Document Number: P0924r1
Date: 2018-11-21
To: SC22/WG21 EWG
Reply to: Nathan Sidwell
nathan@acm.org / nathans@fb.com
Re: Merging Modules, p1103r2

Modules:Context-Sensitive Keyword

Nathan Sidwell

The new `module` & `import` keywords presents some difficulties with converting existing code bases that use them as identifiers, particularly in externally publicized interfaces. This paper discusses avenues available for making `module` & `import` context-sensitive keywords.

1 Background

At Albuquerque'17 I presented two papers that could simplify making `module` context-sensitive, although that was not their main goal:

- P0774 'Module Declaration Location'
- P0787 'Proclaiming Ownership Declarations'

This paper draws on those, but has the goal of making `module` context-sensitive, rather than a desirable side-effect of another change.

That version of the paper proposed full tentative parsing, in the same manner as Most Vexing Parse, and was presented at the Jacksonville'18 meeting. It was not accepted. Consensus was to proceed with `import` and `module` as keywords.

Since that time, context has changed. The ATOM proposal, p0947r0, was presented at the same meeting and EWG elected to merge components of ATOM with the TS, thereby producing p1103. Of significance to this paper, p1103r0 had the following:

1. Header units, which have header names.
2. Removal of proclaimed-ownership-declarations.
3. Removal of import declarations from nested export declarations.

Also, p0713 was accepted, requiring the global module fragment be introduced by a nameless module declaration.

2 Discussion

The restriction of module and import declarations to top-level constructs, and the removal of proclaimed-ownership-declarations, changes the cost of making module and import context-sensitive.

2.1 Full Tentative Parsing

TL;DR: This is the proposal of the r0 paper, retained for historical record.

This disambiguation rule is that such ambiguities are always parsed as a *module-declaration* – if a (top-level) declaration could be a *module-declaration*, it is. This is simple to state. It would change the meaning of the following C++17 source (presume '`module`' is a typedef):

```
// implementation unit 'me', not variable 'me'
module me;

// interface unit 'me', not export of variable 'me'
export module me;
```

This disambiguation occurs, regardless of whether the interpretation is semantically ill-formed.

Function declarations, more complex variable declarations, member declarations and non-global-namespace-scope declarations would remain with their C++17 meaning:

```
class module { /* Unspecified.  */ };
module frob (); // function returning module
module *me; // variable pointing to a module
namespace bits {
  export module me; // exporting variable of type module
}
class thing {
  module me; // data member of type module
};
```

For completeness, the following uses of module, as an identifier, continue unchanged:

```
class module; // class named 'module'
class thing {
 module m; // field 'm' type 'module'
};
int module (); // function called 'module'
```

It is my understanding that the known uses of '`module`' do not fall into cases that would be interpreted as *module-declarations* under this disambiguation.

The disambiguation can usually be implemented with minimal look ahead, not requiring full tentative parsing. If the token after the first *identifier* of a potential *module-name* is '`.`' or '`;`', the declaration

must be a *module-declaration* or ill-formed. If the next two tokens are '`[[`', it could be either reduction, and one must skip to the matching '`]]`' to look[1] for a ';' indicating a *module-declaration*. Otherwise it must be some other declaration (or even *expression-statement*), or ill-formed.

Here are some examples:

```
// module-declarations:
module m;
module m [[ whatever ]];
module m.n;

// declarations (ill-formed if 'module' not a type):
module *m;
module m (…);
module (m);
module m[];
module m, n;
module m [[ whatever ]], n;
```

All those examples parse the same way if preceded by '`export`'.

As mentioned above, this was not accepted.

2.2 Top-Level Disambiguation

As module and import declarations are only valid at the top level, a simple rule is to always interpret them as identifiers at any other level. That is within inner scopes, or within extended linkage or export declarations. Conveniently, this is equivalent to being outside of any bracing.

However, this is insufficiently compatible with C++17, as known sources use '`module`' and/or '`import`' as global scope types or namespaces.

To further disambiguate, we can rely on the module or import declaration syntax begins as:

`export`$_{opt}$ `module` *not-scope-ref*
`export`$_{opt}$ `import` *not-scope-ref*

where *not-scope-ref*, is any token except `::`. Where a declaration does not begin thusly, it should be interpreted as a declaration other than a import or module[2] declaration. Existing code using `import` or `module` as a global-scope typename or template name may need adjusting to insert a leading `::`.

1 Similar lookahead will work for compiler extensions such as '`__attribute__((...))`'.
2 Certain module constructs employ '`module`' but are not themselves module-declarations. For brevity this paper considers them all syntactically module declarations.

2.3 Tokenization

The names of header units are *header-name*s ([lex.header]), which are recognized by the preprocessor in phase 3. The grammar is:

> header-name:
> < h-char-sequence >
> " q-char-sequence "

where the h-char & q-char sequences are composed of any member of the source character set except the final character of their respective header names (> and " respectively) or an end of line. It is implementation defined as to any interpretation of escape sequences, some systems use '\' as a directory separator.

A similar disambiguation rule to that given above could be used within the preprocessor to determine whether the token following '`import`' should be lexed as a header name, if it begins with an appropriate character. However, to implement such disambiguation requires examining tokens after macro expansion – it is only then that brace nesting can be correctly counted.

With *header-name*s the ambiguities are cases such as:

```
template <typename T> class import {};
::import <int> X; // not a header-name
import <int>; // a header-name (with a peculiar name)
```

Unfortunately macro expansion is performed at phase 4, so such an algorithm violates the separation of phases. I had unfortunately misunderstood some of the p1103 semantics, and that this had already been violated, because macro importation from header units appeared to be a path from phase 7 to phase 4. However, that is not the case. P1103 specified that header units were recognized by the preprocessor at phase 4, and the macros therefrom loaded at that point. The header units were again recognized at phase 7 to load their C++ declarations etc.[3]

Such a back propagation proposed would not cause implementation difficulties of the 4 compiler developers queried. However the comment was made that a preprocessor might choose to implement phases 1 to 3 as a first pass and then process phase 4. Such an implementation would clearly have difficulty if phase 4 affected the tokenization of phase 3.

2.3.1 Import-Is-Keyword Locations

Determining when an `import` identifier is at a location where it may be a keyword was expressed with two conditions:

1. outside of any brace nesting, and

3 Implementations may well behave in the manner I had inferred though.

2. immediately after a close-brace or semicolon, with a possibly intervening `export` keyword.

Before macro expansion we can neither count braces, nor unambiguously determine if we are at the start of a declaration. Obviously, if the programmer hides `export` or `import` inside macro expansions, the phase-3 tokenizer will fail completely.

Brace nesting will fail for idioms such as:

```
#define NAMESPACE_BEGIN namespace my_company {
#end NAMESPACE_END }
NAMESPACE_BEGIN /*…*/ NAMESPACE_END
import <frob>;
```

In general, any identifier could be a macro expansion that ends a declaration. Similarly any '`)`' could be the end of a macro invocation with the same effect. We can be certain we are *not* at the start of a declaration after a non-identifier cpp-token that is not a close parenthesis, semicolon or close brace.

We may be reasonably certain that if we have tokenized an unclosed brace outside of a preprocessing directive, that we are not at the top-level. This heuristic will fail for:

```
#ifndef NO_NAMESPACES
namespace foo {
#endif
/* #1 stuff */
#ifndef NO_NAMESPACES
} using namespace foo;
#endif
```

However, considering the region #1 as not at the outer level regardless of `NO_NAMESPACES` is probably non-breaking.

The pretend-it's-pascal idiom of:

```
#define begin {
#define end }

int foo () begin
/* stuff */
end
```

will also break brace-counting, and lead to ambiguities of the form:

```
import < lower || import >= upper ? v = 1 : v = 2;
```

which is a baroque way to express conditional assignment.

If we add the following restrictions, phase-3 tokenization becomes simpler:

1. ‘`export`’ & ‘`import`’ in header-unit declarations must not come from macro expansion.
2. Braces originating in macro expansions *might* not be counted in brace nesting (braces in preprocessor directives are not counted).
3. Braces not arising from macro expansion must be balanced.
4. The preceding token before a header-unit import declaration must be a non-macro expanded close brace or semicolon.

Rule 1 resurrects P0947’s preamble rule, but only for header-unit imports. Heuristic 2 permits implementations to choose to count braces at phase 4, but does not compel them. Rule 3 inhibits arbitrary mix & match of macro-expanded and direct braces. Rule 4 is a modification of P0947’s semicolon rule, applying to the token before the declaration of interest. Generally code formatting tools already have difficulty if Rules 3 & 4 are not followed. Likewise they are often blind to bracing hidden by macro expansions. Thus these rules are likely followed by existing code bases.

If these conditions are satisfied, the preprocessor should tokenize the following characters as a *header-name* at phase 3. If they do not begin with a valid *header-name* initial character, they should be tokenized as normal. If the *header-name* cannot be lexed (an end of line occurs) an error should be issued.

2.3.2 Macro Expansions

Macros that expand to (parts of) import declarations will be problematic, in the same way as macro expanding an include directive or `__has_include` operand:

```
#define NAME <file>
import NAME;   // #1

#define IMPORT(ARG) import ARG
IMPORT(<file>);     // #2

#define INIT import <file>
INIT;          // #3
```

In all these cases, the `<file>` fragment is tokenized as `<`, `file`, `>`. The identifier ‘`file`’ is subject to macro expansion itself. Were it to be spelled ‘`"file"`’, it would be tokenized as a *string-literal*.

Prohibiting macro expansion for an *import-declaration*’s operand may well cause user confusion, particularly as `__has_include` has no such restriction. Allowing such macro expansions, implies the grammar for an import-declaration needs to be extended to permit a *string-literal* and `<`, *pp-tokens*, `>` sequence. Unlike the `__has_include` case, there is ambiguity in where such a *pp-tokens* sequence might end.[4] Consider:

4 `__has_include` is treated as-if a macro, therefore the closing argument parenthesis marks the end.

```
#define NAME <file   // #1 missing >
import NAME;
… >            // #2 sometime later
```

The expansion of NAME does not include a terminating '`>`', should pp-tokens be concatenated up to '`>`', should it stop at a '`;`' (or '`[`'), or perhaps it should signal an error if a newline is encountered?

For avoidance of doubt I suggest that any retokenization of `<`, *pp-tokens*, `>` occur at phase 4 and not in the C++ parsing phase.

2.3.3 Include Translation

Include translation occurs at phase 4. Such translation is explicitly generating a header unit import, and relies on the location of the translation being at the start of an outermost declaration. There is no difficulty generating the *header-name* token.

3 Proposal

As accepted at San Diego'18, both `module` and `import` are context-sensitive keywords. They are part of a module or import declaration iff:

- At the toplevel, outside of any `{…}` nesting, and
- at the beginning of a declaration, possibly preceded by '`export`', and
- not immediately followed by `::`.

Updated p1103 expresses the first two directly in the grammar, and expresses the third in text.

The preprocessor tokenizing rules should specify the rules in Section 2.3.1, but allow implementations to feedback from phase 4, if they so choose. The retokenizing of Section 2.3.2 should be added.

4 Revision History

R0 Original paper, presented Jacksonville'18

R1 Updated paper, from presentation made in San Diego'18