진리는어디에/C++20

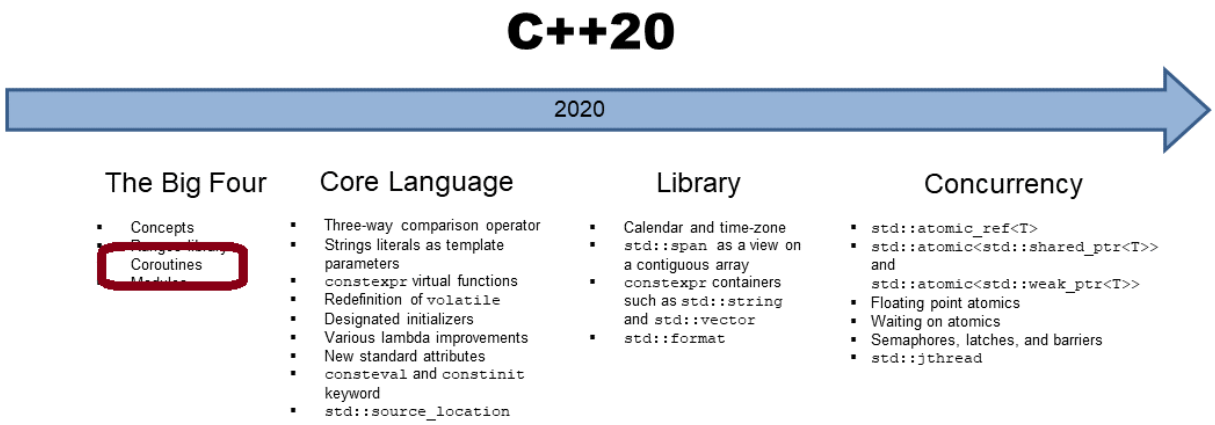
[C++20] 코루틴(Coroutine)

kukuta 2021. 3. 20. 00:53

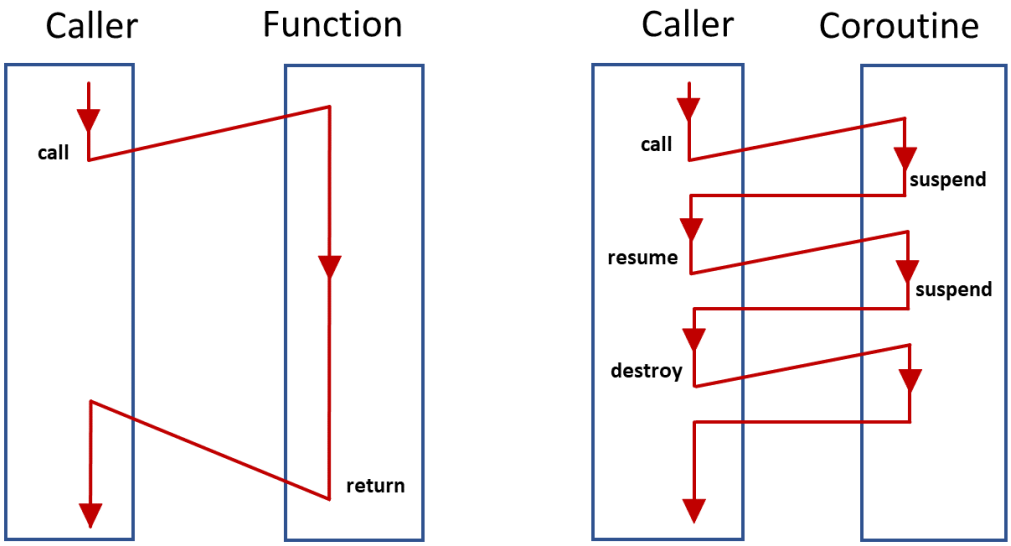
1958년 멜빈 콘웨이에 의해 처음으로 개념이 만들어 졌으며 이미 많은 현대 프로그래밍 언어들에서 지원하고 있지만 C++에만 없던 코루틴이 C++20 부터 드디어 지원됩니다.

코루틴이 지원되는 것은 너무도 반갑고 만세를 불러야 하는 일인데, 역시 C++은 C++이었습니다.
유구한 역사와 전통을 자랑하는 '자유를 주는 대신 책임도 니가 져라'라는 C++의 정신 답게 코루틴 활용 방법에 자유를 준 대신, 편리한 사용성을 가져갔습니다. 덤으로 제대로 사용하지 못하는 경우의 책임도 사용자가 져야 합니다. 만일 C#과 파이썬, 자바스크립트와 같은 다른 언어의 코루틴 경험이 있으신 분이라면 무슨 코루틴 한번 쓰기가 이렇게 어렵냐고 생각하실지도 모르겠습니다.

하지만 이 글을 끝까지 읽어 보시면 C++코루틴도 나름 익숙해지면 사용하기 편리하다는 것을 아실 수 있을겁니다(~~익숙해지면 뭐든 다 편리해 진다는 사실은 비밀~~)



먼저 워밍업으로 코루틴이 뭔지 간단하게 알아 보도록 하겠습니다. 일반적인 루틴, 그러니까 함수와 한번 비교해보죠.



C++에서 일반적인 함수의 진입점은 **함수의 시작 부분**, 딱 하나 입니다. 어떠한 경우라도 함수를 호출하면 함수의 가장 처음 부터 시작합니다. 그리고 도중에 return문을 만나거나 함수의 끝까지 실행 하면 함수를 종료합니다. 그래서 어떤 문서에서는 '함수는 단 하나의 진입 점과 단 하나의 리턴을 가지는 것이다'라고 정의하기도 하고, 다른 문서에서는 return 문이 여러개가 있을 수 있으니 '진입 점은 하나이고 return하는 부분은 여러개 일수 있는 것이 함수'다 라고 하기도 합니다만 가장 중요한 것은 '**일반함수는 한번 진입하고 한번 리턴한다**' 입니다.

일반함수는 한번 진입하고 한번 리턴한다

그럼 일반적인 루틴이 아닌, 코루틴은 어떨까요? 코루틴은 여러개의 진입 점과 여러 개의 중단점을 가질 수 있고, 루틴이 종료되기 전까지 몇 번이든 진입과 중단을 할 수 있습니다. 여기서 '리턴(return)' 이라는 말 대신 '**중단(suspend)**'이라는 용어를 사용한것을 주목해 주세요.

리턴은 함수를 완전히 종료하고 스택 메모리에 할당된 모든 리소스를 해제하는 것을 의미하지만 중단은 힙 메모리 영역에 다시 **재개(resume)** 하기 위해 필요한 모든 정보 - *코루틴 스테이트(coroutine state)라 합니다* - 를 저장하고 제어권을 호출자에게 다시 넘기는 것을 의미 합니다.

그리고 누군가가 다시 리쥼을 요청을 하면 힙 메모리 영역에 저장되어 있던 코루틴 스테이트의 정보를 가져와 복구하여, 중단 되었던 바로 다음부터 함수를 다시 진행 합니다.

이렇게 힙 메모리 영역에 코루틴의 정보를 저장하는 방식을 stackless라고하고 C++에서는 stackless 코루틴을 지원합니다. 반대로 stackful 코루틴이라는 것이 있는데 지금 다루려는 내용에서 너무 벗어나므로 stackless와 stackful에 관련된 자세한 내용은 다른 포스트에서 다루도록 하겠습니다. 당장 지금 중요한 것은 아래 정도로 요약 할 수 있습니다.

- 코루틴은 실행을 중단(suspend) 할 수 있고, 이후 중단 시점 이후 부터 실행을 재개(resume) 할 수 있는 함수 입니다. 심지어 완전히 종료 하기 전까지 몇번이고 반복 할 수 있습니다.
- C++ 코루틴은 ~~Stackless 입니다(아직 중요하지 않습니다).~~ 실행이 '중단' 될 때 '재개'에 필요한 정보들을 heap 메모리 영역에 저장 해놓습니다.

> 코루틴은 몇번이든 실행을 중단(suspend) 할 수 있고,
> 중단 시점 이후 부터 실행을 재개(resume) 할 수 있다.

C++ 코루틴의 규칙

Unity의 C#이나 코틀린, JavaScript에서 코루틴을 사용해보신 경험이 있으신 분들이라면 지금 부터 소개되는 C++ 코루틴이 복잡하고 어렵게 느껴지실 수 있습니다. 다른 프로그래밍 언어에서는 언어 차원에서 기본으로 제공되는 코루틴 반환 객체(예를 들어 C#의 IEnumerator)를 C++은 직접 각자의 입맛에 맞게 정의해서 사용 한다고 생각하면, 아 또 C++이 C++하고 있구나[1] 하고 어느정도 납득 하실수 있을 겁니다.

위와 같은 이유로 C++ 코루틴은 라이브러리라기 보단 프레임워크에 가깝습니다. 프레임워크란 Frame(틀, 규칙) + Work(일, 목적)의 합성어로, 목적을 이루기 위한 일하는 규칙들의 집합이라고 이해하시면 됩니다. 그럼 C++ 코루틴의 규칙은 무엇일까요?

규칙 1. 제약 사항

다음과 같은 경우에는 코루틴을 사용할 수 없습니다.

- 코루틴은 Variadic arguments, 일반 return 문, auto와 Concept 같은 placeholder 리턴 타입을 사용 할 수 없습니다.
- constexpr 함수와 생성자, 소멸자, 메인 함수를 코루틴으로 사용 할 수 없습니다.

규칙 2. co_await, co_yield, co_return 키워드 사용

C++에서 코루틴이라 불리기 위해서는 co_await, co_yield, co_return 키워드들 중 최소한 하나라도 사용 해야 합니다.

- 단순히 코루틴을 중단하기 위해서는 co_await를 사용 합니다.

```
task<> tcp_echo_server() {
  char data[1024];
  for (;;) {
    size_t n = co_await socket.async_read_some(buffer(data));
    co_await async_write(socket, buffer(data, n));
  }
}
```

- 만일 호출자에게 전달 할 값이 있다면 co_yield를 사용합니다.

```
generator<int> iota(int n = 0) {
  while(true)
    co_yield n++;
}
```

- 코루틴을 완전히 종료 하려 한다면 co_return을 사용합니다.

```
lazy<int> f() {
```

조 하세요.

지금은 위 키워드들에 대해서 잘 몰라도 상관 없습니다. 뒤에서 다시 자세히 설명할겁니다. 다만 코루틴이 되기 위해선 '**위 세 키워드 중 최소한 하나라도 사용 해야한다**' 라는 것이 핵심이라는 것만 알아 두시면 됩니다.

co_await, co_yield, co_return

규칙 3. 코루틴 반환 객체

C++코루틴은 C++이 정의한 규칙을 구현한 '코루틴 반환 객체'를 호출자에게 리턴해야 합니다. 코루틴이 실행 되면 '**코루틴 반환 객체**'라는 사용자 정의 타입의 객체를 호출자에게 리턴합니다. 이 '코루틴 반환 객체'야 말로 C++코루틴의 핵심이며 이번 포스트의 대부분의 지면을 할애 할 정도로 중요한 내용입니다. 그리고 코루틴 프레임워크의 대부분의 규칙은 이 코루틴 반환 객체를 중심으로 돌아갑니다. 매우 중요합니다. 집중해주세요.

코루틴 반환 객체를 설명하기 위해선 먼저 C++ 코루틴의 구성을 알아야 합니다. C++ 코루틴은 크게 아래 세가지로 이루어져 있습니다.

- **promise**. "코루틴 내부"에서 관리 되는 객체. 코루틴의 결과나 예외를 이 객체를 통해 호출자에게 전달 하는 용도로 사용합니다.
- **coroutine handle**. "코루틴 외부"에서 관리 되는 객체. 코루틴을 resume하거나 코루틴 프레임을 제거 할 때 사용 됩니다.
- **coroutine state**. "힙 메모리 영역"에 할당 되는 코루틴 상태를 나타내는 객체.
  - coroutine state는 promise 객체를 가진다.
  - coroutine state는 코루틴 함수 파라메터의 복사된 값들을 가진다.
  - coroutine state는 resume을 위한 현재 중단 포인트와 destroy를 위한 지역 변수들을 가진다.
  - 지역 변수와 임시 변수들은 현재 중단 포인트에서 살아 있는 것들을 의미한다

코루틴의 구성을 대략적이나마 파악했으니 이젠 코루틴이 어떻게 동작하는지 살펴 보도록 하겠습니다. 아직 제대로 된 코드 한줄 보여주지 않고 계속 개념 설명만 하니 답답하실 겁니다. 조금만 참고 따라와 주세요.

코루틴 실행 시 일어나는 일

1. 코루틴이 최초 실행 되면 new를 이용해 힙 메모리 영역에 coroutine state를 생성합니다.
2. 코루틴 함수의 모든 인자들을 coroutine state에 복사합니다. 이때 모든 인자들은 move 되거나 복사 됩니다. 단, 레퍼런스들은 그대로 레퍼런스로 남아 있습니다.
   NOTE - 만일 코루틴이 재개(resume) 될 때, 레퍼런스 변수들의 생명 주기가 이미 종료 되었다면 뎅글링 레퍼런스를 참조 할 수 있으므로 코루틴 함수의 인자로 레퍼런스 타입을 사용 할 때는 주의가 필요 합니다.
3. promise 객체의 생성자를 호출 합니다.
   NOTE - 만일 promise 타입의 생성자가 모든 코루틴 함수의 인자를 가지고 있다면 해당 생성자가 호출 됩니다. 그렇지 않다면 기본 생성자가 호출 됩니다.
4. promise.get_return_object() 함수를 호출 합니다. get_return_object() 함수는 "**코루틴 반환 객체"를 생성하여 리턴**하며, 이 값은 로컬 변수에 저장되었다가 최초 코루틴 중단(suspend) 시 코루틴 호출자에게 리턴 됩니다.
5. promise.initial_suspend() 를 호출하고 그 결과를 co_await 오퍼레이터에게 전달 합니다. 일반적으로 initial_suspend() 함수는 게으른 시작(lazily-start)을 위해 suspend_always를 리턴하거나, 즉시 시작(eagerly-start)을 위해 suspend_never를 리턴합니다.
6. co_await promise.initial_suspend() 이후 코루틴이 다시 재게(resume) 되면 코루틴은 그제서야 본문(사용자가 정의한코루틴 함수의 내용들)을 실행 합니다.

무슨 말인지 하나도 머리에 들어오는 것이 없을 겁니다. 걱정 마세요. 뒤에 예제와 함께 한줄 한줄 다시, 반복적으로 설명이 나갑니다. 지금은 그냥 핵심 키워드들만 기억해주세요.

코루틴 핵심 키워드들
coroutine state, promise,
get_return_object(), initial_suspend(),
co_await, suspend_always, suspend_never,
게으른 시작(lazily-start), 즉시 시작(eagerly-start)

다시 한번 말씀 드리지만, 아직 위의 내용들이 이해되지 않으셔도 괜찮습니다. 뒤에 예제와 함께 다시 자세하게 설명 됩니다. 지금은 위와 같은 일련의 과정들이 진행 되고 **promise**라는 객체의 **get_return_object() 함수**와 **initial_suspend() 함수**를 호출 하고 있다는 것만 기억하시면 됩니다.

단락의 제일 처음에 '**C++이 정의한 규칙을 구현한 사용자 정의 타입**'이라는 표현을 썼습니다. promise, get_return_object() 함수, initail_suspend() 함수들이 모두 C++이 정의한 규칙의 일부입니다.

이제 부터 이 규칙들을 어떻게 구현해 나가는지 알아 보겠습니다.

promise_type

위에서 이야기한 규칙에 따라 코루틴 반환 객체를 정의하고 그것을 리턴하고 co_await 키워드를 사용하는 코루틴(foo)을 만들었습니다.

```
1  #include <iostream>
2  #include <coroutine> // 코루틴을 사용하기 위한 헤더
3
4  // 지금은 비어 있지만 앞으로 완성 되어져갈 코루틴 반환 객체
5  class Task
6  {
7  public :
8  };
9
10 // 코루틴 함수
11 //     규칙 2. co_await를 사용한다
12 //     규칙 3. 코루틴 반환 객체(Task)를 리턴한다
13 Task foo()
14 {
15     std::cout << "foo 1" << std::endl;
16     co_await std::suspend_always{};
17     std::cout << "foo 2" << std::endl;
18 }
19
20 int main()
21 {
22     std::cout << "\t main 1" << std::endl;
23     foo();
24     std::cout << "\t main 2" << std::endl;
25 }
```

~~Task의 객체를 리턴하는 곳이 어디에도 없는데 반환 값이 Task라는 것이 이상해 보이긴 하지만 설명은 뒤로 미루고,~~ 위 코드에서 추가 설명이 필요한 16라인 co_await std::suspend_always{}에 대한 설명을 하겠습니다.

co_await 단항 연산자는 awaitable object 인 std::suspend_alway의 객체를 인자로 받습니다. awaitable object 도 코루틴 반환 객체 처럼 일련의 규칙을 구현한 객체입니다. 하지만 자세한 설명은 뒤로 미루고 일단

- 'std::suspend_always는 awaitable object다.
- co_await std::suspend_always{}를 만나면 **코루틴이 중단되고 호출자에게 제어권을 넘긴다**'

정도만 기억해주시기 바랍니다. 참고로 또 다른 awaitable object 인 std::suspend_never는 코루틴을 중단하지 않고 계속 진행 시킵니다.

NOTE - awaitable object 또한 일련의 규칙을 따르기만 하면 커스텀 awaitable object를 만들어 사용하는 것이 가능합니다. 이를 이용해 중단하고 호출자가 아닌 다른 스레드등에게 제어권을 넘겨 멀티 스레드를 싱글 스레드 환경에서 사용하는 것 처럼 하는 다양한 활용법들이 있습니다. 이는 다른 포스트에서 따로 설명합니다.

이제 기본 작성이 완료 되었으니 컴파일 해봅시다. (C++20 컴파일 방법은 [여기]를 참고해주세요)

아마 여러분은 **"class std::coroutine_traits<void>에 promise_type 멤버가 없습니다"**와 비슷한 에러를 보게 되실 겁니다. 컴파일도 안되는 코드를 작성하고 컴파일을 시킨 이유는 다음을 설명하기 위함입니다.

먼저 위에서 살펴본 "코루틴 실행 시 일어나는 일"을 상기해 봅시다.

1. ~~코루틴이 최초 실행 되면 new를 이용해 힙 메모리 영역에 coroutine state를 생성합니다.~~
2. ~~코루틴 함수의 모든 인자들을 coroutine state에 복사합니다.~~
3. promise 객체의 생성자를 호출 합니다.
4. promise.get_return_object() 에서 "**코루틴 반환 객체**"를 생성하여 최초 중단(suspend) 시 호출자에게 전달합니다.
5. promise.initial_suspend() 를 호출하고 그 결과를 co_await 오퍼레이터에게 전달 합니다.
6. co_await promise.initial_suspend() 이후 코루틴이 재게(resume)되면 본문을 실행 합니다.

1, 2번은 C++이 내부적으로 알아서 처리해주므로 우리가 신경 쓸 필요는 없는 부분입니다.

중요한 것은 3번! promise 객체의 생성자 호출 부분입니다. 위 컴파일 에러 메시지로 출력된 promise_type라는 타입의 인스턴스가 여기서 말하는 promise 객체 입니다.

컴파일러가 컴파일 중 co_await 문을 만나게 되면 foo() 함수 안에 아래와 비슷한 코드를 생성합니다. 주석 처리 된 부분은 기존에 여러분이 작성하신 코드고 그렇지 않은 부분이 컴파일러가 생성한 코드들 입니다.

```
// Task foo()
// {
      // 3. promise 객체의 생성자를 호출
      Task::promise_type promise;

      // 4. promise.get_return_object()로 부터 "코루틴 반환 객체(Task)" 생성
      Task task = promise.get_return_object();

      // 5. promise.initial_suspend()를 호출하고, 그 결과를 co_await 에게 전달
      co_await promise.initial_suspend();
      try
      {
//        std::cout << "foo 1" << std::endl;
//        co_await std::suspend_always{};
//        std::cout << "foo 2" << std::endl;
      }
      catch (...)
      {
          promise.unhandled_exception();
      }
      promise.return_void();
      co_await promise.final_suspend();
//}
```

4라인을 보시면 컴파일러는 Task 사용자 정의 클래스 안에 promise_type이 있을 것이라 가정하고 코드를 생성했는데 정작 Task 클래스 안에는 아무것도 없습니다. 네. 이 한가지 규칙을 추가하기 앞에 이 많은 설명들을 한겁니다.

코루틴 반환 객체는 promise_type이라는 클래스 또는 구조체를 내부에 정의하고 있어야 합니다. 정리하면 아래와 같습니다.

- **코루틴 반환 객체는 C++에서 정의한 규칙을 구현한 promise_type 이라는 이름의 타입이 정의되어야 한다(필수).**
- std::coroutine_handle<promise_type> 타입의 멤버 변수가 있어야 한다(선택).
- std::coroutine_handle<promise_type> 을 인자로 받아 멤버 변수를 초기화 하는 생성자가 있어야 한다(선택).
- 소멸자에서 std::coroutine_handle<promise_type> 타입의 코루틴 핸들러 멤버 변수를 해제 해야 한다(선택).

필수라고 명시된 부분은 지켜주지 않으면 컴파일 오류가 발생합니다. 나머지는 제가 코루틴을 사용하다 보니 저렇게 하는것이 더 편해서 임의로 추가한 가이드라인이라고 생각하시면 됩니다.

위에서 "C++에서 정의한 규칙을 구현한 promise_type..."이라고 했습니다. 네. promise_type도 규칙을 가지고 있고 그 규칙들을 만족(구현)해야지만 promise_type으로써 사용 가능합니다.

promise_type은 아래 함수들을 구현해야 합니다(중요합니다!!).

- get_return_object() : 코루틴 중단을 위한 사용자 정의 "코루틴 반환 객체"를 반환 한다.
- initial_suspend() : 코루틴 최초 실행 시 호출. awaitable 객체를 반환 한다.
- unhandled_exception() : 코루틴 수행 중 예외 발생 시 호출
- yield_value() : co_yield를 사용하는 경우 구현. 나중에 코루틴 yield를 설명 할 때 같이 설명
- return_void() : co_return을 사용하는 경우 구현. 나중에 코루틴 종료를 설명 할 때 같이 설명
- final_suspend() : 코루틴 종료 시 호출. 나중에 코루틴 종료를 설명 할 때 같이 설명

위 규칙들을 적용해서 promise_type을 추가한 코루틴 반환 객체(Task)를 다시 정의 해보겠습니다.

```
class Task
{
public:
    // 규칙 1. C++에서 정의된 규칙을 구현한 promise_type 이라는 이름의 타입이 정의되어야 한다.
    struct promise_type
    {
        // 사용자 정의 "코루틴 반환 객체"를 반환 한다
        Task get_return_object()
        {
              return Task { std::coroutine_handle<promise_type>::from_promise(*this) };
        }

        // 코루틴 최초 실행 시 호출. awaitable 객체를 반환 한다.
        auto initial_suspend() { return std::suspend_always{}; }

        // co_return을 사용하는 경우 구현. 나중에 코루틴 종료를 설명 할 때 같이 설명
        auto return_void() { return std::suspend_never{}; }

        // 코루틴 종료 시 호출. 나중에 코루틴 종료를 설명 할 때 같이 설명
        auto final_suspend() { return std::suspend_always{}; }

        // 코루틴 수행 중 예외 발생 시 호출
        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 변수의 destroy를 호출 해야 한다.
    ~Task()
    {
        if (true == (bool)co_handler)
        {
            co_handler.destroy();
        }
    }
};
```

```
main 1
main 2
```

아..도데체 코루틴은 언제 실행 할 수 있는..

이유가 뭘까요? 이유를 알기 위해서는 위에서 살펴본 "코루틴 실행 시 일어나는 일"에서 언급한 게으른 시작과, 중단(supend) 되고 재개(resume) 될 때 컴파러가 생성하는 추가 코드를 살펴 봐야 할 필요가 있습니다. 이제 진짜 끝까지 왔습니다. 조금만 힘내세요.

Suspend & Resume

앞에서 컴파일은 되었지만 전혀 실행 되지 않는 현상을 보았습니다. 원인을 설명하기 위해 '코루틴 실행 시 일어나는 일'을 다시 살펴 보아야 할 필요가 있습니다.

1. ~~코루틴이 최초 실행 되면 new를 이용해 힙 메모리 영역에 coroutine state를 생성합니다.~~
2. ~~코루틴 함수의 모든 인자들을 coroutine state에 복사합니다.~~
3. ~~promise 객체의 생성자를 호출 합니다.~~
4. promise.get_return_object() 에서 코루틴 반환 객체를 생성하여 로컬 변수에 저장 하였다가 **최초 중단(suspend) 시 호출자에게 전달** 합니다.
5. promise.initial_suspend() 를 호출하고 그 결과를 co_await 오퍼레이터에게 전달 합니다.
6. co_await promise.initial_suspend() 이후 **코루틴이 재게(resume)되면 본문을 실행** 합니다.

위 과정 우리가 주목해야 할 부분은 5번입니다.

```
class Task
{
public:
    struct promise_type
    {
        ... 생략 ...
        auto initial_suspend() { return std::suspend_always{}; }
        ... 생략 ...
    };
};

    ... 생략 ...
    Task::promise_type promise;

    // 4. 코루틴 반환 객체를 생성하여 로컬 변수에 저장
    Task task = promise.get_return_object();
```

```
        // co_await std::suspend_always{};
        // std::cout << "foo 2" << std::endl;
    }
    ... 생략 ...
```

우리는 promise_type을 정의 할 때 initial_suspend() 함수에서 무조건 std::suspend_always 객체를 리턴하도록 했습니다. std::suspend_always는 코루틴을 중단하고 제어권을 호출자에게 넘기도록 정의된 awaitable object입니다.

호출자가 foo()를 "최초" 호출하면, foo()함수의 본문 코드가 실행하기 전, initial_suspend()에서 리턴한 std::suspend_always 객체로 인해 코루틴을 중단하고 호출자에게 다시 제어권을 넘겨 버렸기 때문에 코루틴이 전혀 실행 되지 못한것 처럼 보였던 것입니다.

"코루틴 핵심 키워드"중에 하나인 **게으른 시작(lazily-start)**이 바로 이것 입니다. 게으른 시작은 코루틴을 실행하면 바로 사용자가 정의한 내용을 실행하지 않고 호출자에게 제어권을 넘긴 후, 실제 실행은 호출자가 resume를 실행 했을때 부터 시작합니다. 반대로 initial_suspend()에서 suspend_never를 리턴했다면 최초 코루틴 호출 시점에 foo()에서 작성한 코드들이 실행 됩니다. 이것이 위에서 언급한 즉시 시작(eagerly-start) 입니다.

다시 본론으로 돌아와서, 코루틴이 하나도 실행 되지 않은 이유는 알게 되었습니다. 그럼 우리가 작성한 코드가 실행 되게 하기 위해서는 어떻게 해야 할까요?

위에서 "**코루틴이 재게(resume)되면 본문을 실행**"한다고 했습니다. 코루틴 외부에서 코루틴을 재개하거나 코루틴 프레임을 삭제하기 위해서는 coroutine_handle이 필요하다고 했습니다.

coroutine handle
코루틴 외부에서 관리 되는 객체.
코루틴을 resume하거나 코루틴 프레임을 제거 할 때 사용.

```
void std::coroutine_handle<Promise>::operator() const noexcept;
void std::coroutine_handle<Promise>::resume() const noexcept;
```

코루틴의 재개(resume)는 coroutine handle의 ()오퍼레이터나 resume() 함수를 호출하면 됩니다. 다행히 우리는 이미 코루틴 반환 객체(Task)에 코루틴 핸들을 멤버로 가지도록 정의 했었습니다. 그리고 코루틴 반환 객체는 코루틴의 최초 중단 시점에 호출자에게 전달 됩니다.

이제는 실제 코루틴에서 중단 이후 다시 재개하기 위해 메인 함수에서 resume()을 호출하는 코드를 다시 작성해보겠습니다.

```
int main()
{
    // 코루틴 foo()를 실행하면 본문 실행 전에 중단하고 호출자(main)에게 제어권 넘김
    // 최초 중단시 코루틴 반환 객체(Task)를 호출자에게 돌려 준다
    Task task = foo();
    std::cout << "\t main 1" << std::endl;

    // 코루틴 반환 객체의 멤버 coroutine_handle.resume()을 이용해 코루틴 재개
    task.co_handler.resume();
    std::cout << "\t main 2" << std::endl;
    task.co_handler.resume();
}
```

먼저 위에서 설명한 대로 foo() 함수를 실행하면 본문 코드를 실행하기 전에 중단하고 호출자에게 제어권을 넘깁니다. 최초 코루틴 실행시 initial_suspend() 이후 제어권을 넘기면서 Task 객체 리턴합니다.

우리는 이 Task 반환 객체를 저장했다가 다시 resume하고 싶은 시점에 task의 coroutine_hander의 resume() 함수를 호출 해주면 foo() 함수의 마지막 실행 시점으로 돌아 갈 수 있습니다.

위 코드를 컴파일 하고 실행하면 아래와 같은 결과를 볼 수 있습니다.

마치며

이상으로 C++20에서 부터 제공되는 코루틴을 만들어 보고, 하나의 스레드에서 두 함수(호출자와 피호출 함수)가 제어권을 서로 넘기는 방법을 살펴 보았습니다. 이번 포스트는 이정도로 마무리 하도록하고 다음 포스트는 promise를 이용해 코루틴에서 호출자로 값을 반환 하는 것을 알아 보도록 하겠습니다.

부록 1. 같이 보면 좋은 글

- [C++20] 신규 기능 정리
- [C++20] 컴파일
- [C++20] 코루틴(Coroutine) - co_await
- [C++20] 코루틴(Coroutine) - co_yield
- [C++20] 코루틴(Coroutine) - co_return
- [C++20] 코루틴(Coroutine) - done()
- [C++20] 코루틴(Coroutine) - 실전 사용 가이드

부록 2. 전체 예제 코드

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

class Task
{
public:
    // 규칙 1. C++에서 정의된 규칙을 구현한 promise_type 이라는 이름의 타입이 정의되어야 한다.
    struct promise_type
    {
        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 (true == (bool)co_handler)
        {
            co_handler.destroy();
        }
    }
};

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

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

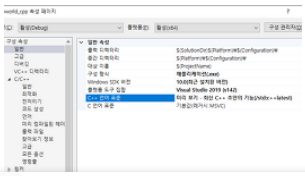
부록 3. G++(or GCC)에서 코루틴 컴파일 시 에러 대처

- [C++20] 컴파일

1. 전통적으로 C++은 사용자의 편의성 보다는 책임과 권한을 동시에 주고 사용자에게 모든걸 맡겨 버려 왔습니다 [본문으로]

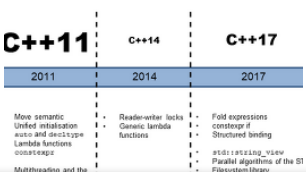
태그 #c++20, #C/C++, #Coroutine

'진리는어디에/C++20' Related Articles



NO IMAGE

NO IMAGE



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

1 ··· 128 129 130 131 132 133 134 135 136 ··· 342