# szczecin::cpp

14.03.2019

# Say hello to: C++ Coroutines

PRESENTER:
JACEK NIJAKI <jacek.nijaki@siili.com>

# What we will talk about?

Coroutines-TS
(Technical Specification - N4775)

Included in C++20 draft (Kona, Feb 2019)
by Gor Nishanov <gorn@microsoft.com>

std::experimental
#include <experimental/coroutine>

# What is a Coroutine?

Generalization of a function

Function:
- can be invoked
- can return to the caller

Coroutine, can the same, and more:
- can suspend execution, and return
- later can be resumed from the point it was previously suspended


invoke
Function
return
invoke
suspend
Coroutine
destroy
resume
return

# How does it work? - activation frame

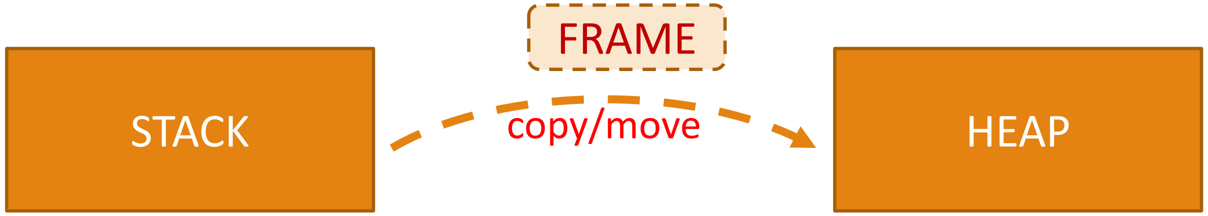
Coroutine can be suspended w/o destroying – stack can’t be used

Coroutine fame is stored on <u>heap</u>

Optimization possible – if lifetime nested in the frame of caller

Activation frame = stack frame + coroutine (heap) frame

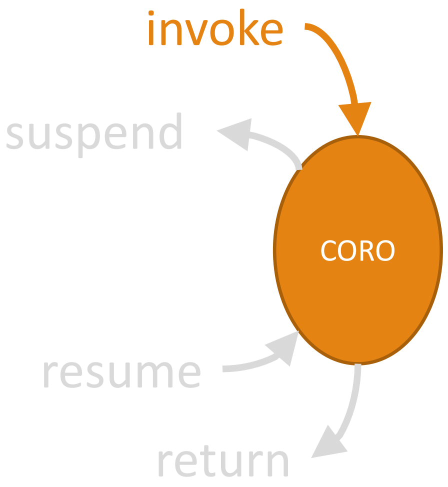
Values are copied or moved to the heap frame



# How does it work? - actions

Invoke

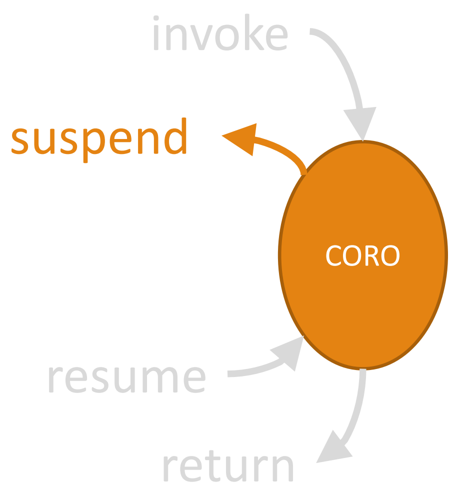
- From caller perspective no difference from function
- Returns to caller when suspended or completed
- Stack frame created same as during ordinary function call
- Finally coroutine creates frame on heap
- Copies params, return address etc. to the coroutine frame



# How does it work? - actions

## Suspend

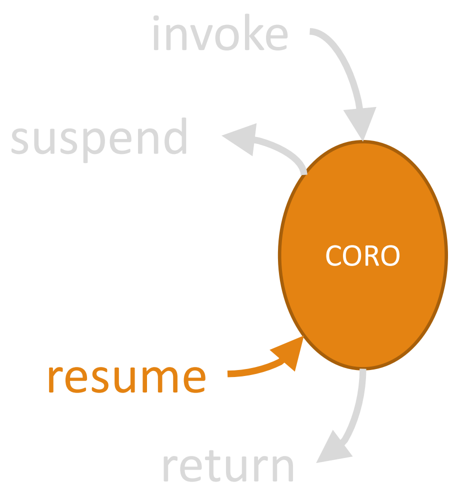
- Possible on defined suspension points
- Drops state to the coroutine frame (including suspension point address)
- When prepared for resumption – considered 'suspended'
- Place for additional logic before returning to the caller (e.g. start real I/O operation)
- Stack frame part is then freed
- Coroutine handle is transferred to the caller (can be used for resumption)



# How does it work? - actions

## Resume

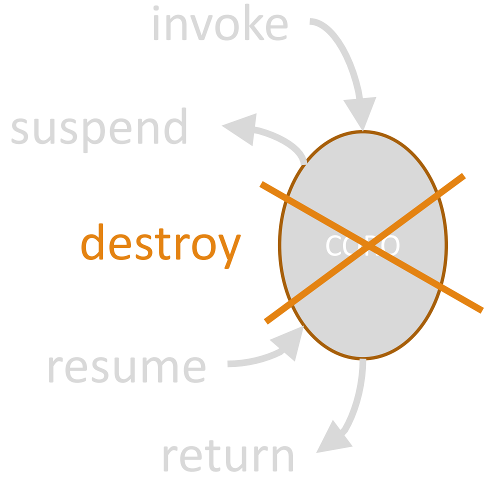
- Done by calling resume() on coroutine handle
- Effectively calls into middle of the function
- Stores stack-frame with caller return address (like normal function)
- Instead of starting the function, execution passed to the point stored in the coroutine-frame



# How does it work? - actions

Destroy

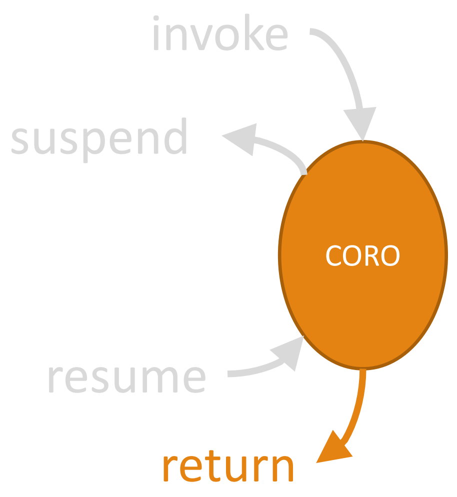
- Destroys coroutine frame w/o resuming
- Can be only performed on suspended coroutine
- Re-activates coroutine activation frame,
- No execution transfer, instead destructors called for variables in scope of last suspension point



# How does it work? - actions

Return

- Different than normal
- Return value is stored somewhere (coroutine specific)
- Can execute additional logic before transferring execution
- Then performs **Suspend** (keeps frame alive) or **Destroy**
- Execution transferred to the caller
- <u>Value passed to the return operation =/= return-value from the coroutine call</u>



SAMPLE: function **`fun()`**, calls a coroutine **`coro(int param)`**

1. before

```
STACK          REGISTERS      HEAP
----------                    -----------
fun()              rsp        -----------
----------         ---
                   rbp
```

SAMPLE: function **fun()**, calls a coroutine **coro(int param)**

2. **coro(42)** is called:

```
STACK              REGISTERS         HEAP
---------------                      -----------
coro()               rsp             -----------
param=42             ---
ret=fun()+0x100      rbp
---------------
fun()
---------------
```

SAMPLE: function **fun()**, calls a coroutine **coro(int param)**

3. coroutine frame allocated

```
STACK               REGISTERS      HEAP
---------------                    -----------
coro()                 rsp         coro()
param=42               ---         param=42
ret=fun()+0x100        rbp         -----------
---------------
fun()
---------------
```

SAMPLE: function **fun()**, calls a coroutine **coro(int param)**

4. Coroutine calls function **subfun()**

```
STACK               REGISTERS      HEAP
---------------                    -----------
subfun()              rsp          coro()
ret=coro()+0x20       ---          param=42
---------------       rbp          -----------
coro()
coroframe
param=42
ret=fun()+0x100
---------------
fun()
---------------
```

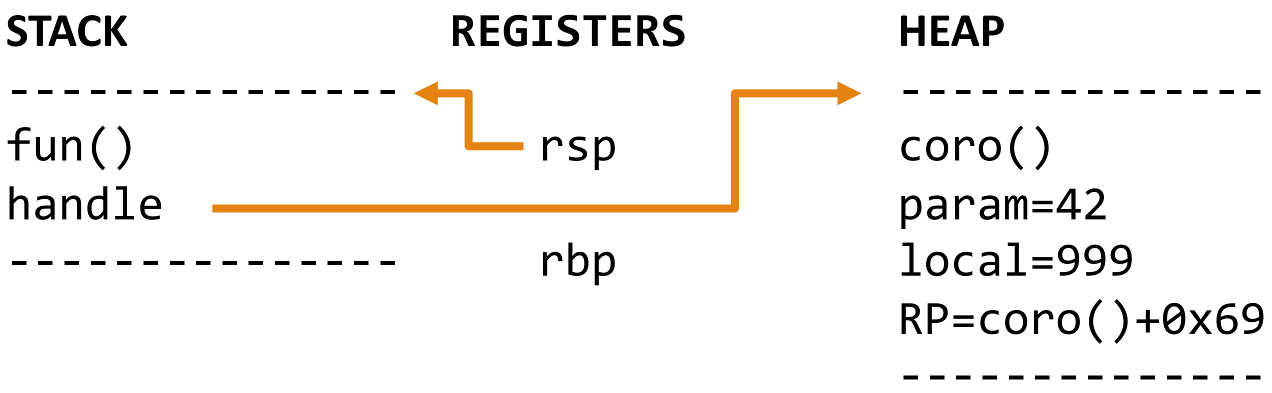
SAMPLE: function **fun()**, calls a coroutine **coro(int param)**

5. **subfun()** returned 999

```
STACK               REGISTERS      HEAP
---------------                    -----------
coro()                 rsp         coro()
param=42               ---         param=42
ret=fun()+0x100        rbp         local=999
---------------                    -----------
fun()
---------------
```

SAMPLE: function **`fun()`**, calls a coroutine **`coro(int param)`**

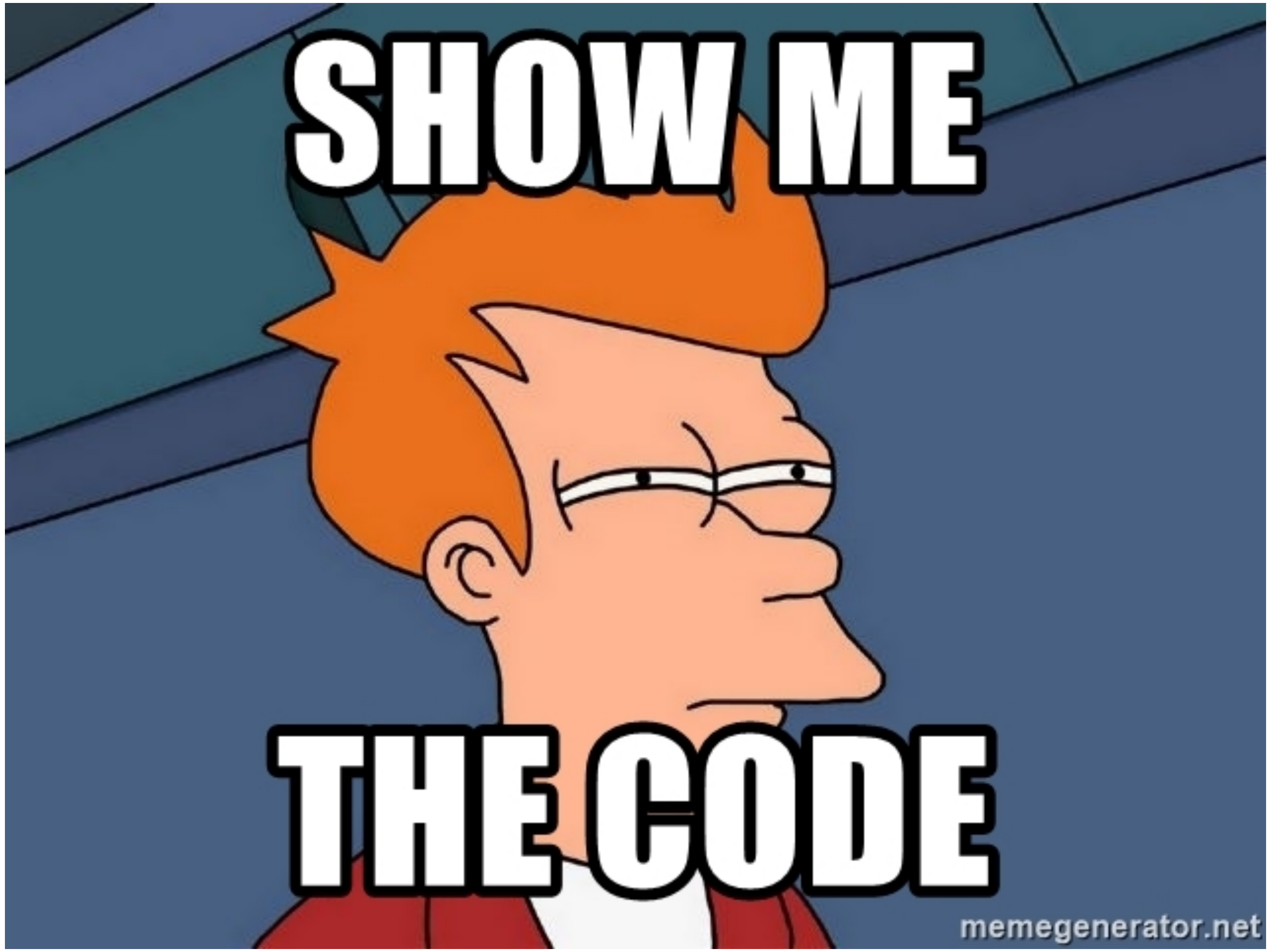
5. **`coro()`** hits suspension point



SAMPLE: function **`fun()`**, calls a coroutine **`coro(int param)`**

5. Some time later… **`coro()`** gets resumed using handle from different function **`fun2()`** (potentially different thread!)

```
STACK               REGISTERS        HEAP
---------------                      -----------
coro()                 rsp           coro()
ret=fun2()+0x91        ---           param=42
---------------        rbp           local=999
fun2()                               -----------
handle
---------------
```


SHOW ME
THE CODE
memegenerator.net

# New keywords

Suspending:

- `co_await`
- `co_yield`

Returning:

- `co_return`

Resuming:

- will get back to this…

# How to create a Coroutine?

Is this a coroutine?

```
std::future<int> do_some_magic();
```

It depends…

Being a coroutine:

- It's an implementation detail
- Function must use (`co_await, co_yield` or `co_return`)
- Has no reflection in function signature/declaration

# How to create a Coroutine?

```cpp
std::future<int> do_some_magic()
{
    return std::async([]
    {
        return 42;
    });
}
```

```cpp
std::future<int> do_some_magic()
{
    int result = co_await std::async([]
    {
        return 42;
    });

    co_return result;
}
```

```cpp
void main()
{
    auto ongoing_magic = do_some_magic();
    int magic_number = ongoing_magic.get();
}
```

# Cool, but why?

You can create:

- Asynchronous tasks
- Generators
- State machines
- Lock-free barriers
- …

<u>Simplify writing asynchronous code!</u>

## Usig <ppltasks.h>

```
void PickImageClick(Platform::Object^ sender,
Windows::UI::Xaml::RoutedEventArgs^ e)
{
    auto picker = ref new FileOpenPicker();
    picker->FileTypeFilter->Append(L".jpg");
    picker->SuggestedStartLocation = PickerLocationId::PicturesLibrary;

    create_task(picker->PickSingleFileAsync()).then([this]
    (Windows::Storage::StorageFile^ file)
    {
        if (nullptr == file)
            return;

        create_task(file->OpenReadAsync()).then([this]
        (Windows::Storage::Streams::IRandomAccessStreamWithContentType^ stream)
        {
            auto bitmap = ref new BitmapImage();
            bitmap->SetSource(stream);
            theImage->Source = bitmap;
            OutputDebugString(L"1. End of OpenReadAsync lambda.\r\n");
        });
        OutputDebugString(L"2. End of PickSingleFileAysnc lambda.\r\n");
    });

    OutputDebugString(L"3. End of function.\r\n");
}
```

## Using C++ coroutines

```
task<void> PickAnImage()
{
    auto picker = ref new FileOpenPicker();
    picker->FileTypeFilter->Append(L".jpg");
    picker->SuggestedStartLocation = PickerLocationId::PicturesLibrary;

    auto file = co_await picker->PickSingleFileAsync();
    if (nullptr == file)
        return;

    auto stream = co_await file->OpenReadAsync();

    auto bitmap = ref new BitmapImage();
    bitmap->SetSource(stream);
    theImage->Source = bitmap;
    OutputDebugString(L"1. End of function.\r\n");
}
```

# Core concepts

Standard introduces new contracts (interfaces) for:

- **Awaitable type**
  - Type which could be co_awaited
- **Promise type**
  - specifies methods for customizing coroutine behavior:
    - what happens when coroutine is called, when returns, etc.
    - customize behavior of `co_await` and `co_yield`
  - communication interface between coroutine and it's caller

<u>Type can be both **Promise** and **Awaitable** at the same time</u>

# Awaitable

```
struct awaitable_type
{
  bool await_ready();
  void await_suspend(coroutine_handle<>);
  auto await_resume(); // becomes a return value from co_await expression
};
```

# co_await magic

```
auto spell = co_await magic_expr
```

```
auto&& awaitable = magic_expr;

if (!awaitable.await_ready())
{
   <suspend coroutine>

   awaitable.await_suspend(coroutine-handle);

   <return-to-caller>
   <resumption-point>
}

return awaitable.await_resume();
```

# How to resume a coroutine?

## Using coroutine handle

```
struct answer_to_life
{
  bool await_ready()
  {
    return false;
  }

  void await_suspend(coroutine_handle<> h) noexcept
  {
    std::thread([h]
    {
      std::this_thread::sleep_for(5s);
      h.resume();
    }).detach();
  }

  auto await_resume()
  {
    return 42;
  }
};
```

```
std::future<int> get_the_answer ()
{
  auto result = co_await answer_to_life(); // 42
  co_return result;
}
```

# What’s inside of coroutine_handle<>?

```
// Type-erased coroutine handle. Can refer to any kind of coroutine.
// Doesn't allow access to the promise object.
template<>
struct coroutine_handle<void>
{
  // Constructs to the null handle.
  constexpr coroutine_handle();

  // Convert to/from a void* for passing into C-style interop functions.
  constexpr void* address() const noexcept;
  static constexpr coroutine_handle from_address(void* addr);

  // Query if the handle is non-null.
  constexpr explicit operator bool() const noexcept;

  // Query if the coroutine is suspended at the final_suspend point.
  // Undefined behaviour if coroutine is not currently suspended.
  bool done() const;

  // Resume/Destroy the suspended coroutine
  void resume();
  void destroy();
};
```

```
// Coroutine handle for coroutines with a known promise type.
// Template argument must exactly match coroutine's promise type.
template<typename Promise>
struct coroutine_handle : coroutine_handle<>
{
  // Access to the coroutine's promise object.
  Promise& promise() const;

  // You can reconstruct the coroutine handle from the promise object.
  static coroutine_handle from_promise(Promise& promise);
};
```

# The Promise

When you write a coroutine that has a body <body-statements>, it's to transformed to something like…

```
{
  co_await promise.initial_suspend();
  try
  {
    <body-statements>

    promise.return_value(42); // assuming co_return 42;
    goto FinalSuspend;
  }
  catch (...)
  {
    promise.unhandled_exception();
  }
FinalSuspend:
  co_await promise.final_suspend();
}
```

# Putting things together

Let’s create a custom coroutine, which can be used like thins:

```
my_custom_coroutine get_the_answer()
{
  auto answer = co_await answer_to_life();
  co_return answer;
}

int main()
{
  auto promise = get_the_answer();
  auto answer = promise.get_result();
  return 0;
}
```

```cpp
using namespace std;
using namespace std::experimental;

struct my_custom_coroutine
{
  struct promise_type
  {
    int _result = 0;

    my_custom_coroutine get_return_object()
    {
      return my_custom_coroutine(coroutine_handle<promise_type>::from_promise(*this));
    }

    auto initial_suspend() { return suspend_never{}; }
    auto final_suspend()   { return suspend_always{}; }

    void return_value(int val)
    {
      _result = val;
    }

    void return_void() {}
  };
  // ...
};
```

Pick one

```
struct my_custom_coroutine
{
  // ...

  coroutine_handle<promise_type> _
  my_custom_coroutine(coroutine_ha
    : _handle(handle)
  {
  }

  ~my_custom_coroutine()
  {
    if (_handle)
    {
      _handle.destroy();
    }
  }

  int get_result()
  {
    return _handle.promise()._resu
  }
};
```

```
my_custom_coroutine get_the_answer()
{
  auto answer = co_await answer_to_life();
  co_return answer;
}


struct answer_to_life
{
  bool await_ready()
  {
    return false;
  }

  void await_suspend(coroutine_handle<> h) noexcept
  {
    std::thread([h]
    {
      std::this_thread::sleep_for(5s);
      h.resume();
    }).detach();
  }

  auto await_resume()
  {
    return 42;
  }
};
```

```
r();
sult();
 for life is: " << result;
```

```
for life is:  0
```

```
r();

sult();
 for life is: " << result;
```

```
for life is: 42
```

# Yielding

Can be used for providing multiple values to the caller, not just one!

`co_yield` is just an abstraction over `co_await`

```
generator<int> produce_int(int start, int end)
{
    for(int val = start; val <= end; ++val)
    {
        co_yield val;
    }
}

int main()
{
    for (auto value : produce_int(5, 10))
    {
        cout << value << endl;
    }
}
```

```
5
6
7
8
9
10
```

# Magic?

```
int main()
{
    for (auto value : produce_int(5, 10))
    {
        cout << value << endl;
    }
}
```

=

```
int main()
{
    generator<int> ints = produce_int(5, 10);
    for (auto it = ints.begin(); it != ints.end(); ++it)
    {
        cout << *it << endl;
    }
}
```

# What happened there?

```
generator_int produce_int(int start, int end)
{
    for(int val = start; val <= end; ++val)
    {
        co_yield val;
    }
}
```

```
generator_int produce_int(int start, int end)
{
    for(int val = start; val <= end; ++val)
    {
        co_await promise.yield_value(val);
    }
}
```

# How to write generator coroutine?

```
struct generator_int
{
    struct promise_type
    {
        const int* _value = nullptr;

        generator_int get_return_object() { /* same as previous*/}

        auto initial_suspend() { return suspend_always{}; }
        auto final_suspend() { return suspend_always{}; }

        auto yield_value(const int& value)
        {
            _value = value;
            return suspend_always{};
        }
    };

    struct iterator : std::iterator<input_iterator_tag, int>
    {
        iterator& operator++(); // if not done, resume coroutine
        const int& operator*(); // return current value
    };
    iterator begin(); // fist resumption, initialize iterator with coroutine handle
    iterator end();   // return empty iterator
};
```

```
void return_value(int val);
void return_void();
auto yield_value(const int & value);
```

# C++ Coroutine design principles

**Scalable**
to billions of concurrent coroutines

**Efficient**
suspend/resume comparable in cost to function call

**Open-ended**
library designers can develop coroutine libs exposing high-level semantics

**Seamless interaction**
with existing facilities with no overhead (e.g. C-style APIs)

**Usable**
in environments where exceptions a are not available or forbidden

# Current status

Part of C++20 draft (from Feb 2019)

Supported in:
- Visual Studio 2015 SP2+ (/await)
- Clang 5+ (-fcoroutines-ts -stdlib=libc++)
- GCC – not supported, in progress

Standard library:
- Nothing in standard at this point
- Visual Studio extends `std::future<>` and brings `std::experimental::generator<>`
- [WIP] provide standard implementations (e.g. **CppCoro** library)

# Wrapping up

- Coroutine additionally can: **Suspend, Resume** or **Destroy**
- Activation frame is stored on **heap**
- To create coroutine use: **co_await, co_yield** or **co_return** in function implementation
- You need **coroutine handle** to resume it
- To create an expression that could be co_awaited: implement **Awaitable** type contract
- To create custom coroutine type: implement struct with **Promise** type contract
- Use **co_yield** to return multiple values from a single coroutine
- “Just” language feature, use e.g. **cppcoro** to get high-level primitives, such as **generators** or **tasks**
- **Try it at home!**

# Useful links

C++ Extensions for Coroutines
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/n4775.pdf

Core Coroutines
http://open-std.org/JTC1/SC22/WG21/docs/papers/2018/p1063r1.pdf

https://github.com/lewissbaker/cppcoro

https://lewissbaker.github.io/

“Introduction to C++ Coroutines”: https://youtu.be/ZTqHjjm86Bw

“Coroutine TS a new way of thinking”: https://youtu.be/pc-MDA1IXqk

“Nano-coroutines to the Rescue!”: https://youtu.be/j9tlJAqMV7U

```
co_await questions();
```

# QUIZ TIME!