진리는어디에/C++20

[C++20] 코루틴(Coroutine) - co_await

kukuta 2021. 3. 24. 02:59

> 안녕하세요.
>
> 저번 시간의 [진리는어디에] - [C++20] 코루틴(Coroutine)에서는 C++20에서 새로이 도입된 코루틴의 기본에 대해 알아 보았습니다. 이 포스팅은 지난 과정에 이어지는 내용이므로 저번 포스팅을 한번 살펴 보시고 오시는 것이 이번 포스팅을 이해하는데 좀 더 도움이 되리라 생각합니다.
>
> 이번 포스팅에서는 지난 시간에 이어 co_await 키워드에 대해 살펴 보는 시간을 갖도록 하겠습니다.

co_await *expr*

`co_await`는 단항 연산자로써, 코루틴의 실행을 중단(suspend)하고 호출자(caller)에게 제어권을 넘기는데 사용 됩니다. co_await의 피연산자(expr)는 co_await operator를 구현했거나 현제 코루틴의 Promise::await_trasnform을 이용해 변환 할 수 있어야 한다고 정의 되어 있습니다.

> The unary operator co_await suspends a coroutine and returns control to the caller. Its operand is an expression whose type must either define operator co_await, or be convertible to such type by means of the current coroutine's Promise::await_transform
>
> from : https://en.cppreference.com/w/cpp/language/coroutines

co_await에서 어떤 일이 일어나는지 알아보기 위해, 지난 시간 [진리는어디에] - [C++20] 코루틴(Coroutine)에서 작성했던 코드의 일부를 발췌해 가져 왔습니다.

```
// 코루틴 객체
class Task
{
public :
    struct promise_type
    {

        // promise_type 구현 부분...생략
    };
```

```
    // 코루틴 객체 구현 부분...생략
};

// 코루틴 함수
Task foo()
{
    std::cout << "foo 1" << std::endl;
    // co_await std::suspend_always{};
    std::suspend_always awaitable_object;
    co_await awaitable_object;
    std::cout << "foo 2" << std::endl;
}
```

컴파일러는 위 foo() 함수의 co_await를 만나면 아래와 비슷한 코드를 생성합니다.

주목해서 보셔야 할 부분은 '**컴파일러가 생성하는 코드 시작**' 부터 '**컴파일러가 생성하는 코드 끝**' 안쪽의 **SUSPEND**와 **RESUME** 위치 입니다. SUSPEND하기 전 await_suspend() 가 호출 되고, 코루틴에 재진입 한 RESUME 이후 바로 await_resume() 이 호출 된다는 것을 기억 하기 바랍니다.

```
Task foo()
{
    // promise 관련 생성된 코드들...생략.
    try
    {
        std::cout << "foo 1" << std::endl;

        std::suspend_always awaitable_object;

        // ==== 컴파일러가 생성하는 코드. 시작 ====
        if(!awaitable_object.await_ready())
        {
            awaitable_object.await_suspend(coroutine_handle);
            // SUSPEND coroutine here!! 여기서 caller로 돌아감
        }

        // RESUME coroutine here!!
        awaitable_object.await_resume();
        // ==== 컴파일러가 생성하는 코드. 끝 ====
        std::cout << "foo 2" << std::endl;
    }
    catch(...)
    {
        // .. 생략 ..
    }

    // .. 생략 ..
}
```

코루틴 함수는 'SUSPEND' 시점에 진행을 중단하고 호출자에게 돌아가게 됩니다. 그 전에 await_ready()를 false와 비교하여, 중단하고 호출자에게 돌아갈지, 무시하고 계속 진행할지를 판단하게 됩니다. 만일 진행을 중단(suspend)하게 된다면 호출자에게 돌아가기 전에 await_suspend를 호출 합니다.

마찬가지로 코루틴(std::coroutine_handle)의 resume()을 호출하여 코루틴 함수로 돌아 오게 된다면 'RESUME' 위치에서 부터 다시 시작하게 됩니다. 당연히 'RESUME' 위치 바로 다음에 오는 await_resume()을 호출 합니다.

C++20에서 제공하는 suspend_always를 예로 좀 더 사세히 살펴 보변 `co_await std::suspend_always{};` 호출 시 suspend_always의 await_ready는 항상 false를 리턴하므로, 이 호출은 항상 코루틴의 실행을 suspend 시킵니다.

```
// from - https://en.cppreference.com/w/cpp/coroutine/suspend_always
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 {}
};
```

반대로 suspend_never를 넘기게 된다면 foo()는 호출자로 리턴하지 않고 다음을 계속 진행 합니다.

```
// from - https://en.cppreference.com/w/cpp/coroutine/suspend_never
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 {}
};
```

위의 suspend_always, suspend_never는 await_ready() 외에는 따로 정의 된 내용이 없어 항상 suspend하거나, 그렇지 않고 계속 진행하거나 외에는 할 수 있는게 없습니다. 하지만 await_suspend, await_resume를 우리 입맛에 맞게 고쳐 준다면 여러 재미있는 일들을 할 수 있습니다.

커스텀 awaitable

이번 장에서는 위에서 언급한 커스텀 awaitable을 이용하여 재미있는 일(?)을 한가지 해보겠습니다. 코루틴을 suspend하고 resume 할 때 원래 작업을 진행하던 스레드가 아닌 새로운 스레드를 생성하여 작업을 진행하는 switch_to_new_thread awaitable 오브젝트를 정의 해보겠습니다. 이 아이디를 이용해 좀 더 발전 시킨다면 항상 콜백을 등록 해줘야 하는 async 작업에서 콜백 없이 async 오퍼레이션이 끝나면 suspend 했던 시점으로 돌아 올 수 있는 코루틴을 만들 수도 있습니다.

```
struct switch_to_new_thread
{

    constexpr bool await_ready() const noexcept { return false; }
    void await_suspend(coroutine_handle<> handle) const noexcept
    {
        std::thread t([handle]()
                {
                    handle.resume();
                })
;
        t.detach();
    }
    constexpr void await_resume() const noexcept {}
};

Task foo()
{
    std::cout << "foo 1" << " " << std::this_thread::get_id() << std::endl;
    co_await switch_to_new_thread{};
    std::cout << "foo 2" << " " << std::this_thread::get_id() << std::endl;
}

int main()
{
    Task task = foo();
    std::cout << "\t\t main 1" << " " << std::this_thread::get_id() << std::endl;
    task.co_handler.resume();
    std::cout << "\t\t main 2" << " " << std::this_thread::get_id() << std::endl;

    // 프로그램의 종료를 방지하기 위해..
    int n;
    std::cin >> n;
}
```

메인 스레드에서 foo()의 "foo 1"까지 실행하다 co_await를 만나면 switch_to_new_thread의 임시 객체를 생성한 후 suspend를 과정을 진행합니다. switch_to_new_thread::await_ready는 false를 리턴하므로 suspend를 하기로 합니다. 그리고 여기서 부터가 중요 합니다. 위에서 호출자(지금은 메인 함수)로 리턴하기 전에 await_suspend를 먼저 호출 한다고 언급 했습니다.

switch_to_new_thread::await_suspend를 살펴 보시면, 내부에서 coroutine_handle을 인자로 받는 람다 표현식을 실행하는 스레드를 생성했습니다. 그리고 람다 표현식 안에서는 coroutine_handle의 resume()을 호출하죠. 이것은 메인 스레드가 아닌 새로운 스레드에서 foo()의 스택을 다시 로드하여 재실행(resume) 할 수 있게 합니다.

위 코드를 컴파일해서 실행 하면 "foo 1"까지는 메인 스레드(예제에서는 스레드 아이디 6524)에서 실행하다 resume 이후 부터는 새로운 스레드(스레드 아이디 7460)에서 나머지 작업을 진행 하는 것을 보실 수 있습니다.

```
        main 1 6524
foo 1 6524
        main 2 6524
foo 2 7460
```

이렇게 awaitable object를 입맛에 맞게 정의 해주면 작업을 새로운 스레드에서 진행 하는 등의 여러 재밌는 일들을 할 수 있습니다. 이 외에도 인터넷을 조금 살펴 보면 많은 재밌는 샘플들이 있으니 관심이 있으시다면 찾아 보는 것도 좋을 것 같습니다.

이상 co_await에 대한 글을 마치도록 하겠습니다.

감사합니다.

부록 1. 전체 코드

```
#include <iostream>
#include <coroutine>
#include <thread>

class Task
{
public:
    // 규칙 1. C++에서 정의된 규칙을 구현한 promise_type 이라는 이름의 타입이 정의되어야 한다.
    struct promise_type
    {
        int value;

        Task get_return_object() { return Task{ std::coroutine_handle<promise_type>::from_promise(*this) }; }
        auto initial_suspend() { return std::suspend_always{}; }
        auto return_void() { return std::suspend_never{}; }
        auto final_suspend() { return std::suspend_always{}; }
        void unhandled_exception() { std::exit(1); }
    };

    // 규칙 2. std::coroutine_handle<promise_type> 타입의 멤버 변수가 있어야 한다.
    std::coroutine_handle<promise_type> co_handler;

    // 규칙 3. std::coroutine_handle<promise_type> 을 인자로 받아 멤버 변수를 초기화 하는 생성자가 있어야 한다.
    Task(std::coroutine_handle<promise_type> handler) : co_handler(handler)
    {
    }

    // 규칙 4. 소멸자에서 std::coroutine_handle<promise_type> 타입의 코루틴 핸들러 멤버 변수를 해제 해야 한다.
    ~Task()
    {
        if (co_handler)
        {
            co_handler.destroy();
        }
    }
};

struct switch_to_new_thread
{
    constexpr bool await_ready() const noexcept { return false; }
    void await_suspend(std::coroutine_handle<> handle) const noexcept
    {
        std::thread t([handle] () {
                        handle.resume();
                    })
;
        t.detach();
    }
    constexpr void await_resume() const noexcept {}
};
```

```
50
51  Task foo()
52  {
53      std::cout << "foo 1" << " " << std::this_thread::get_id() << std::endl;
54      co_await switch_to_new_thread{};
55      std::cout << "foo 2" << " " << std::this_thread::get_id() << std::endl;
56  }
57
58  int main()
59  {
60      Task task = foo();
61      std::cout << "\t\t main 1" << " " << std::this_thread::get_id() << std::endl;
62      task.co_handler.resume();
63      std::cout << "\t\t main 2" << " " << std::this_thread::get_id() << std::endl;
64
65      // 프로그램의 종료를 방지하기 위해..
66      int n;
67      std::cin >> n;
68  }
69
```

부록 2. 같이 보면 좋은 글

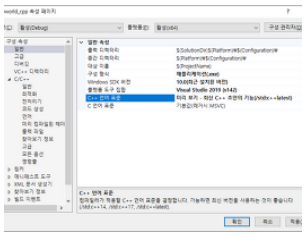
- [C++20] 소개
- [C++20] 코루틴(Coroutine)
- [C++20] 코루틴(Coroutine) - co_yield
- [C++20] 코루틴(Coroutine) - co_return
- [C++20] 코루틴(Coroutine) - done()
- cppreference.com - Coroutines (C++20)
- Understanding C++20 Coroutines, Awaitable Objects and Operator co_await with Real Samples

태그 #c++20, #C/C++, #Coroutine

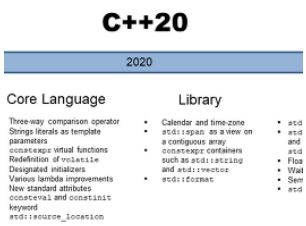
'진리는어디에/C++20' Related Articles

NO IMAGE

[C++20] 코루틴(Coroutine) - co_yield



[C++20] 컴파일



[C++20] 코루틴(Coroutine)

NO IMAGE

[C++20] 람다(lambda)의 변화

유익한 글이었다면 공감(❤) 버튼 꾹!! 추가 문의 사항은 댓글로!!

이름 암호 Secret

여러분의 사랑과 관심이 한 사람을 살립니다. 댓글 한번만 남겨 주세요. 관심 받고 싶습니다!! 하앍하앍!!

댓글달기