Document Number: p1299r1
Date: 2018-10-10
To: SC22/WG21 EWG
Reply to: Nathan Sidwell
nathan@acm.org / nathans@fb.com
Re: N4720 Merging Modules

Module Preamble is Unnecessarily Fragile

Nathan Sidwell

The ATOM proposal introduced the module preamble. The merged document modified that concept for increased flexibility. The rules determining the extent of the preamble have been modified, but it remains fragile. We should consider making the preamble more robust.

1 Background

The ATOM proposal introduced the module preamble concept. That is all imports are a contiguous block as the first set of declarations (after an optional module declaration). This applied to both module unit and non-module unit source code.

The preamble meant there was a clear boundary between specifying dependencies and providing implementation (or interface). Such an ordering permits these dependencies to be discovered by a simple scan of the start of a file – full preprocessing is not needed. GCC implements a `-fmodule-preamble` option that stops preprocessing at the first non-preamble token.

The preamble contents are order-independent. The imports can be reordered without changing program semantics.

It also allowed the interesting implementation technique of deferring processing of imports until the end of the preamble. This permits these dependencies to be found and/or built concurrently, rather than serialized. Again, GCC's module mapper protocol (p1184) permits this use.

1.1 Preamble End (ATOM)

One difficulty was defining exactly when the end of the preamble occurred. While many cases are simple, a particularly vexing case requires backtracking in the preprocessor:

```
    import "legacy.h"; // #1
    #ifndef LEGACY_FLAG
    int not_preamble; // #2
    #endif
```

In this case we reach the `int` keyword at #2 and so the preamble must be over, but we are inside a conditional expansion that could have been affected by an imported macro. The preamble is deemed to have ended immediately after the semicolon at #1. The rule is framed such that the contents of any legacy import is not needed to determine this. If `LEGACY_FLAG` was defined in `"legacy.h"` the conditional inclusion could change. The required backtracking is very hard to implement in some implementations. In discussing this case the following implementation was deemed acceptable:

- Issue a warning [non-error] diagnostic, suggesting a raw semicolon be added at #1 to terminate the preamble before the conditional directive.
- Then restart compiling from the beginning

Performance-wise this is terrible, but expectations were that it would be rare and users would be comfortable adding a marking semicolon.[1] GCC implements the above algorithm by execing itself with an internal *stop-here* option. Such repetition cannot be done if compiling from a non-repeatable input such as `stdin`.

A similar case is:

```
import "legacy.h"; // #1
#ifdef LEGACY_FLAG
int skipped; // #2
#endif
int not_preamble; // #3
```

In this case preamble end is detected at #3 (assuming `LEGACY_FLAG` is only provided by `"legacy.h"`), which is not inside a conditional directive. This is well-formed, possibly mistakenly. The user may well be surprised that the declaration at #2 is skipped even though `"legacy.h"` defined `LEGACY_FLAG`.

These cases turned out to be more common and more confusing than originally thought. The issue was revisited at the Bellevue'18 meeting. The decision was made that the preamble ended at either the first non-import declaration, or at the first expansion of an imported macro. This is a much simpler rule.

1.2 Merged Proposal

The merged proposal dropped the requirement for a preamble in user code, but kept it for module units. Except that global module fragments and legacy header units can contain import declarations at any (global-scope) location. An important case is that of implicitly generated import declarations from translated include directives. It is necessary that legacy header units consistently translate include directives for other legacy headers. Without doing so, the macro definition overriding semantics break down.[2] In particular, the following is well-formed:

1 Other syntaxes were considered, but the semicolon was thought the least worst.
2 It is noted that the case of include translation was already present in the ATOM proposal.

```
// legacy.h
#include "legacy-a.h" // #1 becomes import "legacy-a.h";
#ifdef LEGACY_A_MACRO // #2
#include "legacy-b.h" // #3 becomes import "legacy-b.h";
#endif
```

The translated include at #1 provides `LEGACY_A_MACRO`, which controls the include directive at #3. That include is translated to a second legacy header unit.

Another decision at the Bellevue'18 meeting there could not be post-preamble explicit legacy imports. This makes producing a preprocessed output simpler – only preamble legacy imports need processing.[3] However it is still required that post-preamble include directives be translated to import declarations when compiling.

2 Discussion

The Bellevue'18 simplification of determining the preamble end has two issues:

1. It is still easy to terminate a preamble unexpectedly.
2. Legacy header imports must be processed immediately, in order to update the macro table.

Item 1 will quite probably lead to user frustration. Item 2 causes some loss of the implementation advantages of the preamble, but as discussed below in Section 2.2, does not require immediate loading of all imports.

2.1 Preamble End (Post Bellevue'18)

Having imported macro expansion end the preamble will still be confusing to users. The expansion need not be close to the legacy unit import. There is no indicator in the program source that an imported macro is being tested or expanded, as opposed to a macro defined textually.

The following is ill-formed:[4]

```
module foo;
import "config.h";
#ifdef OS_LINUX
import "linux.h";
#endif
```

Whereas the equivalent non-module unit case would be well formed (because there is no restriction on additional post-preamble imports).

3 Although such a mode of source processing is not part of the standard, it is an expected ability of compilation tools.
4 As described in Section 1.1, prior to the Bellevue'18 decision, this is well formed, but would not import `"linux.h"`.

There is an ambiguity with:

```
module foo;
#define OS_LINUX 1 // or by other textual mechanism
import "config.h";
#ifdef OS_LINUX
import "linux.h";
#endif
```

Is the `OS_LINUX` macro an import, or is the import ignored as a well-formed redefinition? What if the define directive and `"config.h"` are swapped?

2.2 Deferred Loading

The immediate availability of imported macros means legacy import loading cannot not be deferred.[5] But named-module import loading may deferred until a name lookup is required. Deferring to name lookup may be difficult to implement, but they could still be deferred until the next non-import declaration. Such deferring could be implemented in several ways. The pending imports could be loaded as soon as the next declaration is determined to not be an import by:

1. Parser lookahead
2. Tokenizer lookahead

The tokenizer must still cooperate with the lexer to correctly tokenize legacy import `"quoted"` or `<angled>` names. This would allow the concurrent building or location of named-module units.

Such deferred loading could be performed for any contiguous block of import declarations. Legacy imports within that block would not break the block, although they would need importing immediately.

Thus deferred loading is neither more nor less complicated with or without a syntactic preamble.

2.3 Non-Module Sources

Unlike module source units, non-module source has no mechanism to indicate it is aware of modules. There are four cases:

1. It is unaware of modules and does not transitively use them, or
2. It is unaware of modules, but include headers that do use them (or contains headers that are translated to imports), or
3. It uses modules, but also includes headers that themselves import modules, or
4. It only uses modules.

5 Loading refers to any initial loading of the import and its macro table, not completing any lazy loading of exported entity declarations it may contain.

Cases 2 & 3 are essentially the same. Case 4 is the same as module source without a global module fragment. Unfortunately it has no mechanism to indicate so, and that it might benefit from treating its initial block of imports as a module preamble.

This paper proposes no changes in the currently permitted syntax for these cases, but includes the description for comparison.

2.4 Extended Preamble

One alternative to could be permitting imported macro expansion without terminating the preamble. The preamble would end at the first non-import declaration. Legacy header units would be immediately imported. Expansion of imported macros would not terminate the preamble. The ambiguous cases described in Section 2.1 become moot.

This further simplification of the preamble is not more complicated than that proposed at Bellevue’18, and is likely to be less confusing. It does mean that reordering legacy imports within the preamble could affect semantics if that introduced imported macros earlier:

- Conditional preprocessing directives for later imports would be affected.
- Named module names could be affected.

The first is obvious in the source, the latter less so. But no more unexpected than a name used in any other declaration. With the Bellevue’18 decision, such reordering would render the program ill-formed.

Include file reordering can be even more disruptive as an earlier include can affect the semantics of a later include. At least legacy header unit contents are immune from that.

2.5 Loss of Preamble

A different alternative could be removing the concept of a preamble completely. This would allow module unit sources contain imports at any global-namespace location. Clearly removing the concept of a preamble makes any rule for detecting its end superfluous.

Good software practice would encourage a block of imports at the start of a source file, in all sources. That this not be a requirement will give additional flexibility in migrating source bases. However, such flexibility might be unwarranted. Module unit source cannot include a header anywhere other than its global module fragment. Thus there cannot be include headers that might contain embedded import declarations within the module purview. Any non-preamble imports must be within the source.[6]

As with the extended preamble of Section 2.4, it will mean that the order of legacy imports within a block of import declarations could be significant.

6 Or a textually included header, which for the purposes of this section are not headers in the traditional sense, but a mechanism for repeating a token sequence in multiple places.

3 Proposal

Lexing look-ahead is still required for tokenization of legacy header unit names. The rule that the `export` and `import` keywords are not from a macro expansion should be kept.[7] Likewise that the ending semicolon of an import declaration not be from a macro should remain.

Loss of preamble could result in lack of code discipline of module units. Rather, we should encourage the ability of having a preamble in non-module sources. This paper does not suggest such a mechanism, leaving it as an orthogonal issue.

The terminating of the preamble at the first import expansion is confusing and fragile. Such expansions should not terminate the preamble.

4 Revision History

R0 Placeholder.

R1 Title modified. Content added.

7 They cannot come from an imported macro, as those are never permitted to be keywords.