Module Interfaces and Preamble

Gabriel Dos Reis
Microsoft

This document discusses how to distinguish module interfaces (and partitions) from ordinary implementation units, and the placement of module declarations. It also offers formal wording that implement two design choices addressing active issues on the module issue list.

1. Module Interfaces

Proposal P0629R0 provides a good part of the changes for using "export module M;" as the declaration of a module interface unit. Here, completeness, I provide formal wording for the alternative syntax.

Amend paragraph 10.7/3 as follows:

> A *module* is a collection of module units, ~~at most~~ exactly one of which contains a *module-declaration* with `[[interface]]` in its *attribute-specifier-sequence*, *export-declarations* or *exported-fragment-groups* or *module-export-declarations*. Such a distinguished module unit is called the *module interface unit*. Any other module unit is called a module implementation unit.

User feedback

In earlier drafts, the syntax "export module M;" was used to export a module. After the Fall 2016 meeting in Issaquah, Washington, that syntax was removed and replaced by "export import M;" as part of resolution of Module Issue #1. The Visual C++ compiler implemented that resolution. However, the user community reaction has been for the most part extremely negative. So, recycling that syntax does pose concerns. The [[interface]] notation addresses that concern. The syntax is no less first-class than a use of keyword.

1.1. Module Partitions

There is a desire to loosen the requirement of a having all exports of a module declared in a single distinguished translation unit. That distinguished translation unit could be spread over several translation units called *module partitions*. They act collectively like a module interface unit. The suggestion here is to have

- those module units use the [[partition]] attribute in their module declarations
- the module partitions must be collectively translated, and the result of that translation defines the module interface.
  This requirement is novel compared to the traditional translation model that assumes one source file at time. The Visual C++ compiler implementation of modules has been experimenting with

this translation technique for a few releases, so this suggestion is based on some field experiment. It has the advantage or reducing syntax overhead of forward declarations.

The formal wording will be amendment to the same paragraph 10.7/3 as follows:

> A *module* is a collection of module units, ~~at most~~ exactly one of which contains a *module-declaration* with `[[interface]]` in its *attribute-specifier-sequence*, *export-declaration*s or *exported-fragment-group*s or *module-export-declaration*s. Such a distinguished module unit is called the *module interface unit*. Alternatively, if no module unit is a module interface unit, one more module units may contain *module-declarations* with `[[partition]]` in their *attributes-specifier-sequences*; they called module partitions. They shall be translated collectively and the result acts as the module interface unit. Any other module unit is called a module implementation unit.

2. Module Preamble

The current design of modules does not require a module declaration (or an interface declaration) as first declaration. Rather, it allows the programmer to introduce declarations (usually via #include directives) that are owned by the global module. For example,

```
#include <unistd.h>    // most likely non-modular system header
#include <QtCore>      // third party, not yet modularized


module FancyApp.Widget;      // my module starts here
struct FancyWidget : QWidget { /*… */ };
```

This flexibility allows:

1. Seamless migration from header files to module interfaces
2. Source code changes only where semantically needed – that is, a line change adds value
3. Components (especially third-party components) to be modularized at their own pace

This flexibility is being used by developers to move quickly from non-module world to module world, once they figured out architectural boundaries of *their* components. Often the change is just a one-liner to introduce module boundaries.

It has been suggested at various occasions that module declaration should be required as first declaration. With that requirement, it would become necessary to invent a mechanism and/or syntax to allow escaping back to the global module just so one could #include header files. Migration from non-module code to module world now would require more changes than would have been semantically necessary. Not only the mechanism is more complex, it is not clear what the benefits are.

I've heard that with a requirement of module declaration first, module code is aesthetically better. The problem is that "aesthetically better" is a very subjective term. In fact, the opposite is the case with the various syntaxes that have been suggested.

I’ve also heard that with a module declaration first, the compiler could use more efficient processing. Well, the truth is that in practice compilers are invoked in context (that certainly is the case with all implementations so far) and the compiler has all the information it needs for efficient processing strategy long before it even read the first character of the source file!

My recommendation is to proceed with the current design, which is flexible and allows migration effort and changes commensurate to value added.