진리는어디에/C++20

[C++20] 코루틴(Coroutine) - co_yield

kukuta 2021. 3. 29. 00:05

이번 강의는 [C++20] 코루틴(Coroutine)에 이어지는 내용 입니다. 이번 강의를 읽으시기 전에 이전 포스팅을 먼저 읽어 보시길 추천 드립니다.

co_yield

코루틴을 중단(suspend)하고 호출자에게 돌아갈 때, 호출자에게 값을 넘기고 싶다면 co_await대신 co_yield를 사용 하면 됩니다.

```
Generator foo()
{
    //co_await std::suspend_always{};
    co_yield 10;
}
```

하지만 컴파일러는 '**co_yield**' 구문을 만나면 내부적으로 다음과 같은 코드를 생성합니다.

```
Generator foo()
{
    Generator::promise_type promise;
    // ...코드 생략...
    // co_yield 10;
    co_await promise.yield_value(10);
}
```

위와 같이 co_yield 구문은 **co_await promise.yield_value(expr)** 와 같은 구문으로 컴파일러에 의해 변경 됩니다. 때문에, **단순히** co_await를 co_yield 키워드로만 바꾸고 컴파일 한다면 "promise_type 에서 yield_vaule() 함수를 찾을 수 없습니다"와 같은 오류를 보게 될것 입니다.

만일 코루틴이 int 타입을 호출자에게 리턴한다면 promise_type 구조체 선언에 멤버 변수로 int 타입 변수를 추가하고, 그 멤버 변수에 값을 저장하는 yield_value함수를 만들어 주셔야 합니다.

```
class Task
{
    struct promise_type
    {
        // 다른 코드들 생략
        int value;

        std::suspend_always yield_value(int value)
        {
            this->value = value;
            return {};
        }
    };
};

Generator foo()
{
    // co_yield 10;
    co_await promise.yield_value(10);
}
```

컴파일러가 생성한 코드에서 co_await는 yield_value()를 호출하여 promise에 리턴할 값을 저장하고, yield_value에서 리턴 되는 suspend_always에 의해 코루틴을 중단(suspend) 하고 호출자에게 돌아갑니다.

```
Task foo()
{
    int value = 10;
    std::cout << "foo 1" << " " << value << std::endl;

    co_yield value;

    value += 10;
    std::cout << "foo 2" << " " << value << std::endl;
}

int main()
{
    Task task = foo();
    task.co_handler.resume(); // start coroutine
    std::cout << "main 1" << " " << task.co_handler.promise().value << std::endl;
    task.co_handler.resume();
    std::cout << "main 2" << " " << task.co_handler.promise().value << std::endl;
}
```

코루틴에서 yield한 값을 메인함수에서 promise().value를 통해 전달 받아 출력 할 수 있는 것을 볼수 있습니다.

정리

부록 1. 전체 코드

```
class Task
{
public:
    struct promise_type
    {
        int value; // 코루틴에서 호출자에게 리턴 할 값

        Task get_return_object() { return Task{ std::coroutine_handle<promise_type>::from_promise(*this) }; }
        auto initial_suspend() { return std::suspend_always{}; }

        // 코루틴에서 co_yield를 호출하기 위해 필요한 함수
        std::suspend_always yield_value(int v)
        {
            this->value = v;
            return {};
        }

        void return_void() { return; }
        auto final_suspend() noexcept { return std::suspend_always{}; }
        void unhandled_exception() { std::exit(1); }
    };

        std::coroutine_handle<promise_type> co_handler;
    Task(std::coroutine_handle<promise_type> handler) : co_handler(handler)
    {
    }
    ~Task() { if (co_handler) { co_handler.destroy(); }     }
};

Task foo()
{
    int value = 10;
    std::cout << "foo 1" << " " << value << std::endl;

    co_yield value;

    value += 10;
    std::cout << "foo 2" << " " << value << std::endl;
}

int main()
{
    Task task = foo();
    task.co_handler.resume(); // start coroutine
    std::cout << "main 1" << " " << task.co_handler.promise().value << std::endl;
    task.co_handler.resume();
    std::cout << "main 2" << " " << task.co_handler.promise().value << std::endl;
}
```

부록 2. 템플릿(template) 적용

템플릿을 적용하여 코루틴의 리턴 타입이 달라진다고 해도 매번 새로운 타입의 promise_type을 정의하는 것이 아니라 기존 클래스에 템플릿 인자만 다르게 사용하는 방식도 적용 가능합니다. 템플릿 특수화를 통해 템플릿 인자 T의 타입에 따라 다양한 promise_type을 컴파일 타임에 지정 해줄 수도 있습니다.

```
template <class T>
class Coroutine
{
private:
    struct promise_base
    {
        Coroutine get_return_object();
        {
            return Coroutine {};
        }

        INITIAL_SUSPEND initial_suspend()
```

```
        }

        void unhandled_exception()
        {
            throw std::exception("unhandled exception");
        }

        void return_void()
        {
            return;
        }
    };

    template <class R>
    struct promise_type_impl :
public promise_base
    {
        R value;
        std::suspend_always yield_value(const R& value)
        {
            this->value = value;
            return {};
        }
    };

    template <>
    struct promise_type_impl<void> : public promise_base
    {
    };
 public:
    typedef promise_type_impl<typename T> promise_type;
 };
```

부록 3. 같이 보면 좋은 글

- [C++20] 코루틴(Coroutine) - 실전 사용 가이드
- [C++20] 코루틴(Coroutine)
- [C++20] 코루틴(Coroutine) - co_await
- [C++20] 코루틴 관련 글 더 보기

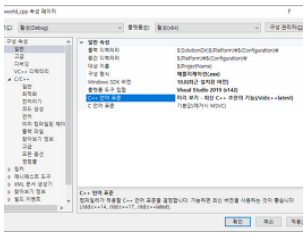
태그 #c++20, #C/C++, #Coroutine

NO IMAGE

[C++20] 코루틴(Coroutine) - 활용

NO IMAGE

[C++20] 코루틴(Coroutine) - done()



[C++20] 컴파일

NO IMAGE

[C++20] 코루틴(Coroutine) - co_await

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