Is there a way that my macro can detect that it’s running in a C++ coroutine?

devblogs.microsoft.com/oldnewthing/20211011-00

October 11, 2021

Raymond Chen

Say you are writing a macro that wants to behave differently depending on whether it is expanded inside a coroutine or not. Specifically, you want to expand to `return` in a regular function, but `co_return` in a coroutine.

```
template<T>
T&& TraceValue(T&& v);

// NOTE: Just a sketch. A real macro would have to do more work,
// but we are focusing on the IF_IN_COROUTINE part.
#define TRACE_RETURN() \
    TraceExit(); IF_IN_COROUTINE(co_return, return)

#define TRACE_RETURN_VALUE(v) \
    IF_IN_COROUTINE(co_return TraceExitValue(v), return TraceExitValue(v))

bool TestSomething()
{
    TRACE_ENTER();
    TRACE_RETURN_VALUE(IsSomethingReady()); // want "return"
}

task<bool> TestSomethingAsync()
{
    TRACE_ENTER();
    TRACE_RETURN_VALUE(IsSomethingReady()); // want "co_return"
}
```

Is it possible to write the magic `IF_IN_COROUTINE` macro which expands either its first or second parameter?

It’s not possible in general, because the decision as to whether a function body is a regular function body or a coroutine function body depends on what is inside the body. Specifically, if the body it contains any `co_await` or co_return statements, then it is a coroutine body. Otherwise, it is a regular function body.

Since the macro is expanded as part of the function body, the decision about whether it is a coroutine or not hasn't yet been made. In fact, the macro's expansion might be the thing that determines whether the function body is a coroutine!

In the second example above, the function body expands to something like this:

```
task<bool> TestSomethingAsync()
{
    TraceEnter(__func__, __FILE_, __LINE__);
#if in coroutine
    co_return TraceExitValue(IsSomethingReady());
#else
    return TraceExitValue(IsSomethingReady());
#endif
}
```

Whether this is a coroutine depends on what the macro chooses!

If the macro detects that this is a coroutine, then the body expands to `co_return TraceExitValue(...)`, and it is that `co_return` that makes the function body a coroutine. But if the macro detects that it's not a coroutine, then the body says `return TraceExitValue(...)`, and since there is no `co_return` or `co_await` statement, the function body is a regular function body.

You thought your macro was passively detecting whether it was in a coroutine, but in fact it is actively controlling the decision!

Now, you might think, "Well, can I just base my decision on the function return type?"

Even if you could detect the return type from a macro (I'm not sure you can), that still wouldn't be good enough. The `task<bool>` might support construction from a `bool`, say to represent an already-completed task, and therefore both `co_return boolValue` and `return boolValue` are legal in the function body.

Basically, you are trying to be a passive predictor of a future that you inadvertently influence. That doesn't work well in science fiction, and it doesn't work well here either.

Bonus paradox: Imagine writing the opposite macro:

```
#define TRACE_RETURN_VALUE(v) \
    IF_IN_COROUTINE(return TraceExitValue(v), co_return TraceExitValue(v))
```

This macro tries to be contrary and says, "Use `return` if I'm in a coroutine, but `co_return` if I'm not."

We could call this Russell's macro since it creates a similar paradox:

```
task<bool> TestSomethingAsync()
{
    TraceEnter(__func__, __FILE_, __LINE__);
#if in coroutine
    return TraceExitValue(IsSomethingReady());
#else
    co_return TraceExitValue(IsSomethingReady());
#endif
}
```

If the coroutine detector says, “This is a coroutine”, then the macro expands to `return`, which makes the function body *not* a coroutine. But if the coroutine detector says, “This is not a coroutine”, then the macro expands to `co_return`, which makes the function body a coroutine after all!

Proof by logical contradiction that a perfect coroutine-detector macro is impossible to write.

Raymond Chen

Follow