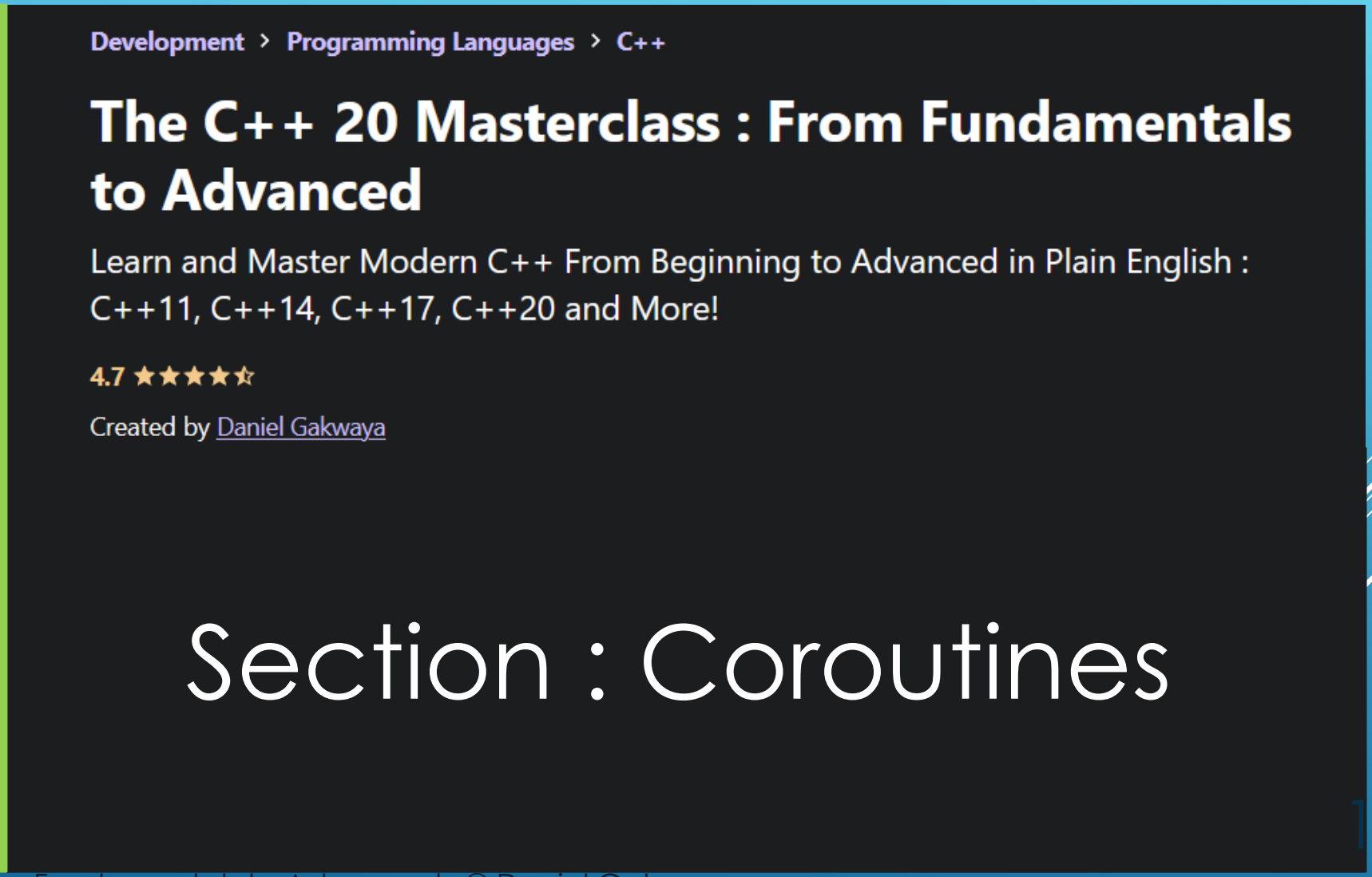
Slides



Section : Coroutines

Slide intentionally left empty

C++20 Coroutines

Concepts

Ranges

Coroutines

Modules

- Coroutines are a generalization of functions in C++
- They are designed to make writing asynchronous code much easier

Function

- Can be called
- Can return something

Coroutine

- Can be called
- Can return something
- Can be paused
- Can be resumed

```
generator<int> generate_numbers()
{
    co_yield 10; // Return 10 and pause
    std::cout << "After stop point #1" << std::endl;

    co_yield 20;
    std::cout << "After stop point #2" << std::endl;

    co_yield 30;
    std::cout << "After stop point #3" << std::endl;
}
```

C++ user code

Coroutine types

Coroutine Infrastructure

```
generator<int> generate_numbers()
{
    co_yield 10; // Return 10 and pause
    std::cout << "After stop point #1" << std::endl;

    co_yield 20;
    std::cout << "After stop point #2" << std::endl;

    co_yield 30;
    std::cout << "After stop point #3" << std::endl;
}
```

generator<T>

C++20

Lazy computations

```
auto f = some_coroutine(); // Store computation info in f

//Use f to manipulate the computation
```

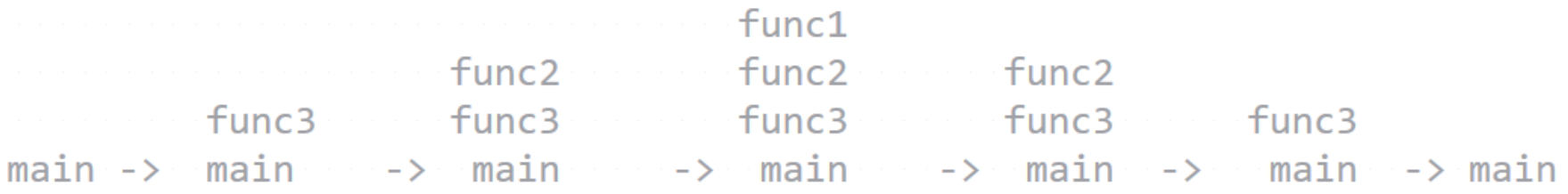
Slide intentionally left empty

Coroutine workflow

Functions

```
#include <iostream>

void func1(){
    //Doing stuff
}

void func2(){
    double a {56};
    double b {100};
    func1();
}

void func3(){
    func2();
}

int main(int argc, char **argv)
{
    func3();
    return 0;
}
```

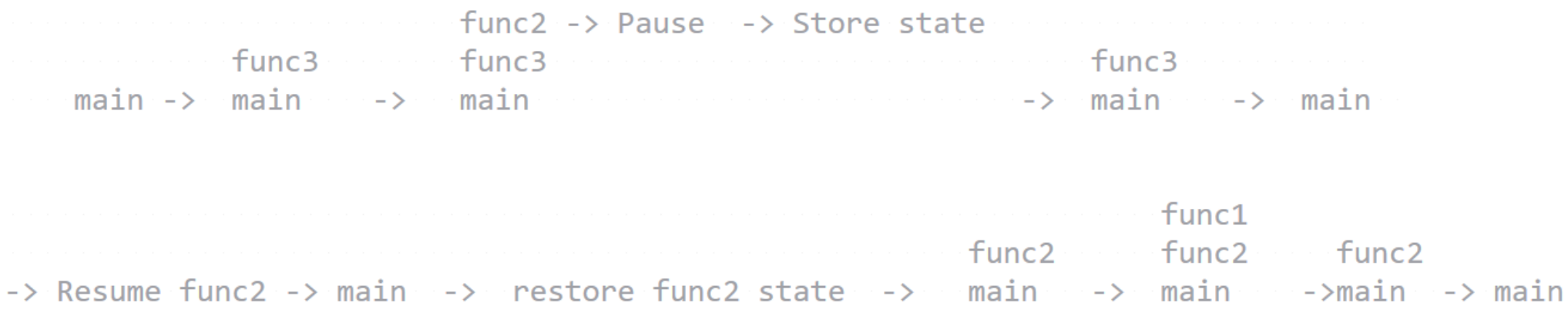
Call stack


main
main
func3
main
func2
func3
main
func1
func2
func3
main
func2
func3
main
func3
main
main

Coroutines

```
#include <iostream>

void func1(){
    //Doing stuff
}

void func2(){
    double a {56};
    double b {100};
    //Suspension point
    func1();
}

void func3(){
    func2();
}

int main(int argc, char **argv)
{
    func3();
    return 0;
}
```

Coroutines : call stack [pause –resume]



Slide intentionally left empty

Coroutine keywords

. C++ 20 introduces three kewords that help pause and resume coroutines.

    . co_yield : suspends the execution and returns a value

    . co_return : completes execution and optionally returns a value

    . co_await : suspends the execution until resumed

. If a function has one of those keywords, it becomes a coroutine. There is no other special syntax for coroutines.

. It's not every function in C++ that can be a coroutine. The functions below can't be coroutines.

- Constexpr functions
- constructors
- destructors
- the main function

. It's not every function in C++ that can be a coroutine. The functions below can't be coroutines.

- Constexpr functions
- constructors
- destructors
- the main function

Coroutine keywords can’t show up in these functions

co_yield

```
#include <iostream>

coro[int] func1(){
    co_yield 45;
    co_yield 46;
    co_yield 47;
    co_return 48;
}

int main(int argc, char **argv)
{
    auto f1 = func1(); // f1 can be thought of as a handle to the computation.
                       // it's called a coroutine type, a generator to be exact in this case.
    std::cout << f1() << std::endl; // 45 . Numbers are generated lazily, on the fly as we need them
    std::cout << f1() << std::endl; // 46
    std::cout << f1() << std::endl; // 47
    std::cout << f1() << std::endl; // 48
    std::cout << f1() << std::endl; // UB : Function yields three numbers and we
                                    // want to get 4 out of it

    return 0;
}
```

co_yield

```
#include <iostream>


coro[int] func2(){
    int start{1};
    while(true){
        co_yield start++;
    }
}

int main(int argc, char **argv)
{

    //func2 generates numbers indefinitely:
    auto f2 = func2(); // f2 is also a handle to the computation

    for(auto i{}; i< 10;++i){ // We just take 10 elements
        std::cout << f2() << std::endl; // 1,2,3,4,5,6,7,8,9,10
    }

    return 0;
}
```

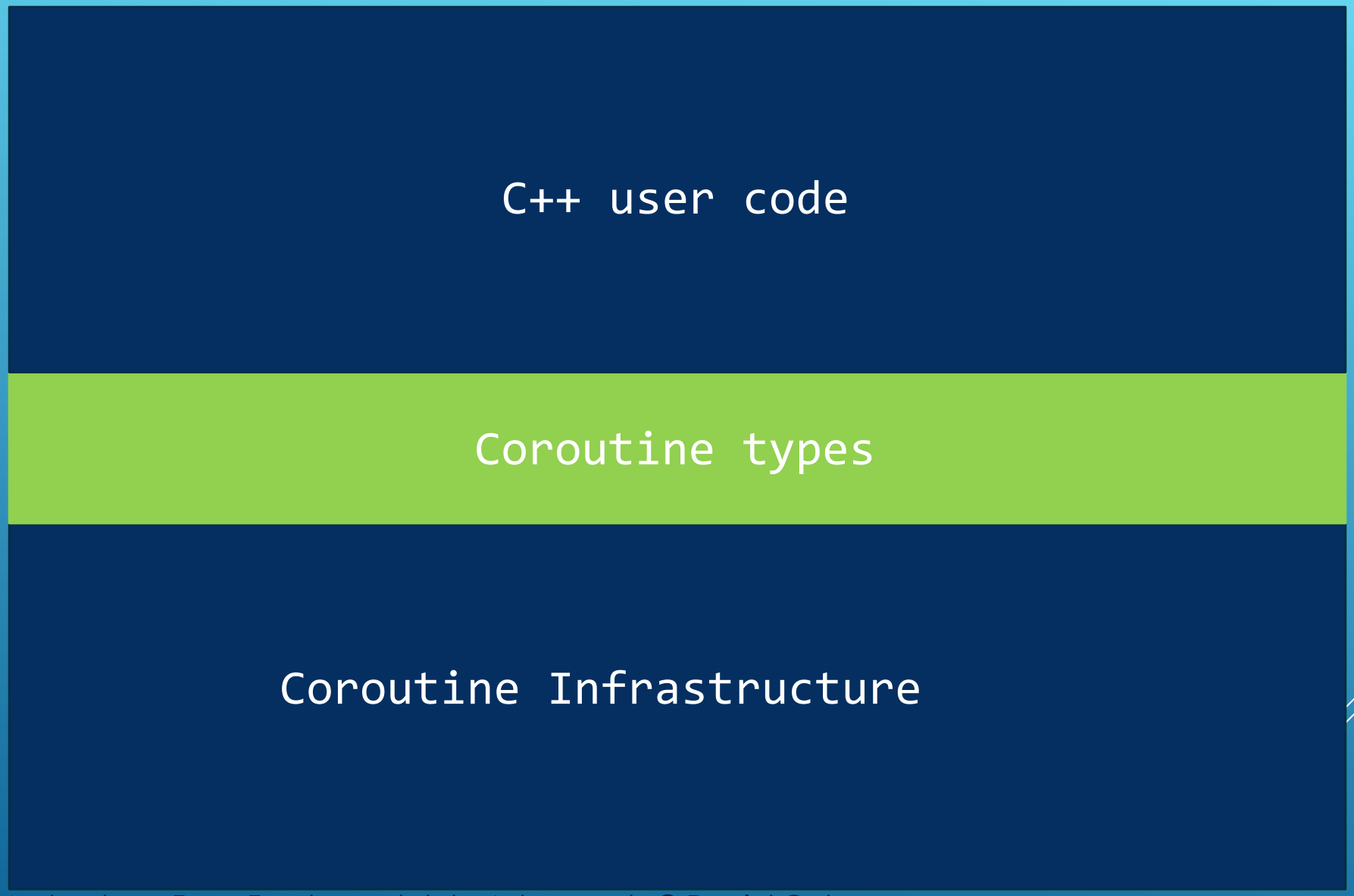
co_return

```
#include <iostream>

coro[int] func3(){
    co_return 55;
}

int main(int argc, char **argv)
{

    auto f3 = func3();
    std::cout << f3() << std::endl;

    return 0;
}
```

co_await

```
coro[int] do_work() {
    std::cout << "Doing first thing... " << std::endl;
    co_await std::suspend_always{}; // suspension point #1
    std::cout << "Doing second thing..." << std::endl;
    co_await std::suspend_always{}; // Suspension point #2
    std::cout << "Doing Third thing..." << std::endl;
}

int main(int argc, char **argv)
{
    auto task = do_work(); //Coroutine suspended on call

    task.resume(); // Runs and hits suspension point #1
    task.resume(); // Runs and hits suspension point #2
    task.resume(); // Runs and hits End of coroutine.
    task.resume(); // DISASTER. RESUMING WHAT YOU HAVEN'T PAUSED.

    std::cout << "Done!" << std::endl;
    return 0;
}
```

So how do we actually run this code?


C++ user code
Coroutine types
Coroutine Infrastructure

Slide intentionally left empty

Coroutine Infrastructure

```
generator<int> generate_numbers()
{
    co_yield 10; // Return 10 and pause
    std::cout << "After stop point #1" << std::endl;

    co_yield 20;
    std::cout << "After stop point #2" << std::endl;

    co_yield 30;
    std::cout << "After stop point #3" << std::endl;
}
```

C++ user code

Coroutine types

Coroutine Infrastructure

```
generator<int> generate_numbers()
{
    co_yield 10; // Return 10 and pause
    std::cout << "After stop point #1" << std::endl;

    co_yield 20;
    std::cout << "After stop point #2" << std::endl;

    co_yield 30;
    std::cout << "After stop point #3" << std::endl;
}
```

generator<T>

C++20


Promise type
Coroutine handle
Awaiter

```
struct CoroType {
    struct promise_type {
        CoroType get_return_object() { return CoroType(this); }
        std::suspend_always initial_suspend() { return {}; }
        std::suspend_always final_suspend() noexcept{ return {}; }
        void unhandled_exception() noexcept
        {
            std::rethrow_exception(std::current_exception());
        }
        void return_void(){};
    };
    CoroType(promise_type* p)
         : m_handle(std::coroutine_handle<promise_type>::from_promise(*p)) {}
    ~CoroType()
     {
         std::cout << "Handle destroyed..." << std::endl;
          m_handle.destroy();
     }
    std::coroutine_handle<promise_type>   m_handle;
};
```

Method	Description
get_return_object(){}	Return value of coroutine type
initial_suspend(){}	Whether function is suspended upon call
final_suspend(){}	Whether coroutine state is destroyed at last suspension point
Unhandled_exception(){}	Handling exceptions thrown into coroutines
Return_value(value){}	Enables the co_return v; syntax
Return_void(){}	Enables the co_return; syntax
Yield_value(value){}	Enables the co_yield v; syntax

Method	Description
bool await_ready(){}	Whether the co_await expression suspends. If false is returned, then await_suspend is called, to (mostly) suspend
void await_suspend(){}	May suspend the coroutine, or schedule the coroutine state for destruction
void await_resume(){}	May return the result of the entire co_await expression

Awaitables

```
struct suspend_never {
    constexpr bool await_ready() const noexcept { return true; }
    constexpr void await_suspend(coroutine_handle<>) const noexcept {}
    constexpr void await_resume() const noexcept {}
};

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

Awaiter

It is possible to create your own awaiters, things that can act as operands to the co_await operator, by setting up structs/classes that overload the co_await operator

Custom Awaitables

```
struct My_Awaitable {
    auto operator co_await(){ return std::suspend_always{}; }
};
```

```
CoroType do_work() {
    std::cout << "Doing first thing... " << std::endl;
    co_await std::suspend_always{};
    std::cout << "Doing second thing..." << std::endl;
    co_await std::suspend_always{};
    std::cout << "Doing Third thing..." << std::endl;
}
```

- C++ 20 doesn't provide actual usable coroutine types like CoroType
- It provides the low level infrastructure to build them (promises, awaitables, coroutine handles,…
- Building your own coroutine types is not recommended. It's only reserved for hard core, highly experienced library developers who really know what they're doing
- It is expected that C++23 will provide high level coroutine types built into C++, ready to use just by including some headers
- If you want to use them know, there are third party libraries that can help, like cppcoro and some others

Slide intentionally left empty

co_await

C++ user code

Coroutine types

Coroutine Infrastructure

```
CoroType do_work() {
    std::cout << "Doing first thing... " << std::endl;
    co_await std::suspend_always{};
    std::cout << "Doing second thing..." << std::endl;
    co_await std::suspend_always{};
    std::cout << "Doing Third thing..." << std::endl;
}
```

CoroType

C++20

Keep in mind

- What we're going to do here works for modern compilers with C++ 20 enabled and support for coroutines
- gcc and msvc have been tested :
  - gcc 10 and gcc 11 require the –fcoroutines switch
  - msvc 2019 (version 16.8.3) and upwards don't require any switch
- Online compilers are also a convenient option :
  - https://wandbox.org is a good choice

```
struct CoroType {
    struct promise_type {
        CoroType get_return_object() { return CoroType(this); }
        std::suspend_always initial_suspend() { return {}; }
        std::suspend_always final_suspend() noexcept{ return {}; }
        void unhandled_exception() noexcept
        {
            std::rethrow_exception(std::current_exception());
        }
        void return_void(){};
    };
    CoroType(promise_type* p)
         : m_handle(std::coroutine_handle<promise_type>::from_promise(*p)) {}
    ~CoroType()
     {
         std::cout << "Handle destroyed..." << std::endl;
          m_handle.destroy();
     }
    std::coroutine_handle<promise_type>   m_handle;
};
```

Awaitables

```
struct suspend_never {
    constexpr bool await_ready() const noexcept { return true; }
    constexpr void await_suspend(coroutine_handle<>) const noexcept {}
    constexpr void await_resume() const noexcept {}
};

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

Putting it all together

```
CoroType do_work() {
    std::cout << "Doing first thing... " << std::endl;
    co_await std::suspend_always{};
    std::cout << "Doing second thing..." << std::endl;
    co_await std::suspend_always{};
    std::cout << "Doing Third thing..." << std::endl;
}


int main(int argc, char **argv)
{

    auto task = do_work(); //Coroutine suspended on call
    task.m_handle(); // This resumes the couroutine. When next suspension point is hit it pauses
    task.m_handle.resume();
    task.m_handle.resume();
    // task.m_handle.resume(); // RECIPE FOR DISASTER. RESUMING WHAT YOU HAVEN'T PAUSED

    std::cout << "Done!" << std::endl;
    return 0;
}
```

Slide intentionally left empty

co_yield

C++ user code

Coroutine types

Coroutine Infrastructure

```
CoroType do_work() {
    co_yield 1;
    co_yield 2;
    co_yield 3;
}
```

generator<T>

C++20

```
struct CoroType {
    struct promise_type {
        int m_value;
        CoroType get_return_object() { return this; }
        std::suspend_always initial_suspend() { return {}; }
        std::suspend_always final_suspend() { return {}; }
        void unhandled_exception() noexcept
        {
            std::rethrow_exception(std::current_exception());
        }
        std::suspend_always yield_value(int val) {
            m_value = val;
            return {};
        }
    };
    CoroType(promise_type* p) : m_handle(std::coroutine_handle<promise_type>::from_promise(*p)) {}
    ~CoroType() { m_handle.destroy(); }
    std::coroutine_handle<promise_type>   m_handle;
};
```

co_yield 2;

co_await promise.yield_value(2);

Awaitables

```
struct suspend_never {
    constexpr bool await_ready() const noexcept { return true; }
    constexpr void await_suspend(coroutine_handle<>) const noexcept {}
    constexpr void await_resume() const noexcept {}
};

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

Awaiter

It is possible to create your own awaiters, things that can act as operands to the co_await operator, by setting up structs/classes that overload the co_await operator

Custom Awaitables

```
struct My_Awaitable {
    auto operator co_await(){ return std::suspend_always{}; }
};
```

Putting it all together

```
CoroType do_work() {
    co_yield 1;
    co_yield 2;
    co_yield 3;
}

int main(int argc, char **argv)
{
    auto task = do_work(); //Coroutine suspended on call
    task.m_handle(); // This resumes the couroutine. When next suspension point is hit it pauses
    std::cout << "value : " << task.m_handle.promise().m_value << std::endl;

    task.m_handle(); // This resumes the couroutine. When next suspension point is hit it pauses
    std::cout << "value : " << task.m_handle.promise().m_value << std::endl;

    task.m_handle(); // This resumes the couroutine. When next suspension point is hit it pauses
    std::cout << "value : " << task.m_handle.promise().m_value << std::endl;

    std::cout << "Done!" << std::endl;
    return 0;
}
```

Slide intentionally left empty

co_return

C++ user code

Coroutine types

Coroutine Infrastructure

```
generator<int> generate_numbers()
{
    co_yield 10; // Return 10 and pause
    std::cout << "After stop point #1" << std::endl;

    co_yield 20;
    std::cout << "After stop point #2" << std::endl;

    co_yield 30;
    std::cout << "After stop point #3" << std::endl;
}
```

generator<T>

C++20

```
struct CoroType {
    struct promise_type {
        int m_value;
        CoroType get_return_object() { return this; }
        std::suspend_always initial_suspend() { return {}; }
        std::suspend_always final_suspend() { return {}; }
        void unhandled_exception() noexcept
        {
            std::rethrow_exception(std::current_exception());
        }
        std::suspend_always yield_value(int val) {
            m_value = val;
            return {};
        }
        void return_value(int val){
             m_value = val;
         }
        void return_void() {
        }
    };
    CoroType(promise_type* p) : m_handle(std::coroutine_handle<promise_type>::from_promise(*p)) {}
    ~CoroType() { m_handle.destroy(); }
    std::coroutine_handle<promise_type>   m_handle;
};
```

Putting it all together

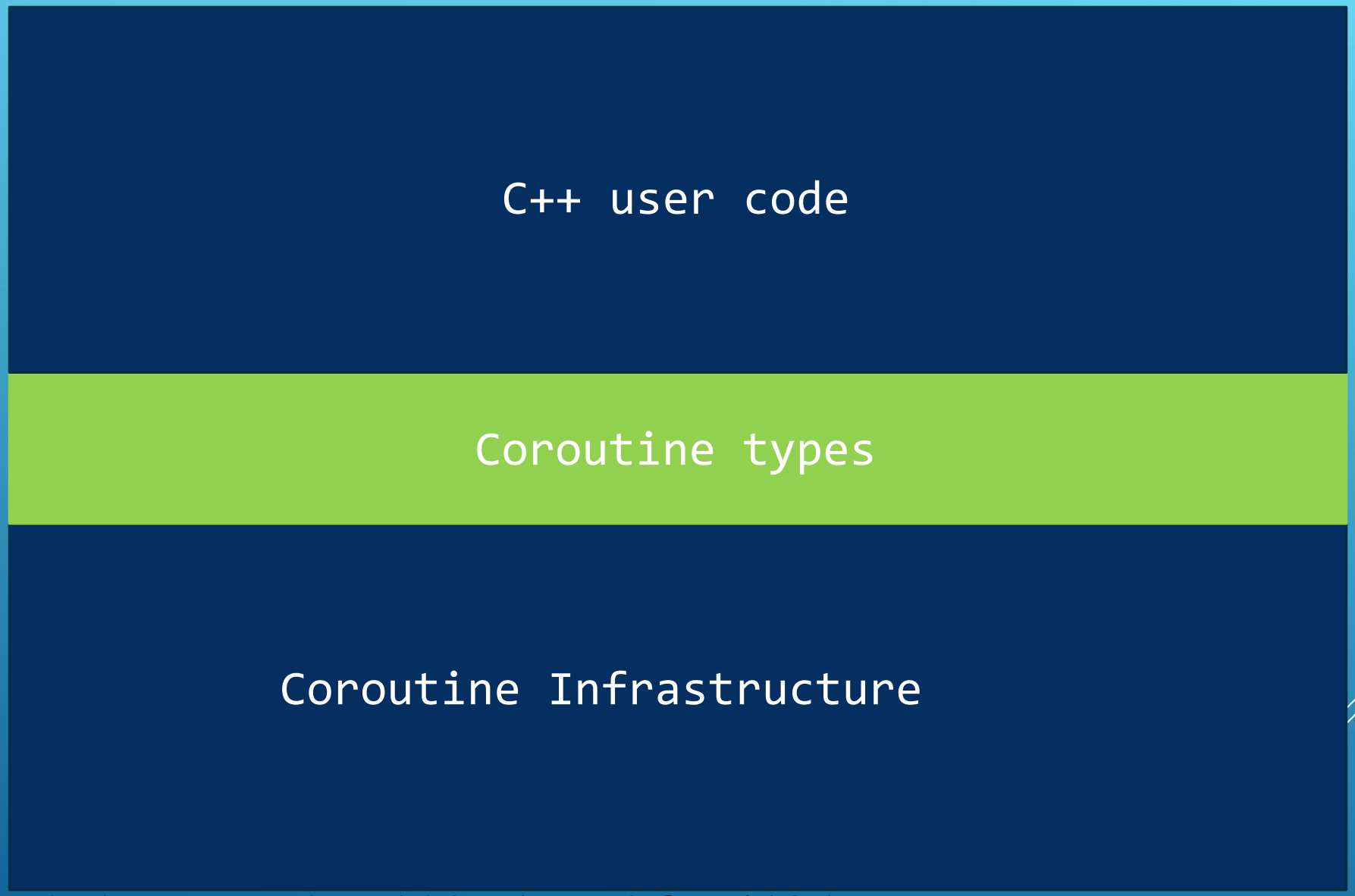
```
CoroType do_work() {
    co_yield 1;
    co_yield 2;
    co_yield 3;
    co_return;
}
int main(int argc, char **argv)
{
    auto task = do_work(); //Coroutine suspended on call

    task.m_handle(); // This resumes the couroutine. When next suspension point is hit it pauses
    std::cout << "value : " << task.m_handle.promise().m_value << std::endl;

    task.m_handle(); // This resumes the couroutine. When next suspension point is hit it pauses
    std::cout << "value : " << task.m_handle.promise().m_value << std::endl;

    task.m_handle(); // This resumes the couroutine. When next suspension point is hit it pauses
    std::cout << "value : " << task.m_handle.promise().m_value << std::endl;

    task.m_handle(); // This resumes the couroutine. When next suspension point is hit it pauses
    std::cout << "value : " << task.m_handle.promise().m_value << std::endl;
    return 0;
}
```

- C++ 20 doesn't provide actual usable coroutine types like generator<T>
- It provides the low level infrastructure to build them (promises, coroutine frames, coroutine handles,…
- Building your own coroutine types is not recommended. It's only reserved for hard core, highly experienced library developers who really know what they're doing
- It is expected that C++23 will provide high level coroutine types built into C++, ready to use just by including some headers
- If you want to use them know, there are third party libraries that can help, like cppcoro
- Before we use cppcoro though, we need to build it and load it into our C++ project
- In the next chapter, we explore basic info and knowledge you need to create and use third party libraries in C++, like cppcoro

Slide intentionally left empty

Custom coroutine types : Generator

```
generator<int> generate_numbers()
{
    co_yield 10; // Return 10 and pause
    std::cout << "After stop point #1" << std::endl;

    co_yield 20;
    std::cout << "After stop point #2" << std::endl;

    co_yield 30;
    std::cout << "After stop point #3" << std::endl;
}
```


C++ user code
Coroutine types
Coroutine Infrastructure

C++ user code

Coroutine types

Coroutine Infrastructure

```
generator<int> generate_numbers()
{
    co_yield 10;
    std::cout << "After stop point #1" << std::endl;

    co_yield 20;
     std::cout << "After stop point #2" << std::endl;

    co_yield 30;
     std::cout << "After stop point #3" << std::endl;
    co_return 40;
}
```

generator<T>

C++20

```
template <typename T>
struct generator
{
    struct promise_type
    {
        T m_value;
        /* ...
    };
    explicit generator(promise_type& p)
        : m_handle(std::coroutine_handle<promise_type>::from_promise(p)){}
    generator() = default;
    ~generator()
    { ...
    T operator()() { ...
private:
    std::coroutine_handle<promise_type> m_handle = nullptr;
};
```

```
struct promise_type
{
    T m_value;

    auto get_return_object() { return generator{ *this }; }
    auto initial_suspend() noexcept { return std::suspend_always{}; }
    auto final_suspend() noexcept { return std::suspend_always{}; }

    void unhandled_exception() noexcept
    {
        std::rethrow_exception(std::current_exception());
    }

    auto yield_value(T const& v)
    {
        m_value = v;
        return std::suspend_always{};
    }

    void return_void() {}
};
```

```
template <typename T>
struct generator
{
    struct promise_type
    { ...
    explicit generator(promise_type& p)
        : m_handle(std::coroutine_handle<promise_type>::from_promise(p)){}
    generator() = default;
    ~generator()
    {
        if (m_handle)
        {
            m_handle.destroy();
        }
    }
    T operator()() {
        assert(m_handle != nullptr);
        m_handle.resume();
        return (m_handle.promise().m_value);
    }
private:
    std::coroutine_handle<promise_type> m_handle = nullptr;
};
```

```
generator<int> generate_numbers()
{
    co_yield 10; // Return 10 and pause
    std::cout << "After stop point #1" << std::endl;

    co_yield 20;
    std::cout << "After stop point #2" << std::endl;

    co_yield 30;
    std::cout << "After stop point #3" << std::endl;
}

generator<int> infinite_number_stream(int start = 0)
{
    auto value = start;
    for (int i = 0;; ++i)
    {
        co_yield value;
        ++value;
    }
}
```

Slide intentionally left empty

Third Party Coroutine Types

C++ user code

Coroutine types

Coroutine Infrastructure

```
generator<int> generate_numbers()
{
    co_yield 10; // Return 10 and pause
    std::cout << "After stop point #1" << std::endl;

    co_yield 20;
    std::cout << "After stop point #2" << std::endl;

    co_yield 30;
    std::cout << "After stop point #3" << std::endl;
}
```

generator<T>

C++20

- Cppcoro
- https://github.com/Quuxplusone/coro

#include “third_party_coro_type.h”

```
unique_generator<int> generate_numbers()
{
    std::cout << "generate_numbers starting" << std::endl;
    co_yield 10; // Return 10 and pause
    std::cout << "After stop point #1" << std::endl;

    co_yield 20;
    std::cout << "After stop point #2" << std::endl;

    co_yield 30;
    std::cout << "After stop point #3" << std::endl;
    std::cout << "generate_numbers ending" << std::endl;
}
```

```
unique_generator<int> infinite_number_stream(int start = 0)
{
    auto value = start;
    for (int i = 0;; ++i)
    {
        std::cout << "In infinite_number stream..." << std::endl;
        co_yield value;
        ++value;
    }
}


unique_generator<int> range(int first, int last)
{
    while (first != last) {
        co_yield first++;
    }
}
```

```
auto g1 = generate_numbers();

std::cout << "value : " << g1() << std::endl;
std::cout  << "value : " << g1() << std::endl;
std::cout << "value : " << g1() << std::endl;
```

No function call operator in <unique_generator>

Changing the interface of unique_generator

```
template<class Ref, class Value = std::decay_t<Ref>>
class unique_generator {
public:
    /* ...
    auto value(){
        coro_.resume();
        return coro_.promise().get();
    }
    auto operator()(){
        coro_.resume();
        return coro_.promise().get();
    }
private:
    explicit unique_generator(handle_t coro) noexcept :
        coro_(coro){}
    handle_t coro_;
};

#endif // INCLUDED_CORO_UNIQUE_GENERATOR_H
```

Slide intentionally left empty

Coroutines : Summary

C++ user code

Coroutine types

Coroutine Infrastructure

```
generator<int> generate_numbers()
{
    co_yield 10;
    std::cout << "After stop point #1" << std::endl;

    co_yield 20;
     std::cout << "After stop point #2" << std::endl;

    co_yield 30;
     std::cout << "After stop point #3" << std::endl;
    co_return 40;
}
```

generator<T>

C++20

- co_await
- co_yield
- co_return

```
template <typename T>
struct generator
{
    struct promise_type
    {
        T m_value;
        /* ...
    };
    explicit generator(promise_type& p)
        : m_handle(std::coroutine_handle<promise_type>::from_promise(p)){}
    generator() = default;
    ~generator()
    { ...
    T operator()() { ...
private:
    std::coroutine_handle<promise_type> m_handle = nullptr;
};
```

```
struct promise_type
{
    T m_value;

    auto get_return_object() { return generator{ *this }; }
    auto initial_suspend() noexcept { return std::suspend_always{}; }
    auto final_suspend() noexcept { return std::suspend_always{}; }

    void unhandled_exception() noexcept
    {
        std::rethrow_exception(std::current_exception());
    }

    auto yield_value(T const& v)
    {
        m_value = v;
        return std::suspend_always{};
    }

    void return_void() {}
};
```

```
template <typename T>
struct generator
{
    struct promise_type
    { ...
    explicit generator(promise_type& p)
        : m_handle(std::coroutine_handle<promise_type>::from_promise(p)){}
    generator() = default;
    ~generator()
    {
        if (m_handle)
        {
            m_handle.destroy();
        }
    }
    T operator()() {
        assert(m_handle != nullptr);
        m_handle.resume();
        return (m_handle.promise().m_value);
    }
private:
    std::coroutine_handle<promise_type> m_handle = nullptr;
};
```

```
generator<int> generate_numbers()
{
    co_yield 10; // Return 10 and pause
    std::cout << "After stop point #1" << std::endl;

    co_yield 20;
    std::cout << "After stop point #2" << std::endl;

    co_yield 30;
    std::cout << "After stop point #3" << std::endl;
}

generator<int> infinite_number_stream(int start = 0)
{
    auto value = start;
    for (int i = 0;; ++i)
    {
        co_yield value;
        ++value;
    }
}
```

- C++20 doesn't provide actual coroutine types for direct use
- It only provides an infrastructure to build them
- Good coroutine types are hard to build and get right
- It's advised to at least start by using third party coroutine type libraries and only build your own when you REALLY know what you're doing

Slide intentionally left empty