C++ Expert training: C++ coroutines [**n4736**]

ir J.P. Lammertink

2020, March 4th

Program

- 13:00h Theory of coroutines
- 14:45h [snack]
- 15:00h Research workshop part 1
- 17:30h [dinner]
- 18:30h Research workshop part 2
- 21:00h End

Each research workshop part

- Choose and share your subject of interest
- Start individually or form a small team
- Research
- A round of 5 minutes presentation (individually)

Fixed presentation format

- My research subject is ...
- I discovered ...
- I still would like to know ...

Goals

- Learn about C++ coroutines [n4736]
- Learn about the CppCoro library
- Learn from each other
- Bonding
- Inspire
- Enjoy

My motivation

I'm wondering...

- frameworks in C++, such as COM, CORBA, SSCF, DDF, QT, ASIO
- they all impose a specific execution architecture
- ...is that OO?

My motivation

Theorem

A framework that includes:

- *serialization/deserializaion*
- *marshalling*
- *inter-process and inter-node communication*
- *asynchronous/overlapped IO/communication*
- *event processing*
- *TCP/IP and serial communication*

always *consists of tightly coupled components.*

Motivation

Theorem

C++20 has enough expression power to decompose such frameworks into loosely coupled components, without introducing (needless) overhead.

Sources

Lewis Baker

- 3 articles on GitHub [**Baker2018**]
- cppcoro library [**CppCoro**]

C++ Coroutines TS (N4736)

What is a coroutine

Definition
A coroutine is a generalisation of a function that allows the function to be *suspended* and then later *resumed*.

Definition
In C++20, a function is a coroutine, when it has at least one of the following keywords in its body:

- `co_await`
- `co_yield`
- `co_return`

From compiler-implementation-perspective, a function is a coroutine, for which the 'activation frame' solely resides on the stack.

‘Normal’ function call

<Call>

- Prepare caller for suspension
- Allocate memory for the activation frame
- Pass parameters
- Write address of resume-point of the caller
- Jump to the begin address of the function
- (Optionally, create local variables)

<Return>

- Optionally, store the return value
- Destroy local variables
- Optionally, destroy parameter objects
- Free memory for the coroutine activation frame
- Restore activation frame of caller
- Jump to the resume-point of caller

Stack

Stack

- A brief history of the stack; Sten Henriksson [**Henriksson**]
- Etymologically related haystack
- First design with stack: PERM II computer 1957
- First realisation: KDF9 computer 1959
- The push down automaton was introduced by Newell et al in 1959
- 'Stack' apears in first publication of E.W. Dijkstra in 1960 (before named 'pushdown store', 'LIFO list' or 'cellar storage')
- VAX computer implemented the CISC instructions pop and push 1978
- Enabler for recursive function calling

Example recursive Fibonacci

```
int recursive_fibonacci(int n)
{
    if (n <= 1) return n;
    else
    {
        int n_2 = recursive_fibonacci(n - 2);
        int n_1 = recursive_fibonacci(n - 1);
        return n_1 + n_2;
    }
}
```

Example recursive Fibonacci main

```
bool recursive_fibonacci_max(int n, int const max, int& f)
{
    if (n <= 1)
    {
        f = n;
        return f <= max;
    }
    else
    {
        int n_2 = recursive_fibonacci(n - 2);
        int n_1 = recursive_fibonacci(n - 1);
        f = n_1 + n_2;
        return n_2 < max - n_1;
    }
}

int main()
{
    for (int f, n = 0; recursive_fibonacci_max(n, 10'000, f); ++n)
    {
        std::cout << f << std::endl;
    }
}
```

Coroutine frame

The complete activation frame of a coroutine consists of two parts:

- Stack part
- Persistent part (often heap; accessable with handle)

<Suspend>

A coroutine function can be suspended in the middle of a function at suspension points indicated with `co_await` or `co_yield`

- Prepare coroutine for suspension
- Write address of resume-point
- (some programmer defined logic can be executed. ⇒ immediatelly resume/continue)
- The stack part of the coroutine activation frame is freed
- Restore activation frame of caller
- Jump to the resume-point of caller

<Resume>

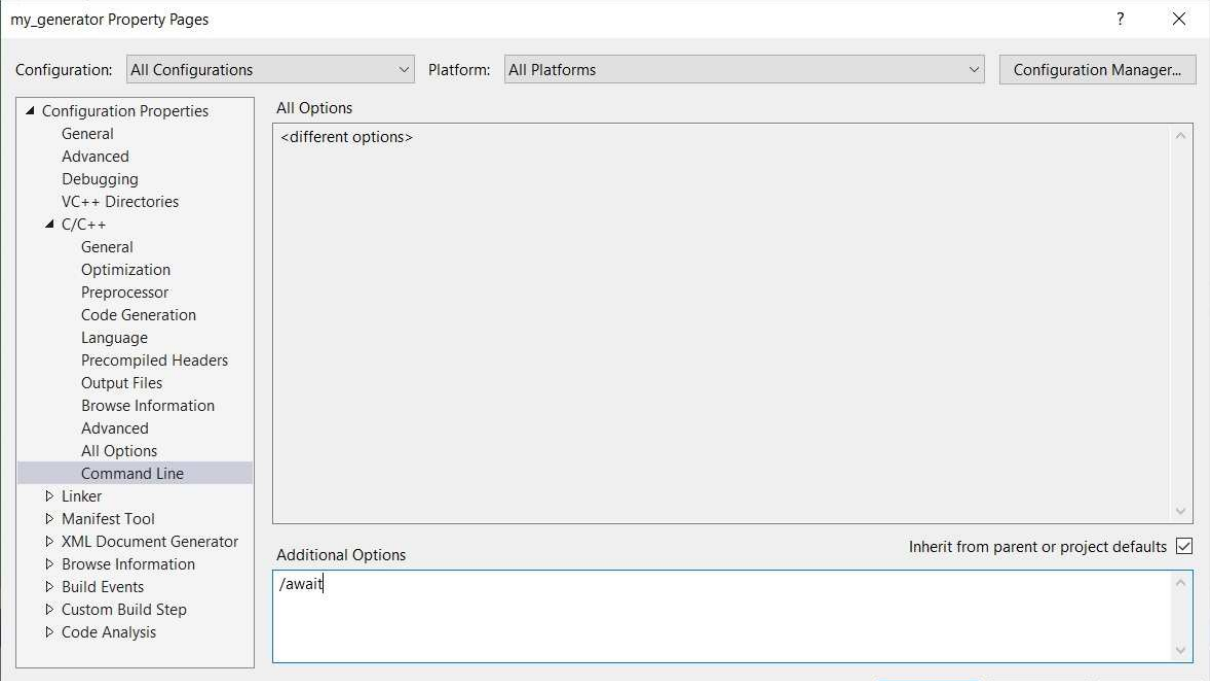
A coroutine can be resumed by calling the `void resume()` method on the coroutine-frame handle.

- Prepare caller for suspension
- Allocate memory for stack part of the activation frame
- Write address of resume-point of the caller
- Jump to the resume-point of the coroutine

<Destroy>

The ‘destroy’ operation destroys the coroutine frame without resuming execution of the coroutine.

- Prepare caller for suspension
- Allocate memory for stack part of the activation frame
- Write address of resume-point of the caller
- Destruct all local variables in-scope
- Free memory for the coroutine activation frame (all)
- Restore activation frame of caller
- Jump to the resume-point of caller


my_generator Property Pages
Configuration: All Configurations
Platform: All Platforms
Configuration Manager...
Configuration Properties
General
Advanced
Debugging
VC++ Directories
C/C++
General
Optimization
Preprocessor
Code Generation
Language
Precompiled Headers
Output Files
Browse Information
Advanced
All Options
Command Line
Linker
Manifest Tool
XML Document Generator
Browse Information
Build Events
Custom Build Step
Code Analysis
All Options
<different options>
Additional Options
Inherit from parent or project defaults
/await

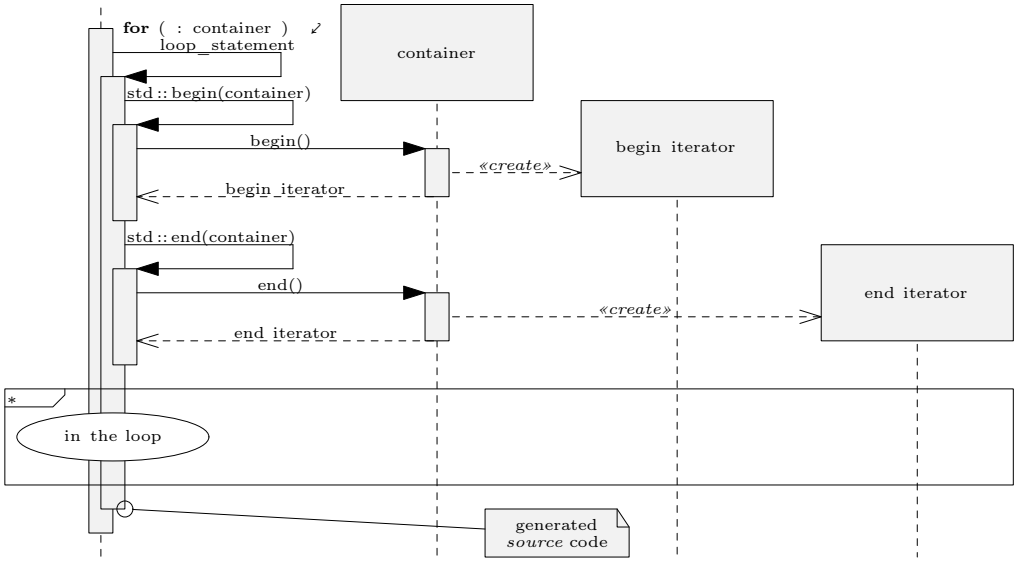
Example main

```
#include <iostream>

int main()
{
    int_generator_t int_generator(my_fibonacci_generator(10'000));
    int f;
    while (int_generator.get(f))
    {
        std::cout << f << std::endl;
    }
}
```

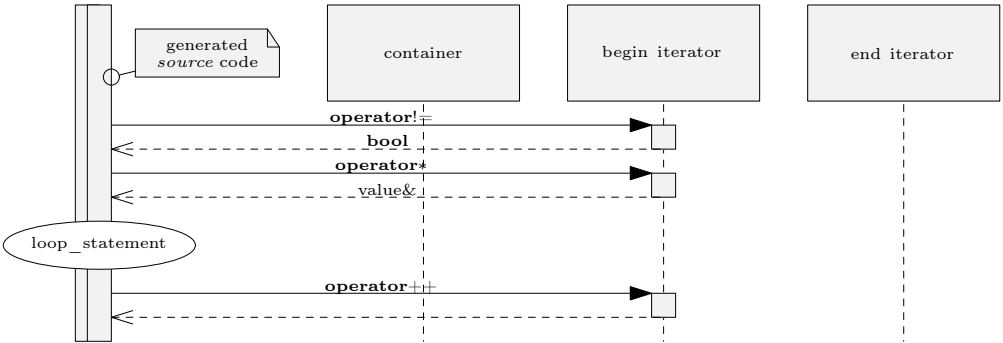
Example coroutine

```
#include <experimental/resumable>

int_generator_t my_fibonacci_generator(int max)
{
    int n_2 = 0;
    if (max < n_2) co_return;
    co_yield n_2;
    int n_1 = 1;
    if (max < n_1) co_return;
    co_yield n_1;
    while (true)
    {
        if (max - n_1 < n_2) co_return;
        int n_0 = n_2 + n_1;
        co_yield n_0;
        n_2 = n_1;
        n_1 = n_0;
    }
}
```

Range-based for loop when ‘range expression’ is a container



Range-based for loop; in the loop



Range-based for loop; in code

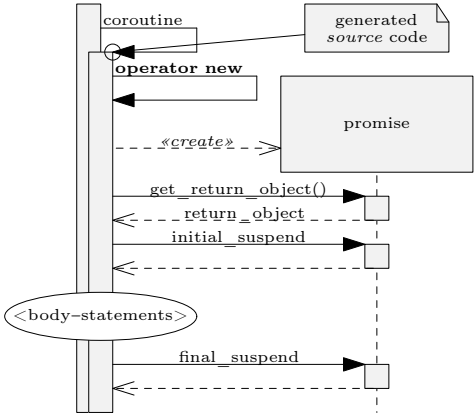
```
for ( init-statement(optional) range_declaration : range_expression )
loop_statement
```

⇒

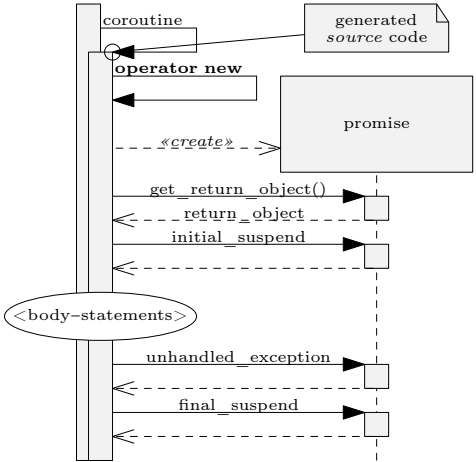
```
{
    init-statement
    auto && __range = range_expression ;
    auto __begin = begin_expr ;
    auto __end = end_expr ;
    for ( ; __begin != __end; ++__begin)
    {
        range_declaration = *__begin;
        loop_statement
    }
}
```

‘Coroutine’ function call

No exception



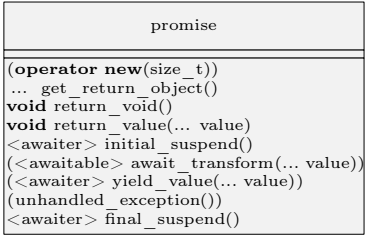
Exception



‘Coroutine’ function call

```
T some_coroutine(P param)
{
    auto* f = new coroutine_frame(std::forward<P>(param));
    auto returnObject = f->promise.get_return_object();
    // This call will return when the coroutine gets to the first
    // suspend-point or when the coroutine runs to completion.
    coroutine_handle<...>::from_promise(f->promise).resume();
    return returnObject;
}

coroutine_handle<...>::resume()
{
    co_await promise->initial_suspend();
    try
    {
        <body-statements>
    }
    catch (...)
    {
        promise->unhandled_exception();
    }
FinalSuspend:
    co_await promise.final_suspend();
}
```

promise_type class



Notion of “promise”

- Sometimes “promise” is like `std::promise` part of promise-future pair
  reminder:
  calling `set_value(...)` on `std::promise`
  results in `get()` to return on `std::future`
- For some use cases “promise” is *not* like `std::promise`
- ⇒ “promise” is like “coroutine state controller”

Example promise_type

```
struct promise_type
{
    int_generator_t get_return_object()
    { ... }
    ... initial_suspend()
    { ... }
    ... final_suspend()
    { ... }
    ...
};
```

Two options to obtain the promise_type

1. Return type of coroutine/function has member-type `promise_type`
2. Using `coroutine_traits`
   asume coroutine `foo` with signature: `T0 foo(T1, T2, ..., Tn)`
   then `promise_type` can be defined under:
   `typename coroutine_traits<T0, T1, T2, ..., Tn>::promise_type;`

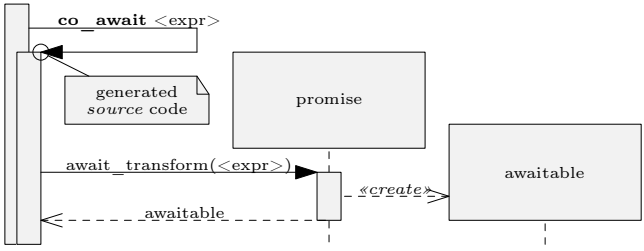
Example promise_type class

```
struct int_generator_t
{
    struct promise_type
    {
        using coro_handle_t =
           std::experimental::coroutine_handle<promise_type>;
        int_generator_t get_return_object()
        {
           return int_generator_t(coro_handle_t::from_promise(*this));
        }
        ...
    };
    int_generator_t(promise_type::coro_handle_t coro_handle)
        : m_coro_handle(coro_handle)
    {}
    promise_type::coro_handle_t m_coro_handle;
};
```

co_await <expr>

`await_transform` exists

⇒



`await_transform` does *not* exist

⇒

`awaitable = <expr>`

co_await <expr>

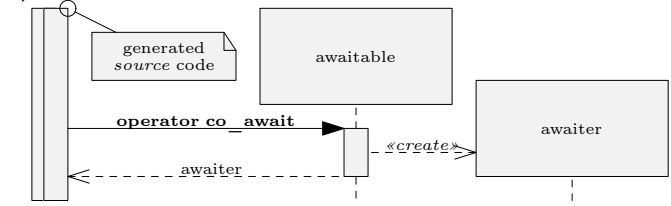
```
co_await <expr>
```

⇒

```
template<typename P, typename T>
decltype(auto) get_awaitable(P& promise, T&& expr)
{
    if constexpr (has_any_await_transform_member_v<P>)
        return promise.await_transform(expr);
    else
        return expr;
}
...

{
    auto&& value = <expr>;
    auto&& awaitable = get_awaitable(promise, value);
    ...
```

operator co_await awaitable

`operator co_await` exists
⇒



`operator co_await` does *not* exist
⇒
`awaiter = ↙ awaitable`

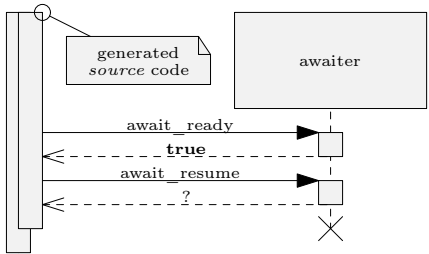
operator co_await awaitable

operator co_await

⇒

```
template<typename Awaitable>
decltype(auto) get_awaiter(Awaitable&& awaitable)
{
    if constexpr (has_member_operator_co_await_v<Awaitable>)
        return awaitable.operator co_await();
    else if constexpr (has_non_member_operator_co_await_v<Awaitable
        &&>)
        return operator co_await(awaitable);
    else
        return awaitable;
}
...

{
    auto&& awaitable = get_awaitable(promise, value);
    auto&& awaiter = get_awaiter(awaitable);
```

await_ready returning true

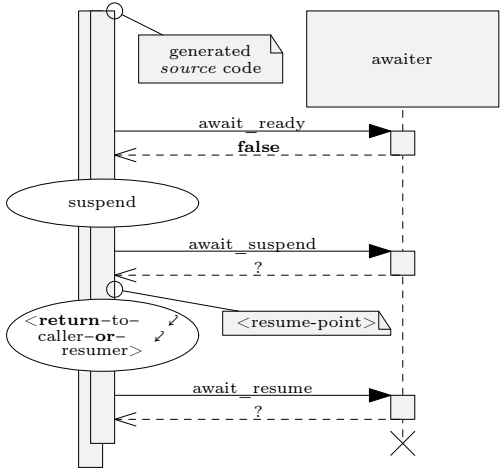


await_ready returning true

```
{
    auto&& value = <expr>;
    auto&& awaitable = get_awaitable(promise, value);
    auto&& awaiter = get_awaiter(awaitable);
    if (!awaiter.await_ready())
        {
            ... /* skipped when await_ready() returns true */
        }
    return awaiter.await_resume();
}
```

Note that `co_await <expr>` may return a value, when `await_resume()` does.

await_ready returning false



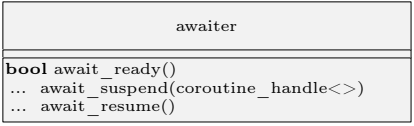
await_ready returning false

```
{
    ...
    if (!awaiter.await_ready())
    {
        using handle_t = std::experimental::coroutine_handle<P>;
        using await_suspend_result_t = decltype(awaiter.await_suspend( ↙
            handle_t::from_promise(p)));
        <suspend-coroutine>
        if constexpr (std::is_void_v<await_suspend_result_t>)
        {
            awaiter.await_suspend(handle_t::from_promise(p));
            <return-to-caller-or-resumer>
        }
        else
        {
            ...
        }
<resume-point>
    }
    return awaiter.await_resume();
}
```

await_ready returning false, await_suspend returning bool

```
{
    ...
    if (!awaiter.await_ready())
    {
        using handle_t = std::experimental::coroutine_handle<P>;
        using await_suspend_result_t = decltype(awaiter.await_suspend( ↙
            handle_t::from_promise(p)));
        <suspend-coroutine>
        if constexpr (std::is_void_v<await_suspend_result_t>)
        ...
        else
        {
            if (awaiter.await_suspend(handle_t::from_promise(p)))
            {
                <return-to-caller-or-resumer>
            }
        }
<resume-point>
    }
    return awaiter.await_resume();
}
```

awaiter class



Example usage of awaiter class

```
struct int_generator_t
{
    struct promise_type
    {
        std::experimental::suspend_always initial_suspend()
        {
            return std::experimental::suspend_always();
        }
        std::experimental::suspend_always final_suspend()
        {
            return std::experimental::suspend_always();
        }
        ...
    };
    ...
};
```

std::experimental::suspend_always

```
struct suspend_always
{
    bool await_ready() noexcept
    {
        return false;
    }
    void await_suspend(coroutine_handle<>) noexcept {}
    void await_resume() noexcept {}
};
```

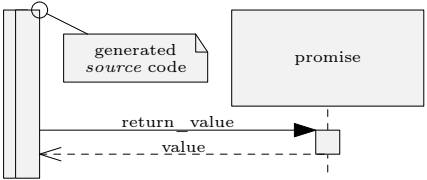
co_yield

`co_yield <expr>`
translates into
`co_await promise.yield_value(<expr>)`

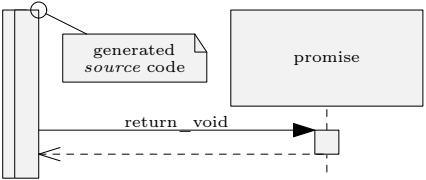
Example co_yield

```
struct int_generator_t
{
    struct promise_type
    {
        ...
        std::experimental::suspend_always yield_value(int v)
        {
            m_v = v;
            return std::experimental::suspend_always();
        }
        ...
        int m_v;
    };
    ...
};
```

co_return



`goto FinalSuspend;`



`goto FinalSuspend;`

If execution runs off the end of a coroutine without a `co_return` statement, then this is equivalent to having a `co_return;` at the end of the function body.

Example return_void

```
struct int_generator_t
{
    struct promise_type
    {
        ...
        void return_void(){}
        ...
    };
    ...
};
```

Handle and promise

- Handle ⇒ promise: `m_coro_handle.promise()`
- Promise ⇒ handle: `coro_handle_t::from_promise(*this)`
- Resuming coroutine: `coro_handle.resume()`
- Check wheter coroutine finished: `coro_handle.done()`

Example handle and promise

```
struct int_generator_t
{
    struct promise_type
    {
        using coro_handle_t = std::experimental::coroutine_handle<
            promise_type>;
        int_generator_t get_return_object()
        {
            return int_generator_t(coro_handle_t::from_promise(*this));
        }
        ...
        int m_v;
    };
    ...
    int_generator_t(promise_type::coro_handle_t coro_handle)
        : m_coro_handle(coro_handle)
    {}
    bool get(int& v)
    {
        m_coro_handle.resume();
        v = m_coro_handle.promise().m_v;
        return !m_coro_handle.done();
    }
    promise_type::coro_handle_t m_coro_handle;
};
```

Example output

```
0
1
1
2
3
5
8
13
21
34
55
89
144
233
377
610
987
1597
2584
4181
6765
```

Example fancy main

```
int main()
{
    for (auto f : my_fibonacci_generator(10'000))
    {
        std::cout << f << std::endl;
    }
}
```

Example range based for logic

```
struct int_generator_t
{
    struct promise_type
    {
        ...
        bool resume()
        {
            coro_handle_t coro_handle(coro_handle_t::from_promise(*  ↙
                this));
            coro_handle.resume();
            return !coro_handle.done();
        }
    };
    struct iterator
    {
        iterator(promise_type& promise) : m_promise(promise)   {}
        bool operator!=(int) const { return m_promise.resume(); }
        int operator*()      const { return m_promise.m_v;      }
        iterator operator++()      { return *this;              }
        promise_type& m_promise;
    };
    ...
    iterator begin(){ return iterator(m_coro_handle.promise()); }
    int end()       { return 0;                                 }
};
```

cppcoro::generator

```
#include <cppcoro/generator.hpp>

cppcoro::generator<int> fibonacci_generator(int max)
{
    int n_2 = 0;
    if (max < n_2) co_return;
    co_yield n_2;
    int n_1 = 1;
    if (max < n_1) co_return;
    co_yield n_1;
    while (true)
    {
        if (max - n_1 < n_2) co_return;
        int n_0 = n_2 + n_1;
        co_yield n_0;
        n_2 = n_1;
        n_1 = n_0;
    }
}

int main()
{
    for (auto f : fibonacci_generator(10'000))
    {
        std::cout << f << std::endl;
    }
}
```

cppcoro

- Coroutine Types
  - task<T>
  - shared_task<T>
  - *generator<T>*
  - recursive_generator<T>
  - async_generator<T>
- Awaitable Types
  - single_consumer_event
  - single_consumer_async_auto_reset_event
  - async_mutex
  - async_manual_reset_event
  - async_auto_reset_event
  - async_latch
  - sequence_barrier
  - multi_producer_sequencer
  - single_producer_sequencer

cppcoro

- Functions
  - sync_wait()
  - when_all()
  - when_all_ready()
  - fmap()
  - schedule_on()
  - resume_on()
- Cancellation
  - cancellation_token
  - cancellation_source
  - cancellation_registration
- Schedulers and I/O
  - static_thread_pool
  - io_service and io_work_scope
  - file, readable_file, writable_file
  - read_only_file, write_only_file, read_write_file
- Networking
  - socket
  - ip_address, ipv4_address, ipv6_address
  - ip_endpoint, ipv4_endpoint, ipv6_endpoint

Speed test

- 5 pieces of work: 1, 3, 5 interleave with 2 and 4
- implementation with hardcoded interleaving
- implementation with coroutines and events
- implementation with `std::async`

Speed test output

```
test 1, hardcoded interleaving:              600ns
test 2, coroutine event implementation:     1100ns
test 3, async implementation:               12000ns
async to coroutine ratio:                      22.8
```

With custom new operator

```
test 1, hardcoded interleaving:              600ns
test 2, coroutine event implementation:     1000ns
test 3, async implementation:               12000ns
async to coroutine ratio:                      28.5
```

References

[1] [Baker, 2018] Lewis S. Baker.
Asymmetric Transfer
https://lewissbaker.github.io/2017/09/25/coroutine-theory
https://lewissbaker.github.io/2017/11/17/understanding-operato
https://lewissbaker.github.io/2018/09/05/understanding-the-pro

[2] [CppCoro] Lewis S. Baker.
CppCoro - A coroutine library for C++

[3] [n4736]
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/n4736

[4] [Henriksson] Sten Henriksson.
A brief history of the stack
Computer Science Department, Lund University, Lund, Sweden

Tech talks

- CppCon 2016: James McNellis “Introduction to C++ Coroutines”
  `https://www.youtube.com/watch?v=ZTqHjjm86Bw`
- CppCon 2016: Gor Nishanov “C++ Coroutines: Under the covers”
  `https://www.youtube.com/watch?v=8C8NnE1Dg4A`