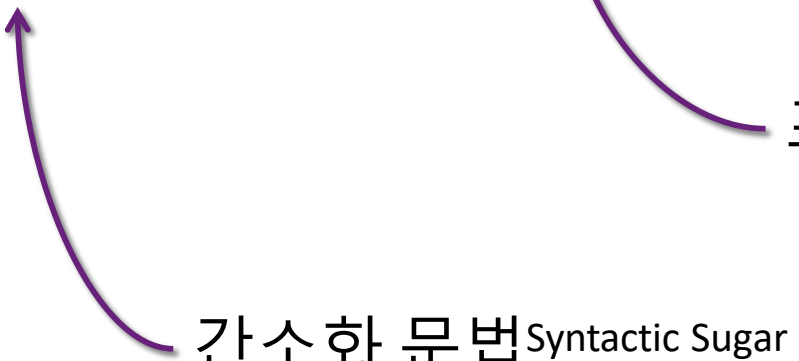
```
struct wait_for_tuple
{
    bool await_ready();
    void await_suspend(coroutine_handle<void>);
    auto await_resume() -> std::tuple<int, bool>;
};


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;
}
```

C3313: ‘v’: variable cannot have the type ‘void’

await_resume()

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

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



co_await 표현식

Awaitable Type의 역할

`co_await` 에서 사용하는 인터페이스

- `co_await` 연산자는 **함수들**을 필요로 한다
  - `await_ready`
  - `await_suspend`
  - `await_resume`

`co_await`을 사용하여...

- 컴파일러는 해당 라인line에 중단점Suspend Point을 생성한다
- 프로그래머는 조건에 맞게 코루틴의 제어 흐름을 중단할 수 있다

Coroutine Promise Requirement (N4736)

Promise Type은 무엇인가? 어떤 내용이 작성되는가?

```
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 Type의 역할

타입 시스템을 사용한 컴파일 시간compile time 검사

- `coroutine_traits<T...>`

코루틴 프레임의 생성/소멸

- Operator `new`/`delete`
- 생성자/소멸자
- `get_return_object`, `get_return_object_on_allocation_failure`

반환return 처리

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

```
#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

컴파일러가 코루틴을 만나면…

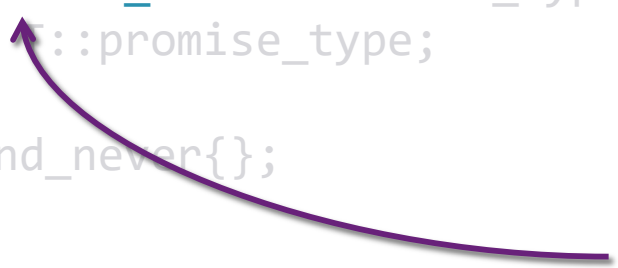
```cpp
#include <experimental/coroutine>
namespace coro = std::experimental;

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{};
}
```

coroutine_traits<T...>을 사용해 검사한다

```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{};
}
```

반환 타입 + 함수 인자 타입

```
#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{};
}
```



이 템플릿 클래스는 대체 무슨 일을?

```
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

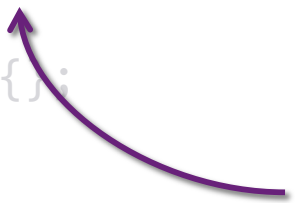
```
template <class>
struct __void_t { typedef void type; };

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>
{
};
```

SFINAE 를 무시하고...

```
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>
{
};
```

핵심만 남겼을 때

```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 이 promise_type 을 가지고 있는가?

coroutine_traits<T...>

coroutine_traits<> 의 응용

설령 return_type이 promise_type을 가지고 있지 않더라도,

프로그래머는 coroutine_traits<T...>의 **템플릿 특수화**를 사용해서

C++ Coroutine의 반환 타입을 지원할 수 있다.

```
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{};
}
```

반드시 return_type::promise_type일 필요는 없다

```cpp
#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**"

E0135가 발생했던 이유

Coroutine Promise Requirement 를 통해서 **컴파일러가 하는 일**은?

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

코루틴을 작성하면...

```
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 ...
}
```

Traits 검사에 문제가 없다면...

```
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();
}
```

Promise를 통한 코드 생성

```
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();
}
```

내 코드는 어디에?

```
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();
}
```

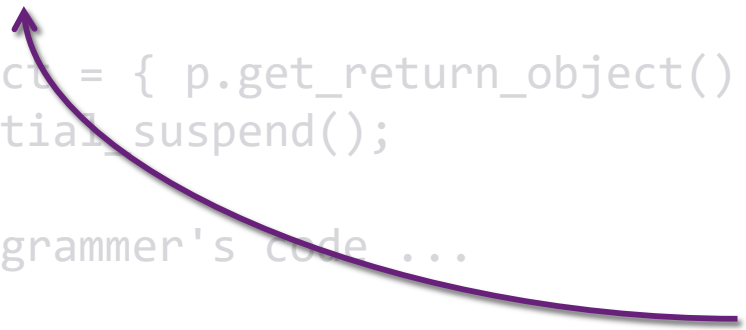
대부분이 promise_type의 멤버함수

```
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-constructor-arguments

Promise: 생성

```
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();
}
```



인자Argument가 불일치 하는 경우,
기본 생성자를 사용

Promise: 생성

```
#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: 생성자/소멸자 예시

```
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();
}
```

Promise를 통한 코드 생성

```
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();
}
```

반환 개체의 생성

```
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

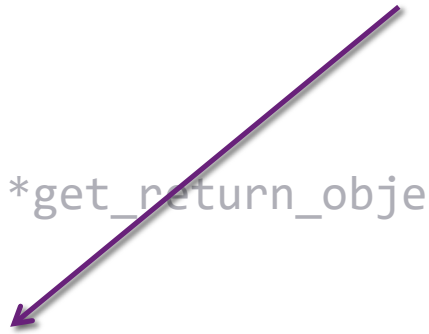
```
struct return_sample
{
    struct promise_type
    {
        auto get_return_object() -> promise_type*
        {
            return this;
        }
        static promise_type *get_return_object_on_allocation_failure() noexcept;
    };

    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() };
}
```

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() };
}
```

반드시 promise_type일 필요는 없음

```
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() };
}
```

실제로는 operator new 를 사용한 동적 할당

```
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() };
}
```

동적할당에 실패하는 경우 사용

```
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();
}
```

예외 처리는 어떻게?

```
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();
}
```

Compiler가 추가한 **최후의** 예외 처리 코드

```
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{};
        }
    };
};
```

Awaitable Type을 반환

```
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
바로 프로그래머의 코드로 진입할 것인가?

```
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

co_return 이후 코루틴 프레임을 파괴할 것인가?

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

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

반환 타입 작성해보기

```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;
    }
};
```

Promise 포인터를 통해 접근

반환 타입의 정의

```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;
    }
};
```

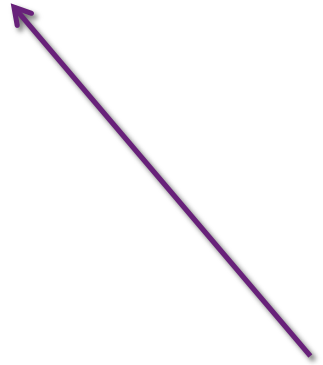
```cpp
struct promise_type
{
    Item* ptr = nullptr;

    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"

+ Promise 타입 정의

```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;
    }
};
```

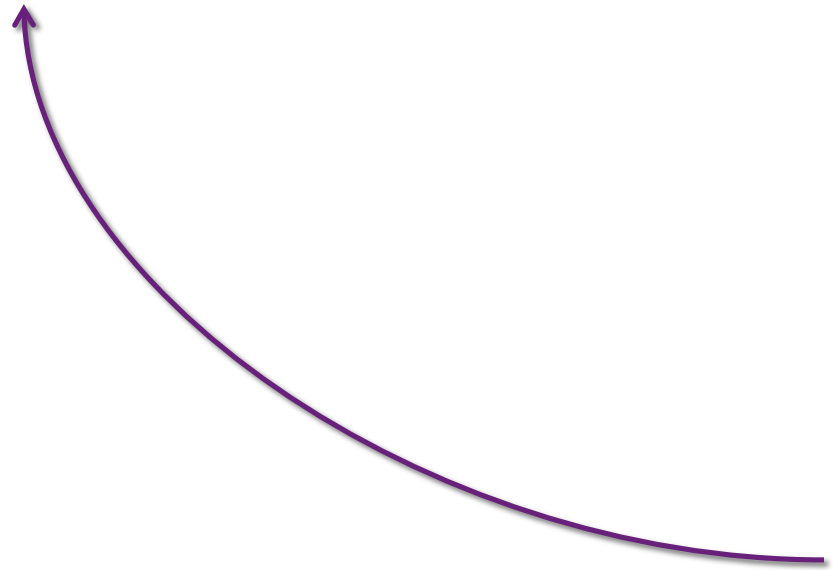
```cpp
struct promise_type
{
    Item* ptr = nullptr;

    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 을 사용하려면
return_value 함수가 필요

```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;
    }
};
```

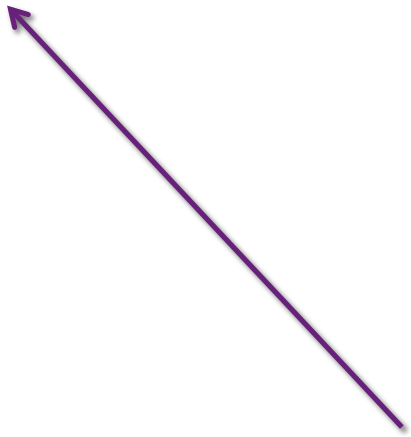


E2665: "pack<int>::promise_type" has no member "return_void"

인자 없이 co_return 한다면?

```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 의 인자가 없다면
return_void 함수를 사용

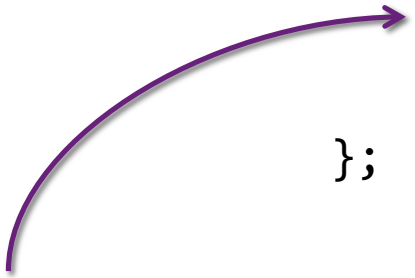
```cpp
#include <experimental/coroutine>
using namespace std::experimental;

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() {}
};
```

두 함수를 모두 정의한다면?

```
#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() {}
};
```



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

```cpp
#include <experimental/coroutine>

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

co_return 표현식을 사용하면...

```
#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: 컴파일러의 코드

```
#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;
}
```

co_return 0; 로부터 생성된 코드

Promise Type의 역할

타입 시스템을 사용한 컴파일 시간compile time 검사

- `coroutine_traits<T...>`

코루틴 프레임의 생성/소멸

- Operator `new`/`delete`
- 생성자/소멸자
- `get_return_object`, `get_return_object_on_allocation_failure`

반환return 처리

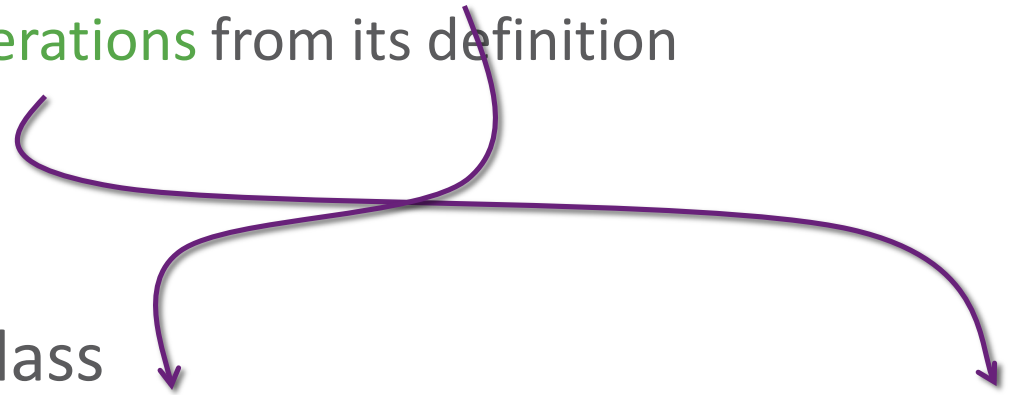
- `co_return`: `return_value`, `return_void`
- `co_yield` : `yield_value`

Coroutine Handle

코루틴 개체를 재개resume/파괴destroy하기 위한 안전한 방법

Coroutine

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



Task Class

- An organized data structure with its member functions

결국 코루틴 프레임을
개체Object처럼 사용할 수 있다는 의미

```
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 타입

```
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

Promise를 포함한 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

보조 함수Helper Function들

타입 시스템Type System과 C++ Coroutine

프로그래머는 타입을 통해 컴파일러의 코드 생성을 제어

- Promise Type
- Awaitable Type

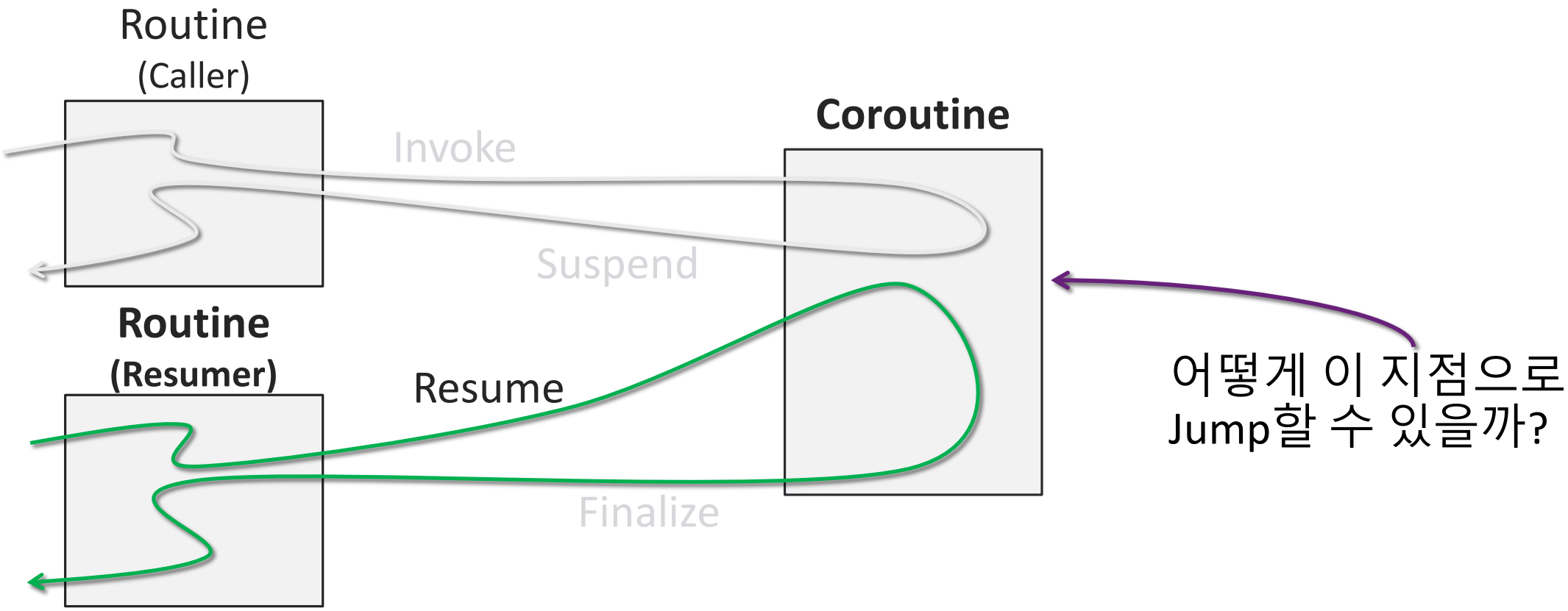
중단suspend/반환return을 위한 연산자 사용

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

그렇다면 **재개**resume는 어떻게 제어하는가?

코루틴Coroutine

코드는 컴파일러가 생성하는데, 그렇다면...



```
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

결국 컴파일러가
지원해야 하는 부분

C++ Coroutine 을 위한 Compiler Intrinsic

Intrinsic: 컴파일러 내장 함수

MSVC 와 Clang 모두 `coroutine_handle<void>` 구현을 위해 intrinsic을 노출.

GCC는 과연 어떤 선택을 할지…

C++ Coroutine 을 위한 Compiler Intrinsic

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`
- `...`

다른 Intrinsic들도 있으나, 사용방법이 불분명

`<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

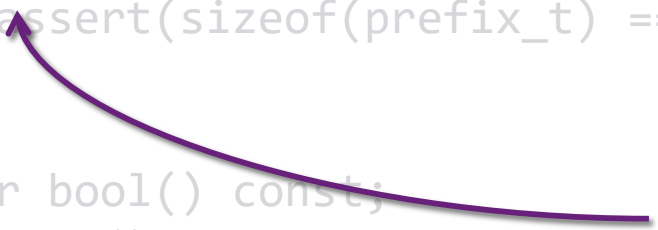
Coroutine Intrinsic: MSVC

```
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*);
};
```



이 부분은?

코루틴 프레임의 구조?

Coroutine Frame에 포함되는 것들

Frame == Routine’s state

서브루틴의 프레임과 비슷하지만, 몇가지 더 추가된다…

CppCon 2016: “Introduction to C++ Coroutines”

What are in the Coroutine Frame?

- 지역 변수
  - 함수 전달인자Argument들
  - 임시 변수들 (+ Awaitable)
  - 반환 값
- Coroutine Frame's Prefix (coroutine_handle<void>에서 사용)
- Promise 개체
- 컴파일러가 사용하는 영역(maybe)

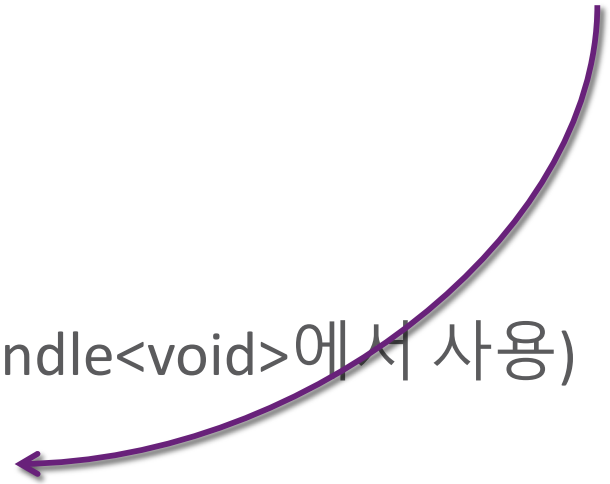
CppCon 2016: "Introduction to C++ Coroutines"

- 지역 변수
  - 함수 전달인자Argument들
  - 임시 변수들 (+ Awaitable)
  - 반환 값
- Coroutine Frame’s Prefix (coroutine_handle<void>에서 사용)
- Promise 개체
- 컴파일러가 사용하는 영역(maybe)

서브루틴과 동일

- 지역 변수
  - 함수 전달인자Argument들
  - 임시 변수들 (+ Awaitable)
  - 반환 값
- Coroutine Frame's Prefix (coroutine_handle<void>에서 사용)
- Promise 개체
- 컴파일러가 사용하는 영역(maybe)

Stack-less 코루틴에서 사용



이들은 어떻게 할당되는가?

Promise Type 을 사용한 코루틴 프레임의 할당/해제

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));
}
```

Frame 관리 코드

```
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));
}
```

대략 이런 타입이 생성된다.

함수의 Frame 타입

```
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));
}
```

promise_type을 통해 관리하는 경우

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));
}
```

정의가 없는 경우,
전역 할당/해제를 사용한다.

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 내부의 Promise 개체

수명Lifetime & 복사/이동 소멸Elision

N4736, 11.4.4

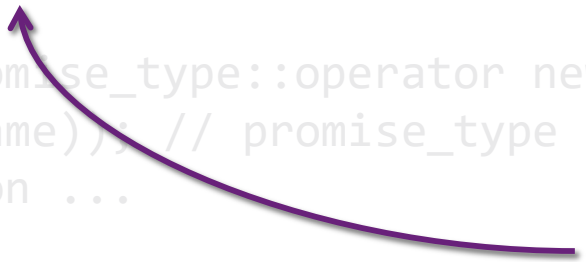
When a coroutine is invoked, a copy is created for each coroutine parameter …
*… **The lifetime of parameter** copies ends immediately after the lifetime of the coroutine promise object ends. …*

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));
}
```



이제 이 부분에 대해서...

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>

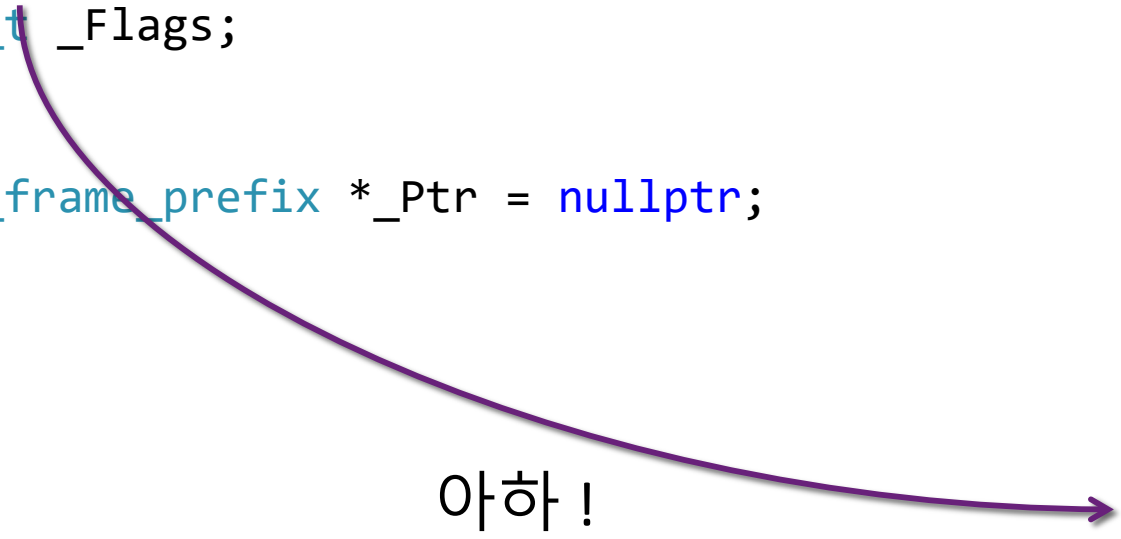
VC++ 에서의 정의

```
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;
};
```

중단 지점Suspend Point의 Index?

<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 0: // final suspended
    goto __destroy_point;
  case 1:
    goto __initial_suspend_point;
  case 2:
    goto __suspend_point_1;
  case 3:
    goto __suspend_point_2;
  // ...
}
```

<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>

호출 규약Calling Convention: `__cdecl`

호출자Calling Function 에 의한 스택 정리Stack clean-up

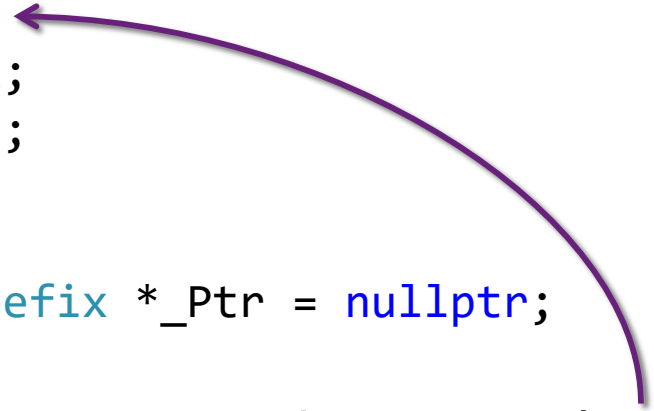
== 반환 타입이 `void` 라면, 정리가 필요하지 않다

== 코루틴의 프레임은 `_Resume_fn` 호출 이후 변경되지 않는다

변수들이 스택이 아니라 (동적할당된) 코루틴 프레임에 위치하므로,
이는 매우 자연스러운 코드!

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;
};
```



결국 이 함수의 호출은 goto 와 동일하다

<experimental/resumable>

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

어떤 정보도 없음:(

하지만 **VC++ 헤더에 Clang-cl 컴파일러를 사용하면**
반드시 Crash가 발생하는 것으로 보아,
MSVC와는 다르다는 것을 알 수 있었다…

libcxx/release_70/include/experimental/coroutine

libc++ 7.0 에서의 정의

```
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

복잡하지는 않다!

```
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 함수가 먼저 배치된다.

코루틴이 final suspended 상태일 때는 `nullptr` 값을 가진다.

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 함수를 호출하면
프레임(과 변수들)의 소멸자가 호출된다.

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

`coroutine_handle<void>` 는 여기까지

다음은 `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

이상한 계산 코드

```
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);
```

주소와 정수가 사용된다?

주소 계산이 확실하다!

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>

VC++의 코드

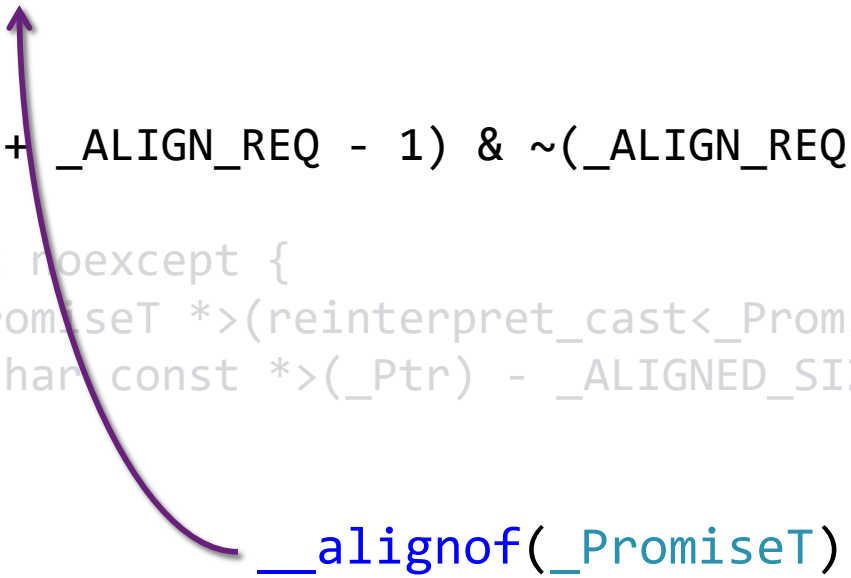
```
static const size_t _ALIGN_REQ = sizeof(void *) * 2;
static const size_t _ALIGNED_SIZE =
    is_empty_v<_PromiseT>
    ? 0
    : ((sizeof(_PromiseT) + _ALIGN_REQ - 1) & ~(_ALIGN_REQ - 1));

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

복잡하지만 , 16의 배수를 반환
(clang의 `__alignof` 과 같음)

<experimental/resumable>

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>

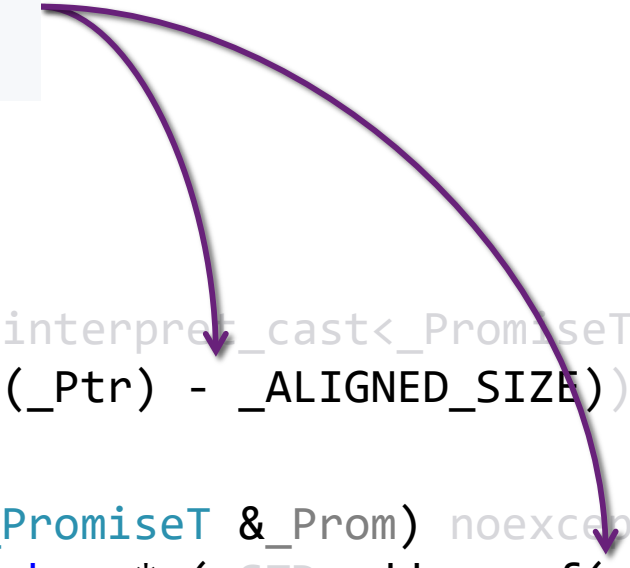
Promise Type 의 정렬 크기

```
_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>

두 함수의 핵심

```
| 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>

MSVC의 메모리 배치

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

약간의 분석 후...

Clang's Frame

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

Clang의 메모리 배치

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

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

두 컴파일러의 Promise Type, Frame Prefix 배치가 다른 것이
clang-cl compiler와 VC++ header를 사용했을때 Crash가 발생하는 이유였다.

`?` 에는 MSVC처럼 index가 위치한다. (변경된 경우, `resume()` 에서 Crash 발생)

그건 그렇고,
`coroutine_handle<void>` 개체는 어떻게 얻을 수 있죠?

`coroutine_handle<void>` 개체를 획득하는 방법

- Promise Type
  호출 단계에서 획득 가능 (`get_return_object`)

- `void*`
  간단한 변환 함수 지원

- Awaitable Type
  중단 단계에서 획득 가능 (`await_suspend`)

```
promise_type &_prom;
```

Promise 에 접근할 수 있다면 …

```
promise_type &_prom;

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

Promise >> Coroutine Handle

```
promise_type &_prom;

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

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

Coroutine Handle >> **Promise**

```
void *ptr;
```

포인터가 있다면 …

```
void *ptr;

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

`void*` >> Coroutine Handle

```
void *ptr;

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

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

Coroutine Handle >> `void*`

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

인자로 전달된다.

Awaitable >> Coroutine Handle

coroutine_handle<P>의 역할

간접적(안전한) 컴파일러 내장함수Compiler Intrinsic의 사용
- done, resume, destroy

Coroutine 프레임의 소멸
- destroy

컴파일러의 배치에 맞는 주소 계산
- 코루틴 프레임 시작부Prefix의 주소
- 코루틴 프레임 내 Promise 개체의 주소 계산

C++ Coroutine 구성요소component 요약

Awaitable, Promise, 그리고 Handle

Awaitable

`co_await`의 피연산자

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

중단Suspension 제어 (== 프로그래머 의도를 반영)

Promise

코루틴 코드 생성

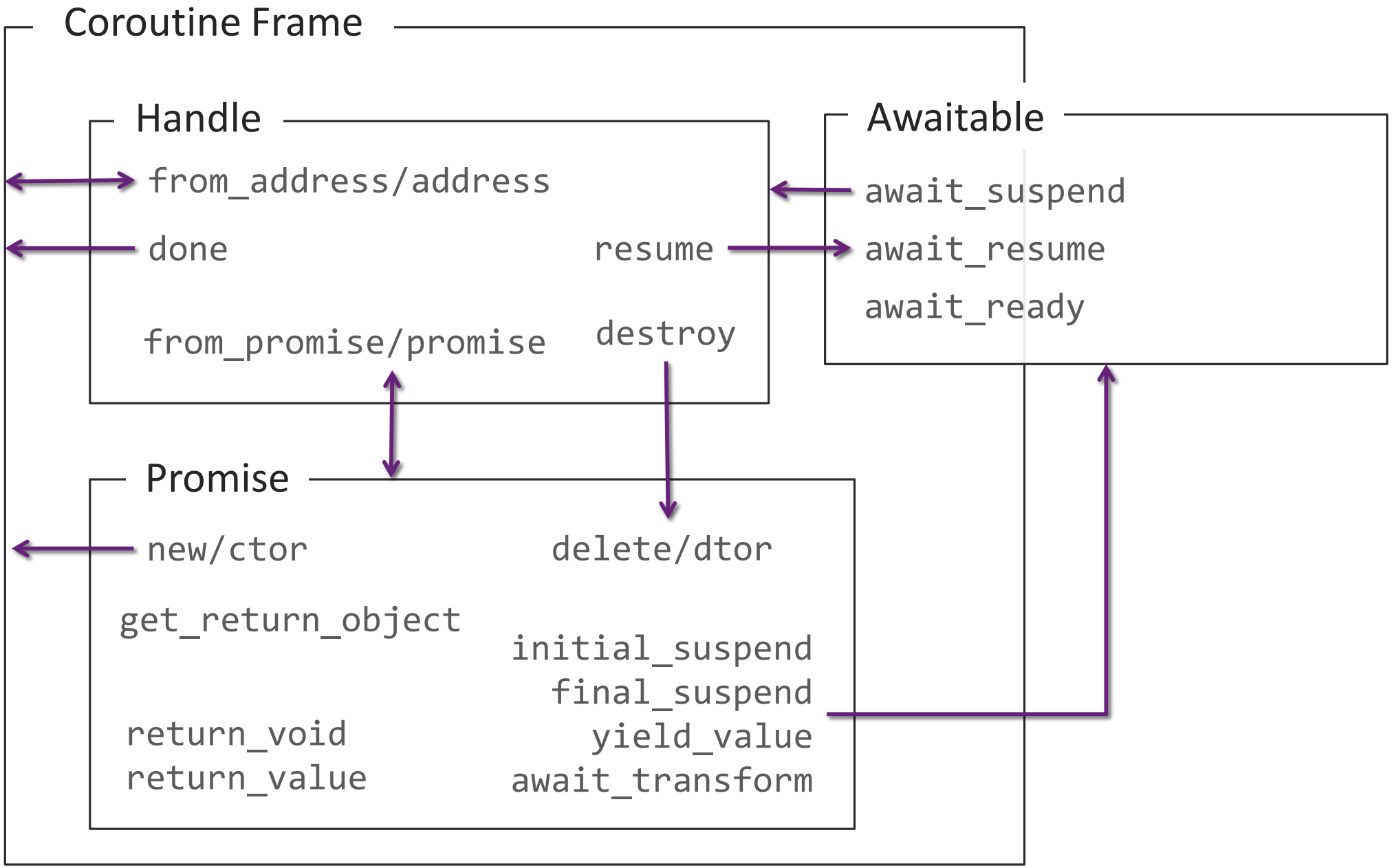
- 프레임의 수명주기
  - 할당/해제
  - Initial / Final suspend
- 반환/예외 처리

Handle

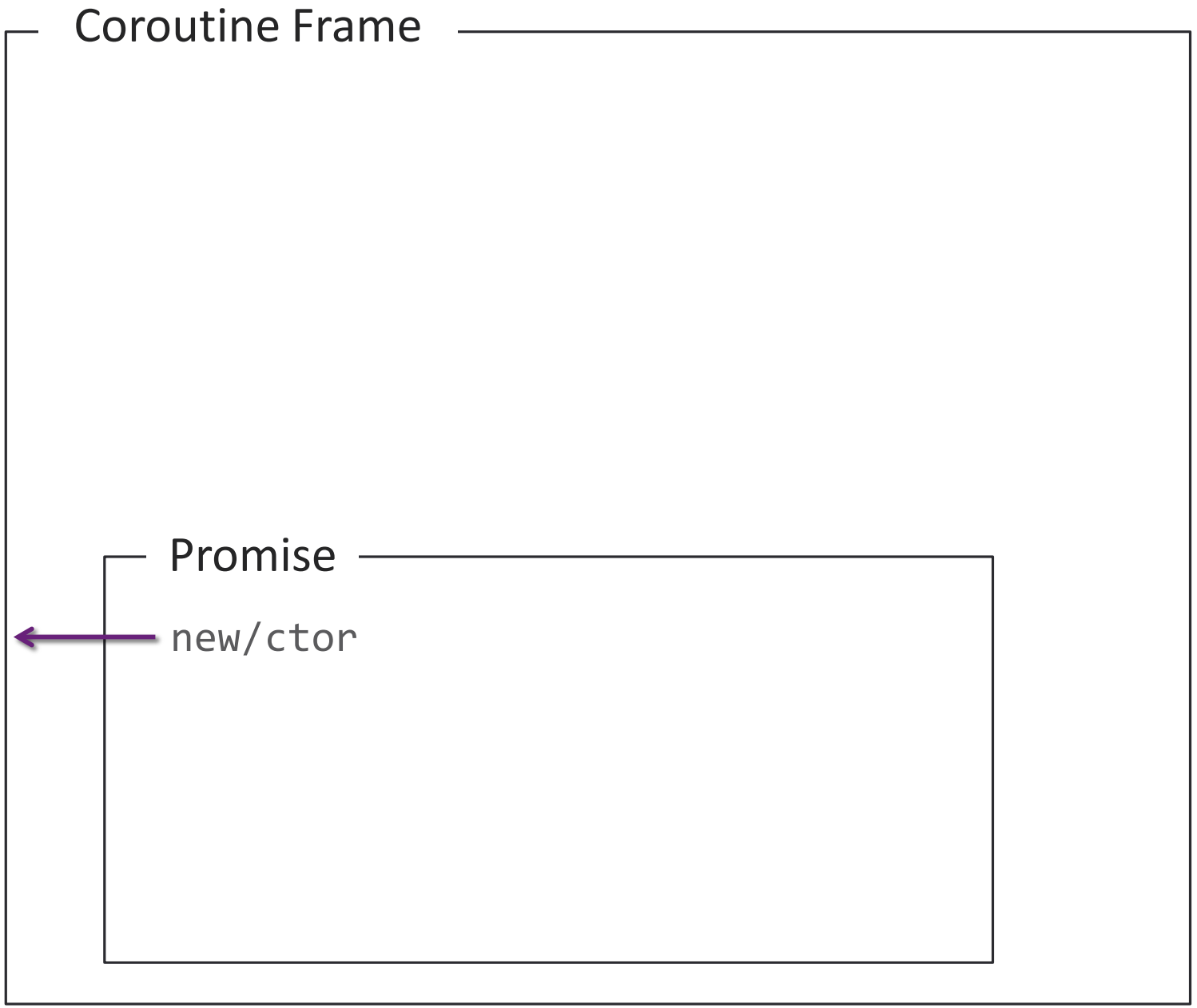
컴파일러가 생성한 구조체와 내장함수로의 인터페이스

- Suspend
- Resume
- Destroy

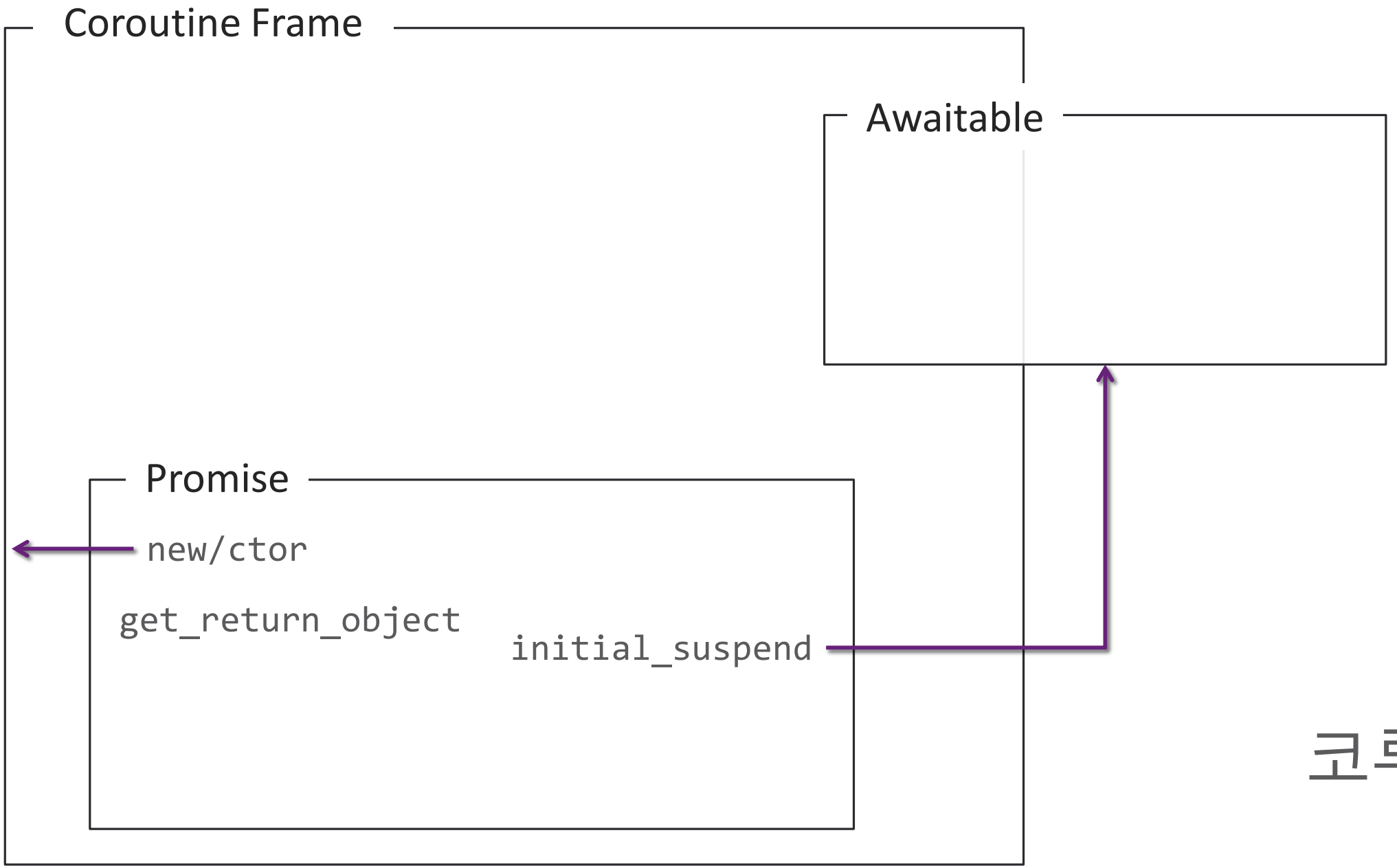
각각의 역할


Coroutine Frame
Handle
from_address/address
done
from_promise/promise
resume
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

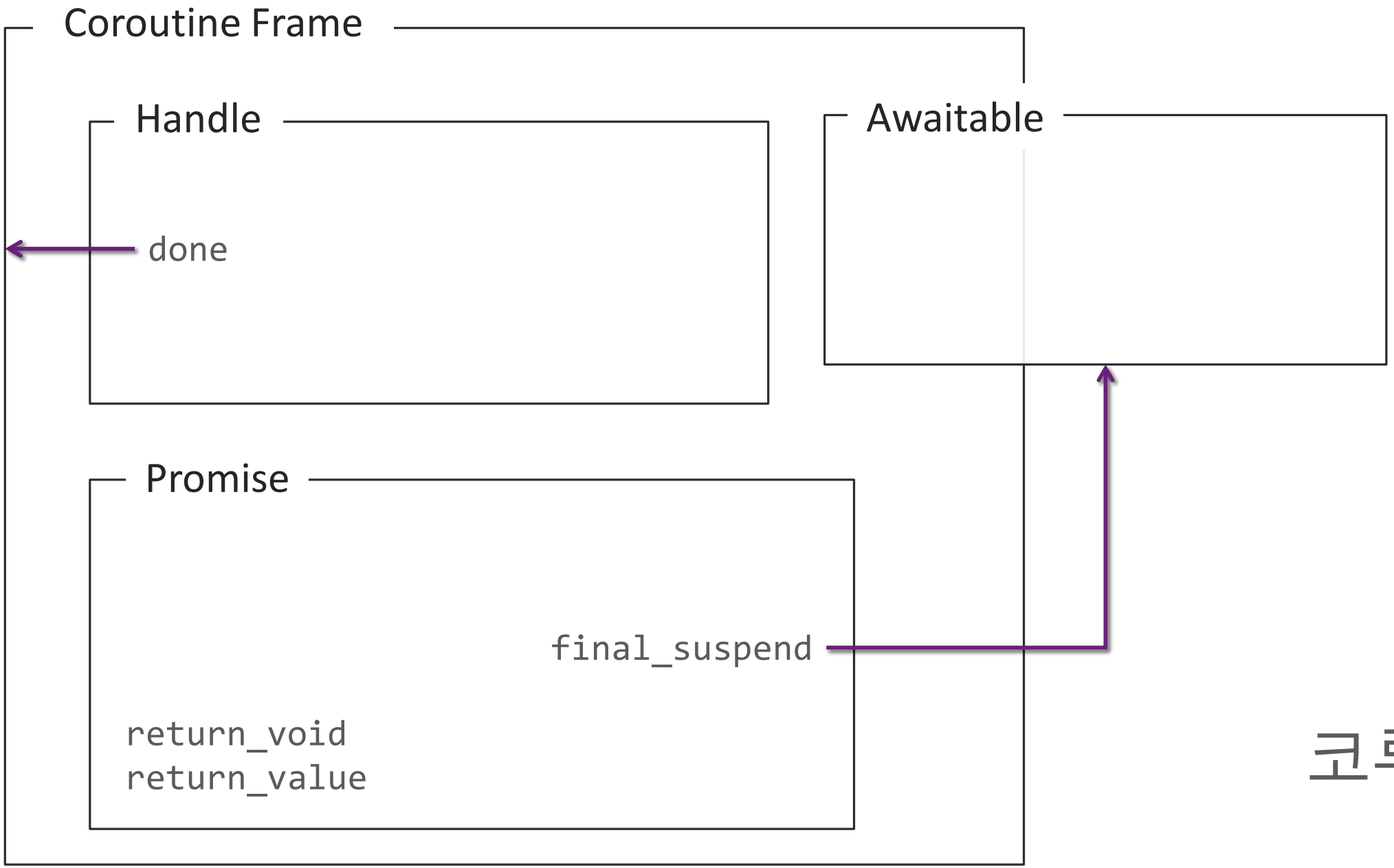
관계도



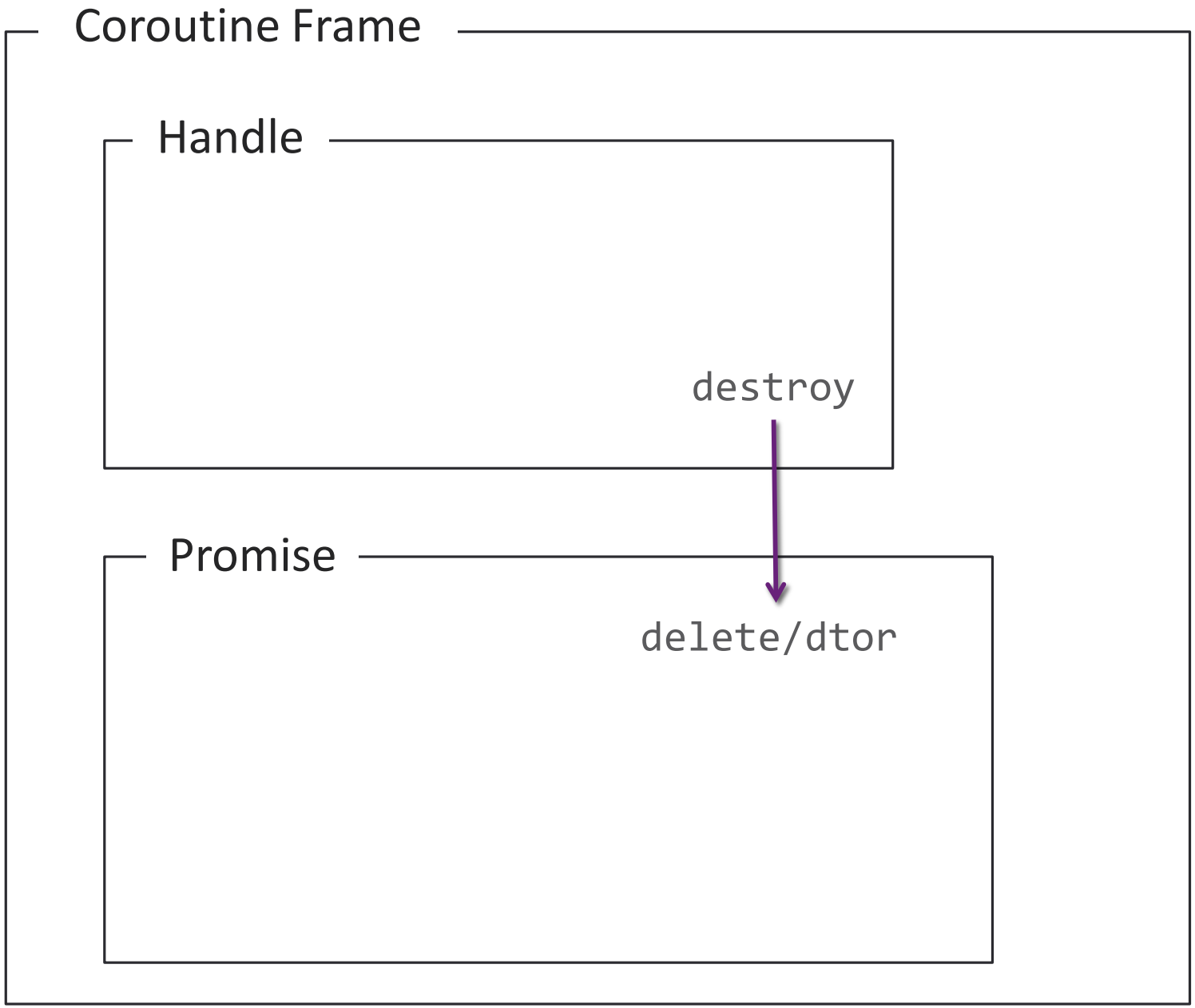
프레임 수명 주기: 생성


Coroutine Frame
Awaitable
Promise
new/ctor
get_return_object
initial_suspend

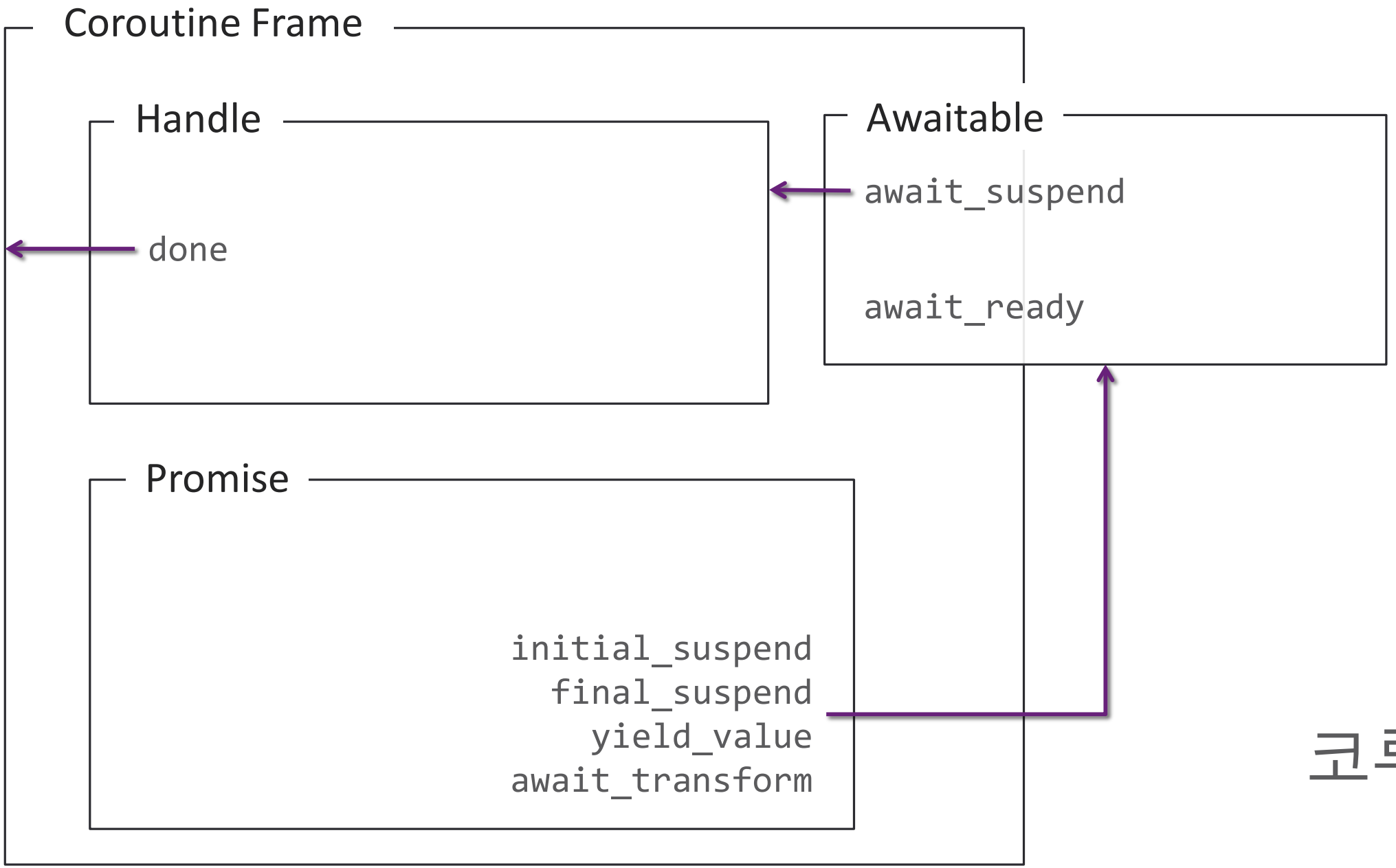
코루틴 연산: 호출


Coroutine Frame
Handle
done
Awaitable
Promise
final_suspend
return_void
return_value

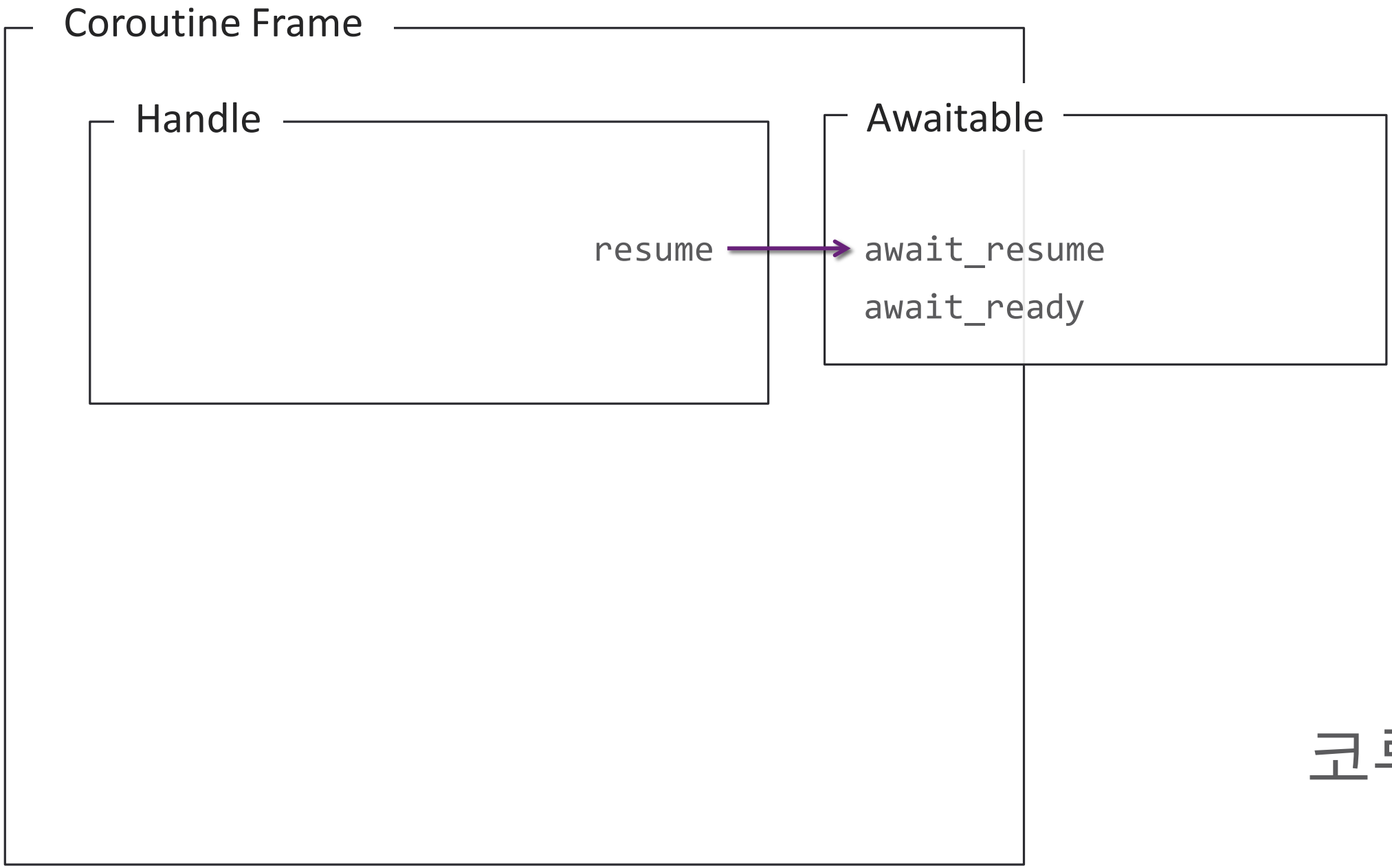
코루틴 연산: 반환



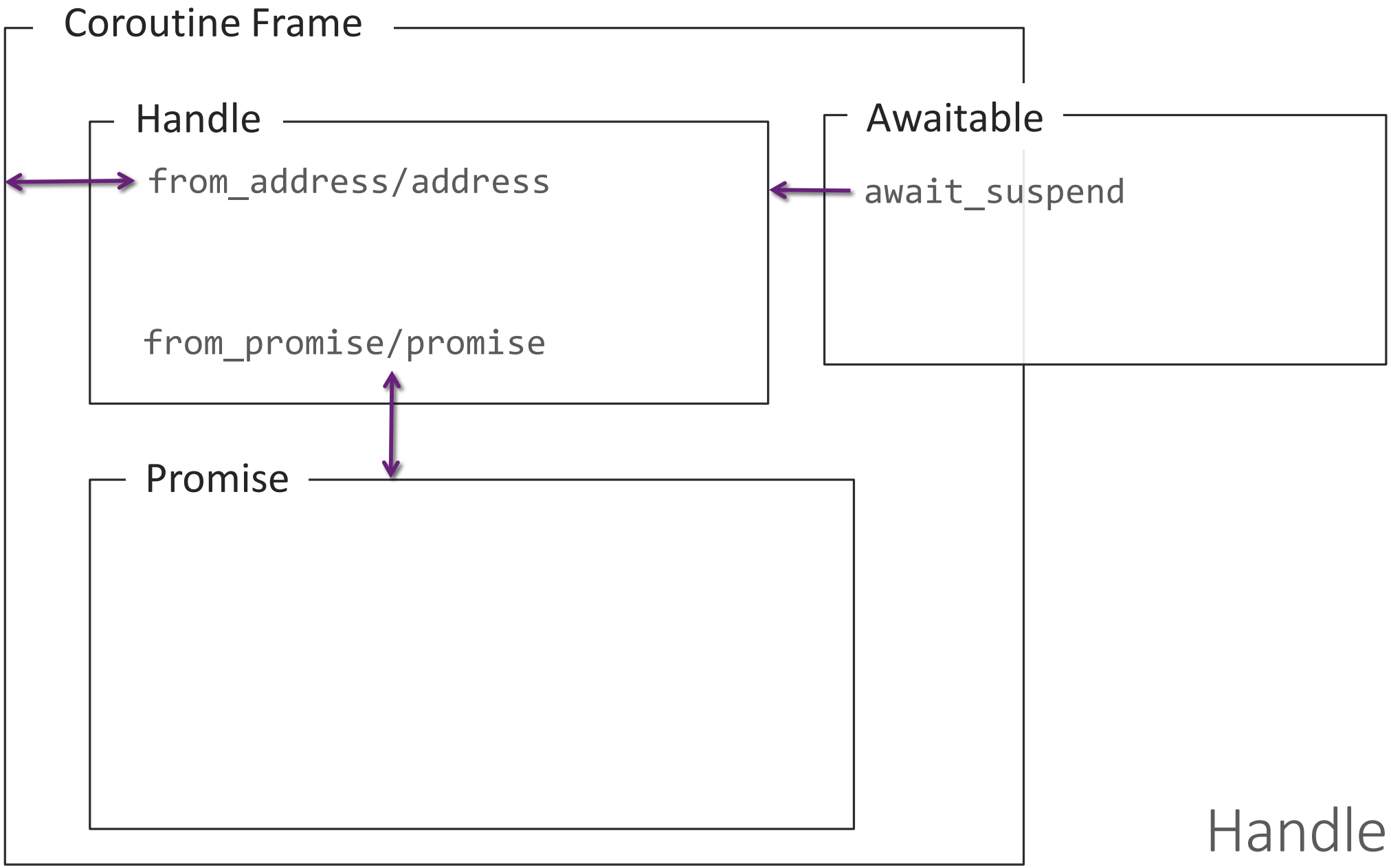
코루틴 연산: 종결(소멸)


Coroutine Frame
Handle
done
Awaitable
await_suspend
await_ready
Promise
initial_suspend
final_suspend
yield_value
await_transform

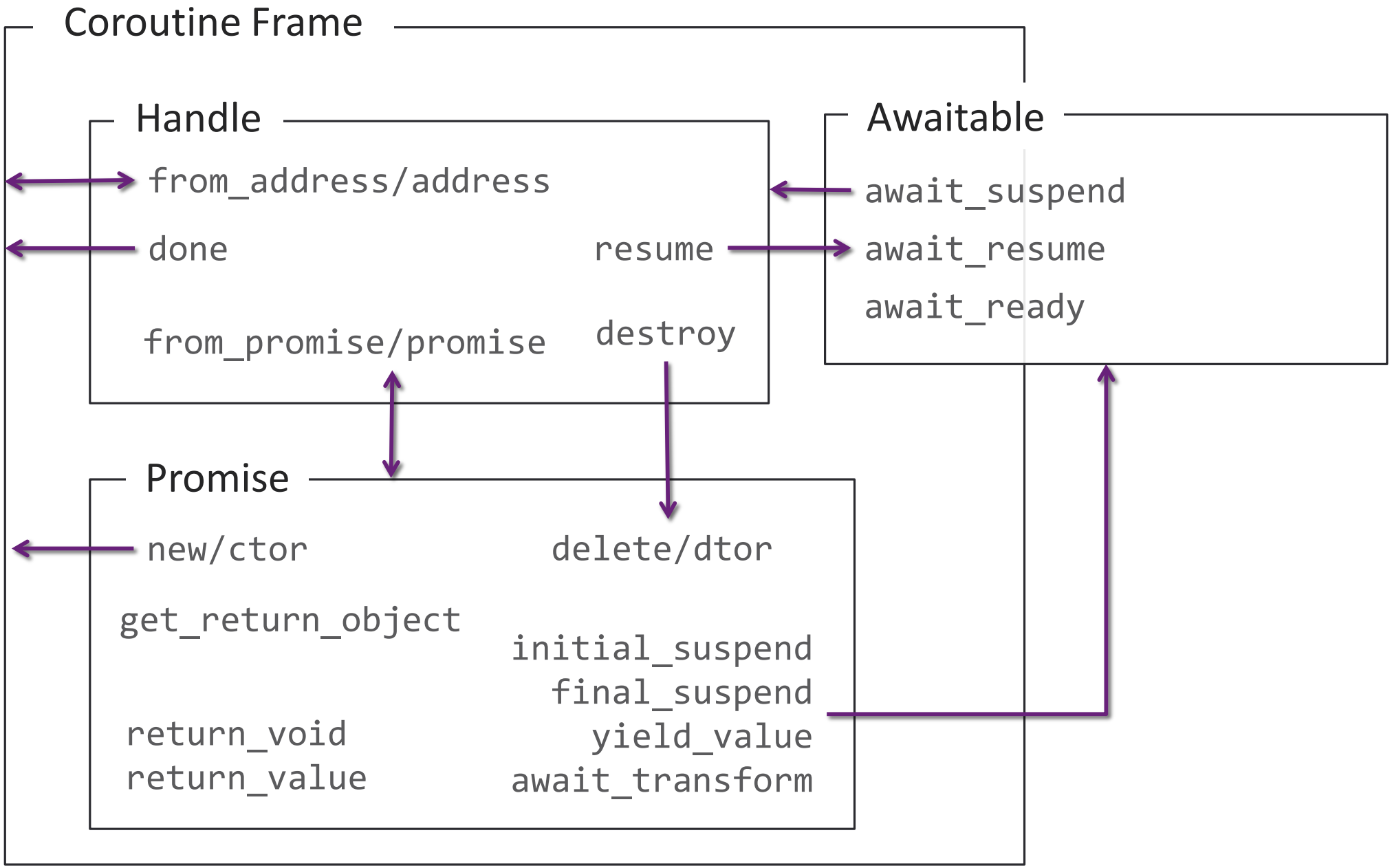
코루틴 연산: 중단


Coroutine Frame
Handle
Awaitable
resume
await_resume
await_ready

코루틴 연산: 재개


Coroutine Frame
Handle
from_address/address
from_promise/promise
Awaitable
await_suspend
Promise

Handle로의 접근


Coroutine Frame
Handle
from_address/address
done
from_promise/promise
resume
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

관계도

감사합니다!

질문 / 발표자료의 오류는 C++ Korea Facebook Group
혹은 luncliff@gmail.com
로 알려주세요!

Coroutine Generator

Understanding `co_yield`

`co_yield` 연산자

`co_return`와 유사하지만, **반환**return보다는 **중단**suspension에 더 무게를 두고 있음

```
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

co_yield 표현식이 사용되면...

```
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);
}
```

프로그래머의 코드는
promise_type::yield_value 함수로 전달된다

2018/n4736.pdf

co_yield 의 의미구조 Semantics

```
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!
}
```

MSVC: 분리된 형태로 생성되는 경우도 허용

<experimental/generator> in VC++

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

    return sum;
}
```

Generator: 사용자 코드

```
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;
}
```

일반적인 input iterator(일방향 진행)와 동일

Generator의 의미구조 Semantics

```
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;
};
```

복사는 불가능, 이동은 가능

<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;
};
```

소멸자에서 코루틴 프레임을 제거

<experimental/generator> in VC++

Generator: 생성자/소멸자

```
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

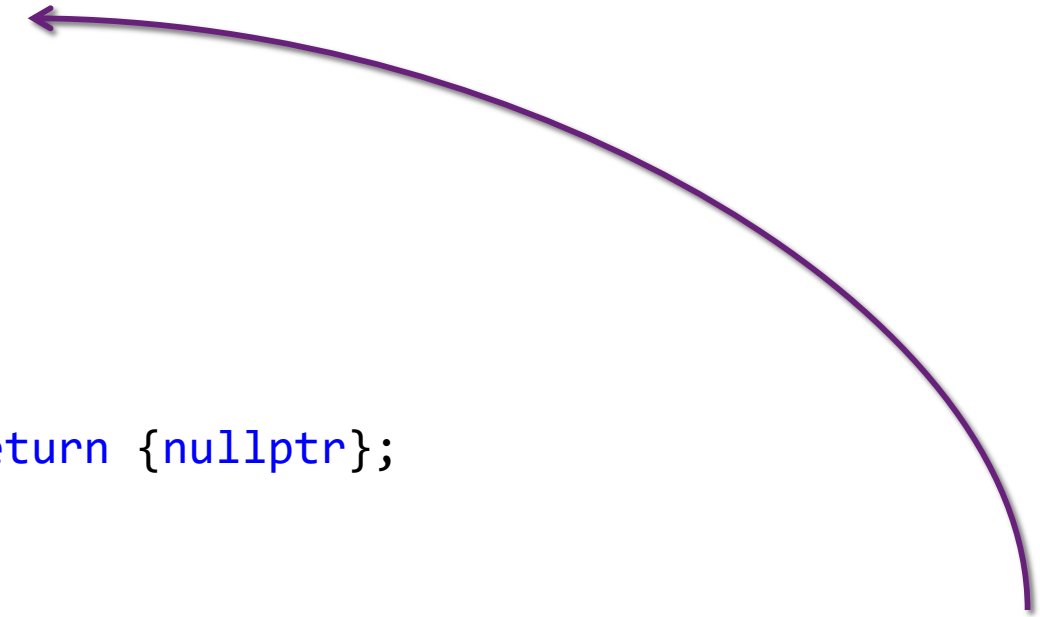
결국 포인터 하나와 동일하다

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;
        }
    };
    // ...
};
```

Promise 개체를 통해서
값에 접근한다

```
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}; }
};
```



전진 Advance == 재개 Resume

```
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;
};
```

단순히 yield된 변수의 주소만 저장한다.

Generator: Promise

이 타입 정말로 안전한가?

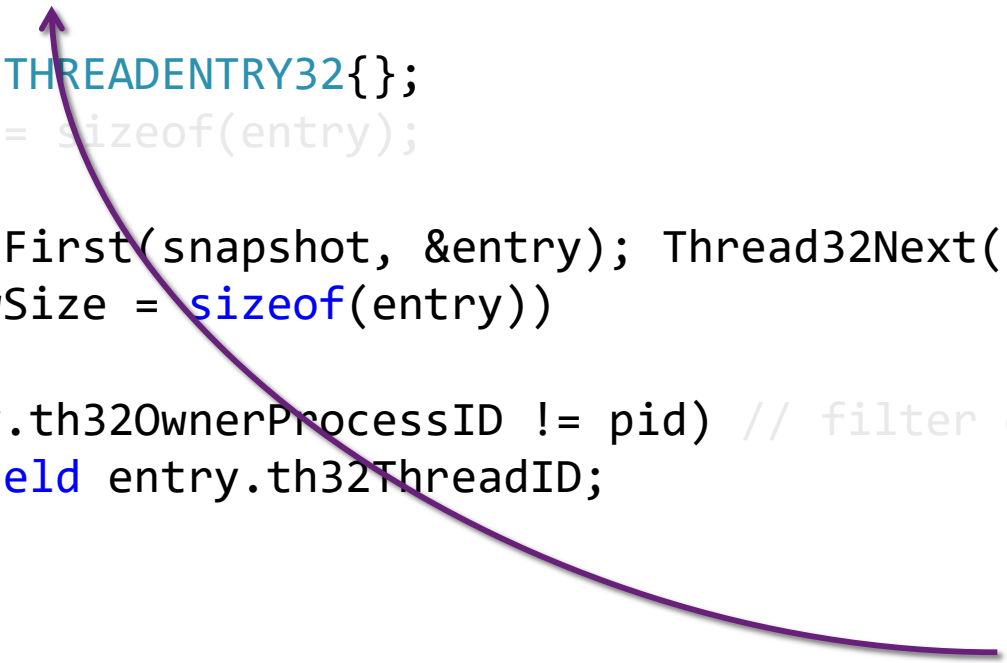
```
auto current_threads() -> generator<DWORD>
{
    auto pid = GetCurrentProcessId();

    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);
}
```

이 코드의 문제점?

```
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);
}
```

만약 호출자가 loop를 완주하지 않으면,
이 라인은 실행되지 않는다 (+ 코루틴 프레임은 소멸되어버림)

```
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;
}
```



소멸자를 사용한 Coverage Leak 예방

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

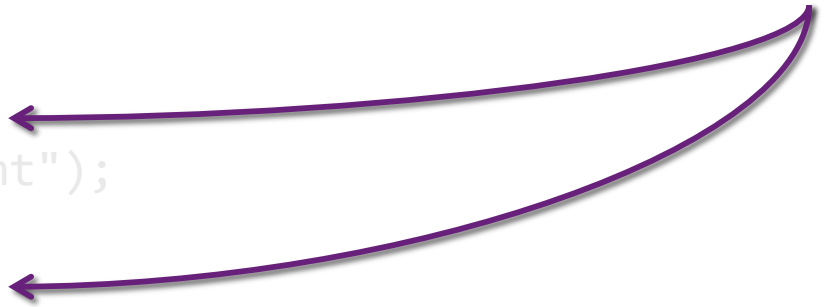
```
auto program(coro_queue& foreground, //
             coro_queue& background) -> return_ignore
{
    using namespace std;
    print_thread_id("invoke");

    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++

```
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;
}
```