Doc No.: P2691R0
Date: 2022-10-14
Group: JTC 1 / SC 22 / WG 21

Allow referencing inline functions with internal linkage from outside their defining header unit

Herb Sutter, Gabriel Dos Reis, Stephan T. Lavavej, Michael Spencer

Background and overview

[P2003] made several suggestions regarding modules, most of which were adopted in Prague (wiki notes are here).

However, this suggestion was not adopted:

> *The second is that it is ill-formed to use internal linkage entities outside of that header unit. This is problematic as it is quite common for headers that want to be compatible with C to use* `static inline` *for inline functions, as C's version of inline works differently.*
>
> *[…]* `static inline` *functions are extremely common today, and a large portion of C headers would not be usable as header units if they were not allowed to be referenced. As header units exists purely to support existing code and code that will likely never move to modules (like C code), it would significantly harm modules adoption to not allow this. The consequences are the same as using* `static inline` *from a textual header today. We're not introducing any new UB, just not stopping people from hitting it.*

The standard currently says that a header that contains a `static inline` function can be turned into a header unit, but referencing the `static inline` function is an error in some cases (allowed in others, deprecated in still others).

[P2003] reported that this was an issue for Apple headers.

Subsequent experience has shown the extent of the problem is wider than anticipated in Prague. We can report that user feedback from field experience with modules has shown that this has become a significant modules adoption blocker, as [P2003] anticipated.

Proposal

(1) Allow referencing inline functions with internal linkage from outside their defining header unit

We should fix this problem for header units, by adopting the direction proposed in [P2003]. This paper re-proposes [P2003]'s proposal that:

- header units and named modules should support the usage of entities declared as `static inline`,
- with the semantics that the translation unit that ODR-uses the entity gets its own "definition" (same as with an `#include` header file).

(2) Allow exporting using-declarations that name entities with internal linkage

Today, named modules explicitly prohibit exporting an entity declared with internal linkage.

However, it would be desirable to enable implementations of [P2465R3] like the following to work even if `time` is declared `static inline`:

```
// in <ctime> when building C++23's module 'std'
#include <time.h>
namespace std {
    export using ::time;
}
```

Certainly, there is evidence that users expect the following program to work:

```
import std;
int main() { time(0); }
```

If we extend the fix to this problem to cover using-declarations, then it will be more broadly applicable, will simplify module `std` implementation techniques, and will also cover the use case where the exported using-declaration of something defined and attaching to the global module. Exporting a using-declaration is a common technique for projecting a modular view over a header. Imagine

```
module;
#include <unistd.h>
#include <sys/stat.h>
export module OS.Filesystem;
namespace FS {
     export using ::stat;    // ::stat might be "static inline"
}
```

If the function `::stat` was defined by the C header `<sys/stat.h>` as a `static inline` function, then the above code won't compile today because the using-declaration is naming something with internal linkage and that is not allowed. Note that the declarations being referred to by the using-declaration are **not** attached to the named module `OS.Filesystem`.

However, this would need wordsmithing… the rule that

> … `E` is not a function or function template and `D` contains an id-expression, type-specifier, nested-name-specifier, template-name, or concept-name denoting `E` …

could exclude using-declarations, but that could still restrict using `std::time` or `std::stat` in inline functions (and other potential exposures) in module interface units or module partitions.

Discussion

Why can't C++ implementers just go fix their headers?

This problem manifests for any header that wants to be consumable from both C and C++, not just standard headers.

Even for just the standard headers, the C++ implementation vendor is not always able to change the C headers. Even when they can, it can be a lot of work and still not end up being a general solution. For example, here is a bug where we could in principle remove `static inline` from a standard C function, but it could not be done quickly (because a different team owns the header) and so in the meantime we had to implement a workaround:

> **Bug 1163516: Visual Studio can't find `time()` function using modules and `std.core`**
>
> Current status: For the `time` family functions specifically, one of us (Lavavej) found a fairly elaborate workaround without changing the UCRT header itself. However, we believe that the workaround won't handle all scenarios because it involves defining another `std::time`, so scenarios of the form `#include <ctime>` followed by `import std;` won't be able to call `std::time` unambiguously. This is the best we can do until the actual header can be changed.

But of course we cannot in general change the affected headers, most of which are not owned by the C++ implementation. This example is just to illustrate the difficulties that this problem creates.

References

[P2003R0] M. Spencer. "Fixing Internal and External Linkage Entities in Header Units" (WG 21 paper, January 2020).

[P2465R3] S. Lavavej et al. "Standard Library Modules `std` and `std.compat`" (WG 21 paper, March 2022).