Document Number: P0629R0
Date: 2017-03-17
To: SC22/WG21 CWG/EWG
Reply to: Nathan Sidwell
nathan@acm.org

Distinguishing Module Interface From Module Implementation

Gabriel Dos Reis, Jason Merrill, Nathan Sidwell

Alternative mechanisms for source-level distinction of module interface and implementation TUs.

1 Background

The current Modules TS (n4637) specifies that a module-declaration starts the purview of the module interface TU or a module implementation TU:

```
module M;
```

A compiler is unable to determine whether a particular module-declaration denotes the interface or an implementation. Unfortunately, it must know at that point because implementation TUs read data from the corresponding interface in order to make names declared therein immediately available.

Programmers will probably have similar confusion, and at the very least resort to comments specifying intent.

The most extreme example is a TU consisting solely of a module-declaration. Is it an interface that exports nothing, or an implementation that implements nothing?

The Modules TS leaves it as an implicit requirement that the compiler's invocation resolves this ambiguity – by special command line option, distinguished file suffix or other mechanism at the implementors discretion. Such ambiguity could lead to implementation divergence, and prohibit particular implementation strategies (for instance providing module search path capability to an existing build system via the environment). This has impact on build and development systems, forcing them to recognize interface files as C++ (if, perhaps they have a different file suffix), and/or different compiler invocation commands.

The implicit requirement has subtly split the language. It envisions 3 different modes of translation:

- Compiling a module interface, or
- Compiling a module implementation, or
- Neither of the above

Of course the third alternative could be considered a subset of one of the other two. But it remains that the first two need information not provided in the program source. This is an undesirable situation.

2 Direction

At the Kona '17 meeting EWG discussion led to the consensus that:

- Interface and implementation module-declarations should be distinct.
- A preference was for annotating the interface's module-declaration.

This paper explores alternative ways of specializing the interface's module-declaration.

In either case the amended *module-declaration* is referred to as an exporting *module-declaration*. The new section 7.7 [dcl.module] is amended as follows:

2 A *module* is a collection of module units, ~~at most~~exactly one of which contains an exporting *module-declaration* and optionally *export-declaration*s, ~~or~~ *exported-fragment-group*s ~~or~~and *module-export-declaration*s. Such a distinguished module unit is called the *module interface unit*. Any other module unit is called a *module implementation unit*.

2.1 Attribute

One approach is to specify a standard attribute in the module-declaration (whose grammar already permits attributes):

```
module M [[interface]];
```

In the absence of the attribute, command line or other options could (continue) to distinguish interface from implementation.

One drawback is that rather than being an optimization, debugging or other optional hint, this attribute mandates a semantic difference. It also reserves an identifier, `interface`, that is now unavailable for macro use.

2.2 First Class Syntax

An alternative would be to augment the grammar. The simplest change appears to be denoting the interface with:

```
export module M;
```

This has the pleasant feature of using the existing keyword '`export`' to mark the translation unit that is exporting the module's interface. All the occurrences of '`export`' continue to appear only in the interface TU.

The grammar in the amended paragraph 3.5/1 would insert an additional optional keyword:

module-declaration

  export$_{opt}$ module module-name attribute-specifier-seq$_{opt}$;

Toolchains could continue to offer their existing mechanism for distinguishing a module interface TU – in effect offering a C++-interface compilation mode.

3 Conclusion

The attribute approach is aesthetically unpleasing. If accepted, it shall forever appear as a wart. Even if toolchains require or permit additional options for the interface, programmers will need to distinguish interface from implementation in the program source itself.

Our recommendation is the new syntax approach. It is a very localized simple grammar change. Leveraging it so that an implementation no longer needs a special invocation mode may of course be more involved. However, even if implementations remain unchanged in that regard, more informative diagnostics could still be delivered as the programmer has made her intent clearer.

Whichever scheme is chosen, implementations could continue to offer their existing mechanism to disambiguate interface from implementation.