C++20 Coroutines

What's next?

Dawid Pilarski

dawid.pilarski@panicsoftware.com
blog.panicsoftware.com
dawid.pilarski@tomtom.com

Introduction

Agenda

Introduction

Quick refresh about the coroutines.

Missing coroutines parts

RVO for the co_await

return_value or/and return_void

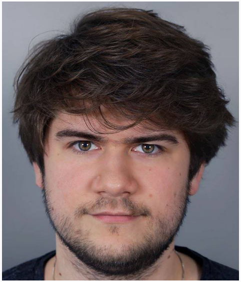
Questions...

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

Who am I?

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.

What are 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.

What are the coroutines?

Function can be:

- called
- returned from

What are the coroutines?

Coroutine can be:

- called
- returned from
- suspended

What are the coroutines?

Coroutine can be:

- called
- returned from
- suspended
- resumed from

What are the coroutines?

Coroutine can be:

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

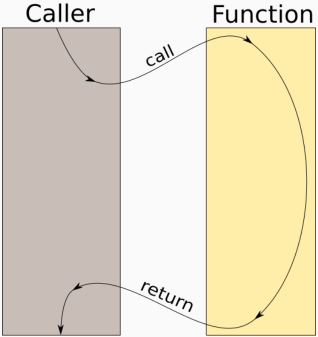
What are the coroutines?

Coroutine can be:

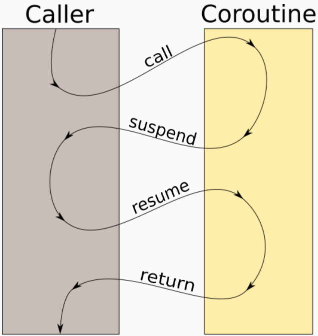
- called
- returned from
- suspended
- resumed from
- created
- destroyed

Coroutine flowchart

Function's flow:



Coroutine flow:



How to implement custom coroutine types.

Creating custom coroutine type is not easy:

- C++ provides keywords only.

How to implement custom coroutine types.

Creating custom coroutine type is not easy:

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

How to implement custom coroutine types.

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)

How to implement custom coroutine types.

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)

How to implement custom coroutine types.

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.

Coroutine declaration

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

- Whether the function is a coroutine depends on it's definition.

Coroutine declaration

```
// 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.

Promise_type

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

Promise_type controls coroutine's behavior.

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

Promise_type

Promise_type controls coroutine's behavior.

- `awaitable initial_suspend();`
- `awaitable final_suspend();`

- `suspension at the beginning`
- `suspension at the end`

Promise_type

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

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

Keywords and promise_type

Promise_type is also responsible for keyword's actions:

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

Keywords and promise_type

Promise_type is also responsible for keyword's actions:

- `co_return V;`
- `co_return;`

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

Keywords and promise_type

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

co_await

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

co_await

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()`

co_await

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

co_await

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

co_await

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()`
  - `await_suspend(coroutine_handle<P>)` returning
    - void
    - bool
    - another `coroutine_handle`
  - `T await_resume()`

Missing coroutines parts

Type erasure

asynchronous RAII

RAII - Resource Acquisition Is Initialization.

what does it mean in practice?

Release the resources in the destructor.

RAII vs. coroutines

How do coroutines differ?

RAII vs. coroutines

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;
}
```

RAII vs. coroutines

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;
}
```

RAII vs. coroutines

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;
}
```

RAII vs. coroutines

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!

RAII vs. coroutines

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

It's not only about proper error handling

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
}
```

It’s not only about proper error handling

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
}
```

It’s not only about proper error handling

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
}
```

It's not only about proper error handling

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
}
```

It's not only about proper error handling

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
}
```

generator example usage

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

generator example usage

```
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

generator example usage

```
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 from file might be consumed

generator example usage

```
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 from file might be consumed
- proper cleanup needs to be performed anyway on `coroutine_handle::destroy()`

No issue for the given synchronous coroutine

```
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
}
```

No issue for the given synchronous coroutine

```
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()

No issue for the given synchronous coroutine

```
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()

No issue for the given synchronous coroutine

```
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()

asynchronous generators

```
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

asynchronous generators

```
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 early destroy - *no cleanup*

on loop finished

cleanup

Current solution to the problem

```
async_generator<std::string> lines(const path& file_path) {
  auto opened_file = co_await async_open(file_path);
  while(getline(stream, line)){
    // remember to resume coroutine before destroy!
    auto cancellation_token =
         co_yield co_await async_read_line(opened_file);
    if(cancellation_token){
      goto clean_up;
    }
  }

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

Current solution to the problem

```
async_generator<std::string> lines(const path& file_path) {
  auto opened_file = co_await async_open(file_path);
  while(getline(stream, line)){
    // remember to resume coroutine before destroy!
    auto cancellation_token =
          co_yield co_await async_read_line(opened_file);
    if(cancellation_token){
      goto clean_up;
    }
  }

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

Current solution to the problem

```cpp
async_generator<std::string> lines(const path& file_path) {
  auto opened_file = co_await async_open(file_path);
  while(getline(stream, line)){
    // remember to resume coroutine before destroy!
    auto cancellation_token =
          co_yield co_await async_read_line(opened_file);
    if(cancellation_token){
      goto clean_up;
    }
  }

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

Current solution to the problem

```
async_generator<std::string> lines(const path& file_path) {
  auto opened_file = co_await async_open(file_path);
  while(getline(stream, line)){
    // remember to resume coroutine before destroy!
    auto cancellation_token =
         co_yield co_await async_read_line(opened_file);
    if(cancellation_token){
      goto clean_up;
    }
  }

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

Current solution to the problem

```
async_generator<std::string> lines(const path& file_path) {
  auto opened_file = co_await async_open(file_path);
  while(getline(stream, line)){
    // remember to resume coroutine before destroy!
    auto cancellation_token =
          co_yield co_await async_read_line(opened_file);
    if(cancellation_token){
      goto clean_up;
    }
  }

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

Proposed coroutines improvement

- make co_await create it's own 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()`

Example

```cpp
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
}
```

Example

```
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
}
```

Summary

- Currently it's difficult to correctly implement asynchronous generators

Summary

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

Summary

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

Summary

- 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

Summary

- 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

What is RVO?

RVO - Return Value Optimization.

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

What is RVO?

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();
```

RVO on regular functions

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};
}
```

Why RVO is not possible with co_await

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();
}
```

Why RVO is not possible with co_await

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();
}
```

Why RVO is not possible with co_await

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();
}
```

How to resolve the issue [P1663R0]

1. Remove await_resume function.

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

How to resolve the issue [P1663R0]

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();
}
```

How to resolve the issue [P1663R0]

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>;
}
```

How to resolve the issue [P1663R0]

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

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

Extensions to the coroutine_handle

Two additional functions in the coroutine_handle are needed.

- set_value(T)

Extensions to the coroutine_handle

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)

Extensions to the coroutine_handle

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

```cpp
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.

RVO summary

pros	cons
very simplified awaiter concept	

RVO summary

pros	cons
• very simplified awaiter concept	
• savings in CPU cycles	

RVO summary

pros

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

cons

RVO summary

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

RVO summary

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

RVO summary

pros	cons

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

RVO summary

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)

RVO summary

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)

RVO summary

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

return value [and|or] return void

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

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

return value [and|or] 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?

Motivating example: implementation simplifications

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

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

Motivating example: implementation simplifications

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

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

Motivating example: implementation simplifications

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

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

How implementors would like to implement it

```
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>{}
}
```

How implementors would like to implement it

```
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>{}
}
```

How implementors would like to implement it

```cpp
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(){
    //...
  }
};
```

Tail recursive coroutines

What's the tail recursive coroutines?

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

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

Tail recursive coroutines

What's the tail recursive coroutines?

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

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

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

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

tail call / no tail call

How does tail call work?

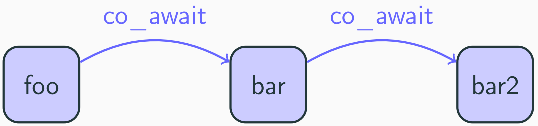
First, how does regular call work?

How does tail call work?

First, how does regular call work?

How does tail call work?

First, how does regular call work?

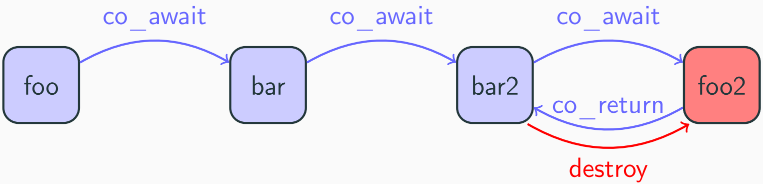


How does tail call work?

First, how does regular call work?

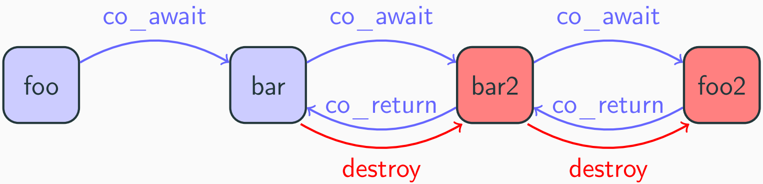
How does tail call work?

First, how does regular call work?



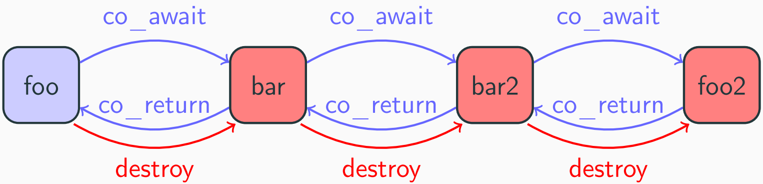
How does tail call work?

First, how does regular call work?



How does tail call work?

First, how does regular call work?



Conclusion:

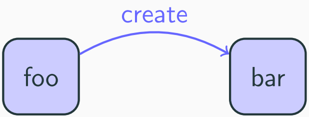
- At peak 4 coroutine frames had to be allocated
- Only after returning to the calling coroutine, called one can be destroyed

How does tail-call work?

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

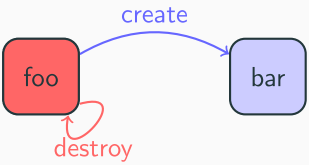
How does tail-call work?

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



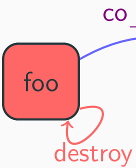
How does tail-call work?

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



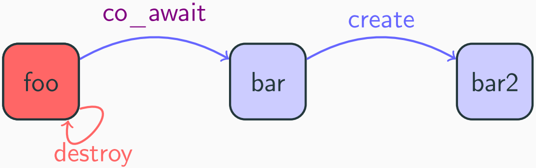
How does tail-call work?

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



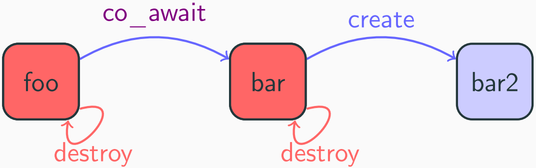
How does tail-call work?

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



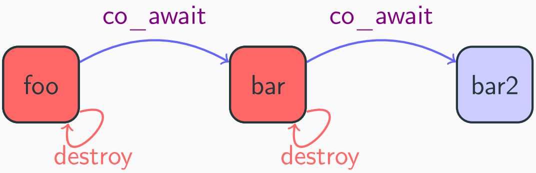
How does tail-call work?

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



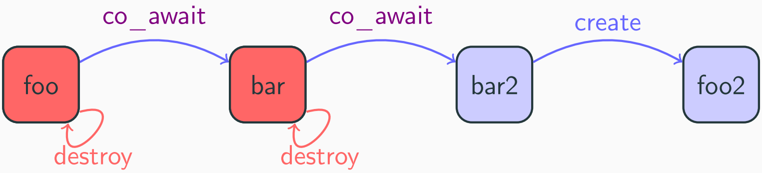
How does tail-call work?

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



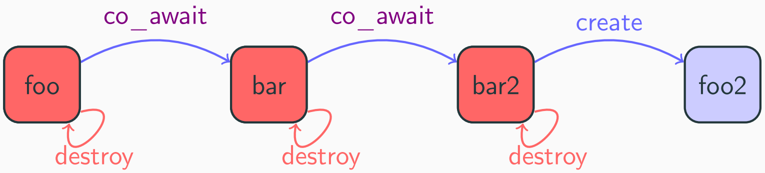
How does tail-call work?

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



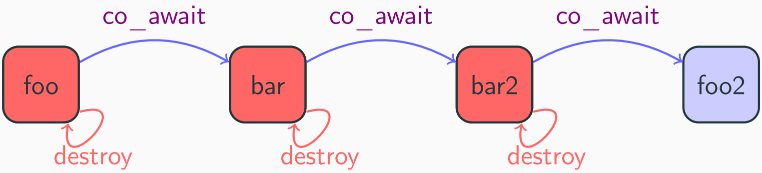
How does tail-call work?

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



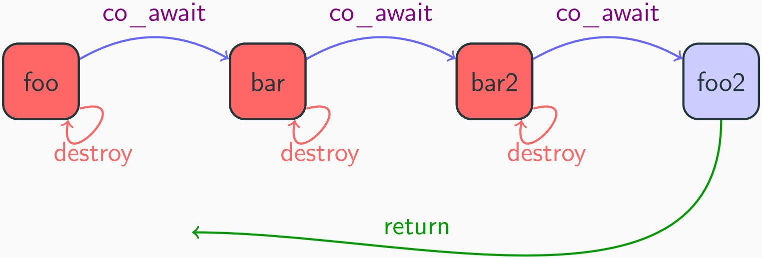
How does tail-call work?

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.



Thank you for your attention!

Special thank you! goes to:

- Gor Nishanov
- Lewiss Baker

for making coroutines

Bibliography and further reading

- 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?

Questions?