C++ Coroutines: Understanding the promise type

Sep 5, 2018

This post is the third in the series on the C++ Coroutines TS (N4736).

The previous articles in this series cover:

- Coroutine Theory
- Understanding operator co_await

In this post I look at the mechanics of how the compiler translates coroutine code that you write into compiled code and how you can customise the behaviour of a coroutine by defining your own **Promise** type.

Coroutine Concepts

The Coroutines TS adds three new keywords: `co_await`, `co_yield` and `co_return`. Whenever you use one of these coroutine keywords in the body of a function this triggers the compiler to compile this function as a coroutine rather than as a normal function.

The compiler applies some fairly mechanical transformations to the code that you write to turn it into a state-machine that allows it to suspend execution at particular points within the function and then later resume execution.

In the previous post I described the first of two new interfaces that the Coroutines TS introduces: The **Awaitable** interface. The second interface that the TS introduces that is important to this code transformation is the **Promise** interface.

The **Promise** interface specifies methods for customising the behaviour of the coroutine itself. The library-writer is able to customise what happens when the coroutine is called, what happens when the coroutine returns (either by normal means or via an unhandled exception) and customise the behaviour of any `co_await` or `co_yield` expression within the coroutine.

Promise objects

The **Promise** object defines and controls the behaviour of the coroutine itself by implementing methods that are called at specific points during execution of the coroutine.

> *Before we go on, I want you to try and rid yourself of any preconceived notions of what a "promise" is. While, in some use-cases, the coroutine promise object does indeed act in a similar role to the `std::promise` part of a `std::future` pair, for other use-cases the analogy is somewhat stretched. It may be easier to think about the coroutine's promise object as being a "coroutine state controller" object that controls the behaviour of the coroutine and can be used to track its state.*

An instance of the promise object is constructed within the coroutine frame for each invocation of a coroutine function.

The compiler generates calls to certain methods on the promise object at key points during execution of the coroutine.

In the following examples, assume that the promise object created in the coroutine frame for a particular invocation of the coroutine is `promise`.

When you write a coroutine function that has a body, `<body-statements>`, which contains one of the coroutine keywords (`co_return`, `co_await`, `co_yield`) then the body of the coroutine is transformed to something (roughly) like the following:

```
{
  co_await promise.initial_suspend();
  try
  {
    <body-statements>
  }
  catch (...)
  {
    promise.unhandled_exception();
  }
FinalSuspend:
  co_await promise.final_suspend();
}
```

When a coroutine function is called there are a number of steps that are performed prior to executing the code in the source of the coroutine body that are a little different to regular functions.

Here is a summary of the steps (I'll go into more detail on each of the steps below).

1. Allocate a coroutine frame using `operator new` (optional).
2. Copy any function parameters to the coroutine frame.
3. Call the constructor for the promise object of type, `P`.
4. Call the `promise.get_return_object()` method to obtain the result to return to the caller when the coroutine first suspends. Save the result as a local variable.
5. Call the `promise.initial_suspend()` method and `co_await` the result.

6. When the `co_await promise.initial_suspend()` expression resumes (either immediately or asynchronously), then the coroutine starts executing the coroutine body statements that you wrote.

Some additional steps are executed when execution reaches a `co_return` statement:

1. Call `promise.return_void()` or `promise.return_value(<expr>)`
2. Destroy all variables with automatic storage duration in reverse order they were created.
3. Call `promise.final_suspend()` and `co_await` the result.

If instead, execution leaves `<body-statements>` due to an unhandled exception then:

1. Catch the exception and call `promise.unhandled_exception()` from within the catch-block.
2. Call `promise.final_suspend()` and `co_await` the result.

Once execution propagates outside of the coroutine body then the coroutine frame is destroyed. Destroying the coroutine frame involves a number of steps:

1. Call the destructor of the promise object.
2. Call the destructors of the function parameter copies.
3. Call `operator delete` to free the memory used by the coroutine frame (optional)
4. Transfer execution back to the caller/resumer.

When execution first reaches a `<return-to-caller-or-resumer>` point inside a `co_await` expression, or if the coroutine runs to completion without hitting a `<return-to-caller-or-resumer>` point, then the coroutine is either suspended or destroyed and the return-object previously returned from the call to `promise.get_return_object()` is then returned to the caller of the coroutine.

Allocating a coroutine frame

First, the compiler generates a call to `operator new` to allocate memory for the coroutine frame.

If the promise type, `P`, defines a custom `operator new` method then that is called, otherwise the global `operator new` is called.

There are a few important things to note here:

The size passed to `operator new` is not `sizeof(P)` but is rather the size of the entire coroutine frame and is determined automatically by the compiler based on the number and sizes of parameters, size of the promise object, number and sizes of local variables and other compiler-specific storage needed for management of coroutine state.

The compiler is free to elide the call to `operator new` as an optimisation if:

- it is able to determine that the lifetime of the coroutine frame is strictly nested within the lifetime of the caller; and
- the compiler can see the size of coroutine frame required at the call-site.

In these cases, the compiler can allocate storage for the coroutine frame in the caller's activation frame (either in the stack-frame or coroutine-frame part).

The Coroutines TS does not yet specify any situations in which the allocation elision is guaranteed, so you still need to write code as if the allocation of the coroutine frame may fail with `std::bad_alloc`. This also means that you usually shouldn't declare a coroutine function as `noexcept` unless you are ok with `std::terminate()` being called if the coroutine fails to allocate memory for the coroutine frame.

There is a fallback, however, that can be used in lieu of exceptions for handling failure to allocate the coroutine frame. This can be necessary when operating in environments where exceptions are not allowed, such as embedded environments or high-performance environments where the overhead of exceptions is not tolerated.

If the promise type provides a static `P::get_return_object_on_allocation_failure()` member function then the compiler will generate a call to the `operator new(size_t, nothrow_t)` overload instead. If that call returns `nullptr` then the coroutine will immediately call `P::get_return_object_on_allocation_failure()` and return the result to the caller of the coroutine instead of throwing an exception.

Customising coroutine frame memory allocation

Your promise type can define an overload of `operator new()` that will be called instead of global-scope `operator new` if the compiler needs to allocate memory for a coroutine frame that uses your promise type.

For example:

```
struct my_promise_type
{
  void* operator new(std::size_t size)
  {
    void* ptr = my_custom_allocate(size);
    if (!ptr) throw std::bad_alloc{};
    return ptr;
  }

  void operator delete(void* ptr, std::size_t size)
  {
    my_custom_free(ptr, size);
  }
```

```
  ...
};
```

"But what about custom allocators?", I hear you asking.

You can also provide an overload of `P::operator new()` that takes additional arguments which will be called with lvalue references to the coroutine function parameters if a suitable overload can be found. This can be used to hook up `operator new` to call an `allocate()` method on an allocator that was passed as an argument to the coroutine function.

You will need to do some extra work to make a copy of the allocator inside the allocated memory so you can reference it in the corresponding call to `operator delete` since the parameters are not passed to the corresponding `operator delete` call. This is because the parameters are stored in the coroutine-frame and so they will have already been destructed by the time that `operator delete` is called.

For example, you can implement `operator new` so that it allocates extra space after the coroutine frame and use that space to stash a copy of the allocator that can be used to free the coroutine frame memory.

For example:

```
template<typename ALLOCATOR>
struct my_promise_type
{
  template<typename... ARGS>
  void* operator new(std::size_t sz, std::allocator_arg_t, ALLOCATOR& allocato
  {
    // Round up sz to next multiple of ALLOCATOR alignment
    std::size_t allocatorOffset =
      (sz + alignof(ALLOCATOR) - 1u) & ~(alignof(ALLOCATOR) - 1u);

    // Call onto allocator to allocate space for coroutine frame.
    void* ptr = allocator.allocate(allocatorOffset + sizeof(ALLOCATOR));

    // Take a copy of the allocator (assuming noexcept copy constructor here)
    new (((char*)ptr) + allocatorOffset) ALLOCATOR(allocator);

    return ptr;
  }

  void operator delete(void* ptr, std::size_t sz)
  {
    std::size_t allocatorOffset =
      (sz + alignof(ALLOCATOR) - 1u) & ~(alignof(ALLOCATOR) - 1u);

    ALLOCATOR& allocator = *reinterpret_cast<ALLOCATOR*>(
```

```
      ((char*)ptr) + allocatorOffset);

    // Move allocator to local variable first so it isn't freeing its
    // own memory from underneath itself.
    // Assuming allocator move-constructor is noexcept here.
    ALLOCATOR allocatorCopy = std::move(allocator);

    // But don't forget to destruct allocator object in coroutine frame
    allocator.~ALLOCATOR();

    // Finally, free the memory using the allocator.
    allocatorCopy.deallocate(ptr, allocatorOffset + sizeof(ALLOCATOR));
  }
}
```

To hook up the custom `my_promise_type` to be used for coroutines that pass `std::allocator_arg` as the first parameter, you need to specialise the `coroutine_traits` class (see section on `coroutine_traits` below for more details).

For example:

```
namespace std::experimental
{
  template<typename ALLOCATOR, typename... ARGS>
  struct coroutine_traits<my_return_type, std::allocator_arg_t, ALLOCATOR, ARG
  {
    using promise_type = my_promise_type<ALLOCATOR>;
  };
}
```

Note that even if you customise the memory allocation strategy for a coroutine, **the compiler is still allowed to elide the call to your memory allocator**.

Copying parameters to the coroutine frame

The coroutine needs to copy any parameters passed to the coroutine function by the original caller into the coroutine frame so that they remain valid after the coroutine is suspended.

If parameters are passed to the coroutine by value, then those parameters are copied to the coroutine frame by calling the type's move-constructor.

If parameters are passed to the coroutine by reference (either lvalue or rvalue), then only the references are copied into the coroutine frame, not the values they point to.

Note that for types with trivial destructors, the compiler is free to elide the copy of the parameter if the parameter is never referenced after a reachable `<return-to-caller-or-resumer>` point in the coroutine.

There are many gotchas involved when passing parameters by reference into coroutines as you cannot necessarily rely on the reference remaining valid for the lifetime of the coroutine. Many common techniques used with normal functions, such as perfect-forwarding and universal-references, can result in code that has undefined behaviour if used with coroutines. Toby Allsopp has written a great article on this topic if you want more details.

If any of the parameter copy/move constructors throws an exception then any parameters already constructed are destructed, the coroutine frame is freed and the exception propagates back out to the caller.

Constructing the promise object

Once all of the parameters have been copied into the coroutine frame, the coroutine then constructs the promise object.

The reason the parameters are copied prior to the promise object being constructed is to allow the promise object to be given access to the post-copied parameters in its constructor.

First, the compiler checks to see if there is an overload of the promise constructor that can accept lvalue references to each of the copied parameters. If the compiler finds such an overload then the compiler generates a call to that constructor overload. If it does not find such an overload then the compiler falls back to generating a call to the promise type's default constructor.

Note that the ability for the promise constructor to "peek" at the parameters was a relatively recent change to the Coroutines TS, being adopted in N4723 at the Jacksonville 2018 meeting. See P0914R1 for the proposal. Thus it may not be supported by some older versions of Clang or MSVC.

If the promise constructor throws an exception then the parameter copies are destructed and the coroutine frame freed during stack unwinding before the exception propagates out to the caller.

Obtaining the return object

The first thing a coroutine does with the promise object is obtain the `return-object` by calling `promise.get_return_object()`.

The `return-object` is the value that is returned to the caller of the coroutine function when the coroutine first suspends or after it runs to completion and execution returns to the caller.

You can think of the control flow going something (very roughly) like this:

```
// Pretend there's a compiler-generated structure called 'coroutine_frame'
// that holds all of the state needed for the coroutine. It's constructor
// takes a copy of parameters and default-constructs a promise object.
struct coroutine_frame { ... };

T some_coroutine(P param)
{
  auto* f = new coroutine_frame(std::forward<P>(param));

  auto returnObject = f->promise.get_return_object();

  // Start execution of the coroutine body by resuming it.
  // This call will return when the coroutine gets to the first
  // suspend-point or when the coroutine runs to completion.
  coroutine_handle<decltype(f->promise)>::from_promise(f->promise).resume();

  // Then the return object is returned to the caller.
  return returnObject;
}
```

Note that we need to obtain the return-object before starting the coroutine body since the coroutine frame (and thus the promise object) may be destroyed prior to the call to `coroutine_handle::resume()` returning, either on this thread or possibly on another thread, and so it would be unsafe to call `get_return_object()` after starting execution of the coroutine body.

The initial-suspend point

The next thing the coroutine executes once the coroutine frame has been initialised and the return object has been obtained is execute the statement `co_await promise.initial_suspend();`.

This allows the author of the `promise_type` to control whether the coroutine should suspend before executing the coroutine body that appears in the source code or start executing the coroutine body immediately.

If the coroutine suspends at the initial suspend point then it can be later resumed or destroyed at a time of your choosing by calling `resume()` or `destroy()` on the coroutine's `coroutine_handle`.

The result of the `co_await promise.initial_suspend()` expression is discarded so implementations should generally return `void` from the `await_resume()` method of the awaiter.

It is important to note that this statement exists outside of the `try` / `catch` block that guards the rest of the coroutine (scroll back up to the definition of the coroutine body if

you've forgotten what it looks like). This means that any exception thrown from the `co_await promise.initial_suspend()` evaluation prior to hitting its `<return-to-caller-or-resumer>` will be thrown back to the caller of the coroutine after destroying the coroutine frame and the return object.

Be aware of this if your `return-object` has RAII semantics that destroy the coroutine frame on destruction. If this is the case then you want to make sure that `co_await promise.initial_suspend()` is `noexcept` to avoid double-free of the coroutine frame.

> *Note that there is a proposal to tweak the semantics so that either all or part of the `co_await promise.initial_suspend()` expression lies inside try/catch block of the coroutine-body so the exact semantics here are likely to change before coroutines are finalised.*

For many types of coroutine, the `initial_suspend()` method either returns `std::experimental::suspend_always` (if the operation is lazily started) or `std::experimental::suspend_never` (if the operation is eagerly started) which are both `noexcept` awaitables so this is usually not an issue.

Returning to the caller

When the coroutine function reaches its first `<return-to-caller-or-resumer>` point (or if no such point is reached then when execution of the coroutine runs to completion) then the `return-object` returned from the `get_return_object()` call is returned to the caller of the coroutine.

Note that the type of the `return-object` doesn't need to be the same type as the return-type of the coroutine function. An implicit conversion from the `return-object` to the return-type of the coroutine is performed if necessary.

> *Note that Clang's implementation of coroutines (as of 5.0) defers executing this conversion until the return-object is returned from the coroutine call, whereas MSVC's implementation as of 2017 Update 3 performs the conversion immediately after calling `get_return_object()`. Although the Coroutines TS is not explicit on the intended behaviour, I believe MSVC has plans to change their implementation to behave more like Clang's as this enables some interesting use cases.*

Returning from the coroutine using `co_return`

When the coroutine reaches a `co_return` statement, it is translated into either a call to `promise.return_void()` or `promise.return_value(<expr>)` followed by a `goto FinalSuspend;`.

The rules for the translation are as follows:

- `co_return;`
  -> `promise.return_void();`
- `co_return <expr>;`
  -> `<expr>; promise.return_void();` if `<expr>` has type `void`
  -> `promise.return_value(<expr>);` if `<expr>` does not have type `void`

The subsequent `goto FinalSuspend;` causes all local variables with automatic storage duration to be destructed in reverse order of construction before then evaluating `co_await promise.final_suspend();`.

Note that if execution runs off the end of a coroutine without a `co_return` statement then this is equivalent to having a `co_return;` at the end of the function body. In this case, if the `promise_type` does not have a `return_void()` method then the behaviour is undefined.

If either the evaluation of `<expr>` or the call to `promise.return_void()` or `promise.return_value()` throws an exception then the exception still propagates to `promise.unhandled_exception()` (see below).

Handling exceptions that propagate out of the coroutine body

If an exception propagates out of the coroutine body then the exception is caught and the `promise.unhandled_exception()` method is called inside the `catch` block.

Implementations of this method typically call `std::current_exception()` to capture a copy of the exception to store it away to be later rethrown in a different context.

Alternatively, the implementation could immediately rethrow the exception by executing a `throw;` statement. For example see folly::Optional However, doing so will (likely - see below) cause the the coroutine frame to be immediately destroyed and for the exception to propagate out to the caller/resumer. This could cause problems for some abstractions that assume/require the call to `coroutine_handle::resume()` to be `noexcept`, so you should generally only use this approach when you have full control over who/what calls `resume()`.

Note that the current Coroutines TS wording is a little unclear on the intended behaviour if the call to `unhandled_exception()` rethrows the exception (or for that matter if any of the logic outside of the try-block throws an exception).

My current interpretation of the wording is that if control exits the coroutine-body, either via exception propagating out of `co_await promise.initial_suspend()`, `promise.unhandled_exception()` or `co_await promise.final_suspend()` or by the coroutine running to completion by `co_await p.final_suspend()` completing synchronously then the coroutine frame is automatically destroyed before execution returns to the caller/resumer. However, this interpretation has its own issues.

A future version of the Coroutines specification will hopefully clarify the situation. However, until then I'd stay away from throwing exceptions out of `initial_suspend()`, `final_suspend()` or `unhandled_exception()`. Stay tuned!

The final-suspend point

Once execution exits the user-defined part of the coroutine body and the result has been captured via a call to `return_void()`, `return_value()` or `unhandled_exception()` and any local variables have been destructed, the coroutine has an opportunity to execute some additional logic before execution is returned back to the caller/resumer.

The coroutine executes the `co_await promise.final_suspend();` statement.

This allows the coroutine to execute some logic, such as publishing a result, signalling completion or resuming a continuation. It also allows the coroutine to optionally suspend immediately before execution of the coroutine runs to completion and the coroutine frame is destroyed.

Note that it is undefined behaviour to `resume()` a coroutine that is suspended at the `final_suspend` point. The only thing you can do with a coroutine suspended here is `destroy()` it.

The rationale for this limitation, according to Gor Nishanov, is that this provides several optimisation opportunities for the compiler due to the reduction in the number of suspend states that need to be represented by the coroutine and a potential reduction in the number of branches required.

Note that while it is allowed to have a coroutine not suspend at the `final_suspend` point, **it is recommended that you structure your coroutines so that they do suspend at `final_suspend`** where possible. This is because this forces you to call `.destroy()` on the coroutine from outside of the coroutine (typically from some RAII object destructor) and this makes it much easier for the compiler to determine when the scope of the lifetime of the coroutine-frame is nested inside the caller. This in turn makes it much more likely that the compiler can elide the memory allocation of the coroutine frame.

How the compiler chooses the promise type

So lets look now at how the compiler determines what type of promise object to use for a given coroutine.

The type of the promise object is determined from the signature of the coroutine by using the `std::experimental::coroutine_traits` class.

If you have a coroutine function with signature:

```
task<float> foo(std::string x, bool flag);
```

Then the compiler will deduce the type of the coroutine's promise by passing the return-type and parameter types as template arguments to `coroutine_traits`.

```
typename coroutine_traits<task<float>, std::string, bool>::promise_type;
```

If the function is a non-static member function then the class type is passed as the second template parameter to `coroutine_traits`. Note that if your method is overloaded for rvalue-references then the second template parameter will be an rvalue reference.

For example, if you have the following methods:

```
task<void> my_class::method1(int x) const;
task<foo> my_class::method2() &&;
```

Then the compiler will use the following promise types:

```
// method1 promise type
typename coroutine_traits<task<void>, const my_class&, int>::promise_type;

// method2 promise type
typename coroutine_traits<task<foo>, my_class&&>::promise_type;
```

The default definition of `coroutine_traits` template defines the `promise_type` by looking for a nested `promise_type` typedef defined on the return-type. ie. Something like this (but with some extra SFINAE magic so that `promise_type` is not defined if `RET::promise_type` is not defined).

```
namespace std::experimental
{
  template<typename RET, typename... ARGS>
  struct coroutine_traits<RET, ARGS...>
  {
    using promise_type = typename RET::promise_type;
  };
}
```

So for coroutine return-types that you have control over, you can just define a nested `promise_type` in your class to have the compiler use that type as the type of the promise object for coroutines that return your class.

For example:

```
template<typename T>
struct task
{
  using promise_type = task_promise<T>;
```

```
  ...
};
```

However, for coroutine return-types that you don't have control over you can specialise the `coroutine_traits` to define the promise type to use without needing to modify the type.

For example, to define the promise-type to use for a coroutine that returns `std::optional<T>`:

```
namespace std::experimental
{
  template<typename T, typename... ARGS>
  struct coroutine_traits<std::optional<T>, ARGS...>
  {
    using promise_type = optional_promise<T>;
  };
}
```

Identifying a specific coroutine activation frame

When you call a coroutine function, a coroutine frame is created. In order to resume the associated coroutine or destroy the coroutine frame you need some way to identify or refer to that particular coroutine frame.

The mechanism the Coroutines TS provides for this is the `coroutine_handle` type.

The (abbreviated) interface of this type is as follows:

```
namespace std::experimental
{
  template<typename Promise = void>
  struct coroutine_handle;

  // Type-erased coroutine handle. Can refer to any kind of coroutine.
  // Doesn't allow access to the promise object.
  template<>
  struct coroutine_handle<void>
  {
    // Constructs to the null handle.
    constexpr coroutine_handle();

    // Convert to/from a void* for passing into C-style interop functions.
    constexpr void* address() const noexcept;
    static constexpr coroutine_handle from_address(void* addr);

    // Query if the handle is non-null.
```

```
    constexpr explicit operator bool() const noexcept;

    // Query if the coroutine is suspended at the final_suspend point.
    // Undefined behaviour if coroutine is not currently suspended.
    bool done() const;

    // Resume/Destroy the suspended coroutine
    void resume();
    void destroy();
  };

  // Coroutine handle for coroutines with a known promise type.
  // Template argument must exactly match coroutine's promise type.
  template<typename Promise>
  struct coroutine_handle : coroutine_handle<>
  {
    using coroutine_handle<>::coroutine_handle;

    static constexpr coroutine_handle from_address(void* addr);

    // Access to the coroutine's promise object.
    Promise& promise() const;

    // You can reconstruct the coroutine handle from the promise object.
    static coroutine_handle from_promise(Promise& promise);
  };
}
```

You can obtain a `coroutine_handle` for a coroutine in two ways:

1. It is passed to the `await_suspend()` method during a `co_await` expression.
2. If you have a reference to the coroutine's promise object, you can reconstruct its `coroutine_handle` using `coroutine_handle<Promise>::from_promise()`.

The `coroutine_handle` of the awaiting coroutine will be passed into the `await_suspend()` method of the awaiter after the coroutine has suspended at the `<suspend-point>` of a `co_await` expression. You can think of this `coroutine_handle` as representing the continuation of the coroutine in a continuation-passing style call.

Note that the `coroutine_handle` is **NOT** and RAII object. You must manually call `.destroy()` to destroy the coroutine frame and free its resources. Think of it as the equivalent of a `void*` used to manage memory. This is for performance reasons: making it an RAII object would add additional overhead to coroutine, such as the need for reference counting.

You should generally try to use higher-level types that provide the RAII semantics for coroutines, such as those provided by cppcoro (shameless plug), or write your own higher-

level types that encapsulate the lifetime of the coroutine frame for your coroutine type.

Customising the behaviour of `co_await`

The promise type can optionally customise the behaviour of every `co_await` expression that appears in the body of the coroutine.

By simply defining a method named `await_transform()` on the promise type, the compiler will then transform every `co_await <expr>` appearing in the body of the coroutine into `co_await promise.await_transform(<expr>)`.

This has a number of important and powerful uses:

It lets you enable awaiting types that would not normally be awaitable.

For example, a promise type for coroutines with a `std::optional<T>` return-type might provide an `await_transform()` overload that takes a `std::optional<U>` and that returns an awaitable type that either returns a value of type `U` or suspends the coroutine if the awaited value contains `nullopt`.

```
template<typename T>
class optional_promise
{
  ...

  template<typename U>
  auto await_transform(std::optional<U>& value)
  {
    class awaiter
    {
      std::optional<U>& value;
    public:
      explicit awaiter(std::optional<U>& x) noexcept : value(x) {}
      bool await_ready() noexcept { return value.has_value(); }
      void await_suspend(std::experimental::coroutine_handle<>) noexcept {}
      U& await_resume() noexcept { return *value; }
    };
    return awaiter{ value };
  }
};
```

It lets you disallow awaiting on certain types by declaring `await_transform` overloads as deleted.

For example, a promise type for `std::generator<T>` return-type might declare a deleted `await_transform()` template member function that accepts any type. This basically disables use of `co_await` within the coroutine.

```
template<typename T>
class generator_promise
{
  ...

  // Disable any use of co_await within this type of coroutine.
  template<typename U>
  std::experimental::suspend_never await_transform(U&&) = delete;

};
```

It lets you adapt and change the behaviour of normally awaitable values

For example, you could define a type of coroutine that ensured that the coroutine always resumed from every `co_await` expression on an associated executor by wrapping the awaitable in a `resume_on()` operator (see `cppcoro::resume_on()`).

```
template<typename T, typename Executor>
class executor_task_promise
{
  Executor executor;

public:

  template<typename Awaitable>
  auto await_transform(Awaitable&& awaitable)
  {
    using cppcoro::resume_on;
    return resume_on(this->executor, std::forward<Awaitable>(awaitable));
  }
};
```

As a final word on `await_transform()`, it's important to note that if the promise type defines *any* `await_transform()` members then this triggers the compiler to transform *all* `co_await` expressions to call `promise.await_transform()`. This means that if you want to customise the behaviour of `co_await` for just some types that you also need to provide a fallback overload of `await_transform()` that just forwards through the argument.

Customising the behaviour of `co_yield`

The final thing you can customise through the promise type is the behaviour of the `co_yield` keyword.

If the `co_yield` keyword appears in a coroutine then the compiler translates the expression `co_yield <expr>` into the expression `co_await promise.yield_value(<expr>)`. The

promise type can therefore customise the behaviour of the `co_yield` keyword by defining one or more `yield_value()` methods on the promise object.

Note that, unlike `await_transform`, there is no default behaviour of `co_yield` if the promise type does not define the `yield_value()` method. So while a promise type needs to explicitly opt-out of allowing `co_await` by declaring a deleted `await_transform()`, a promise type needs to opt-in to supporting `co_yield`.

The typical example of a promise type with a `yield_value()` method is that of a `generator<T>` type:

```
template<typename T>
class generator_promise
{
  T* valuePtr;
public:
  ...

  std::experimental::suspend_always yield_value(T& value) noexcept
  {
    // Stash the address of the yielded value and then return an awaitable
    // that will cause the coroutine to suspend at the co_yield expression.
    // Execution will then return from the call to coroutine_handle<>::resume(
    // inside either generator<T>::begin() or generator<T>::iterator::operator
    valuePtr = std::addressof(value);
    return {};
  }
};
```

Summary

In this post I've covered the individual transformations that the compiler applies to a function when compiling it as a coroutine.

Hopefully this post will help you to understand how you can customise the behaviour of different types of coroutines through defining different your own promise type. There are a lot of moving parts in the coroutine mechanics and so there are lots of different ways that you can customise their behaviour.

However, there is still one more important transformation that the compiler performs which I have not yet covered - the transformation of the coroutine body into a state-machine. However, this post is already too long so I will defer explaining this to the next post. Stay tuned!

Comments

Comments are welcome in this GitHub issue

Asymmetric Transfer

Lewis Baker

lewissbaker

lewissbaker

Some thoughts on programming, C++ and other things.