

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`

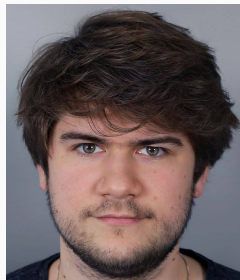


Time is rather tight.
Please hold your questions till the end.



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



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

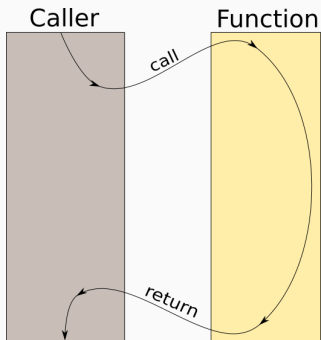


Coroutine can be:

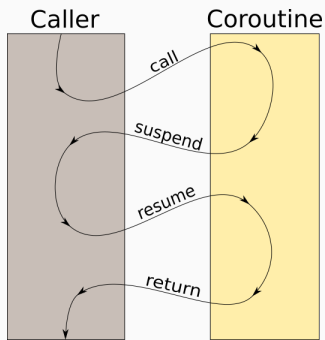
- called
- returned from
- suspended
- resumed from
- created
- destroyed



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**.



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:



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:
 - `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:
 - `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:
 - `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:
 - `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:
 - `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:
 - `bool await_ready()`
 - `awaiter_suspend(coroutine_handle<P>)` returning
 - `void`
 - `bool`
 - another `coroutine_handle`
 - `T await_resume()`

Missing coroutines parts





RAII - Resource Acquisition Is Initialization.

what does it mean in practice?

Release the resources in the destructor.



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

Add solution



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;  
}
```

issues:

- at the `break;` we are destroying coroutine



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

issues:

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



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

issues:


- at the break; we are destroying coroutine
- not all lines 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 loop finished

on destroy

~ifstream()



```
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);  
    }  
  
    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);  
    while(getline(stream, line)){  
        co_yield co_await async_read_line(opened_file);  
    }  
  
    co_await async_close(opened_file);  
}
```

on loop finished

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);  
    }  
    co_await async_close(opened_file);  
}
```

on destroy - *no cleanup*

on loop finished

cleanup

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

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



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);
}

};
```



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 [and|or] return void



Thank you for your attention!

Special thank you! goes to:



- Gor Nishanov
- Lewiss Baker

for making coroutines



- 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?