Document Number:	P0924r0
Date:	2018-02-09
To:	SC22/WG21 EWG
Reply to:	Nathan Sidwell

nathan@acm.org / nathans@fb.com

Re: Working Draft, Extensions to C ++ for Modules, n4720

Modules:Context-Sensitive Keyword

Nathan Sidwell

The new `module` keyword presents some difficulties with converting existing code bases that use `module` as an identifier, particularly in externally publicized interfaces. This paper discusses avenues available for making `module` a context-sensitive keyword.

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.

The module keyword is used in two new declarations:

module-declaration: `export`$_{opt}$ `module` *module-name attributes*$_{opt}$`;`

proclaimed-ownership-declaration: `extern` `module` *module-name* `:` *declaration*

A *module-declaration* may appear, at most, once in a translation unit, and *proclaimed-ownership-declaration*s are discouraged. Thus a reserved keyword is rather extravagant.

1.1 Module-declaration

The two forms of a module-declaration:

`export` `module` *module-name attributes*$_{opt}$ `;`
`module` *module-name attributes*$_{opt}$ `;`

present ambiguities when `module` is a *typedef-name* or *class-name*, and the *module-name* is a plain *identifier*. As well as being valid *module-declarations*, they may be parsed as declarations of a variable of type `module`, the first also being exported. (Of course, outside of module interface purview such an export is semantically invalid.)

1.1.1 Explicit Disambiguation Rule

The simplest 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`'.

1.1.2 Module-Declaration as First Declaration

P0774 proposed requiring the *module-declaration* to be the first declaration of a translation unit, and adding syntax to place entities in the global module. It suggested the following grammar:

translation-unit:
  module-preamble$_{opt}$
  declaration-seq$_{opt}$

module-preamble:
  module-declaration global-module-declaration$_{opt}$

module-declaration:
  `export`$_{opt}$ `module` *module-name attribute-specifier-seq*$_{opt}$ `;`

global-module-declaration:
  `module` `{` *declaration-seq*$_{opt}$ `}`

Requiring the *module-declaration* to be first clearly makes it possible for `module` to be context-sensitive. Also, there would be no requirement for the *global-module-declaration* to be introduced by `module`, but could use the more mnemonic identifier '`global`':[2]

global-module-declaration:
  `global` `{` *declaration-seq*$_{opt}$ `}`

1 Similar lookahead will work for compiler extensions such as '`__attribute__((...))`'.
2 Daveed Vandevoorde suggestion during the Albuquerque '17 presentation.

With this change, there can be no *typedef-name* in scope called ‘`module`’, and thus disambiguation as *module-declaration* is extremely straight-forwards. If the first token is ‘`module`’, or the first two tokens are ‘`export module`’, the TU starts with a *module-declaration*. Otherwise it does not.

1.2 Proclaimed-ownership-declaration

The proclaimed-ownership-declaration grammar of:

> *proclaimed-ownership-declaration* : `extern module` *module-name* `:` *declaration*

is not ambiguous with an `extern` declaration of an entity with type ‘`module`’, because no such declaration can end with ‘`:`’ – ‘`typedef int module; extern module x:` ...’ would be syntactically ill-formed.

However, it may be wise to clarify this in a similar manner to the *module-declaration* above.

1.2.1 Alternative Syntax

Amongst the changes P0787 proposed was changing the syntax of a *proclaimed-ownership-declaration* to avoid the `module` keyword entirely. Its uses there came from earlier syntax for module exporting, and was not reconsidered when the current ‘`export`$_{opt}$ `import` *module-name* `;`’ syntax was developed. To recap, p0787 suggested:

> `import` *module-name* `: extern` *declaration*

as syntax. The emphasis being that we’re declaring an entity exported by the named module. An alternative approach might be:

> `extern export` *module-name* `:` *declaration*

Here the emphasis is that we’re declaring something that is being exported by the named module.

1.2.2 Module Partitions

P0775 ‘Module Partitions’ suggested an alternative approach that would remove the need for *proclaimed-ownership-declaration*s. It was positively received, but I have had insufficient time to advance it at this stage.

2 Proposal

I propose

- Making ‘`module`’ a context-sensitive keyword. I.e., it behaves as a regular identifier, except in specific cases.

- Not changing the syntax of *proclaimed-ownership-declaration*.
- Not changing the location requirements of a *module-declaration*
- Adding a disambiguation rule as specified in Section 1.1.1.

3 Changes to Modules-TS Draft

Modify [lex.name] to add ‘`module`’ to Table 4 as an identifier with special meaning.

Modify [lex.key] to add ‘`import`’ to Table 5 as an unconditional keyword. (i.e. do not add ‘`module`’).

Add the following disambiguation rule to [dcl.module.unit]:

> There is an ambiguity in the grammar between *module-declaration*s and a *declaration*, when ‘`module`’ is a *typedef-name* or *class-name*, and a single identifier is used as the *module-name*. Such ambiguities are resolved as *module-declarations*. *[Note:* Parenthesizing the identifier will cause it to be parsed as a declaration. *– end note]*

Add suitable examples to [dcl.module.unit].

Add the following to [dcl.module.proclaim]

[Note: No ambiguity exists between a *proclaimed-ownership-declaration* and a *declaration*, because no namespace-scope *declaration* may contain a ‘`:`’ after the *declarator*. *– end note]*