What does it mean when I get a mismatch from MSVC for _COROUTINE_ABI?

devblogs.microsoft.com/oldnewthing/20230111-00

January 11, 2023

Raymond Chen

A customer was compiling some code that uses coroutines. When they linked in the third party Contoso library, they got a linker error:

```
contoso.lib(cpp.obj) : error LNK2038: mismatch detected for '_COROUTINE_ABI': value
'1' doesn't match value '2' in main.obj
```

First of all, what does this error message literally mean?

It's saying that the `detect_mismatch` pragma was used twice with the same name but different values. In this case, it's saying that `cpp.obj` inside `contoso.lib` did a

```
#pragma detect_mismatch("_COROUTINE_ABI", "1")
```

but `main.obj` (presumably an object file that is part of the customer's project) did a

```
#pragma detect_mismatch("_COROUTINE_ABI", "2")
```

The two values of `_COROUTINE_ABI` do not match, so the linker reported a mismatch error.

Okay, so now we understand what triggers an error message of this form. Now to figure out what triggers this specific mismatch.

From the name "`_COROUTINE_ABI`", it's apparent that this symbols is tracking which coroutine ABI the compiler is using. Even if you hadn't inferred that from the name, a search of the language header files will turn up a pair of hits.

```
// in <experimental/coroutine>
#ifndef _ALLOW_COROUTINE_ABI_MISMATCH
#pragma detect_mismatch("_COROUTINE_ABI", "1")
#endif // _ALLOW_COROUTINE_ABI_MISMATCH

// in <coroutine>
#ifndef _ALLOW_COROUTINE_ABI_MISMATCH
#pragma detect_mismatch("_COROUTINE_ABI", "2")
#endif // _ALLOW_COROUTINE_ABI_MISMATCH
```

Recall that MSVC supports two coroutine ABIs. The legacy pre-standardization coroutines in `<experimental/coroutine>` use a single `resume` method and encode whether the method is being resumed or destroyed by the even/odd-ness of the state index. The standardized coroutines in `<coroutine>` use a pair of `resume` and `destroy` methods, which is ABI-compatible with clang and gcc coroutines, allowing interop among all three implementations.

The two kinds of coroutines (pre-standardization and common-ABI) are not interoperable. If you pass a coroutine of one kind to a function that expects a coroutine of the other kind, it may not operate correctly.

That is the problem that the `detect_mismatch` is trying to prevent. Your module is mixing code that is using experimental coroutines with code that uses standard coroutines, and if the streams ever cross, bad things will happen.

You can study the details of the error message to find the two parts that are using mutually incompatible coroutine ABIs.

Okay, so now that we've identified the problem, how do we solve it?

Well, it depends on how you use the Contoso library.

If the Contoso library's use of coroutines is entirely internal to the implementation of the Contoso library, then there is no cross-contamination, and the mismatch will not occur in practice.

Note that if coroutines are used by internal inline functions in public Contoso header files, then that still counts as externally visible, because the code for the inline function will be generated by the *client* of the library, not locked away inside `contoso.lib`.

If you can successfully avoid crossing the streams, then you can define the symbol `_ALLOW_COROUTINE_ABI_MISMATCH` to disable the mismatch detection.

Of course, if you disable the detection, and the streams do cross at some point, then bad things will happen, and it's all your fault.

If you are unable to avoid crossing the streams, then you have to get the Contoso library and your program to agree on which coroutine ABI it is using. If there is a C++20 version of the Contoso library (or if you can make one by recompiling it), then you can use that version.

Otherwise, you'll have to downgrade your program to use pre-standardization coroutines. One way is to downgrade the entire program to C++17 with `/await`. But you may not want to throw away that much baby with the bathwater, since you'll lose all the C++20 features.

You can also use the `/await` switch in combination with `/std:c++20` or `/std:c++latest` to say "I want pre-standardization coroutines" with as much of the newer language features as possible, provided they don't create compatibility issues.

This hassle with the coroutine ABI is why compiler toolchain vendors are so concerned about ABI breaking changes. Any time you make an ABI breaking change, you have to deal with pulling everybody over the line at the same time. A compiler upgrade is no longer something you can opt into for only part of a project.