STL Continuation C++20

C++ for Developers

C++20: The big four

- Concepts
- Modules
- Coroutines
- Ranges (and views)

Before we begin

To easier deal with C++20 features, you might need to update:

- Your Intellisense settings in VSCode
- The Makefile look in the CFLAG and change the -std flag to -std=c++20

Concepts

C++ for Developers

Concepts

Define a limitation of types for your templated classes and functions.

Defined in:

<concepts>

<type_traits>

```cpp
template <typename T>
requires (std::is_integral_v<T> || std::is_floating_point_v<T>)
T square(T x) {
    return x * x;
}
```

```cpp
template <typename S>
requires (std::convertible_to<S, std::string>)
int countCharacters (S s) {
    int count = 0;
    for (const auto& char c : s) {
        std::cout << c;
        count++;
    }

    std::cout << std::endl;

    return count;
}
```

Why are they used?

- Better guards for other developers using your code.
- Compile-time errors
- Self-documenting code
- Prevents misuse early

Some more documentation on concepts

https://en.cppreference.com/w/cpp/language/constraints.html

Modules

C++ for Developers

Modules

Replacing the pre-processor.

- Instead of text-replacement
- Using a compiled interface

Probably recognizes the keyword **import** from other languages.

This works similarly in C++. Using compiled files instead of expanding code through pre-processing.

Ranges

C++ for Developers

Ranges

What is a range?

In C++, a range is defined by a begin iterator and an end sentinel, which together mark the start and end of the sequence.

What is a sentinel?

A sentinel marks the end of a range. It acts like an “end iterator,” but it doesn’t have to be of the same type as the iterator returned by begin().

Sentinel

You can think of a sentinel as an *abstract concept* that represents where iteration should stop. It might be the same type as the iterator, or it might represent a more flexible boundary.

For example: *stopping early or marking an “infinite” range.*

Thanks to sentinels, a range no longer has to be strictly bound between a matching begin() and end() iterator. It can end wherever the sentinel says it does.

std::ranges::

Enables us to work with the STL containers without defining the iterator pairs (.begin() and .end()). Instead we just pass the container instead:

```
std::for_each(movieVector.begin(), movieVector.end(), printMovie);
```

```
std::ranges::for_each(movieVector, printMovie);
```

How do we work with ranges?

Ranges removes the necessity to work with iterator pairs. A lot of the <algorithm> functions work with std::ranges as well.

```
if (std::ranges::all_of(movieArray, searchFunction)) {
    std::cout << "The array only contains the movie: " << titleTarget << std::endl;
} else {
    std::cout << "The array has mixed movies!\n";
}
```

```
const auto find_it = std::ranges::find_if(movieList, searchFunction);
if (find_it != movieList.end()) {
    std::cout << "The list contains the movie: " << find_it->getTitle() << std::endl;
}
```

```
auto remove_range = std::ranges::remove_if(movieVector, [&yearTarget](const auto& movie){
    return movie.getYearOfRelease() == yearTarget;
});
movieVector.erase(remove_range.begin(), remove_range.end());
```

Ranges

Look at all these Ranges:

https://en.cppreference.com/w/cpp/algorithm.html

LET’S LOOK AT SOME CODE!

https://github.com/lafftale1999/cpp_for_developers/blob/main/week_6/5_ranges/main.cpp

lafftale for developers

Ranges concepts

`ranges::range` (C++20)	specifies that a type is a range, that is, it provides a begin iterator and an end sentinel (concept)
`ranges::borrowed_range` (C++20)	specifies that a type is a `range` and iterators obtained from an expression of it can be safely returned without danger of dangling (concept)
`ranges::approximately_sized_range` (C++26)	specifies that a range can estimate its size in constant time (concept)
`ranges::sized_range` (C++20)	specifies that a range knows its size in constant time (concept)
`ranges::view` (C++20)	specifies that a range is a view, that is, it has constant time copy/move/assignment (concept)
`ranges::input_range` (C++20)	specifies a range whose iterator type satisfies `input_iterator` (concept)
`ranges::output_range` (C++20)	specifies a range whose iterator type satisfies `output_iterator` (concept)
`ranges::forward_range` (C++20)	specifies a range whose iterator type satisfies `forward_iterator` (concept)
`ranges::bidirectional_range` (C++20)	specifies a range whose iterator type satisfies `bidirectional_iterator` (concept)
`ranges::random_access_range` (C++20)	specifies a range whose iterator type satisfies `random_access_iterator` (concept)
`ranges::contiguous_range` (C++20)	specifies a range whose iterator type satisfies `contiguous_iterator` (concept)
`ranges::common_range` (C++20)	specifies that a range has identical iterator and sentinel types (concept)
`ranges::viewable_range` (C++20)	specifies the requirements for a `range` to be safely convertible to a `view` (concept)
`ranges::constant_range` (C++23)	specifies that a range has read-only elements (concept)

Views

C++ for Developers

View

A view is a lightweight, non-owning wrapper that provides a customized way to access or transform elements of an underlying sequence without copying or modifying it.

Views do not store elements themselves—they simply reference another range or container.

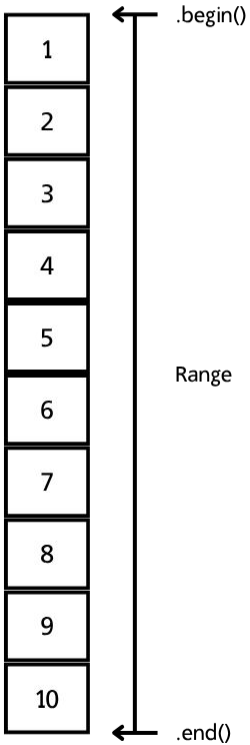
They are also *lazily evaluated*, meaning that the operations defined by a

What are they used for?

Views create lazy, lightweight streams of elements that can be looped over or transformed without allocating new containers or copying data.

They work similarly to streams in Java or LINQ in C#, allowing you to build composable pipelines that filter, transform, or combine data on demand.

Vector Container



Filter

```
auto filter = std::views::filter(numbers, [](const auto& i){
    return i % 2 == 0;
});
```

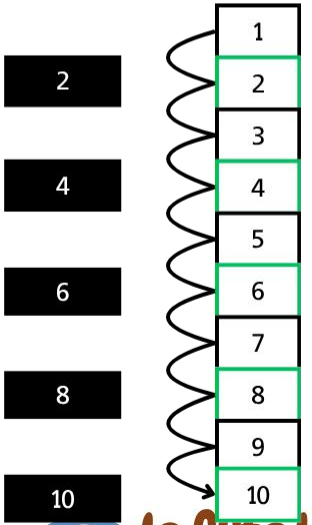
Creates a wrapper around the container. This contains:

- A reference to numbers.
- The predicate (in this case a lambda function).

The variable filter is now a *view object*.

View

```
for (auto i : filter) {
    std::cout << i << std::endl;
}
```



lafftale for developers

Maybe too easy example

Imagine trying to create a program for movie recommendations.

- std::vector<Movie> movies = getMovies()
- Loop through the container
  - Only movies starting on S
  - In the genre Documentary
  - Released in 1995
  - Read them as Strings

Maybe too easy example

Imagine trying to create a program for movie recommendations.

- std::vector<Movie> movies = getMovies()
- Loop through the container
  - Only movies starting on S
  - In the genre Documentary
  - Released in 1995
  - Handle them as class MovieRecommendations

With views:

is_match is a lambda used as *predicate* for filtering.

```
auto is_match = [](const auto& m){
    const auto& title = m.getTitle();
    return
        !title.empty() && title.at(0) == 'S' &&
        m.getYearOfRelease() == "1995" &&
        std::ranges::any_of(m.getGenres(), [](const auto& g){
            return g == GENRES::DOCUMENTARY;
        });
};
```

to_recommendation is a lambda used as a *transformation function*.

```
auto to_recommendation = [](const auto& m){
    return MovieRecommendation(m.getTitle());
};
```

We compose these functions using *std::views::filter* and *std::views::transform* to create a **view pipeline** over r_movies.

This pipeline filters and transforms the movies on demand, without copying or allocating new containers

```
for (const auto& r : r_movies |
    std::views::filter(is_match) |
    std::views::transform(to_recommendation))
    {
    std::cout << r.getName() << std::endl;
}
```

Views

Look at all these Views:

https://en.cppreference.com/w/cpp/ranges.html

LET'S LOOK AT SOME CODE!

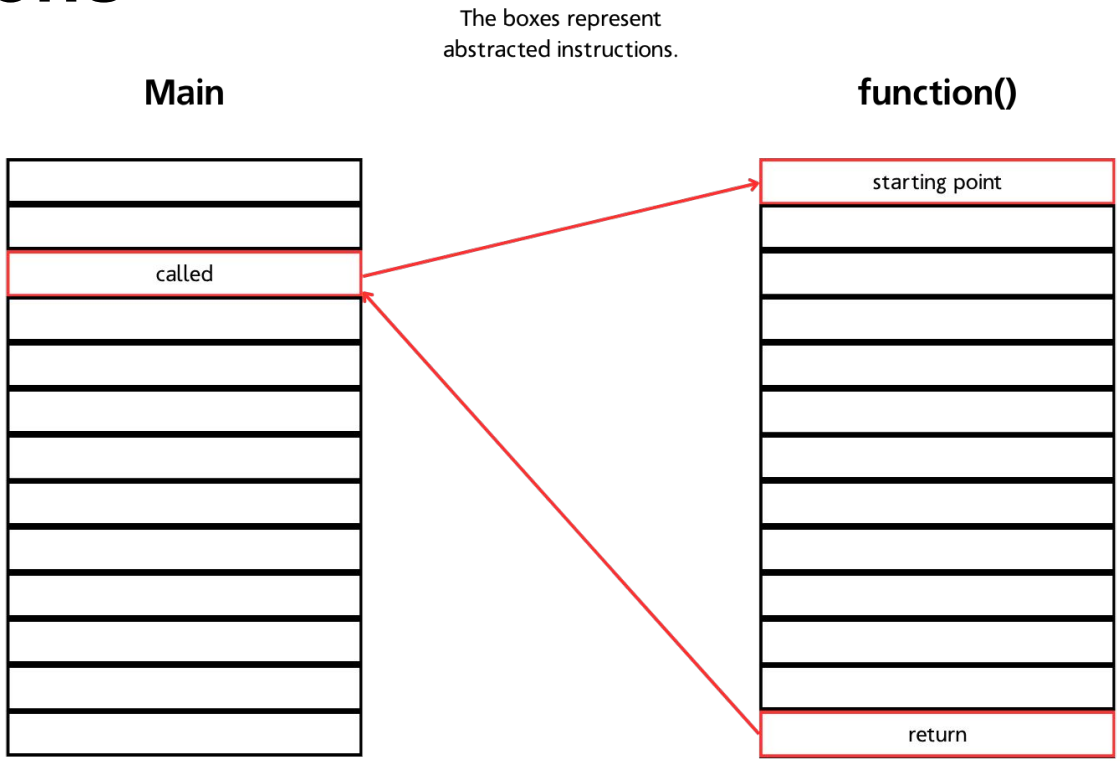
https://github.com/lafftale1999/cpp_for_developers/blob/main/week_6/6_views/main.cpp

lafftale for developers

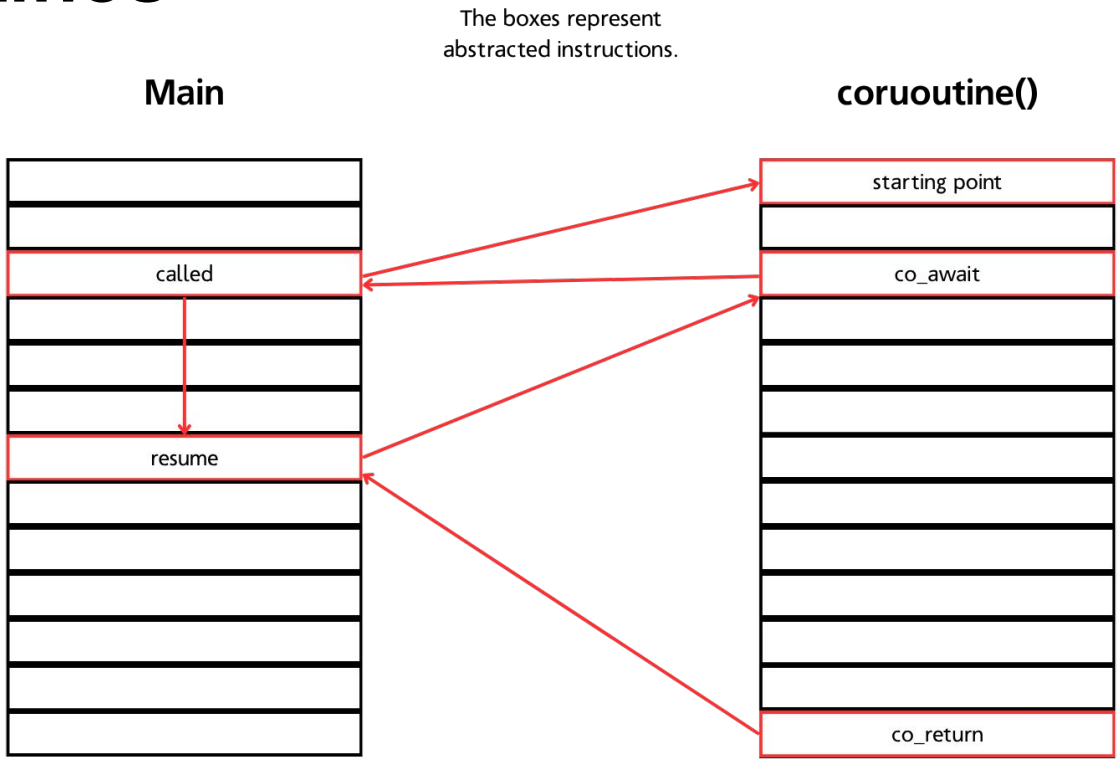
Coroutines

C++ for Developers

Functions



Coroutines



lafftale
for developers

Coroutines

A coroutine is a function that can suspend execution to be resumed later.

It is not decided by its function prototype, but by its implementation. A function is a coroutine if it contains:

- co_return (ends)
- co_await (suspends)
- co_yield (suspends)

Keyword Table

co_return Ends the coroutine and optionally produces a final result for the caller.

co_yield Produces (yields) a value to the caller and suspends the coroutine, allowing it to be resumed later.

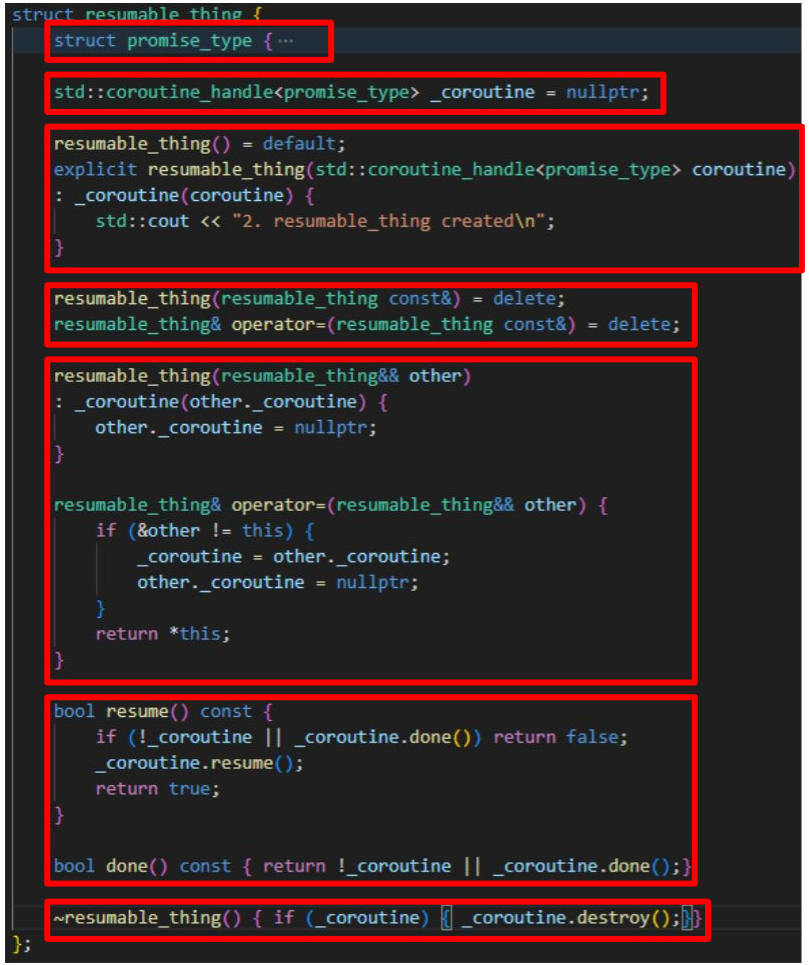
co_await Suspends the coroutine while it waits for another asynchronous or awaitable operation to complete.

Keyword	Action	State
co_return	Output	Ended
co_yield	Output	Suspended
co_await	Input	Suspended

Coroutines

A coroutine minimal requirements are:

- A return type that defines a nested promise_type
- promise_type must implement at least:
  - wrapper_type get_return_object()
  - initial_suspend()
  - final_suspend()
  - unhandled_exception()
  - return_void() or return_value()



promise_type the implementation of the *promise object*

handle to the coroutine *promise object*

Constructors
Default and one explicit. This means it will not accept implicit parsing of parameters.

deleting of copy constructor and assignment

defining move constructor and move assignment

constructor when the object hasn't been created.

assignment it is already was created.

interfacing the states and methods of our coroutine frame.

destructor

lafftale for developers

Coroutine Frame

A compiler-generated object for each coroutine that persists across suspension. It contains:

Promise Object
Instance of our promise_type that controls the coroutine's behaviour.

Parameters and local variables
Data that needs to survive suspensions.

Bookkeeping fields
Internal numbers and flags to keep track of where to continue execution.

Awaiter storage
Temporary storage for awaitable objects in the coroutine.

Exception storage
Instead of crashing the coroutine, the exceptions can be stored here and handled.

```
struct promise_type {
    resumable_thing get_return_object() {
        std::cout << "1. get_return_object()\n";
        return resumable_thing(std::coroutine_handle<promise_type>::from_promise(*this));
    }

    auto initial_suspend() noexcept {
        std::cout << "3. initial_suspend()\n";
        return std::suspend_never{};
    }

    auto final_suspend() noexcept {
        std::cout << "final_suspend()\n";
        return std::suspend_always{};
    }

    void return_void() noexcept {std::cout << "return_void()\n";}

    void unhandled_exception() {
        std::terminate();
    }

};
```

coroutine

By including co_await in this function, we have now declared it a coroutine.

```
resumable_thing counter() {
    std::cout << "counter: called\n";
    for (unsigned i = 1; ; ++i) {
        co_await std::suspend_always{};
        std::cout << "counter: resumed #" << i << std::endl;
    }
}
```

co_await

What does co_await do?

- Evaluates the expression to get an awaitable object
- Calls await_ready() - checking if result is already ready
- If not - call await_suspend() - suspends and returns to caller.
- Once result is ready - the awaitable object is resumed

And the co_routine then continues.

```
auto awaitable&& = expression…

if (!awaitable.await_ready) {
        awaitable.await_suspend(handle);
    // abstracted “starting point after
    returning”
}

awaitable.await_resume();
```

lafftale for developers

Helper types

```
_EXPORT_STD struct suspend_always {
    _NODISCARD constexpr bool await_ready() const noexcept {
        return false;
    }

    constexpr void await_suspend(coroutine_handle<>) const noexcept {}
    constexpr void await_resume() const noexcept {}
};
```

suspend_always{}
Suspends execution immediately

```
_EXPORT_STD struct suspend_never {
    _NODISCARD constexpr bool await_ready() const noexcept {
        return true;
    }

    constexpr void await_suspend(coroutine_handle<>) const noexcept {}
    constexpr void await_resume() const noexcept {}
};
```

suspend_never{}
Continues execution

coroutine

By understanding this, we now also can understand how to create awaitable objects. We need to implement await_ready(), await_suspend() and await_resume()

```
resumable_thing counter() {
    std::cout << "counter: called\n";
    for (unsigned i = 1; ; ++i) {
        co_await std::suspend_always{};
        std::cout << "counter: resumed #" << i << std::endl;
    }
}
```

lafftale for developers

redefining an awaitable object

```
auto await_transform(std::string) noexcept {
    struct awaiter {
        promise_type& promise;
        constexpr bool await_ready() const noexcept { return true; }
        std::string await_resume() const noexcept {
            return std::move(promise.incoming_message);
        }
        void await_suspend(std::coroutine_handle<>) const noexcept {}
    };

    return awaiter{*this};
}
```

promise_type& promise A reference stored in the awaiter that binds to the coroutine's *promise object*.

Definition of:

- await_ready()
- await_resume()
- await_suspend()

return awaiter{*this}
returns an awaitable object with *this as the reference to the *promise object*.

redefining an awaitable object

```
auto await_transform(std::string) noexcept {
    struct awaiter {
        promise_type& promise;
        constexpr bool await_ready() const noexcept { return true; }
        std::string await_resume() const noexcept {
            return std::move(promise.incoming_message);
        }
        void await_suspend(std::coroutine_handle<>) const noexcept {}
    };

    return awaiter{*this};
}
```

```
Task messages() {
    co_yield "Hello!\n";

    std::cout << co_await std::string{};

    co_return "I'm good, how about you?\n";
}
```

auto&& awaitable =
promise.await_transform(std::string{}).

if (!awaitable.await_ready()) {
    awaitable.await_suspend();
}

awaitable.await_resume();

await_resume moves the *incoming_message* to the empty string in:

std::cout << co_await std::string{};

Different types of coroutines

Task

A coroutine that performs work without returning a value.

Generator

A coroutine that performs work and returns a value (either through co_return or co_yield).

Where do we use coroutines?

- Networking – non-blocking I/O
- Game Development
- File / Disk Operations
- UI / Simulation
- Task Scheduling / Pipelines

Whenever operations are asynchronous or long-running, coroutines let you write code that looks synchronous but runs concurrently. Letting us avoid callbacks, complex state machines or thread overhead.


lafftale
for developers