Document number P0273R1

Date 2016-10-17

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.

Changes since R0

The following changes were approved by EWG in Jacksonville:

- The module-declaration specifying that a translation unit is a module must be the first declaration in the source file. Vote: **6 | 14 | 9 | 2 | 0**[1]
- Module partitions, allowing a single module (and its notion of module ownership) to be split across multiple interface files. Vote: **11 | 12 | 3 | 4 | 0**
- Module implementations begin with "`module implementation M`" stanza rather than "`module M`". Vote: **10 | 11 | 8 | 3 | 0**

Wording changes for the above will be provided as a separate paper, but not for this meeting as it is anticipated that CWG will be fully occupied resolving NB comments for C++17.

R0 suggested that module names be string literals denoting a filesystem location for the corresponding file. Committee feedback was that allowing modules to be found based on

[1] Per WG21 convention for 5-way polls, these results are:
Strongly in favor | in favor | neutral | against | strongly against

their name may be useful, but the names should be specified as a dotted sequence of identifiers rather than a string. An implementation might, for instance, specify a mapping from dotted names to paths by replacing periods with directory separators and searching along some set of include paths, but this need not be specified by the Modules TS. Alternative implementation strategies, where no such mapping is necessary because the build system is aware of module dependencies, should also be supported. By moving this concern from the realm of the TS to that of the implementation, we require no further EWG discussion, and leave open multiple avenues of investigation so that more experience can be gained from the TS and fed back into the Standard.

The section on module ownership has been removed, as the Kona updates to the Lenexa proposal addressed the relevant concerns.

Specific changes

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.

Prior EWG discussion

Previous discussion on this topic led to several EWG polls, which we take as guidance to proceed with a proposal to allow modules in the Modules TS to export macros:

Allow exporting and importing macros somehow in some specification? **12 | 11 | 4 | 5 | 1**
Should we have just one TS (including macro import/export)? **5 | 11 | 13 | 3 | 0**

Questions were raised about providing an import syntax that would either produce an error if the imported module exports macros or filter out macros, but there was no clear consensus for proceeding in either of those directions, so our proposed solution does not provide such facilities at this time.

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.

In order to support legacy module partitions (see below), we propose one more feature:

```
#export_macros
```

This directive instructs the preprocessor to export all macros defined by the current translation unit. The purpose of this directive is to permit an incremental transition away from "`import legacy`"; see below for details.

Legacy module imports

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 import:

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

This declaration is equivalent to an import of a module containing

```
module unique ;
#export_macros
export module {
      #include string-literal
}
```

(where *unique* is the name of a unique module; this module should be the same for all legacy imports of the same file[2], and shall be different for legacy imports of two different files). This precise equivalence allows an incremental transition away from legacy module imports, whenever the library in question is ready to do so.

Our `WrapFoo` module interface then contains this:

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

as a way of declaratively stating the intent to import 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

[2] We leave determination of "the same file" implementation-defined. Unlike other facilities requiring a notion of "same file" such as #pragma once, the consequence of two handles to the same file being determined as "different" is merely that equivalent modules will be (harmlessly) imported twice, rather than a change in program semantics.

```
#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.)

In addition, we propose that the existing rule that the pre-existing rule that namespace scope variables of const-qualified type implicitly receive internal linkage be removed for a variable declared within a module interface unit (thus, such a variable can be exported, and receives module linkage by default if not exported):

```
module my.constants;
const int kFoo = 123;
export const int kBar = 456; // ok
export int getFoo();

module implementation my.constants;
int getFoo() { return kFoo; } // ok, module linkage

module implementation somewhere.else;
```

```
import my.constants;
int a = kBar; // ok
int b = kFoo; // error, not exported
```

This change has been discussed offline with Gabriel Dos Reis and at least partially matches the existing behavior of the MSVC implementation.