Coroutines in the C++

Introduction to the C++ coroutines

Dawid Pilarski

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

Introduction

Agenda

Introduction

Why do we need coroutines?

Why do we need language support for the coroutines?

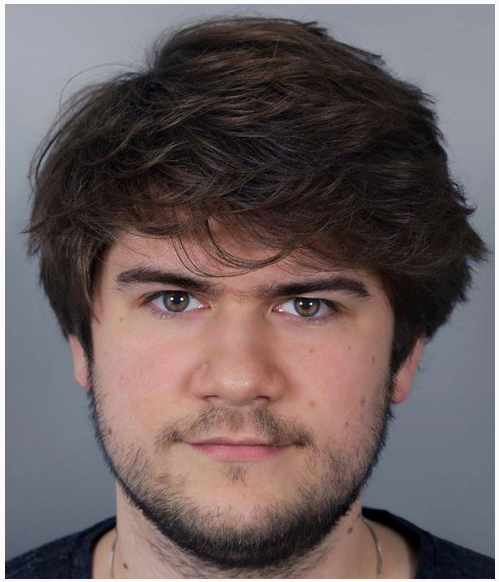
How to implement your own coroutine types?

Questions...

Ask questions any time.
Don't be afraid to interrupt me :)

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

Why do we need 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

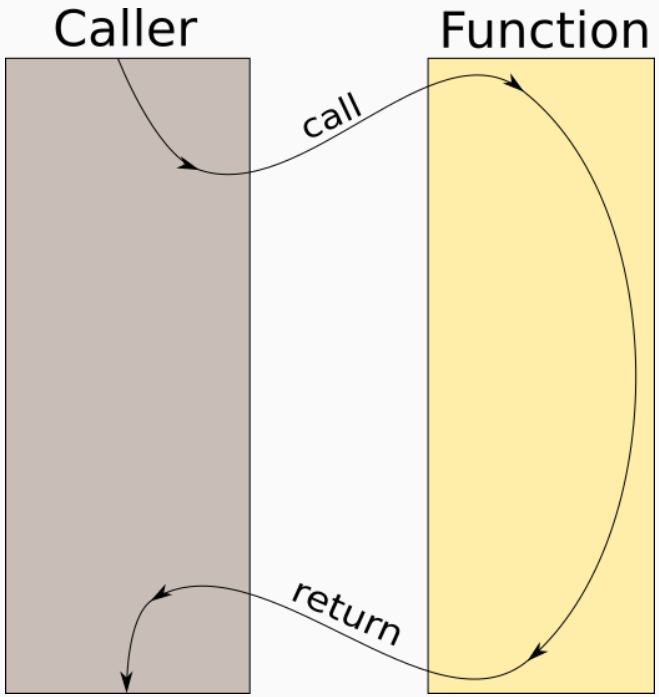
What are the coroutines?

Coroutine can be:

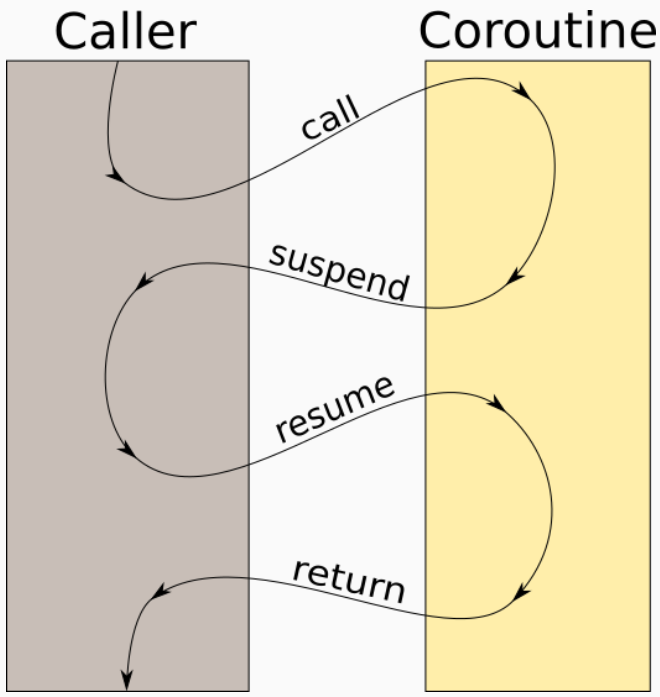
- called
- returned from
- suspended
- resumed from
- created
- destroyed

Coroutine flowchart

Function's flow:



Coroutine flow:



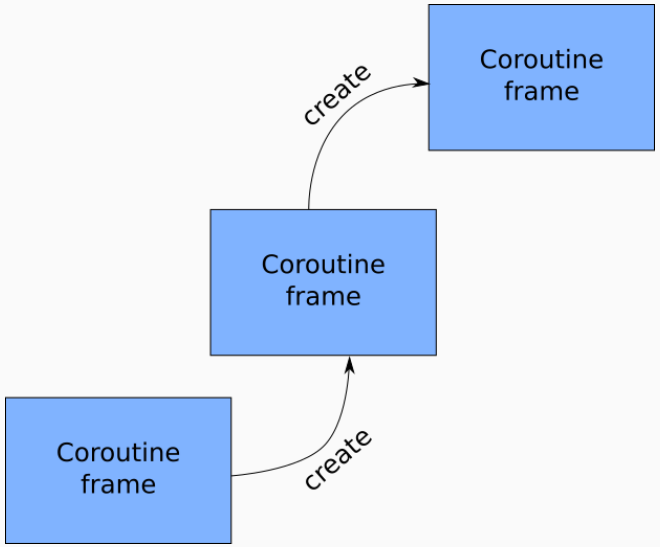
Coroutines use cases

- cooperative multitasking
- generators
- easier state-based computations
- prevent active waiting in the programs

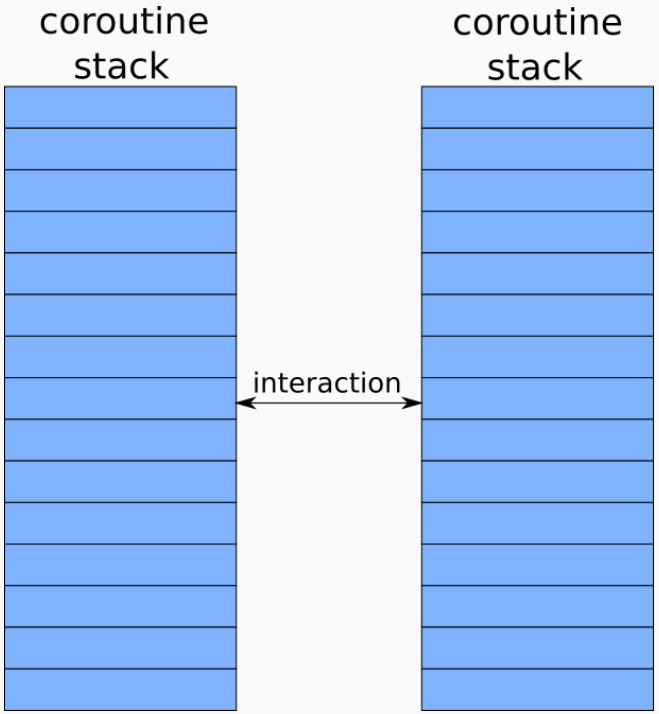
Why do we need language support for the coroutines?

C++ coroutines vs library coroutines

Language based



Library based



C++ vs library - summary

- Need to allocate the stack for the Fiber/Coroutine

C++ vs library - summary

- Need to allocate the stack for the Fiber/Coroutine
- Can be suspended from the top level functions and below

C++ vs library - summary

- Need to allocate the stack for the Fiber/Coroutine
- Can be suspended from the top level functions and below
- Allocation of the memory in advance

C++ vs library - summary

- Need to allocate the stack for the Fiber/Coroutine
- Can be suspended from the top level functions and below
- Allocation of the memory in advance
- One allocation per whole stack.

C++ vs library - summary

Boost.Fiber

- Need to allocate the stack for the Fiber/Coroutine
- Can be suspended from the top level functions and below
- Allocation of the memory in advance
- One allocation per whole stack.

built-in coroutines

- Need to allocate the frame for the Coroutine

C++ vs library - summary

Boost.Fiber

- Need to allocate the stack for the Fiber/Coroutine
- Can be suspended from the top level functions and below
- Allocation of the memory in advance
- One allocation per whole stack.

built-in coroutines

- Need to allocate the frame for the Coroutine
- Can be suspended only from the top level function

C++ vs library - summary

Boost.Fiber

- Need to allocate the stack for the Fiber/Coroutine
- Can be suspended from the top level functions and below
- Allocation of the memory in advance
- One allocation per whole stack.

built-in coroutines

- Need to allocate the frame for the Coroutine
- Can be suspended only from the top level function
- Minimal memory allocations

C++ vs library - summary

Boost.Fiber

- Need to allocate the stack for the Fiber/Coroutine
- Can be suspended from the top level functions and below
- Allocation of the memory in advance
- One allocation per whole stack.

built-in coroutines

- Need to allocate the frame for the Coroutine
- Can be suspended only from the top level function
- Minimal memory allocations
- Multiple allocations

How to implement your own coroutine types?

What's the issue?

Creating custom coroutine type is not easy:

- C++ provides keywords only.

What's the issue?

Creating custom coroutine type is not easy:

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

What's the issue?

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)

What's the issue?

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)

What's the issue?

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);
```

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;`
- `void return_value(V);`

Keywords and promise_type

Promise_type is also responsible for keyword's actions:

- `co_return V;`
- `co_return;`

- `void return_value(V);`
- `void return_void();`

Keywords and promise_type

Promise_type is also responsible for keyword's actions:

- `co_return V;`
- `co_return;`
- `co_yield V;`
- `void return_value(V);`
- `void return_void();`
- `awaitable yield_value();`

Implementing generator: the API

```
template <typename T>
class generator {
public:
  T next();
  ~generator();

  struct promise_type;
private:
  using coro_handle_t = std::coroutine_handle<promise_type>;
  coro_handle_t coro_handle_;
  generator(coro_handle_t handle);
};
```

Implementing generator: the promise_type

```
template <typename T>
struct generator<T>::promise_type {
  std::optional<T> recent_value;

  generator<T> get_return_object() {
    return generator<T>::coro_handle_t::from_promise(*this);
  }

  std::suspend_always initial_suspend() { return {}; }
  std::suspend_always final_suspend() { return {}; }

  void return_void() {}

  std::suspend_always yield_value(T value) {
    recent_value.emplace(value);
    return {};
  }

  void unhandled_exception() {
    throw;
  }
};
```

Implementing generator

```
template <typename T>
generator<T>::generator(coro_handle_t handle) : coro_handle_(handle){}

template <typename T>
generator<T>::~generator(){
        coro_handle_.destroy();
}

template <typename T>
T generator<T>::next(){
  if(coro_handle_.done())
    throw "nothing to resume";

  coro_handle_.resume();
  return *coro_handle_.promise().value_;
};
```

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(T)` support, or
- implement 3 functions:
  - `bool await_ready()`
  - `await_suspend(coroutine_handle<T>)` returning
    - void
    - bool
    - another `coroutine_handle`
  - `T await_resume()`

co_await example - single_consumer_event

```
class single_consumer_event{
  void set(){
    is_set_ = true;
    if(coro_handle_) coro_handle_.resume();
  }

  void reset(){
    is_set_=false;
    coro_handle_=nullptr;
  }

  operator co_await(){
    return awaiter{*this};
  }

private:
    bool is_set_=false;
    std::coroutine_handle<void> coro_handle_;
    struct awaiter;
};
```

co_await example - single_consumer_event

```
struct single_consumer_evern::awaiter{
  single_consumer_event& parent;
  bool await_ready(){
    return parent.is_set_;
  }

  void await_suspend(std::coroutine_handle<void> new_handle){
    parent.coro_handle_ = new_handle;
  }

  void await_resume(){};
};
```

Thank you for attention

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?