Coroutines Tutorial Extremely Rough Draft (convert to HTML)

What are coroutines?

C++ coroutines, introduced in C++20, offer a streamlined approach to asynchronous and concurrent programming. They enable you to craft asynchronous code that closely resembles synchronous programming, enhancing code clarity and maintainability.

These coroutines can temporarily pause their execution at designated points, later resuming from where they left off. This pause-and-resume behavior allows other tasks to execute in the interim. Furthermore, they operate without relying on a traditional stack, suspending execution by returning to the caller and storing the necessary data separately from the stack. In essence, this stackless mechanism stores the coroutine's data, or coroutine frame, on the heap.

Different types of coroutines?

Generally, coroutines can be categorized into two primary types. The first type is the generator coroutine, designed for producing sequences, while the second type is the task coroutine, typically employed for asynchronous programming.

Generator coroutines exhibit a unique behavior. They generate sequences of values in a lazy, on-demand manner, yielding each value individually when requested by the consumer. This is achieved through the use of the co_yield keyword, which enables you to generate values without the need to compute the entire sequence in advance. Generator coroutines are particularly valuable when implementing iterators or implementing a mechanism for efficient lazy sequence processing. Below is an example of a simple generator coroutine.

```cpp
#include <iostream>
#include <coroutine>

generator<int> generateNumbers(int from, int to) {
    for (int i = from; i <= to; ++i) {
        co_yield i;
    }
}

int main() {
    for (int number : generateNumbers(1, 5)) {
        std::cout << number << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

In this example, we define the generator coroutine called generateNumbers that yields a sequence of numbers using co_yield from “from” to “to” (change variable names).
Output: 1 2 3 4 5

On the other hand, task coroutines serve the purpose of enabling you to await asynchronous operations without causing the calling thread to become blocked. These coroutines make extensive use of the co_await keyword, which effectively pauses their execution until the awaited operation has concluded. Various operations that can be awaited encompass I/O tasks (which we'll delve into shortly), network requests, and timers. To illustrate the concept, here's a straightforward task coroutine that emulates an asynchronous operation.

```
#include <iostream>
#include <coroutine>
#include <chrono>

std::task<void> asyncTask() {
    // Simulate an asynchronous task
    std::cout << "Async task started." << std.endl;

    // Simulate an asynchronous operation
    co_await std::suspend_for(std::chrono::seconds(2));

    std::cout << "Async task completed." << std.endl;
}

int main() {
    // Start the asynchronous task using 'resume'
    asyncTask().resume();

    // Continue with other work while the async task is in progress
    std::this_thread::sleep_for(std::chrono::seconds(3));

    std::cout << "Main function continues to run "
              << "asynchronously." << std::endl;

    return 0;
```

We define a task coroutine called asyncTask that simulates an asynchronous operation using std::suspend_for. Using the co_await keyword to suspend execution, we allow other tasks to run concurrently during the delay. In the main function, we start the coroutine by calling resume(), and while the async task is in progress, the main function continues to execute other code concurrently.

Output:
Async task started.
Main function continues to run asynchronously.
Async task completed.

Why use Coroutines?

Another reason why using coroutines is useful is with non-blocking I/O. They are particularly useful for handling requests, or interacting with databases ultimately improving the responsiveness of applications. When a coroutine encounters an I/O operation, it can yield control back to the event loop, allowing other tasks to run. Coroutines are often integrated with event loops, which are a fundamental component of event-driven programming. The event loop manages the scheduling and execution of coroutines, ensuring that events are processed efficiently and that the application remains responsive. As you will see in the example below, coroutines can pause and resume execution while waiting for I/O operations to complete, allowing the program to continue processing other tasks in the meantime.

```cpp
// Asynchronous operation to read a file
std::task<std::string> readFileAsync(const std::string& filename) {
    co_await std::suspend_always{}; // Simulate some initial work

    // Perform a file I/O operation asynchronously
    std::ifstream file(filename);
    if (!file.is_open()) {
        co_return "Error: File not found.";
    }

    std::string content;
    char buffer[256];
    while (!file.eof()) {
        file.read(buffer, sizeof(buffer));
        content.append(buffer, file.gcount());
        co_await std::suspend_always{}; // Simulate asynchronous I/O
    }
    file.close();

    co_return content;
}

int main() {
    std::string filename = "example.txt";
    std::string fileContent = readFileAsync(filename).await_resume();

    std::cout << "File content:\n" << fileContent << std::endl;
    return 0;
}
```

Additionally, if you are worried about resources such as memory, coroutines are extremely useful for resource management. Coroutines are more resource efficient than creating a thread or process for every asynchronous task. They can be executed on a limited number of threads, which saves memory and resources.

How do we implement coroutines? (I'm going to put this before the why section)

Use c++20
Include <coroutine>

Define a coroutine function.

By the cppreference definition, "a function is a coroutine if its definition contains any of the following:"

- Co_await – used for suspending execution until resumed
- Co_yield – used for suspending execution and return a value
- Co_return – used for completing execution and return a values

Additionally, coroutines can have different return types – similar to a regular function – that depends on the specific behavior you want to achieve. Here are a few examples of different return types:

- void – when the coroutine doesn't return any meaningful result. Quite suitable for coroutines that primarily have side effects or perform asynchronous operations without producing a value
- std::suspend_never – when you want the coroutine to run to completion without ever suspending. Appropriate for lightweight, non-blocking tasks that don't need to wait other operations.
- std::suspend_always – when you want the coroutine to suspend immediately upon entry and never resume. Useful for cases when you don't intend to perform any work in the coroutine

- std::coroutine_handle<> – a low-level coroutine used for cusomizing coroutine handling.
- std::task<T> or std::task<void> – should be used for high level asynchronous programming. Suitable for representing asynchronous operations as tasks and can be used with libraries that provide abstractions
- Custom promise types - For more complex coroutines that need to return values, manage state, or implement advanced coroutine behavior

How to implement a customizable corouotine?

Define the promise_type structure or class

- The promise_type structure is part of the coroutine and is responsible for managing the coroutine's state and returning the final result. It must include certain member functions and data members:

- Esesntial members:

    - *get_return_object()* – required; returns an instance of the object (this is where the coroutine's result is stored)
    - *unhandled_exception()* – required; for proper exception handling within the coroutine
        - Used to propagate exceptions when an exception is unhandled
    - *return_void()* – required if coroutine returns void; allows the promie to finalize the coroutine's results
    - *initial_susend()* – typically required; specifies whether the coroutine should suspend immediately upon entering. Decides whether to perform some initial setup or suspend right away.
        - Use when you want to control the suspension behavior of the coroutine when it starts
    - *final_suspend()* – typically required; specifies whether the coroutine should suspend after the last value (generator), or when the coroutine is complete
        - Use when you want to control the suspension behavior of the coroutine after producing all values or completing its task
        -

- Commonly used members:

    - *yield_value()* – allows the coroutine to yield a value using co_yield
        - Commonly used in generator coroutines
    - *await_suspend()* – allows custom logic to be applied when the coroutine is suspended
    - *await_resume()* – used when the corouotine resumes after a suspend, usually to retrieve the result of the an awaited taks
    - *get_return_object_on_allocation_failure()* – function is used to handle resource allocation failures when creating the coroutine promise
        - Can provide an alternative object in case of allocation failure
    - rethrow_if_nested(task<...>) – used for proper exception handling in nested tasks; ensure that exceptions thrown in nested tasks are rethrown correctly

Here is a great example of a customizable coroutine

```
struct Chat {

    struct promise_type {
        std::string msgOut{}, msgIn{}; // this stores values going into or coming out of the coroutine

        void unhandled_exception() noexcept {}; // what to do when there's an exception
        Chat get_return_object() { return Chat(this); } // coroutine creation
        std::suspend_always initial_suspend() noexcept { return{}; } // startup
        std::suspend_always yield_value(std::string msg) noexcept // value from co_yield
        {
            msgOut = std::move(msg);
            return {};
        }

        auto await_transform(std::string) noexcept // value from co_await
        {
            struct awaiter { //> these can be customized instead of using suspend_always or suspend_never
                promise_type& pt;
                constexpr bool await_ready() const noexcept { return true; }
                std::string await_resume() const noexcept { return std::move(pt.msgIn); }
                void await_suspend(std::coroutine_handle<>) const noexcept {}

            };

            return awaiter{*this};
        }

        void return_value(std::string msg) noexcept { msgOut = std::move(msg); } // value from co_return
        std::suspend_always final_suspend() noexcept { return {}; }  // ending

    };
```

We create a coroutine called Chat with the promise_type struct nested inside. At the bottom, the chat coroutine has a return_value function that is callable from the co_return keyword. We also set final_suspend to suspend_always, meaning when the coroutine is finished, it will always suspend, or in other words, terminate.

Taking a look into the promise_type struct, you will notice it's nested inside the chat coroutine. we include member variables to store values that will come in and go out of the coroutine. As mentioned before, we include the essential member functions unhandled_exception() which can be customized to handle exceptions, get_return_object() which returns the coroutine object when first created, initial_suspend() which returns suspend_always, and final_suspend() which also returns suspend_always.

You will also find some commonly used member functions such as yield_value that stores the value from the co_yield keyword into the msgOut member variable. Additionally, the await_transform member function is part of the coroutine's promise type, and can be called using the co_await keyword. Inside the await_transform function contains a local awaiter structure which is responsible for controlling the behavior of the coroutine when co_await is used. You will notice the awaiter object has:

- promise_type &pt – the awaiter struct constructor takes a reference to the promise type allowing the awaiter to access the state and control the behavior of the coroutine
- await_ready() – called when the co_await expression is evaluated. Will return a boolean indicating whether the awaited value is immediately available
- await_resume() – called when the coroutine is ready to resume after suspension. In this case, it retrieves the awaited value from the coroutines promise type, specifically moving pt.msgIn and returning it as a string
- await_suspend(std::coroutine_handle<>) – This is called when the coroutine is about to be suspended after the co_await expression is evaluated. In this case, it doesn't perform any specific suspension action

Moving on we will look more closely at the chat structure

```cpp

    using Handle = std::coroutine_handle<promise_type>;  // shortcut
    Handle mCoroHdl{};

    explicit Chat(promise_type *p) : mCoroHdl{Handle::from_promise(*p)} {} // get the handle from the promise
    Chat(Chat && rhs) : mCoroHdl{std::exchange(rhs.mCoroHdl, nullptr)} {} // move only

    /// Destructor
    ~Chat()
    {
        if (mCoroHdl) { mCoroHdl.destroy(); }
    }

    std::string listen() // activate the coroutine and listen
    {
        if (not mCoroHdl.done()) { mCoroHdl.resume(); }
        return std::move(mCoroHdl.promise().msgOut);
    }

    void answer(std::string msg) const // send data to the coroutine and activate it
    {
        mCoroHdl.promise().msgIn = msg;
        if (not mCoroHdl.done()) { mCoroHdl.resume(); }
    }

};
```

As mentioned before, we use coroutine_handle<promise_type> which allows us to customize our coroutine handling and create an alias named Handle and declare mCoroHdl as a member variable of type Handle which will be used to control the execution of the coroutine.

We create a Chat instance and initialize mCoroHdl with the coroutine handle created from the provided promise. Additionally, we create the move constructor for Chat that takes an rvalue reference to another Chat instance and moves the coroutine handle from rhs to the current object. Although it’s not necessary, it can be beneficial if you want to optimze memory and performance, are worried about resource management, or just need to transfer ownership. For instance, if your object is managing a large buffer, it would be highly beneficial to transfer ownership rather than copying it.

Continuing, we create a listen function which is used to activate the coroutine and listen for a response. It first checks if the coroutine is done, if not, it calls resume() to advanced the coroutine's execution. In this example, it returns the restored response in the promise's msgOut member. Lastly, we create an answer function used to send data to the coroutine and activate it by storing the provided message in the promise's msgIn member and again checks if the coroutine is done. If not, it resumes.

```
61
62  Chat Fun()
63  {
64      co_yield "Hi, what's your name?\n"; // suspends coroutine w/ output message
65      std::string name = co_await std::string{}; // suspends coroutine w/ input message
66      co_return "Nice to meet you, " + name + "!\n"; // ends coroutine and returns final message
67
68  }
69
70
71  int main()
72  {
73      std::string input;
74      Chat chat = Fun();
75      std::cout << chat.listen();
76
77      std::getline(std::cin, input);
78      chat.answer(input);
79
80      std::cout << chat.listen();
81      return 0;
82
83  }
84
```

Finally, we define the Fun() coroutine function. We create a Chat object named chat and begin the coroutine's execution and the initial message is displayed with chat.listen(). We use chat.answer() to send the user's input as a response to the coroutine which will continue executing after the co_await line on line 65. And at the end, we call chat.listen() activating the coroutine to return the final message.

This is a great example of how a coroutine can be used to simulate the conversation, with the coroutine yielding messages, awaiting user input, and finally returning a response.

Here is another example of a customizable generator coroutine that interleaves two vectors:

```cpp
#include <iostream>
#include <coroutine>
#include <vector>

struct Generator {

    struct promise_type {
        int val{};

        Generator get_return_object() { return Generator(this); }
        std::suspend_never initial_suspend() noexcept { return {}; }
        std::suspend_always final_suspend() noexcept { return {}; }
        std::suspend_always yield_value(int v)
        {
            val = v;
            return {};
        }
        void unhandled_exception() {}
    };

    using Handle = std::coroutine_handle<promise_type>;
    Handle mCorohdl{};

    explicit Generator(promise_type* p) : mCorohdl{Handle::from_promise(*p)} {}

    Generator (Generator&& rhs) : mCorohdl{std::exchange(rhs.mCorohdl, nullptr)} {}
    ~Generator()
    {
        if (mCorohdl) { mCorohdl.destroy(); }
    }

    int value() const { return mCorohdl.promise().val; }
    bool finished() const { return mCorohdl.done(); }
    void resume()
    {
        if (not finished()) { mCorohdl.resume(); }
Generator interleaved(std::vector<int> a, std::vector<int> b)
{
    auto lamb = [](std::vector<int>& v)->Generator {
        for (const auto &e : v) {
            co_yield e;
        }
    };

    auto x = lamb(a);
    auto y = lamb(b);

    while (not x.finished() or not y.finished()) {
        if (not x.finished()) {
            co_yield x.value();
            x.resume();
        }

        if (not y.finished()) {
            co_yield y.value();
            y.resume();
        }
    }
}
```

Instead of going line by line, I'll give more or less a summary now that we know the basics.

```cpp
int main()
{
    std::vector a{1, 3, 5, 7, 9};
    std::vector b{2, 4, 6, 8, 10};

    Generator g(interleaved(std::move(a), std::move(b)));

    while (not g.finished())
    {
        std::cout << g.value() << "\n";
        g.resume();
    }

    std::cout << std::endl;
    return 0;
}
```

Generator Struct:

- Custom coroutine type designed to generate and yield values
- Defines the promise_type struct ultimately customizing the coroutine behavior
- The generator type uses a coroutine handle (std::corourtine_handle) to manage the coroutine's execution
- Provides a method to check if the coroutine is finished, obtain the current yielded value, and manually resume the coroutine
- Struct manages the destruction of the coroutine handle in its destructor

Interleaved Function:

- Takes two vectors as input (a & b)
- Inside the function, it defines the lambda function that takes a reference to the vector and creates a Generator for it
- The lambda iterates through the input vector and yields each element in the vector using co_yield
- Interleaves the values from the two Generator instances, x and y, in the while loop
- It yields values from x and y and if either x or y are not finished, it resumes the respective generator

Main function:

- The two vectors are defined (a & b)
- Interleaved function is called and its results are stored in a generator called g

- The values are printed to the console, and the generator is manually resumed

I hope this tutorial was helpful and you enjoyed. Again this is a ***rough*** draft..

References:

https://en.cppreference.com/w/cpp/language/coroutines
https://stackoverflow.com/questions/71153205/c-coroutine-when-how-to-use
https://www.scs.stanford.edu/~dm/blog/c++-coroutines.html
https://www.geeksforgeeks.org/coroutines-in-c-cpp/