

C++20 Coroutines

What's next?

Dawid Pilarski

dawid.pilarski@panicsoftware.com

blog.panicsoftware.com

dawid.pilarski@tomtom.com

Introduction



Introduction

Quick refresh about the coroutines.

Missing coroutines parts

RVO for the `co_await`

`return_value` or/and `return_void`

Summary

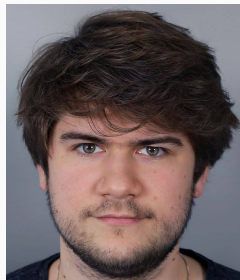


Time is rather tight.
Please hold your questions till the end.



Dawid Pilarski

- Senior Software Developer in TomTom
- Member of the ISO/JTC1/SC22/WG21
- Member of the PKN KT (programming languages)
- C++ blog writer



Quick refresh about the
coroutines.



Subroutine Is a sequence of program instructions that performs a specific task, packaged as a unit.

Function Is a subroutine

Coroutine Is generalization of the function.



Function can be:

- called
- returned from

What are the coroutines?



Coroutine can be:

- called
- returned from
- suspended



Coroutine can be:

- called
- returned from
- suspended
- resumed from



Coroutine can be:

- called
- returned from
- suspended
- resumed from
- created

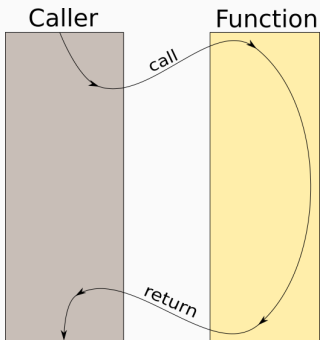


Coroutine can be:

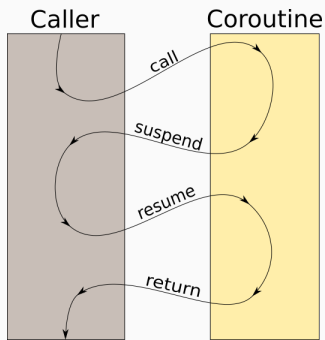
- called
- returned from
- suspended
- resumed from
- created
- destroyed



Function's flow:



Coroutine flow:





Creating custom coroutine type is not easy:

- C++ provides keywords **only**.



Creating custom coroutine type is not easy:

- C++ provides keywords **only**.
- **Developer must implement** what keywords do.



Creating custom coroutine type is not easy:

- C++ provides keywords **only**.
- **Developer must implement** what keywords do.

This means:

- Implementation of `promise_type` (~6 functions)



Creating custom coroutine type is not easy:

- C++ provides keywords **only**.
- **Developer must implement** what keywords do.

This means:

- Implementation of `promise_type` (~6 functions)
- Implementation of the `co_await` keyword (~3 functions)



Creating custom coroutine type is not easy:

- C++ provides keywords **only**.
- **Developer must implement** what keywords do.

This means:

- Implementation of `promise_type` (~6 functions)
- Implementation of the `co_await` keyword (~3 functions)

You need to remember to implement on average **9 functions**.



```
// returned-type    name    arguments
//|-----| /-----| /-----|
generator<int> fibonacci (int from_value);
```

- Whether the function is a coroutine depends on [it's definition](#).



```
// returned-type    name        arguments
//|-----| /-----| /-----|
generator<int> fibonacci (int from_value);
```

- Whether the function is a coroutine depends on [it's definition](#).
- If function is a coroutine it's [return type must support coroutines](#).



```
// returned-type    name        arguments  
///-----/ /-----/ /-----/  
generator<int> fibonacci (int from_value);
```

- Whether the function is a coroutine depends on *it's definition*.
- If function is a coroutine it's *return type must support coroutines*.
- Compiler knows the function is a coroutine by presence of keywords *co_await, co_return, co_yield*



Type supports coroutines **if it has promise_type**.

promise_type can be:

- member of the class
- member of the specialization of the `coroutine_traits<returned_type>`



Promise_type controls coroutine's behavior.

- `awaitable initial_suspend();`
- suspension at the beginning



Promise_type controls coroutine's behavior.

- `awaitable initial_suspend();`
- `awaitable final_suspend();`
- suspension at the beginning
- suspension at the end



Promise_type controls coroutine's behavior.

- `awaitable initial_suspend();`
- `awaitable final_suspend();`
- `return_type`
`get_return_object();`
- suspension at the beginning
- suspension at the end
- `how to create`
`return_type`



Promise_type controls coroutine's behavior.

- `awaitable initial_suspend();`
- `awaitable final_suspend();`
- `return_type`
`get_return_object();`
- `void unhandled_exception();`
- suspension at the beginning
- suspension at the end
- how to create
`return_type`
- handling unhandled exception



`Promise_type` is also responsible for keyword's actions:

- `co_return V;`
- `p.return_value(V);`



`Promise_type` is also responsible for keyword's actions:

- `co_return V;`
- `co_return;`
- `p.return_value(V);`
- `p.return_void();`



`Promise_type` is also responsible for keyword's actions:

- `co_return V;`
- `co_return;`
- `co_yield V;`
- `p.return_value(V);`
- `p.return_void();`
- `co_await p.yield_value();`



In order to support `co_await` expressions, the argument (awaitable) must:

- have `awaiter` operator `co_await` defined, or



In order to support `co_await` expressions, the argument (awaitable) must:

- have awaiter operator `co_await` defined, or
- have global awaiter operator `co_await(A)` support, or



In order to support `co_await` expressions, the argument (awaitable) must:

- have awaiter operator `co_await` defined, or
- have global awaiter operator `co_await(A)` support, or
- implement 3 functions (be awaiter itself):



In order to support `co_await` expressions, the argument (awaitable) must:

- have awaiter operator `co_await` defined, or
- have global awaiter operator `co_await(A)` support, or
- implement 3 functions (be awaiter itself):
 - `bool await_ready()`



In order to support `co_await` expressions, the argument (awaitable) must:

- have awaiter operator `co_await` defined, or
- have global awaiter operator `co_await(A)` support, or
- implement 3 functions (be awaiter itself):
 - `bool await_ready()`
 - `await_suspend(coroutine_handle<P>) returning`



In order to support `co_await` expressions, the argument (awaitable) must:

- have awaiter operator `co_await` defined, or
- have global awaiter operator `co_await(A)` support, or
- implement 3 functions (be awaiter itself):
 - `bool await_ready()`
 - `await_suspend(coroutine_handle<P>) returning`
 - `void`



In order to support `co_await` expressions, the argument (awaitable) must:

- have awaiter operator `co_await` defined, or
- have global awaiter operator `co_await(A)` support, or
- implement 3 functions (be awaiter itself):
 - `bool await_ready()`
 - `await_suspend(coroutine_handle<P>)` returning
 - `void`
 - `bool`



In order to support `co_await` expressions, the argument (awaitable) must:

- have awaiter operator `co_await` defined, or
- have global awaiter operator `co_await(A)` support, or
- implement 3 functions (be awaiter itself):
 - `bool await_ready()`
 - `await_suspend(coroutine_handle<P>)` returning
 - `void`
 - `bool`
 - `another coroutine_handle`



In order to support `co_await` expressions, the argument (awaitable) must:

- have awaiter operator `co_await` defined, or
- have global awaiter operator `co_await(A)` support, or
- implement 3 functions (be awaiter itself):
 - `bool await_ready()`
 - `await_suspend(coroutine_handle<P>)` returning
 - `void`
 - `bool`
 - another `coroutine_handle`
 - `T await_resume()`



In order to support `co_await` expressions, the argument (awaitable) must:

- have awaiter operator `co_await` defined, or
- have global awaiter operator `co_await(A)` support, or
- implement 3 functions (be awaiter itself):
 - `bool await_ready()`
 - `await_suspend(coroutine_handle<P>) returning`
 - `void`
 - `bool`
 - another `coroutine_handle`
 - `T await_resume()`
- `promise_type.await_transform(A)` must produce object which type has above properties

Missing coroutines parts



RAII - Resource Acquisition Is Initialization.



How do coroutines differ?



How do coroutines differ?

```
task<std::vector<char>>
read_file(const path& file_path){

    auto opened_file = co_await async_open(path);
    auto content = co_await async_read(opened_file);
    co_await async_close(opened_file);

    co_return content;
}
```



How do coroutines differ?

```
task<std::vector<char>>
read_file(const path& file_path){

    auto opened_file = co_await async_open(path);
    auto content = co_await async_read(opened_file);
    co_await async_close(opened_file);

    co_return content;
}
```



How do coroutines differ?

```
task<std::vector<char>>
read_file(const path& file_path){

    auto opened_file = co_await async_open(path);
    auto content = co_await async_read(opened_file);
    co_await async_close(opened_file);

    co_return content;
}
```



How do coroutines differ?

```
task<std::vector<char>>
read_file(const path& file_path){

    auto opened_file = co_await async_open(path);
    auto content = co_await async_read(opened_file);
    co_await async_close(opened_file);

    co_return content;
}
```



How do coroutines differ?

```
task<std::vector<char>>
read_file(const path& file_path){

    auto opened_file = co_await async_open(path);
    auto content = co_await async_read(opened_file);
    co_await async_close(opened_file);

    co_return content;
}
```



How do coroutines differ?

```
task<std::vector<char>>
read_file(const path& file_path){

    auto opened_file = co_await async_open(path);
    auto content = co_await async_read(opened_file);
    co_await async_close(opened_file);

    co_return content;
}
```

No RAII to close the file!



How do coroutines differ?

```
task<std::vector<char>>
read_file(const path& file_path){

    auto opened_file = co_await async_open(path);
    auto content = co_await async_read(opened_file);
    co_await async_close(opened_file);

    co_return content;
}
```

No RAII to close the file!

Possible leak when `async_read` throws



Consider following scenario:

```
generator<std::string> lines(const path& file_path) {  
    ifstream stream(file_path.string());  
    std::string line;  
    while(getline(stream, line)){  
        co_yield line;  
    }  
    // stream closes file  
}
```



Consider following scenario:

```
generator<std::string> lines(const path& file_path) {  
    ifstream stream(file_path.string());  
    std::string line;  
    while(getline(stream, line)){  
        co_yield line;  
    }  
    // stream closes file  
}
```



Consider following scenario:

```
generator<std::string> lines(const path& file_path) {  
    ifstream stream(file_path.string());  
    std::string line;  
    while(getline(stream, line)){  
        co_yield line;  
    }  
    // stream closes file  
}
```



Consider following scenario:

```
generator<std::string> lines(const path& file_path) {  
    ifstream stream(file_path.string());  
    std::string line;  
    while(getline(stream, line)){  
        co_yield line;  
    }  
    // stream closes file  
}
```



Consider following scenario:

```
generator<std::string> lines(const path& file_path) {  
    ifstream stream(file_path.string());  
    std::string line;  
    while(getline(stream, line)){  
        co_yield line;  
    }  
    // stream closes file  
}
```



```
for(const auto& line : lines("myfile.txt")){  
    if(starts_with(line, "string I am looking for"))  
        break;  
}
```



```
for(const auto& line : lines("myfile.txt")){  
    if(starts_with(line, "string I am looking for"))  
        break;  
}
```

- at the break; we are destroying coroutine



```
for(const auto& line : lines("myfile.txt")){  
    if(starts_with(line, "string I am looking for"))  
        break;  
}
```

- at the break; we are destroying coroutine
- not all lines from file might be consumed



```
for(const auto& line : lines("myfile.txt")){  
    if(starts_with(line, "string I am looking for"))  
        break;  
}
```

- at the break; we are destroying coroutine
- not all lines from file might be consumed
- proper cleanup needs to be performed anyway on `coroutine_handle::destroy()`



```
generator<std::string> lines(const path& file_path) {  
    ifstream stream(file_path.string());  
    std::string line;  
    while(getline(stream, line)){  
        co_yield line;  
    }  
    // stream closes file  
}
```



```
generator<std::string> lines(const path& file_path) {  
    ifstream stream(file_path.string());  
    std::string line;  
    while(getline(stream, line)){  
        co_yield line;  
    }  
    // stream closes file  
}
```

↑ ~ifstream()



```
generator<std::string> lines(const path& file_path) {  
    ifstream stream(file_path.string());  
    std::string line;  
    while(getline(stream, line)){  
        co_yield line;  
    }  
    // stream closes file  
}
```

on loop finished

~ifstream()



```
generator<std::string> lines(const path& file_path) {  
    ifstream stream(file_path.string());  
    std::string line;  
    while(getline(stream, line)){  
        co_yield line;  
    }  
    // stream closes file  
}
```

(on destroy

on loop finished

~ifstream()



```
async_generator<std::string> lines(const path& file_path) {  
    auto opened_file = co_await async_open(file_path);  
    std::optional<std::string> opt_line;  
    while(opt_line = co_await  
            async_read_line(opened_file)){  
        co_yield *opt_line;  
    }  
  
    co_await async_close(opened_file);  
}
```



```
async_generator<std::string> lines(const path& file_path) {  
    auto opened_file = co_await async_open(file_path);  
    std::optional<std::string> opt_line;  
    while(opt_line = co_await  
            async_read_line(opened_file)){  
        co_yield *opt_line;  
    }  
  
    co_await async_close(opened_file);  
}
```





```
async_generator<std::string> lines(const path& file_path) {  
    auto opened_file = co_await async_open(file_path);  
    std::optional<std::string> opt_line;  
    while(opt_line = co_await  
            async_read_line(opened_file)){  
        co_yield *opt_line;  
    }  
  
    co_await async_close(opened_file);  
}
```



```
async_generator<std::string> lines(const path& file_path) {  
    auto opened_file = co_await async_open(file_path);  
    std::optional<std::string> opt_line;  
    while(opt_line = co_await  
            async_read_line(opened_file)){  
        co_yield *opt_line;  
    }  
  
    co_await async_close(opened_file);  
}
```



```
async_generator<std::string> lines(const path& file_path) {  
    auto opened_file = co_await async_open(file_path);  
    std::optional<std::string> opt_line;  
    while(opt_line = co_await  
            async_read_line(opened_file)){  
        co_yield *opt_line;  
    }  
  
    co_await async_close(opened_file);  
}
```

 cleanup



```
async_generator<std::string> lines(const path& file_path) {  
    auto opened_file = co_await async_open(file_path);  
    std::optional<std::string> opt_line;  
    while(opt_line = co_await  
            async_read_line(opened_file)){  
        co_yield *opt_line;  
    }  
  
    co_await async_close(opened_file);  
}
```

cleanup

on loop finished



```
async_generator<std::string> lines(const path& file_path) {  
    auto opened_file = co_await async_open(file_path);  
    std::optional<std::string> opt_line;  
    while(opt_line = co_await  
            async_read_line(opened_file)){  
        co_yield *opt_line;  
    }  
    co_await async_close(opened_file);  
}
```

on early destroy - *no cleanup*

cleanup

on loop finished



```
async_generator<std::string> lines(const path& file_path) {  
    auto opened_file = co_await async_open(file_path);  
    std::optional<std::string> opt_line;  
    while(opt_line = co_await  
            async_read_line(opened_file)){  
        // remember to resume the coroutine before destroying  
        auto cancellation_token = co_yield *opt_line;  
        if(cancellation_token) break;  
    }  
  
    co_await async_close(opened_file);  
}
```



```
async_generator<std::string> lines(const path& file_path) {  
    auto opened_file = co_await async_open(file_path);  
    std::optional<std::string> opt_line;  
    while(opt_line = co_await  
            async_read_line(opened_file)){  
        // remember to resume the coroutine before destroying  
        auto cancellation_token = co_yield *opt_line;  
        if(cancellation_token) break;  
    }  
  
    co_await async_close(opened_file);  
}
```



```
async_generator<std::string> lines(const path& file_path) {  
    auto opened_file = co_await async_open(file_path);  
    std::optional<std::string> opt_line;  
    while(opt_line = co_await  
            async_read_line(opened_file)){  
        // remember to resume the coroutine before destroying  
        auto cancellation_token = co_yield *opt_line;  
        if(cancellation_token) break;  
    }  
  
    co_await async_close(opened_file);  
}
```




```
async_generator<std::string> lines(const path& file_path) {  
    auto opened_file = co_await async_open(file_path);  
    std::optional<std::string> opt_line;  
    while(opt_line = co_await  
            async_read_line(opened_file)){  
        // remember to resume the coroutine before destroying  
        auto cancellation_token = co_yield *opt_line;  
        if(cancellation_token) break;  
    }  
  
    co_await async_close(opened_file);  
}
```



```
async_generator<std::string> lines(const path& file_path) {  
    auto opened_file = co_await async_open(file_path);  
    std::optional<std::string> opt_line;  
    while(opt_line = co_await  
            async_read_line(opened_file)){  
        // remember to resume the coroutine before destroying  
        auto cancellation_token = co_yield *opt_line;  
        if(cancellation_token) break;  
    }  
  
    co_await async_close(opened_file);  
}
```



- create special function in the awaiter : `~operator co_await`



- create special function in the awaiter : `~operator co_await`
- `~operator co_await` gets co-awaited at the end of the scope



- create special function in the awaiter : `~operator co_await`
- `~operator co_await` gets co-awaited at the end of the scope
- instead of `destroy()` you will invoke `set_done()`_{why}



```
async_generator<std::string> lines(const path& file_path) {  
    auto opened_file = co_await async_open(file_path);  
    while(getline(stream, line)){  
        co_yield co_await async_read_line(opened_file);  
    }  
  
    //async close happens as a part of cleanup  
}
```



```
async_generator<std::string> lines(const path& file_path) {  
    auto opened_file = co_await async_open(file_path);  
    while(getline(stream, line)){  
        co_yield co_await async_read_line(opened_file);  
    }  
}
```

```
//async close happens as a part of cleanup  
}
```



- Currently it's difficult to correctly implement asynchronous generators



- Currently it's difficult to correctly implement asynchronous generators
 - coroutine bodies



- Currently it's difficult to correctly implement asynchronous generators
 - coroutine bodies
 - coroutine type, because we cannot simply destroy the coroutine



- Currently it's difficult to correctly implement asynchronous generators
 - coroutine bodies
 - coroutine type, because we cannot simply destroy the coroutine
- In the space of asynchronous operations we have got no RAII idiom



- Currently it's difficult to correctly implement asynchronous generators
 - coroutine bodies
 - coroutine type, because we cannot simply destroy the coroutine
- In the space of asynchronous operations we have got no RAII idiom
- With adoption of the proposal it will get better

RVO for the `co_await`



RVO - Return Value Optimization.

Allows to avoid unnecessary copy or move construction of the values returned from the function.



RVO - Return Value Optimization.

Allows to avoid unnecessary copy or move construction of the values returned from the function.

For example:

```
std::vector<int> foo(){  
    return {1,2,3,4,5};  
}  
  
// no copy or move construction  
// invoked  
auto _ = foo();
```



regular function

```
std::vector<int> foo(){  
    return {1,2,3,4,5};  
}
```

transformed by compiler into:

```
void foo(std::vector<int>* ptr){  
    new(ptr) std::vector<int>  
        {1,2,3,4,5};  
}
```




expression

```
co_await event;
```

transformed by compiler into:

```
{  
    auto&& awaiter = transform(event);  
    if(!awaiter.await_ready()){  
        <coroutine suspend>  
        awaiter.await_suspend();  
    }  
    <coroutine resume>;  
    awaiter.await_resume();  
}
```



expression

```
co_await event;
```

1. On

`await_suspend`

coroutine gets
executed

transformed by compiler into:

```
{  
    auto&&awaiter = transform(event);  
    if(!awaiter.await_ready()){  
        <coroutine suspend>  
        awaiter.await_suspend();  
    }  
    <coroutine resume>;  
    awaiter.await_resume();  
}
```



expression

`co_await event;`

1. On

`await_suspend`

coroutine gets
executed

2. On

`await_resume`

result is returned

transformed by compiler into:

```
{  
    auto&&awaiter = transform(event);  
    if(!awaiter.await_ready()){  
        <coroutine suspend>  
        awaiter.await_suspend();  
    }  
    <coroutine resume>;  
    awaiter.await_resume();  
}
```



1. Remove
 `await_resume`
 function.

```
{  
    auto&& awaiter = transform(event);  
    if(!awaiter.await_ready()){  
        <coroutine suspend>  
        awaiter.await_suspend();  
    }  
    <coroutine resume>;  
    awaiter.await_resume();  
}
```



1. Remove

`await_resume`
function.

2. `await_suspend`

will create return
result

```
{  
    auto&& awaiter = transform(event);  
    if(!awaiter.await_ready()){  
        <coroutine suspend>  
        awaiter.await_suspend();  
    }  
    <coroutine resume>;  
    awaiter.await_resume();  
}
```



1. Remove `await_resume` function.
2. `await_suspend` will create return result
3. Remove `await_ready` function.

```
{  
    auto&& awaiter = transform(event);  
    if(!awaiter.await_ready()){  
        <coroutine suspend>  
        awaiter.await_suspend();  
    }  
    <coroutine resume>;  
}
```



1. Remove `await_resume` function.

```
{  
    auto&& awaiter = transform(event);  
    <coroutine suspend>  
    awaiter.await_suspend();  
    <coroutine resume>;  
}
```
2. `await_suspend` will create return result
3. Remove `await_ready` function.



Two additional functions in the `coroutine_handle` are needed.

- `set_value(T)`



Two additional functions in the `coroutine_handle` are needed.

- `set_value(T)`
- `set_value_from(Arg...)`



Two additional functions in the `coroutine_handle` are needed.

- `set_value(T)`
- `set_value_from(Arg...)`
- `set_exception(exception_ptr)`



Two additional functions in the `coroutine_handle` are needed.

- `set_value(T)`
- `set_value_from(Arg...)`
- `set_exception(exception_ptr)`

On coroutine resumption the compiler will generate code to check whether the exception was saved with `set_exception` and will rethrow it when needed.

Example of the `yield_value`



```
template <typename T> class task<T>::promise_type{  
    // ....  
  
    template <typename U> requires ConvertibleTo<U, T>  
    void return_value(U&& value){  
        handle.set_value<T>(std::forward<U>(value));  
    }  
  
    template <typename... Args>  
        requires Constructible<T, Args...>  
    void return_value(std::in_place_construct<Args&&...>  
        ctor_args){  
        handle.set_value_from<T>(ctor_args);  
    }  
  
};
```

Example of the `yield_value`



```
template <typename T> class task<T>:::promise_type{  
    // ....
```

```
    template <typename U> requires ConvertibleTo<U, T>  
    void return_value(U&& value){  
        handle.set_value<T>(std::forward<U>(value));  
    }
```

```
    template <typename... Args>  
        requires Constructible<T, Args...>  
    void return_value(std::in_place_construct<Args&&...>  
        ctor_args){  
        handle.set_value_from<T>(ctor_args);  
    }
```

```
};
```

Example of the `yield_value`



```
template <typename T> class task<T>::promise_type{
// ....

template <typename U> requires ConvertibleTo<U, T>
void return_value(U&& value){
    handle.set_value<T>(std::forward<U>(value));
}

template <typename... Args>
    requires Constructible<T, Args...>
void return_value(std::in_place_construct<Args&&...>
    ctor_args){
    handle.set_value_from<T>(ctor_args);
}

};
```

Example of the `yield_value`



```
template <typename T> class task<T>::promise_type{
// ....

template <typename U> requires ConvertibleTo<U, T>
void return_value(U&& value){
    handle.set_value<T>(std::forward<U>(value));
}

template <typename... Args>
    requires Constructible<T, Args...>
void return_value(std::in_place_construct<Args&&...>
    ctor_args){
    handle.set_value_from<T>(ctor_args);
}

};
```

Example of the `yield_value`



```
template <typename T> class task<T>::promise_type{
// ....

template <typename U> requires ConvertibleTo<U, T>
void return_value(U&& value){
    handle.set_value<T>(std::forward<U>(value));
}

template <typename... Args>
    requires Constructible<T, Args...>
void return_value(std::in_place_construct<Args&&...>
    ctor_args){
    handle.set_value_from<T>(ctor_args);
}

};
```


Example of the `yield_value`



```
template <typename T> class task<T>:::promise_type{
// ....

template <typename U> requires ConvertibleTo<U, T>
void return_value(U&& value){
    handle.set_value<T>(std::forward<U>(value));
}

template <typename... Args>
    requires Constructible<T, Args...>
void return_value(std::in_place_construct<Args&&...>
    ctor_args){
    handle.set_value_from<T>(ctor_args);
}

};
```

How do compiler know the result of the `co_await`?



With removal of the `await_ready` the compiler no longer knows about the `co_await` returned type.

We will need to guide the compiler. The proposal P1663R0 proposes to add member `await_result_type` to the Awaiter.



pros

- very simplified awaiter concept

cons



pros

- very simplified awaiter concept
- savings in CPU cycles

cons



pros

- very simplified awaiter concept
- savings in CPU cycles
 - Avoiding unnecessary move construction

cons



pros

- very simplified awaiter concept
- savings in CPU cycles
 - Avoiding unnecessary move construction
- savings in memory

cons



pros

- very simplified awaiter concept
- savings in CPU cycles
 - Avoiding unnecessary move construction
- savings in memory
 - no temporary variable created

cons



pros

- very simplified awaiter concept
- savings in CPU cycles
 - Avoiding unnecessary move construction
- savings in memory
 - no temporary variable created
 - allocated coroutine state is smaller

cons



pros

- very simplified awaiter concept
- savings in CPU cycles
 - Avoiding unnecessary move construction
- savings in memory
 - no temporary variable created
 - allocated coroutine state is smaller

cons

- removing `await_ready` makes `co_await` always suspend the coroutine (even if not needed)



pros

- very simplified awaiter concept
- savings in CPU cycles
 - Avoiding unnecessary move construction
- savings in memory
 - no temporary variable created
 - allocated coroutine state is smaller

cons

- removing `await_ready` makes `co_await` always suspend the coroutine (even if not needed)
- a need to support RVO manually (with the help of `construct_in_place`)



pros

- very simplified awaiter concept
- savings in CPU cycles
 - Avoiding unnecessary move construction
- savings in memory
 - no temporary variable created
 - allocated coroutine state is smaller

cons

- removing `await_ready` makes `co_await` always suspend the coroutine (even if not needed)
- a need to support RVO manually (with the help of `construct_in_place`)
- proposed RVO does not consider synchronous coroutines - only `co_await` keyword.

return _value or/and
return _void



right now it's not possible to implement both in the same scope.

- `return_value()`
- `return_void()`



right now it's not possible to implement both in the same scope.

- `return_value()`
- `return_void()`

Why would we even need that?



```
task<int> foo(){  
    co_return 42;  
}
```

```
task<void> start(){  
    std::cout << (co_await foo()) << std::endl;  
    // implicit co_return;  
}
```



```
task<int> foo(){  
    co_return 42; ← return_value(42);  
}
```

```
task<void> start(){  
    std::cout << (co_await foo()) << std::endl;  
    // implicit co_return;  
}
```




```
task<int> foo(){  
    co_return 42; ← return_value(42);  
}
```

```
task<void> start(){  
    std::cout << (co_await foo()) << std::endl;  
    // implicit co_return;  
}
```

↑ return_void()



```
template <typename T>
struct task<T>::promise_type{
    // ...
    void return_void()
        requires std::is_same<T, void>{}

    template <typename U>
    void return_value(U&& val)
        requires not std::is_same<T, void>{}
}
```



```
template <typename T>
struct task<T>::promise_type{
    // ...
    void return_void()
        requires std::is_same<T, void>{}

    template <typename U>
    void return_value(U&& val)
        requires not std::is_same<T, void>{}
}
```



```
template <typename T>
struct task<T>::promise_type{
    // ...
    void return_void()
        requires std::is_same<T, void>{}

    template <typename U>
    void return_value(U&& val)
        requires not std::is_same<T, void>{}
}
```

But that's not the way it works.

How implementors have to implement it?



```
template <typename T>
struct task<T>::promise_type{
    //...
    void return_value(){
        //...
    }
};
```

How implementors have to implement it?



```
template <typename T>
struct task<T>::promise_type{
    //...
    void return_value(){
        //...
    }
};
```

```
template <>
struct task<void>::promise_type{
    //...
    void return_void(){
        //...
    }
};
```



What's the tail recursive coroutines?

```
task<int> bar(){  
    co_return 42;  
}
```

```
task<int> foo(){  
    co_return co_await bar();  
}
```



What's the tail recursive coroutines?

```
task<int> bar(){  
    co_return 42;  
}
```

```
task<int> bar(){  
    co_return 42;  
}
```

```
task<int> foo(){  
    co_return co_await bar();  
}
```

```
task<int> foo(){  
    co_return bar();  
}
```

tail call / no tail call

How does regular/tail call work?



First, how does regular call work?



foo

How does regular/tail call work?



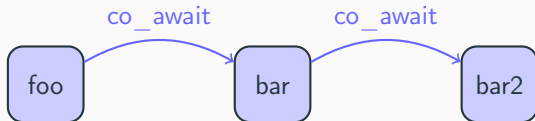
First, how does regular call work?



How does regular/tail call work?



First, how does regular call work?



How does regular/tail call work?



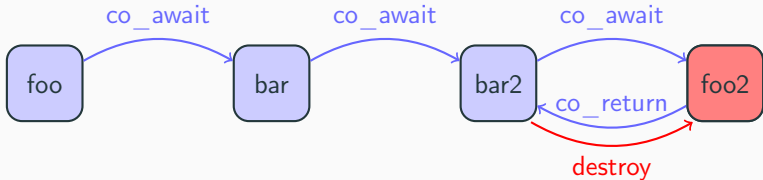
First, how does regular call work?



How does regular/tail call work?



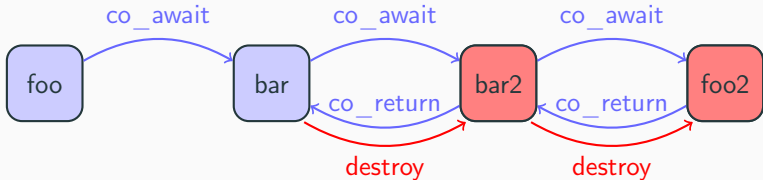
First, how does regular call work?



How does regular/tail call work?



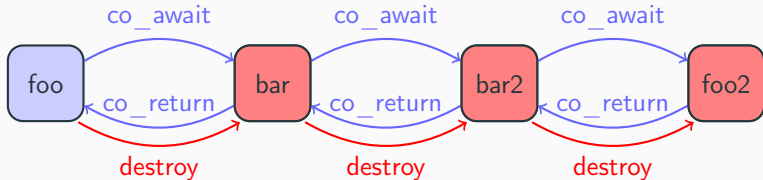
First, how does regular call work?



How does regular/tail call work?



First, how does regular call work?



Conclusion:

- At peak 4 coroutine frames had to be allocated
- Only after returning to the calling coroutine, called one can be destroyed

How does tail-call work?



In case of tail-call we first destroy the coroutine and then call another one.

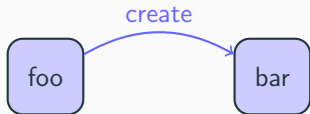


foo

How does tail-call work?



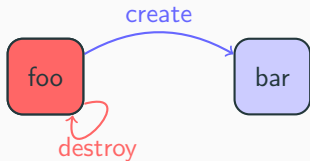
In case of tail-call we first destroy the coroutine and then call another one.



How does tail-call work?



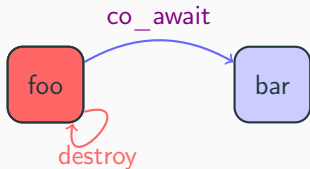
In case of tail-call we first destroy the coroutine and then call another one.



How does tail-call work?



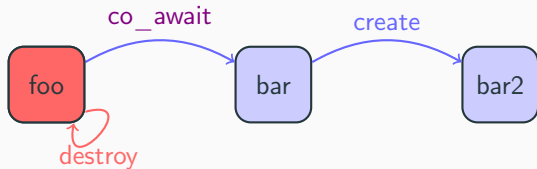
In case of tail-call we first destroy the coroutine and then call another one.



How does tail-call work?



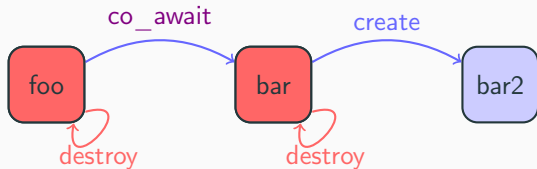
In case of tail-call we first destroy the coroutine and then call another one.



How does tail-call work?



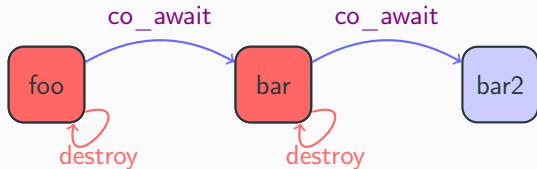
In case of tail-call we first destroy the coroutine and then call another one.



How does tail-call work?



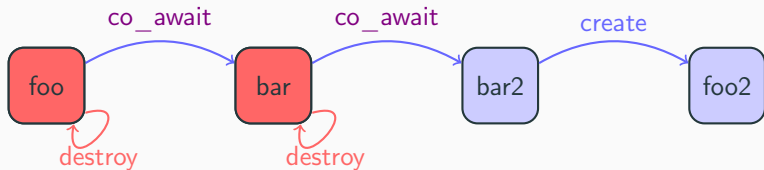
In case of tail-call we first destroy the coroutine and then call another one.



How does tail-call work?



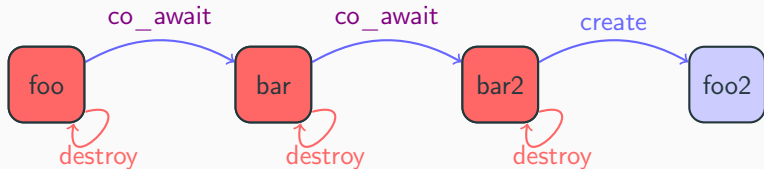
In case of tail-call we first destroy the coroutine and then call another one.



How does tail-call work?



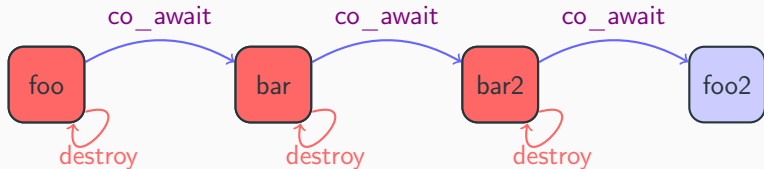
In case of tail-call we first destroy the coroutine and then call another one.



How does tail-call work?



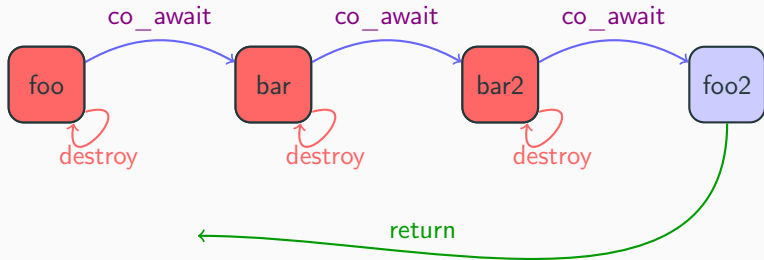
In case of tail-call we first destroy the coroutine and then call another one.



How does tail-call work?



In case of tail-call we first destroy the coroutine and then call another one.





Tail call is implementable.

But only for non-void returning types.

Why it's not implementable for void types.



```
template <>
struct task<void>::promise_type{
    //...

    void return_void(){}

};
```



```
template <>
struct task<void>::promise_type{
    //...

    void return_void(){}

    void return_value(task<void>&&){
        // ...
    }
};
```

Why it's not implementable for void types.



```
template <>
struct task<void>::promise_type{
    //...

    void return_void(){}

    void return_value(task<void>&&){
        // ...
    }
};
```

Both cannot
appear in the
same scope!

Thank you for your attention!

Special thank you! goes to:



- Gor Nishanov
- Lewiss Baker

for making coroutines

Summary



Summary



- Lewiss Baker's Assymetric transfer blog
- newest C++ draft
- My blog - blog.panicsoftware.com
- James McNellis - "Introduction to the C++ Coroutines"
- Gor Nishanov - any video about the coroutines
- Toby Allsopp - "Coroutines: what can't they do?"



Questions?