Document number: P0142R0
Date: 2016-02-15
Working groups: WG21
Reply-To: gdr@microsoft.com
Revises: N4465

A Module System for C++ (Revision 4)

Gabriel Dos Reis Mark Hall Gor Nishanov

Abstract

We present a design of a module system for C++, along with rationale for the design choices. The document focuses on programmer's view of modules (both production and consumption) and how to better support modular programming in the large, componentization, scalable compilation, and semantics-aware developer tools.

Contents

1 Introduction

The lack of direct language support for componentization of C++ libraries and programs, combined with increasing use of templates, has led to serious impediments to compile-time scalability, and programmer productivity. It is the source of lackluster build performance and poor integration with cloud and distributed build systems. Furthermore, the heavy-reliance on header file inclusion (*i.e.* copy-and-paste from compilers' perspective) and macros stifle flowering of C++ developer tools in increasingly semantics-aware development environments.

Responding to mounting requests from application programmers, library developers, tool builders alike, this report proposes a module system for C++ with a

handful of clearly articulated goals. The proposal is informed by the current state of the art regarding module systems in contemporary programming languages, past suggestions [4, 6], experiments such as Clang's [2, 5], ongoing implementation in the Microsoft C++ compiler, and practical constraints specific to C++ ecosystem, deployment, and use. The design is minimalist; yet, it aims for a handful of fundamental goals

1. componentization;
2. isolation from macros;
3. scalable build;
4. support for modern semantics-aware developer tools.

Furthermore, the proposal reduces opportunities for violations of the One Definition Rule (ODR), and increases practical type-safe linking. An implementation of these suggestions is ongoing in the Microsoft C++ compiler.

2 Changers from Previous Revisions

2.1 Changes from N4465

Incorporate feedback from the Core Working Group (CWG), mostly clarifying terminology and design points:

- addition of a new linkage: module linkage
- allow block-scope extern declaration to match previous declarations in current module unit.
- strong module ownership is deferred

3 The Problems

The primary issue we face when trying to scale compilation of C++ libraries and programs to billions of lines of code is how C++ programmers author software components, compose, and consume them.

3.1 The Existing Compilation and Linking Model

C++ inherited its linking model from C's notion of *independent* compilation. In that model, a program is composed of several translation units that are processed independently of each other. That is, *each translation unit is processed with no knowledge or regard to any other translation units it might be composed with in an eventual program*. This obviously poses inter-translation units communication and coherence challenges. The communication problem is resolved via the notion of *name linkage*: a translation unit can reference, by name, an entity defined in another translation – provided the entity's name is *external*. All that the consuming translation unit needs to do (because it is translated independently and with no knowledge of that entity's defining translation unit) is to brandish a "matching" declaration for the referenced entity. The following example illustrates the concept. Consider the program composed of the translation units 1.cc and 2.cc:

1.cc (producer of `quant`)

```
int quant(int x, int y) {
  return x*x + y*y;
}
```

2.cc (consumer of `quant`)

```
extern int quant(int, int);
int main() {
  return quant(3, 4);
}
```

The program is well-formed and the calls to `quant` (in 2.cc) is resolved to the definition in translation unit 1.cc. Please note that none of the translation units mentions anything about each other: 2.cc (the consumer of the definition of `quant`) does not say anything about which translation unit is supposed to provide the definition of `quant`. In particular, the program composed of the translation units 2.cc and 3.cc, defined as follows

3.cc (another producer of `quant`)

```
#include <stdlib.h>
int quant(int x, int y) {
  return abs(x) + abs(y);
}
```

is also well-formed. This linking model, whereby translation units do not take explicit dependencies and external names are resolved to whatever provides them, is the bedrock of both C and C++ linking model. It is effective, but low-level and brittle. It also underscores the problem of coherency across translation units with declarations of entities with external linkage; in another words it poses continuing vexing *type-safe linking* challenges [1, §7.2c].

3.2 Header Files and Macros

The conventional and most popular C++ software organization practice rests upon a more than four decades old linking technology (§3.1) and a *copy-and-paste* discipline. Components communicate via sets of so-called *external names* designating externally visible entry points. To minimize risks of errors of various sorts, these names are typically declared in *header files*, conventionally placed in backing storage of the hosting environment filesystem. A given component uses a name defined in another component by including the appropriate header file via the preprocessor directive `#include`. This constitutes the basic information sharing protocol between producers and consumers of entities with external names. However, from the compiler's point of view, the content of the header file is to be *textually copied* into the including translation unit. It is a very simple engineering technology that has served the C and C++ community for over forty years. Yet, over the past seventeen years, this source file inclusion model has increasingly revealed itself to be ill-suited for modern C++ in large scale programming and modern development environments.

The header file subterfuge was invented as a device to mitigate the coherency problem across translation units. When used with care, it gives the illusion that there is only one "true source" of declaration for a given entity. However, it has several frequent practical failure points. It commonly leads to inefficient use of computing resources; it is a fertile source of bugs and griefs, some of which have been known since the discovery of the preprocessor. The contents of header files are vulnerable to macros and the basic mechanism of textual inclusion forces a compiler to process the same declarations over and and over in every translation unit that includes their header files. For most "C-like" declarations, that is probably tolerable. However, with modern C++, header files contain *lot* of executable codes. The scheme scales very poorly. Furthermore, because the preprocessor is largely independent of the core language, it is impossible for a tool to understand (even grammatically) source code in header files without knowing the set of macros and configurations that a source file including the header file will activate. It is regrettably far too easy and far too common to under-appreciate how much macros are (and have been) stifling development of semantics-aware programming tools and how much of drag they constitute for C++, compared to alternatives.

3.3 The One Definition Rule

C++ is built around the principle that any entity in a well-formed program is defined exactly once. Unfortunately, the exact formulation of this rule isn't that simple, primarily because of unfortunate but unavoidable consequences of the copy-and-

paste technology implied by the preprocessor directive `#include`. The outcome is that the arcane formal rules are variously interpreted by implementers (violation of the ODR results in undefined behavior), doubt, uncertainty, and sometimes outright willful disregard, from library writers and application programmers. Quoting Bjarne Stroustrup in the EWG reflector message c++std-lib: "*Every word in the C and C++ definitions about 'ODR' are there to work around the fact that we cannot identity the one true definition and have to compare definitions instead*."

Having a single, authoritative place that provides the declaration (and definition) of an entity reduces the risks of declaration mismatches going undetected, and improvements to type safe linkage.

4 Goals and Principles

The design described in these pages aims at supporting sound software engineering practices for large scale programming (e.g. componentization), scalable uses of development resources (e.g. build throughput, build frameworks), semantics-aware development tools (e.g. code analysis), etc.

4.1 The Preprocessor

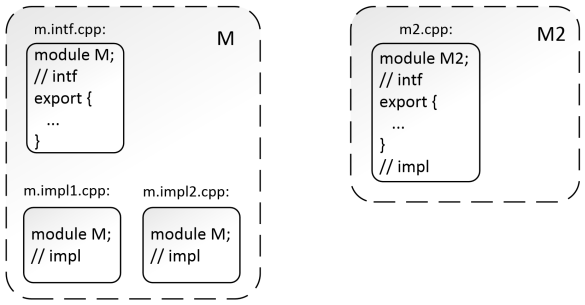
While many of the problems with the existing copy-and-paste methodology can be directly tied to the nature of the preprocessor, this proposal suggests neither its eradication nor improvements of it. Rather, the module system is designed to co-exist with and to minimize reliance on the preprocessor. We believe that the preprocessor has been around for far too long and supports far too many creative usage for its eradication to be realistic in the short term. Past experience suggests that any improvements to the preprocessor (for modularization) is likely to be judged either not enough or going too far. We concluded that whatever the problems are with the preprocessor, any modification at that level to support modularization is likely to add to them without fundamentally moving the needle in the right direction.

A central tenet of this proposal is that a module system for C++ has to be an *evolution* of the conventional compilation model. The immediate consequence is that it has to inter-operate with the existing source file inclusion model while solving the significant problems and minimizing those that can't be completely solved.

4.2 Componentization and Interface

For effective componentization, we desire direct linguistic support for designating a collection of related translation units, a *module*, with well-defined set of entry

points (external names) called the *module's interface*. The (complete) interface should be available to any consumer of the module, and a module can be consumed only through its interface. Usually, a module contains many more entities than those listed in its exported declarations. Only entities explicitly listed by the module interface are available for consumption (by name) outside the module. A translation unit constituent of a module is henceforth called a *module unit*. A module should have a symbolic name expressible in the language, so that it can be used by importing translation unit (consumer) to establish an explicit symbolic dependency.



4.3 Scoping Abstraction

One of the primary goals of a module system for C++ is to support structuring software components at *large scale*. Consequently, we do not view a module as a minimal abstraction unit such as a class or a namespace. In fact, it is highly desirable that a C++ module system, given existing C++ codes and problems, does not come equipped with new sets of name lookup rules. Indeed, C++ already has at least seven scoping abstraction mechanisms along with more than half-dozen sets of complex regulations about name lookup. We should aim at a module system that does not add to that expansive name interpretation text corpus. We suspect that a module system not needing new name lookup rules is likely to facilitate mass-conversion of existing codes to modular form. Surely, if we were to design C++ from scratch, with no backward compatibility concerns or existing massive codes to cater to, the design choices would be remarkably different. But we do not have that luxury.

4.4 Separation

A key property we require from a module system for C++ is *separation*: a module unit acts semantically as if it is the result of fully processing a translation unit

from translation phases 1 through 7 as formally defined by the C++ standards [3, Clause 2]. In particular, a module unit should be immune to macros and any preprocessor directives in effect in the translation unit in which it is imported. Conversely, macros and preprocessor directives in a module unit should have no effect on the translation units that import it.

Similarly, a declaration in an importing module unit should have no effect –in general– on the result of overload resolution (or the result of name lookup during the first phase of processing a template definition) performed in the imported module and its module units. That is, module units and modules should be thought of as "fully backed" translation units.

A corollary of the separation principle is that the order of consecutive import declarations should be irrelevant. This enables a C++ implementation to separately translate individual modules, cache the results, and reuse them; therefore potentially bringing significant build time improvements. This contrasts with the current source file inclusion model that generally requires re-processing of the same program text over and over.

4.5 Composability

Another key primary purpose of a module system is to allow independent components to be developed independently (usually by distinct individuals or organizations) and combined seamlessly to build programs. In particular, we want the ability to compose independent modules that *do not export* the same symbols in a program without worrying about possible duplicate definition clashes from their defining modules (see §5.4.) Therefore, a corollary of the composability requirement is that of *ownership and visibility* of declarations: a module owns declarations it contains, and its non-exported entities have no relationship with entities from other modules. This is the *strong module ownership* model. At the Spring 2015 Meeting in Lenexa, there were concerns that the strong module ownership may require ABI breakage. Consequently, the Evolution Working Group (EWG) has adopted a weaker version, called the *weak module ownership* model: Only non-exported entities are owned by the modules containing their declarations; it is ill-formed (but no diagnostic required) for a program to contain two modules exporting two entities of the same kind/type with the same name from the same namespace.

Operationally, there are various ways an implementation may achieve this effect. E.g. decorating an entity's linkage name with its owning module's name, two-level linking namespace, etc. However, we believe that the notion of linkage should not be elevated above where it belongs, and existing implementations have access to far more elaborate linkage mechanisms than formally acknowledged and

acknowledgeable by the C++ standards.

4.6 Coexistence with Header File

In an ideal world with modules, the usage of the time-honored header files should be rare, if not inexistent. However, realistically we must plan for a transitional path where programs involve components written today in the source-file-inclusion model, and new module-based components or existing components converted to use modules. Furthermore, conversion from heavy macro-based header files are likely to combine parts that are safely modularized with old-style macro interfaces – until the world has completely moved to pure module systems and the preprocessor has vanished from the surface of planet Earth, Solar System.

We acknowledge that the principle of coexistence with source file inclusion does pose significant constraints and brings complications into the design space, e.g. with respect to the ODR.

4.7 Runtime Performance

Moving an existing code to a brave new module world, or writing new codes with modules, should not in any way degrade its runtime performance characteristics. In particular, we do not seek a module system requiring a compiler to perform automatic "boxing" of object representation (exposed in class private members) –in attempts to reducing re-compilation– via opaque data pointers à la `pImpl` idiom.

5 Design Choices

The principles and goals just outlined confine us to parts of the module system design space. We have to make further design decisions. Ideally, it should be easy to transform an existing program `#include`ing header files to consume modules, e.g.:

```
import std.vector;     // #include <vector>
import std.string;     // #include <string>
import std.iostream;   // #include <iostream>
import std.iterator;   // #include <iterator>

int main() {
  using namespace std;
  vector<string> v = {
    "Socrates", "Plato", "Descartes", "Kant", "Bacon"
  };
  copy(begin(v), end(v), ostream_iterator<string>(cout, "\n"));
}
```

That is, it should be a matter of mechanical replacement of header files with corresponding module import declarations and nothing else.

5.1 Module Declaration

The first design decision to make is whether it is necessary for a translation unit to declare itself as a module unit, or if the fact that it is a module unit is the result of some compiler invocation command line switches or some external configuration files.

Given that a module unit is expected to possess a strong ownership semantics, unlike a mere preprocessing unit, it is important that the rules of interpretation are reflected syntactically as opposed to being guessed from the translation environment, the build environment, or implementation-defined command line invocation switches. Consequently, we propose that a module unit be introduced by a declaration:

`module` *module-name* `;`

This declaration means that subsequent declarations in the current translation unit are part of the module nominated by *module-name*. For simplicity, there can be at most one module declaration per translation unit. In a previous design, we required the module declaration to be the first declaration in a module unit; that requirement wasn't necessary (as acknowledged at the time). To support gradual transition from the current compilation model to a world with module we allow toplevel declarations to preceed the module declaration in a module unit. Such declarations do not belong to the nominated module. They belong to the global module (§5.7).

Rule 1 *A translation unit may contain at most one module declaration. The resulting translation unit is referred to as a* module unit.

Note: A module can span several module units — all of which must declare the module they belong to. Like most declarations in C++, it may be necessary to allow attributes on module declarations.

5.1.1 Module Names and Filenames

Having decided on the necessity to have a module declaration, the next question is whether the *module-name* should have any relationship at all with the filename of the source file containing the module unit. We believe that prescribing any such relationship will be too rigid, impractical compared to the flexibility offered today by the source file inclusion model – see examples in §3.1.

We propose a hierarchical naming scheme for the name space of *module-name* in support of submodules, see §5.5.

5.2 Module Interface

It is desirable, from composability perspective, that the language has direct support for expressing a module interface separately from its implementation. This raises at least two questions:

1. Should a module interface declaration be required in a source file distinct from the file that contains its implementation?

2. Should both the interface and implementation of a module be contained in a single source file?

The answers to both questions should be "no". Requiring a module interface declaration to be provided in a file distinct from the implementation file, while in general sound advice, is too constraining a requirement to accommodate all C++ libraries and programs. It should be *possible* for a module author to provide a single source file containing both the module interface and implementations (of both exported and non-exported entities) have the compiler automatically generate the appropriate information containing exported declarations.

Similarly, requiring both interface and implementation to be contained in the same file is too constraining and misses sound engineering practice. Furthermore, we would like to support the scenario where a single module interface is provided, but several independent implementations are made available and the selection of the actual implementation needed to make up a program is left to the user.

5.2.1 Syntax

A module publishes its external entry points through exported declarations of the form

`export` *toplevel-declaration*

or

`export` `{` *toplevel-declaration-seq* `}`

The braces in this context do not introduce a scope, they are used only for grouping purposes. A *toplevel-declaration* is either an import declaration (see §5.3), a module-export declaration, or an ordinary declaration. An import declaration states that all declarations exported by the nominated module are made available (e.g. the

names are available) to the importing translation unit. A module-export declaration in the exported section means that the names exported by the nominated module are transitively accessible to any consumer (e.g. importing translation unit) of the current module.

Rule 2 *No export declaration shall mention a non-exported entity, or an entity with internal linkage or no linkage.*

Note: An entity may be declared as exported, and later defined without the `exported` keyword. Such an entity is still considered exported; only the properties that were computed in the export declaration are exported to the module consumers. In particular, a class that is only declared (but not defined) in an export declaration appears incomplete to the module's consumers even if the class is completed later in the same module unit that declares the interface. Similarly a default argument not present in the export declaration is not visible to the module's consumers. See §5.2.3.

5.2.2 Ownership

Only names exported from a module can be referenced externally to the module. Furthermore, non-exported names cannot be source of ODR violation across two distinct modules; however duplicate definitions in the same module is ill-formed.

alfa.cxx

```
struct S1 {...};
module Alfa;
struct S2 {...};
export namespace Alfa {
  struct S3 {...};
}
```

bravo.cxx

```
struct S1 {...};
module Bravo;
struct S2 {...};
export namespace Bravo {
  struct S3 {...};
}
```

In the example above, modules `Alfa` and `Bravo` contributes structure `S1` to the global module, and they are subject to the usual ODR constraints. Both of them define their own structures `S2` and `S3`. Despite both `Alfa` and `Bravo` defining structure `S2`, it is possible for another module to import both in the same program.

5.2.3 Exported Class Properties

An occasionally vexing rule of standard C++ is that of controls access, not visibility. E.g. a private member of a class is visible to, but not accessible to non-member entities. In particular, any change to a private member of a class is likely to trigger

re-processing of any translation unit that depends on that class's definition even if the change does not affect the validity of dependent units. It is tempting to solve that problem with a module system. However, having two distinct sets of rules (visibility and accessibility) for class members strikes us as undesirable and potentially fertile source of confusion. Furthermore, we want to support mass-migration of existing codes to modules without programmers having to worry about class member name lookup rules: if you understand those rules today, then you do not have to learn new rules when you move to modules and you do not have to worry about how the classes you consume are provided (via modules or non-modules).

That being said, we believe the visibility vs. accessibility issue is a problem that should be solved by an orthogonal language construct, irrespectively of whether a class is defined in a module interface declaration or in an ordinary translation unit. There are proposals (e.g. "uniform call syntax") independently of modules that also need the existing rule to be revisited.

Rule 3 *In general, any property of a class (e.g. completeness) that is computed in the export declaration part of a module is made available to importing modules as is.*

That is if a class is declared (but not defined) in the interface of a module, then it is seen as an incomplete type by any importing module, even it is defined later in the declaring module in a non-export declaration.

5.2.4 Should There Be an Interface Section at All? How Many?

It is sensible to imagine a design where a module interface is inferred or collected from definitions that have special marker (e.g. `export`), instead of requiring that the interface be declared at one place. A major downside of that design is that for an import declaration to be useful (e.g. to effectively consume the module's interface), its entirety needs to be produced in a sort of preprocessing phase by some tools that would scan all modules units making up the module. Therefore, it appears that any perceived theoretical benefit is outweighed by that practical implication.

5.2.5 Should a Module Interface Be Closed?

For practical reasons similar to those exposed in §5.2.4, we require a module interface to be declared "once for all" at a unique place. This does not preclude extensions of a module. Indeed submodules (see §5.5) can be used to extend modules through composition and/or module-export declaration of submodules.

5.2.6 Alternate Syntax for Module Interface

Several suggestions have been made as alternatives to the currently proposed syntax. In particular, it was observed that if the interface section should immediately follow a module declaration, then both could be combined into a single declaration. That is, instead of writing

```
module M;
export {
   int f(int);
   double g(double, int);
}
```

one could simply write

```
module M {
   int f(int);
   double g(double, int);
}
```

we considered this but eventually rejected this notation since it is too close to classes and namespaces (seen as good by some) but is deceptive in that modules do not introduce any scope of their own – see §4.3. That syntax was also part of the oiginal module suggestion by Daveed Vandevoorde, but met resistance [6, §5.11.2]. Furthermore, export declarations and non-export declarations can be freely mixed.

We also avoided reusing the access spcifiers `public`, `private` to delimit *visibility* boundaries in a module.

5.3 Import Declaration

A translation unit makes uses of names exported by other modules through import declarations:

`import` *module-name* `;`

An import declaration can appear only at the global scope. It has the effect of making available to the importing translation unit all names exported from the nominated module. Any class completely defined along with all its members are made visible to the importing module. An incomplete class declaration in the export of a module (even if later completed in that module unit) is exported as incomplete.

Note: An alternate syntax for module importation that avoids a third keyword could be

`using module` *module-name* `;`

but the semantics of transitive exports might not be obvious from the notation.

5.4 Visibility and Ownership

Consider the following two translation units:

m1.cc

```
module M1;
export int f(int, int);

// not exported, local to M1
int g(int x) {
  return x * x;
}
// definition of f exported by M1
int f(int x, int y) {
  return g(x) + g(y);
}
```

m2.cc

```
module M2;
export bool g(int, int);
import std.math;
// not exported, local to M2
int f(int x, int y) {
  return x + y;
}
// definition of g exported by M2
int g(int x, int y) {
  return f(abs(x), abs(y));
}
```

where module `M1` defines and exports a symbol `f(int,int)`, defines but does not export symbol `g(int)`; conversely, module `M2` defines and exports symbol `g(int,int)` defines but does not export symbol `f(int,int)`. It is possible to build a program out of `M1` and `M2`

main.cc

```
import M1;
import M2;
int main() {
  return f(3,4) + g(3,4);
}
```

without ODR violation because each non-exported symbol is *owned* by the containing module.

5.5 Submodules

It is frequent for a component to consist of several relatively independent subcomponents. For example, the standard library is made out of a few components: core runtime support (part of any freestanding implementation), the container and algorithm library (commonly referred to as the STL), the mighty IO streams library, etc. Furthermore each of these components may be subdivided into smaller subcomponents. For example, the container library may be divided into sequence containers, associative containers, unordered containers, etc.

We propose a hierarchical naming of modules as a mechanism to support submodules, and extensions of modules by submodules. A submodule is in every

aspect a module in its own right. As such, it has an interface and constituent module units, and may itself contain submodules. For example, a module named `std.vector` is considered a submodule of a module named `std`. The one distinctive property of a submodule is that its name is only accessible to modules that have access to its parent, provided it is explicitly exported by the parent module.

A submodule can serve as cluster of translation units sharing implementation-detail information (within a module) that is not meant to be accessible to outside consumers of the parent module.

5.6 Aggregation

The current design supports expression of components that are essentially aggregates of other components. Here is an example of standard sequence containers component:

Standard sequence container module

```
module std.sequence;
export {
  module std.vector;
  module std.list;
  module std.array;
  module std.deque;
  module std.forward_list;
  module std.queue;
  module std.stack;
}
```

Note that module aggregates are different from submodules in that there is no relationship between the name of a module aggregate and modules it exports. The two notions are not mutually exclusive. For example, the module `std.sequence` as shown above is both a submodule of `std` and an aggregate module.

5.7 Global Module

To unify the existing compilation model with the proposed module system, we postulate the existence of a global module containing all declarations that do not appear inside any module (the case for all C++ programs and libraries in the pre-module era.) Only names with external linkage from the global module are accessible across translation units.

5.8 Module Ownership and ODR

As concluded in §4.5, a module has ownership of all declarations it contains. So, just about how much ownership is it?

Does a module definition implicitly establish a namespace? No, a module definition does not establish any namespace; and no particular syntax is needed to access a name made visible by an import declaration. All exported symbols belong to the namespace in which they are declared. In particular, the definition of a namespace can span several modules. In the following example,

parsing.cxx

```
module Syntax;
export namespace Calc {
  class Ast {
    //...
  };
}
```

vm.cxx

```
module Evaluator;
import Syntax;
// use Ast from module Syntax
namespace Calc {
  int eval(const Calc::Ast*);
}
```

the name `Calc` in the modules `Syntax` and `Evaluator` refers to the same namespace. The parameter type of the function `eval` involves the type `Calc::Ast` defined and exported by module `Syntax`.

Note: It is not possible for a translation unit to provide a declaration for an entity that it does not own. That is, a translation unit cannot use "`extern`" declaration to claim a matching declaration for an entity (with external linkage) declared in a different module unit. This restriction does not apply to entities in the global module (§5.7).

5.9 Constexpr and Inline Functions

We propose *no* fundamental change to the rules governing constexpr or inline functions. Any exported constexpr or inline function must be defined in the module unit providing the interface of the owning module. The definition itself need not be exported.

5.10 Templates

Standard C++'s compilation model of templates relies on copy-and-paste of their definitions in each translation unit that needs their instantiations. With the module ownership principle, each exported declaration of a template is made available to

importing translation units. As ever the two-phase name lookup applies whether a template definition is exported or not.

5.10.1 Definitions

Definitions for templates listed in a module interface are subject to constraints similar to those for inline functions. Furthermore, a class template that is only declared (but not defined) in an export declaration is seen as an incomplete class template by importing translation units.

5.10.2 Explicit Instantiations

An explicit instantiation is exported when it appears in an export declaration. The semantics is that the definition resulting from that instantiation is globally available to all importing translation units. For example, given the module

vec.cpp

```
module Vector;
export {
  template<typename T> struct Vec {
    // ...
  };
  // Explicit instantiation for commonly used specialization
  template struct Vec<int>;
}
```

the definition of the class `Vec<int>` is exported to any translation unit that imports `Vector`. This provides a mechanism for template authors to "pre-compute" common instantiations and share them across translation unit. Notice that this has effects similar to a C++11-style extern declaration of a specialization combined with an explicit instantiation in an appropriate translation unit.

Conversely, any explicit instantiation not in an export declaration is not exported; therefore the resulting definition is local to the containing translation unit. If a specialization is requested in another translation unit, that would otherwise match the non-exported instantiation, the usual rules for template specializations applies as well as the ODR.

5.10.3 Implicit instantiations

Any implicit specialization of a non-exported template is local to the requesting translation unit. For specializations of exported templates, we distinguish two cases:

1. the template argument lists (whether explicitly specified or deduced) refer only exported entities: the resulting instantiation is exported, and considered available to all importing modules. For all practical purposes, builtin types are considered exported.

2. at least one entities referenced in the template argument list is non-exported. By necessity, the request and the referenced entity must belong to the current translation unit. The resulting definition is non-exported and is local to the containing translation unit.

In each case, ODR is in effect. The rules are designed to allow maximum sharing of template instantiations and to increase consistency of definitions generated from templates, across translation units.

5.10.4 Template explicit specializations

A template explicit specialization is morally an ordinary declaration, except for the fact that it shares the same name pattern as specializations of its primary template. As such it can be exported if its primary template is exported and its template argument list involves only exported entities. Conversely, an explicit specialization of an exported may be declared non-exported. In that case, the declaration (and definition) is local to that module unit, and is unrelated to any other specialization that might be implicitly generated or explicitly defined non-exported in other translation units. For example, in the program

vec-def.cpp

```
module Vector;
export {
   template<typename T> struct Vec; // incomplete
   template<> struct Vec<int> { ... }; // complete
}
// Completed Vec<double>, but definition not exported
template<> struct Vec<double> { .... };
```

vec-use.cpp

```
import Vector;
int main() {
  Vec<int> v1 { ... }; // OK
  Vec<double> v2 { ... }; // ERROR: incomplete type
}
```

the class `Vec<int>` is exported as a complete type, so its use in the definition of the variable `v1` is fine. On the other hand, the type expression `Vec<double>` in vec-use.cpp refers to an implicit instantiation that of `Vec`, which is an incomplete type.

If an explicit specialization of a template, or a partial specialization of a template is declared exported, then its primary template must be exported.

5.11 The Preprocessor

It is not possible for a module to export a macro, nor is it possible for a macro in an importing module to affect the imported module. Components that need to export macros should continue to use header files, with module-based subcomponents for the parts that are well behaved. For example, an existing library that provides interfaces controlled by a preprocessor macro symbol `UNICODE` can modularize its constituents and continue to provide a traditional header file-based solution as follows:

Header file C.h

```
#ifndef C_INCLUDED
#define C_INCLUDED
#ifdef UNICODE
   import C.Unicode;
#else
   import C.Ansi;
#endif // C_INCLUDED
```

5.11.1 Macro-heavy header files

This proposal does not address the problem of macro-heavy header file. Such header files tend to be provided, in majority, by C-style system headers. We will note that often they contain fairly modularizable sub-components that are easily provided by submodule interfaces. Consequently, they can still use module interfaces for subcomponents while controlling their availability via macro guards in header files.

Can a module unit include a header file? Absolutely yes! Remember that the effect of file inclusion via `#include` is that of textual copy-and-paste, not modular declaration. Furthermore, any macro defined in that header file is in effect (until subject to an `#undef` directive). However, what is not possible is for the macros defined in that module to have any effect on any translation unit that imports it.

We anticipate that header files will continue to serve their purpose of delivering macro definitions even when they contain module imports that bring into scope modularized components.

5.12 Separate Compilation vs. On Demand

Since modules act semantically as a collection of self-contained translation units that have been semantically analyzed from translation phase 1 through 7, it is legitimate –from practical programming point of view– to ask whether a module nominated in an import declaration is required to have been separately processed prior to the module requiring it, or whether such module is analyzed on the fly or on demand. For all practical purposes, the answer is likely to be implementation-defined (to allow various existing practice), but our preference is for separate translation.

5.13 Mutually importing modules

With the source file inclusion model, the `#include` graph dependency must be acyclic. However, classes –and in general, most abstraction facilities– in real world programs don't necessarily maintain acyclic *use* relationship. When that happens, the cycle is typically "broken" by a forward declaration usually contained in one of the (sub)components. In a module world that situation needs scrunity. For simplicity of the analysis, let's assume that two modules `M1` and `M2` use each other.

5.13.1 Both Modules Use Each Other Only in Implementation

This situation is easy, and in fact is not really an cyclic dependency. Indeed, since module interface artefacts are separated by the compiler from module unit implementations, the acyclicity of use graph is still maintained.

5.13.2 One (But Not Both) Uses the Other at the Interface Level

Again, this situation is simple since acyclicity is maintained at the interface specfication level and an obvious ordering suggests itself. This situation is common and naturally supported by the proposal.

5.13.3 Both Use Each Other at the Interface Level

This situation is much rarer; the interfaces of `M1` and `M2` should be considered logically as part of a single larger module and treated as such, even though it is convenient from the programmer's perspective to physically split the entities in two distinct source files. Nevertheless, it is possible for the programmer to set up a

(delicate) processing order for the compiler to translate the *interface* parts of both modules, and then consume them independently.

6 Visibility and Parsing

6.1 Visibility vs. Accessibility

C++, as it exists, today make a distinction between *visibility* and *accessibility* of names. Almost by definition, names are always visible even if they have restrictive access specifiers. This implies that an exported class is forced to expose every and each of its members, even if some of those members are meaningful only to entities that are within the boundary of the owning module. This is most unfortunate and stands in the way of realizing one of the key benefits of modules: componentization at scale. Previous designs suggested an `internal` visibility specifier. That specifier can be used in combination with existing access specifiers to limit the visibility of class members. A class member with `internal` visibility specifier means that it is visible to module members, but not outside the owning module boundary. That specifier was rejected at the Spring 2015 meeting in Lenexa. Similarly, EWG rejected the idea of not exporting inaccessible members.

6.2 Parsing

The C++ grammar requires that, in most parsing contexts, names be unambiguously categorized into *type-name*s, *template-name*s, *namespace-name*s, etc. for further correct parsing progress. This requirement (resulting from grammar ambiguities) is most unfortunate as it stands in the way of reducing reliance on header files for forward declarations. Componentization, as suppored by modules, offers a unique opportunity to reduce the conceptual need for forward declarations, and therefore the need for header files within module boundaries. We believe this is an important problem to solve in order to achieve effective componentization at scale.

7 Tools Support

The abstract operational model of a module is that it contains everything that is ever to be known about its constituents module units. In particular, we envision that a high quality implementation will provide library interfaces for querying the elaborated forms of declarations, hosting environmental values, including translation command line switches, target machines, optional source form, binary form, etc. Ideally, a module would provide all that is necessary for a code analysis tool.

A library interface to internal representation of modules will be the subject of a separate proposal.

8 Build Systems

We acknowledge that most build systems work on file stamps. We aim for a module system that does not disturb that invariant. Ideally, modules should continue to work smoothly with existing build systems. For that reason, we have placed restrictions on where exported constexpr functions, exported inline functions, and exported template definitions should be located in module definitions.

9 Migration

The module system suggested in this proposal supports bottom up componentization of libraries, and everywhere consumption of modules in libraries and application programs. In another words, a non-modularized component can consume a module. However unprincipled header file inclusion in a module component may prove problematic.

Tools support will be key to a successful migration of the C++ planet to a module world. For example, a tool for detecting macro definition and usage dependencies in a translation unit will be useful. A tool for detecting multiple declarations of the same entity across source files will be needed to assist in gradual migration of existing source codes. Similarly, a tool turning an existing header file into an initial module interface (when configuration parameters are fixed) is valuable in top-down conversion scenarios when (manual) bottom-up conversion is not appropriate.

Acknowledgment

Numerous people provided feedback on the initial design effort and early drafts of this proposal, via face-to-face meetings, private conversations, or via the Module Study Group (SG2) reflector. We thank the following individuals: Jean-Marc Bourguet, Jonathan Caves, Ian Carmichael, Ale Contenti, Lawrence Crowl, Pavel Curtis, Galen Hunt, Loïc Joly, Leif Kornstädt, Aaron Lahman, Artur Laksberg, Sridhar Madhugiri, Reuben Olinsky, Andrew Pardoe, Dale Rogerson, Cleiton Santoia, Robert Schumacher, Richard Smith, Michael Spencer, Jim Springfield, Bjarne Stroustrup, Herb Sutter. Special thanks to the numerious early adopters from the C++ community who tried and provided feedback on early implementations of the design as outlined in this paper.

References

[1] Margaret E. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.

[2] Douglas Gregor. Modules. `http://llvm.org/devmtg/2012-11/Gregor-Modules.pdf`, November 2012.

[3] International Organization for Standards. *International Standard ISO/IEC 14882. Programming Languages — C++*, 4th edition, 2014.

[4] Bjarne Stroustrup. #scope: A simple scope mechanism for the C/C++ preprocessor. Technical Report N1614=04-0054, ISO/IEC JTC1/SC22/WG21, April 2004.

[5] Clang Team. Clang 3.5 Documentation: Modules. `http://clang.llvm.org/docs/Modules.html`, 2014.

[6] Daveed Vandevoorde. Module in C++. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3347.pdf`, December 2012. N3347=12-0037.