Corrotinas em C++

Introdução

Corrotinas em C++ são uma extensão da funcionalidade das funções que permitem pausar e retomar a execução em pontos específicos. Introduzidas no C++20, elas são usadas para simplificar a programação assíncrona e concorrente, proporcionando uma maneira eficiente de gerenciar tarefas sem bloquear a execução.

1. Definição e Sintaxe

- Definição: Uma corrotina é uma função que pode ser suspensa e retomada, permitindo a execução de código de forma não sequencial.
- Sintaxe:

```
co_return expr;       // Retorna um valor da corrotina
co_await expr;        // Suspende a corrotina até que expr esteja pronto
co_yield expr;        // Suspende a corrotina e retorna um valor temporário
```

2. Exemplo de Corrotina Simples

- Exemplo:

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

struct Coro {
    struct promise_type {
```

```
        Coro get_return_object() { return {}; }
        std::suspend_never initial_suspend() { return {}; }
        std::suspend_never final_suspend() noexcept { return {}; }
        void return_void() {}
        void unhandled_exception() { std::exit(1); }
    };
};

Coro exemplo() {
    std::cout << "Início da corrotina
";
    co_await std::suspend_always{};
    std::cout << "Fim da corrotina
";
}

int main() {
    auto handle = exemplo();
    handle.resume();
    return 0;
}
```

3. Uso de `co_await`

- Definição: `co_await` é usado para pausar a execução de uma corrotina até que uma expressão esteja pronta. A expressão pode ser um futuro, uma tarefa ou qualquer tipo que suporte a espera.

- Exemplo:

```
#include <iostream>
#include <coroutine>
#include <future>

std::future<void> exemplo() {
    std::cout << "Início da corrotina
";
    co_await std::suspend_always{};
    std::cout << "Fim da corrotina
";
}

int main() {
    auto fut = exemplo();
    fut.get();
    return 0;
}
```

4. Uso de `co_yield`

- Definição: `co_yield` é usado para retornar temporariamente um valor da corrotina sem finalizá-la.
- Exemplo:

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

```
struct Generator {
    struct promise_type {
        int current_value;
        Generator get_return_object() { return Generator{ this }; }
        std::suspend_always initial_suspend() { return {}; }
        std::suspend_always final_suspend() noexcept { return {}; }
        std::suspend_always yield_value(int value) {
            current_value = value;
            return {};
        }
        void return_void() {}
        void unhandled_exception() { std::exit(1); }
    };

    struct iterator {
        std::coroutine_handle<promise_type> handle;
        bool done;

        iterator(std::coroutine_handle<promise_type> h, bool d)
            : handle(h), done(d) {}

        iterator& operator++() {
            handle.resume();
            done = handle.done();
            return *this;
        }
```

```
        int operator*() const { return handle.promise().current_value; }

        bool operator!=(const iterator& other) const { return done != other.done; }
    };

    std::coroutine_handle<promise_type> handle;

    Generator(promise_type* p)
        : handle(std::coroutine_handle<promise_type>::from_promise(*p)) {}

    iterator begin() {
        handle.resume();
        return { handle, handle.done() };
    }

    iterator end() { return { handle, true }; }
};

Generator generate_numbers() {
    for (int i = 0; i < 10; ++i) {
        co_yield i;
    }
}

int main() {
```

```
    for (int value : generate_numbers()) {
        std::cout << value << "
";
    }
    return 0;
}
```

5. Tipos de Promessas

- Definição: Corrotinas utilizam um tipo de promessa (`promise_type`) que define como a corrotina é inicializada, suspensa e finalizada. A promessa gerencia o estado da corrotina e fornece pontos de personalização.
- Exemplo:

```
struct Coro {
    struct promise_type {
                                                Coro get_return_object() { return Coro{
std::coroutine_handle<promise_type>::from_promise(*this) }; }
        std::suspend_always initial_suspend() { return {}; }
        std::suspend_always final_suspend() noexcept { return {}; }
        void return_void() {}
        void unhandled_exception() { std::exit(1); }
    };

    std::coroutine_handle<promise_type> handle;

    Coro(std::coroutine_handle<promise_type> h) : handle(h) {}
```

```
    ~Coro() {
        if (handle) handle.destroy();
    }

    void resume() { handle.resume(); }
};
```

Dicas de Boas Práticas

- Clareza: Use corrotinas para simplificar o código assíncrono e evitar aninhamentos profundos de callbacks.
- Eficiência: Utilize `co_await` e `co_yield` para gerenciar a execução sem bloquear a thread.
- Gerenciamento de Recursos: Garanta que recursos alocados dentro da corrotina sejam liberados adequadamente.

Esta seção abrange os conceitos sobre corrotinas em C++. Para mais detalhes, consulte a documentação oficial: https://en.cppreference.com/w/cpp/language/coroutines