Document number P0273R0

Date 2016-02-12

Authors Richard Smith <richard@metafoo.co.uk>
Chandler Carruth <chandlerc@google.com>
David Jones <dlj@google.com>

Audience Evolution

Proposed modules changes from implementation and deployment experience

Introduction

N4465, presented at the 2015-05 WG21 meeting in Lenexa (hereafter referred to as “the Lenexa proposal”), proposes a modules extension for C++. While that proposal provides a good basis for a modules proposal, we believe it is lacking in a few key aspects which will be critical to satisfying the requirements of our users and the broader C++ community. This paper describes a set of modifications to that proposal, based in part on user experience with the C++ modules implementation in Clang, that we believe will better address the needs of the C++ community and significantly aid in the broad adoption of modules.

All syntax appearing in this work is hypothetical. Any resemblance to real syntax, living or dead, is purely coincidental.

Specific changes

Location of *module-declaration*

The Lenexa proposal suggests that a module interface unit be divided into two sections:

```
// declarations in global module
#include "legacy.h"

module Module.Name;

// declarations in module "Module.Name"
export int my_frob();
int n = MY_LEGACY_VALUE;
```

The `module Module.Name;` line in the middle is the *module-declaration* that establishes to the compiler -- and to the reader -- that the current translation unit is part of a module. Note that, in the Lenexa proposal, this can occur anywhere within the file.

This approach causes practical problems for tools that wish to establish a correspondence between on-disk files and modules (for instance, a build or code analysis tool that wishes to know where `Module.Name` is defined): the entire contents of all files must be scanned to find this introducer. This also results in awkward code organization for modules that expose an interface conforming to a legacy ABI, as such code must declare its legacy entities in the global module.

This proposal suggests a different syntax for introducing modules: the *module-declaration* that indicates a source file is a module unit must be the first declaration within the translation unit, and designates that file as the module interface unit. Consequently, this requires minor changes to how entities within the global module are declared:

```
module Module.Name;

// declarations in module "Module.Name"
export int my_frob();

module {
      // declarations in global module
      #include "legacy.h"
}

// declarations in module "Module.Name"
int n = MY_LEGACY_VALUE;
```

The braced global module scope can only declare entities in the global module. The generalized syntax:

```
module other.module.name { … }
```

is *not* proposed. As noted below, we do not propose any mechanism for declaring entities from modules other than the current module and the global module.

This proposal also explicitly allows declarations within the global module to be exported; it is not clear whether the Lenexa proposal intends to permit that. This is discussed in detail below because exporting from the global module correctly presents specific challenges.

In order to assist programmers and tools in determining whether a source file is a module interface unit or a module implementation unit, we propose that module implementation units use a slightly different form of *module-declaration*, as follows:

`module implementation` *module-name* `;`

This declaration form has two effects:

1. It declares that the current translation unit is part of the implementation of module *module-name* (therefore entities owned by the module can be defined in this source file).
2. It imports the interface unit of the module (exported and the non-exported entities declared within the interface unit become visible).

(Note that the implementation *module-declaration* appears in each implementation file, as opposed to the interface *module-declaration*, which appears only once, in the module interface unit.)

Two-level linkage

To what extent should entities be "owned" by the modules that declare them? There are a spectrum of possible answers here, between the following extremes:

- No ownership semantics exist. Modules only control name visibility at the source level. Two separate modules can provide the same declaration (or, for entities that can have multiple identical definitions under the ODR, the same definition), and declare (or define) the same entity. This is today's C++.
- Full ownership semantics: if two modules provide identical declarations, they declare different entities. This can be implemented in a traditional translation model by mangling the module name into the entity's name, and admits a number of non-traditional implementation strategies (such as a module tag that is understood by the linker, or ordinal-based lookup within a single linker-level symbol representing the entire module).

Both extremes provide advantages, and both have problems.

What's wrong with full ownership semantics?

Imbuing modules with full ownership semantics adds a new flavor of namespacing to C++, one that is separate from the existing namespace concept. Under the Lenexa proposal, there is no interaction between namespaces and modules (for instance, there is no way to import a module so that its contents appear in a namespace other than the global namespace). We believe that is the right choice for C++, but in order for it to function, the exported interface of a module must still follow the namespace discipline.

In a modules system with full ownership semantics, we expect that libraries -- especially those written by inexperienced users of C++ modules -- will frequently abandon namespace discipline because the two-level linkage semantics prevents most problems. However, the language provides no way to resolve the inevitable name conflicts: there is no way to explicitly qualify a name with a module name, or partially import a module's interface, for instance. The net result would be similar to ODR violations today which benignly escape diagnosis (until, all of a sudden, they don't), or violations which are introduced through a long chain of transitive dependencies not reflected by the #includes: it leads users of such libraries to encounter mysterious conflicts they cannot resolve. Introducing a two-level linkage mechanism makes it easier to design non-composable libraries, when we should be making it easier to design composable libraries.

More generally, we think that C++ should have only one large-scale namespacing mechanism (beyond the scope of an individual source file or module), and that namespaces should continue to be that mechanism.

Ownership of exported entities

In order to make it clear that modules should not be used as a substitute for namespaces, we propose that declarations that are exported by a module do not have full ownership semantics. They are still owned by that module (and can only be declared within that module), but it is an error for two modules to own and export equivalent or conflicting declarations with the same name.

It is acceptable, however, for two modules to own and export functions that form a single overload set, and for aggregate modules to explicitly export other imported modules. Consistent with the Lenexa proposal’s aggregates, the re-exported symbols are not re-declared; they remain owned by their original declaring module.

Ownership of non-exported entities

We propose that full ownership semantics are provided for non-exported entities. We see no reason to require declarations within a module to shield themselves from accidental collisions with entities external to the module, as is commonly done today using anonymous namespaces or static variable/function definitions.

The notional model we propose for non-exported entities is very much akin to implicitly wrapping an inline namespace (whose name is unique to the module) around all non-exported entities.

Ownership in the global module

This proposal follows the Lenexa proposal in permitting entities in the global module to be exported from module interface units, albeit with modified syntax, and so it remains possible for a translation unit to import the definition of such an entity from multiple modules. We specifically propose that [basic.def.odr]/6 continues to apply for these cases:

> “[T]he behavior is as if there were a single definition of [the entity].”

(An implementation is free to assume that the multiple definitions are identical, and can therefore choose to discard all but one of the definitions.)

Proposed semantics

As agreed by EWG in Lenexa, we propose that:

- Declarations of non-exported entities under the purview of different modules declare distinct entities.
- If two modules declare the entities with the same name (and signature, for functions or function templates), and both of them are exported, the program is ill-formed (but no diagnostic is required unless the program names the entities at a point where both are visible).

In order to support single-pass implementations that use name mangling to implement this two-level linkage model (and do not wish to mangle a module name into an exported entity), we propose an additional rule: if any declaration of an entity is exported, the first declaration of that entity must be exported.

Cyclic dependencies and module partitions

As noted in section 4.13 of N4465, cyclic dependencies between modules are not permitted in the Lenexa proposal. Consider this pair of classes:

```
export class WidgetPiece {
public:
  void frob(Widget *w);
  // ...
};

export class Widget {
  std::vector<WidgetPiece> pieces;
public:
  void frob() { for (auto &piece : pieces) piece.frob(this);
}
};
```

The definition of class `Widget` needs the definition of class `WidgetPiece` to be visible, and the definition of class `WidgetPiece` needs a declaration of class `Widget` to be visible. Under the Lenexa proposal, we are required to place both class definitions in the same module: if they were defined in separate modules, the module defining `WidgetPiece` would need a forward-declaration of class `Widget`, and forward-declaring an entity from another module is not permitted.

We think this is a reasonable restriction: such cyclic dependencies indicate a tight coupling between the components, such that there is no meaningful layering between them. However, the Lenexa proposal also restricts a module to a single module interface unit, which means that both classes must additionally be defined in the same source file (or textually `#include`d into that source file), which prevents the module interface from being arranged in a way that is natural and convenient for the domain, and we believe the combination of these restrictions to be unreasonably constraining -- in order to factor their source code into multiple files as they desire, our poor user must again use textual inclusion and include guards, with all of their problems!

The purpose of the restriction to a single interface unit is to establish a single point of truth for the definition of the contents of a module. This is important for both authors of the code, readers of the code, and tools. We propose an extension that retains this single point of truth but permits source code to be more freely organized:

Module partitions

Under this proposal, a module's interface can optionally be split across multiple translation units, known as *module partitions*. A module partition is a translation unit that forms part of the interface of a module. Unlike textual #inclusion into a single module interface unit which is translated wholesale, partitions can be translated independently; this does not preclude translation as part of the complete module, depending on the desires of the implementation.

Module partitions are imported by the module interface unit with the following syntax:

`import partition` *string-literal* `;`

The *string-literal* identifies the source file that defines the imported partition. That file is translated and incorporated into the current module, much as if it were a separate module, except that entities that it declares are under the purview of the current module.

The *identifier* `partition` is a context-sensitive keyword and may still be used as an identifier in contexts other than those specified here (such as in the standard library function `std::partition`). In order to support parallel module partition compilations, imports (including partition imports) must occur before any declarations in the translation unit.

Module partitions and their imports are required to form a directed acyclic graph with a single root node, the *module interface unit*. That is, all partitions of a module must be reachable through `import partition` declarations from the module interface unit.

As in the Lenexa proposal, the module interface unit begins with a *module-declaration* of the form

`module` *module-name* `;`

Each other module partition begins with a *module-declaration* of the form

`module partition` *module-name* ;

The partitions of a module must nominate the "correct" *module-name*; if a partition imports another partition with a different *module-name* (or a non-partition imports a partition), the program is ill-formed.

Example

WidgetPiece.cppm

```
module partition Widget; // This is a partition of module Widget

export class Widget; // OK, forward-declaration of
                     // entity from this module
export class WidgetPiece {
public:
  void frob(Widget *w);
  // ...
};
```

Widget.cppm

```
module Widget; // This is the main interface unit of module Widget
import partition "WidgetPiece.cppm";

export class Widget {
  std::vector<WidgetPiece> pieces;
public:
  void frob() { for (auto &piece : pieces) piece.frob(this); }
};
```

The interface of module `Widget` includes both class `Widget` and class `WidgetPiece`.

Exported macros

Some important and extremely common C++ libraries choose to expose macros as part of their public interface. These include:

Qt
MFC
ICU
Some of the boost libraries
Google Mock and Google Test
CxxTest
The C++ standard library (NULL, offsetof, feature test macros, …)

If we wish to provide a fully-modular experience for people using these libraries, we should not relegate these interfaces to a second-class position. Instead, we should provide a way for these macros to be intentionally exported by a module, in the cases where the macro is a deliberate part of the design of the library.

Can't we keep the macros in a separate, #included file?

In some cases, the library maintainers may be happy to accommodate this. But in other cases, there will be resistance to what some see as a forced artificial refactoring of the library in order to placate an unnecessary restriction. These users will not embrace the modules system, and this in turn will hinder adoption.

Further, in some cases (such as the boost.preprocessor library), the compilation time cost of reading the macro definitions and building the preprocessor data structures is significant, and we cannot avoid repeating this cost across translation units unless we provide a way to pretranslate the macros.

The arguments against including macros in modules are focused on accidental macro interference, from macros the user did not intend to import. This does not seem like a realistic problem for macros that are intentionally exported as part of the interface of a library that is intentionally imported into end user code. However, it does argue that macros should not be exported *by default* from modular compilations.

Proposed solution

We propose that, at the point of macro definition within a module, the module author can explicitly nominate that the macro is to be exported, by inserting the token `export` between the `#` and `define` tokens:

```
#export define CHECK_EQ(x, y) \
  ::my::lib::CheckImpl((x), (y), #x, #y, __FILE__, __LINE__)
```

Exported macro definitions are expected to be unique: if two modules export macros with the same name, those modules are imported into a translation unit, and the macro name is used, the program is rejected due to ambiguity.

Legacy module partitions

In order for a modules system for C++ to be successful, it must be possible to incrementally and gradually transition existing code. And we must accept that some code will *never* be transitioned to a C++ module system, perhaps because it is too costly to change, or it is C code, or must compile with earlier compilers, or the license prohibits modifications.

Consider the case of a modular C++ library that wraps a legacy library, and re-exports some of its interface:

```
module WrapFoo;
export module {
      #include "foo_widget.hpp"
}
export std::unique_ptr<Widget> make_widget(...);
```

The above code is subtly broken. Consider a user of that code, which has itself not been transitioned to modules yet:

```
import WrapFoo;
#include "other_library.hpp" // #includes "foo_widget.hpp"
// …
```

This will not compile: the compiler will see repeated definitions of every entity defined by "foo_widget.hpp", because it is textually included after a module containing it is imported. The problem is that we have violated a fundamental rule for correct usage of textual headers:

> **if the declarations from a textual header are visible, the include guard macro for that header must also be visible**

In order to fix this, the `WrapFoo` module must export the include guard macro of "foo_widget.hpp".

Careful ordering of imports and `#include`s alone cannot address these issues in the face of module partitions. Consider the somewhat more complex example where the user of WrapFoo above is itself a module with several partitions:

WrapBar.cppm

```
module WrapBar;
import partition “WrapBarPart1.cppm”;
import partition “WrapBarPart2.cppm”;
// ...
```

WrapBarPart1.cppm

```
module partition WrapBar;
import WrapFoo;
// ...
```

WrapBarPart2.cppm

```
module partition WrapBar;
import partition “WrapBarPart1.cppm”;

export module {
  #include “bar.hpp” // #includes “foo_widget.hpp” eventually
}
```

This will end up necessitating the inclusion of the bar.hpp header file after `WrapFoo` has already been imported, and declarations from foo_widget.hpp have been made visible as a consequence.

Proposed solution

We propose to solve this with a new form of module partition import:

```
import legacy string-literal ;
```

This declaration is equivalent to an import of a module partition containing

```
module partition module-name ;
export module {
      #include string-literal
}
```

(where *module-name* is the name of the current module) except that, in addition to exporting all the declarations from the nominated file, it also implicitly exports every macro defined at the end of preprocessing the named file.

Our `WrapFoo` module interface then contains this:

```
import legacy "foo_widget.hpp"
```

as a way of declaratively stating the intent to provide the legacy interface from "foo_widget.hpp".

Transparent migration

With the above features, an implementation can choose to provide a transparent migration path to modules for code that already intends to provide a modular, self-contained interface from its header files. This requires an implementation to be informed of the set of headers that it should treat as modular. It can then treat

```
#include HDR
```

(where *HDR* names such a modular header) as an import of an implicitly-generated module

```
module unique-name ;
import legacy HDR ;
```

That transparent migration path is not proposed by this proposal, but we explicitly intend for it to be a natural (and conforming, as mapping from `#include`s to source files is implementation-defined) extension.

Exports and internal linkage

As per the Lenexa proposal, we do not permit internal-linkage entities to be exported from a module. That creates problems when mass-exporting the contents of a legacy header, which may contain internal linkage entities. We propose a slight refinement to the Lenexa proposal's rule: for an isolated export declaration such as

```
	export static int f();
```

the declaration is ill-formed, but for an internal-linkage declaration appearing in an export block, such as

```
	export {
		static int f();
		// other things
	}
```

the internal-linkage entity is simply not exported. (This differs from Clang's approach, where such an entity is still exported, but in the case of ambiguity between multiple definitions of distinct but equivalent internal-linkage entities, an arbitrary selection is made.)

Module names as strings

An important consideration for naming modules is establishing the mapping from a module name to the module interface unit. This turns into a practical concern not just for compilers, but also for a wide variety of tools, from build systems to editors. When dealing with large code bases, this can even be a significant scaling limitation as every tool has to access a large database to establish this mapping.

We propose that instead of using arbitrary names for modules, we avoid inventing a module naming system entirely, and refer to modules by referring to the module interface file. While we expect this would be implemented much the same way as header file names are handled with `#include`, the idea would *not* introduce any kind of textual relationship; it would simply utilize the location of the module interface unit as the distinguishing identifier of the module.

Implementations already have several well established convenience features for such references such as relative lookup. We expect this would often allow module partition and implementation units to easily refer to a module interface unit in the same directory. This would be durable across directory restructuring and other changes as well.

We would be happy allowing either the existing <> delimited or `""` delimited strings, with the existing implementation defined mappings. We would also be happy simplifying things to only have a single `""` delimited option. However, it seems preferable to reuse the system that C++ tools already have for mapping these strings to files, in order to ease the introduction of modules awareness to these tools.

For example, this would make the syntax for importing the standard's vector and iostreams code:

```
	import <vector>;
```

```
import <iostream>;
```

This would also remove the need for a name in the *module-declaration*. However, a name could be provided there to provide a more human-readable alias or other useful information. We’re not specifically suggesting that at the moment, though, and would be happy with:

```
module;
```

By embedding the mapping from modules to their interface files in the source code, we believe this will significantly simplify tooling and build systems which interact with modular C++ code.