N1964=06-0034
2006-02-27
Daveed Vandevoorde
daveed@vandevoorde.com

Modules in C++

(Revision 3)

1 Introduction

Modules are a mechanism to package libraries and encapsulate their implementations. They differ from the C approach of translation units and header files primarily in that all entities are defined in just one place (even classes, templates, etc.). This paper proposes a module mechanism (similar to that of Modula-2) as an extension to namespaces with three primary goals:

- Significantly improve build times of large projects
- Enable a better separation between interface and implementation
- Provide a viable transition path for existing libraries

While these are the driving goals, the proposal also resolves a number of other long-standing practical C++ issues wrt. initialization ordering, run-time performance, etc.

1.1 Document Overview

This document explores a variety of mechanisms and constructs: Some of these are "basic features" which I regard as essential support for modules, whereas a number are just "interesting ideas" that are thought to be potentially desirable. Section 2 introduces the "basic features", which is followed (in Section 3) by some practical notes on how to transition to the basic feature set. An analysis of the benefits offered by the basic features is presented in Section 4.

Section 5 then presents the "premium features": Ideas that naturally fit the module formulation (e.g., an enhanced model for dynamic libraries). Section 6 covers rather extensive technical notes, including syntactic considerations. Section 7 is a repository of ideas that have been considered and rejected. We conclude with acknowledgments.

1.2 About Syntax

Most examples in this revision of the paper use a syntax that is different from the syntax proposed in previous revisions. It is hoped that this syntax (which introduces a new keyword **`import`**, unlike the original examples) improves the "first reading" experience.

2 Basic Features By Example

2.1 Import Directives

The following example shows a simple use of a **module namespace** (or simply, **module**). In this case, the module namespace encapsulates the standard library.

```
import namespace std;  // Module import directive.
int main() {
  std::cout << "Hello World\n";
}
```

The first statement in this example is a **module import directive** (or simply, an import directive). Such a directive makes a namespace available in the translation unit. In contrast to #include preprocessor directives, module import directives are insensitive to macro expansion (except wrt. to the identifiers appearing in the directive itself, of course).

Nonmodule namespaces are henceforth called **open namespaces**. The unqualified term "namespace" then refers to both modules and open namespaces. It may be possible to allow a namespace to consist both of a modular part and an open part, a situation we will refer to as a mixed namespace. The basic feature set does not generally allow for **mixed namespaces** (the global namespace being an exception; see 2.6), but we do discuss the possibility as a "premium option" in subsection 5.5.

2.2 Module Definitions

Let's now look at the definition (as opposed to the use) of a module namespace.

```
// File_1.cpp:
export namespace Lib {  // Module definition.
  import namespace std;
public:
  struct S {
    S() { std::cout << "S()\n"; }
  };
}

// File_2.cpp:
import namespace Lib;
int main() {
  Lib::S s;
}
```

Import directives only make visible those members of a module namespace that were declared to be "public" (these are also called **exported members**). To this end, the access labels "`public:`" and "`private:`" (but not "`protected:`") are extended to apply not only to class member declarations, but also to namespace member declarations.

Note that the constructor of `S` is an inline function. Although its definition is separated (in terms of translation units) from the call site, it is expected that the call will in fact be expanded inline using simple compile-time technology (as opposed to the more elaborate link-time optimization technologies available in some of today's compilation systems).

Module definitions cannot make use of nonmodule declarations. This constraint is imposed by implementation requirements, and implies that modules can only be transitioned to in a bottom-up fashion. For example, a standard library implementation must be converted to a module before a library depending on `std` can be similarly converted.

Variables with static storage duration defined in module namespaces are called **module variables**. Because modules have a well-defined dependency relationship, it is possible to define a sound run-time initialization order for module variables.

2.3 Transitive Import Directives

Importing a module is transitive only for public import directives:

```
// File_1.cpp:
export namespace M1 {
public:
  typedef int I1;
}

// File_2.cpp:
export namespace M2 {
public:
  typedef int I2;
}

// File_3.cpp:
export namespace MM {
public:
  import namespace M1;  // Make exported names from M1
                        // visible here and in code
                        // that imports MM.
private:
  import namespace M2;  // Make M2 visible here, but
                        // not in clients.
}

// File_4.cpp:
import namespace MM;
M1::I1 i1;  // Okay.
M2::I2 i2;  // Error: M2 invisible.
```

2.4 Private Declarations

Our next example demonstrates the interaction of module namespaces and private class member visibility.

```
// File_1.cpp:
export namespace Lib {
public:
  struct S { void f() {} };  // Public f.
  class C { void f() {} };   // Private f.
}

// File_2.cpp:
import namespace Lib;  // Private members invisible.
struct D: Lib::S, Lib::C {
  void g() {
    f();  // Not ambiguous: Calls S::f.
  }
};
```

The similar case using nonmodule namespaces would lead to an ambiguity, because private members are visible even when they are not accessible. In fact, within module namespaces private members must remain visible as the following example shows:

```
export namespace Err {
public:
  struct S { int f() {} };  // Public f.
  struct C { int f(); };    // Private f.
  int C::f() {}  // C::f must be visible for parsing.
  struct D: S, C {
    void g() {
      f();  // Error: Ambiguous.
    }
  };
}
```

It may be useful to underscore at this point that the separation is only a matter of visibility: The invisible entities are still there and may in fact be known to the compiler when it imports a module. The following example illustrates a key aspect of this observation:

```
export namespace Singletons {
public:
  struct Factory {
    // ...
  private:
    Factory(Factory const&);  // Disable copying
  };
  Factory factory;
}


// Client file:
import namespace Singletons;
Singleton::Factory competitor(Singleton::factory);
  // Error: No copy constructor
```

Consider the initialization of the variable **`competitor`**. In nonmodule code, the compiler would find the private copy constructor and issue an access error. With modules, the user-declared copy constructor still exists (and therefore not generated in the client file), but, because it is invisible, a diagnostic will be issued just as in the nonmodule version of such code. Subsection 5.4 proposes an additional construct to handle a less common access-based technique that does not so easily translate into modularized code.

2.5 Module Partitions

A module may be partitioned into **module partitions** to allow only part of the module to become visible at one time. For example, the standard header <vector> might be structured as follows:

```
  import namespace std["vector_hdr"];
     // Load definitions from std, but only those
     // those from the "vector_hdr" partition should
     // be made visible.
  // Definitions of macros (if any):
  #define ...
```

The corresponding definition has the following general form:

```
  export namespace std["vector_hdr"] {
  public:
   import namespace std["allocator_hdr"];
    // Additional declarations and definitions
  }
```

The partition name is an arbitrary string literal, but it must be unique among the partitions of a module. All partitions must be named, except if a module consists of just one partition.

The partition mechanism is also a convenient vehicle to spread module namespaces across multiple translation units. For example:

```
  // File_1.cpp:
  export namespace Lib["part 1"] {
    struct Helper {  // Not exported.
      // ...
    };
```

```
}

// File_2.cpp:
export namespace Lib["part 2"] {
  import namespace Lib["part 1"];
public:
  struct Bling: Helper {  // Okay.
    // ...
  };
}
```

Unlike open namespaces, module namespaces cannot be extended without this partition mechanism (i.e., modules don't have truly open scopes). This example also illustrates that when importing a module partition within the same module, all declarations (not just the exported ones) are visible.

The dependency graph of the module partitions in a program must form a directed acyclic graph. Note that this does not imply that the dependency graph of the complete modules cannot have cycles.

2.6 Global Namespace Mapping

Namespaces (both modules and open namespaces) can be marked "global" to express that the names of their members are reserved in the global namespace:

```
export namespace std["new_hdr"] { // Module namespace
public:
  import namespace std["stddef_hdr"];
  [["global"]] namespace core {  // Open namespace
    void* operator new(std::size_t);
    // ...
  }
}
```

(The double bracketed construct preceding the keyword namespace is is a **namespace attribute list**. Additional attributes are discussed in section 5. This example also shows an ordinary nonmodule namespace nested in a module namespace. The possibility of nested modules is discussed in subsection 5.3.)

This facility is primarily meant to enable a binary compatible transition from pre-module C++: Global module members can be code-generated as if in the global namespace, but they are otherwise treated as member of their namespace. Note that this effectively means that the global namespace may be a mixed namespace: Partly implemented using nonmodular code, and partly implemented through modular declarations.

In the case of the standard library this is particularly important because module declarations can generally not make use of nonmodule declarations. Since some standard library declarations (e.g., `::operator new`) belong to the global namespace, they could otherwise not be fit in a module. That would prevent the declarations in **`std`** from using these global declarations, which in turn would make the modularization of the standard library impossible.

3 Usage Notes

3.1 Expected Implementation Strategy

Although the C++ standard places few restrictions on how to implement the C++ language, by far the most common approach is that translation units are compiled down to object files and that these object files are then linked together into libraries and applications.

I expect that in the case of a translation unit containing a module partition, a C++ compiler would still produce an associated object file. The compiler would also produce a **module file** containing all the information needed to process an import directive for the corresponding module. This module file is the counterpart of header files in the nonmodule world. Section 6 delves more deeply into the details of how compilers can effectively implement a module file policy.

3.2 Transitioning a Single-Namespace Library

Many (perhaps most) modern C++ libraries are placed in their own namespace. If that is the case, transitioning such a library to a module-based distribution is fairly straightforward. A nonmodule implementation will normally consists of files somewhat like the following:

```
// Implementation File lib_1.cpp
#include <basics.h>
#include <lib.h>
namespace Lib {
  // Various declarations and definitions
}
```

This would be turned into something like:

```
// Implementation File lib_1.cpp
export namespace Lib["lib_1.cpp"] {
  import namespace basics;
public:
  import namespace Lib["lib.h"];
  // Various declarations and definitions
}
```

The declarations of the module members meant to be accessible from client code must be made **`public:`** or **`private:`** as appropriate (and the storage class specifiers **`extern`** and **`static`** can be dropped when they appear on namespace scope declarations).

In this example, **`Lib["lib.h"]`** is presumably a module partition obtained by applying a similar transformation to a header file **`lib.h`**.

4 Benefits

The capabilities implied by the basic features presented above suggest the following benefits to programmers:

- Improved (scalable) build times
- Shielding from macro interference
- Shielding from private members
- Improved initialization order guarantees
- Avoidance of undiagnosed ODR problems
- Global optimization properties (exceptions, side-effects, alias leaks, …)
- Possible dynamic library framework
- Smooth transition path from the #include world
- Halfway point to full exported template support

The following subsections discuss these in more detail.

4.1 Improved (scalable) build times

It would seem that build times on typical C++ projects are not significantly improving as hardware and compiler performance have made strides forward. To a large extent, this can be attributed to the increasing total size of header files and the increased complexity of the code it contains. (An internal project at Intel has been tracking the ratio of C++ code in ".cpp" files to the amount of code in header files: In the early nineties, header files only contained about 10% of all that project's code; now, well over half the code resides in header files.) Since header files are typically included in many other files, the growth in build cycles is generally superlinear wrt. to the total amount of source code.

Module namespaces address this issue by replacing the textual inclusion mechanism (whose processing time is proportional to the amount of code included) by a precompiled module attachment mechanism (whose processing times can be proportional to the number of imported declarations). The property that client translation units need not be recompiled if private module definitions change can be retained.

Experience with similar mechanisms in other languages suggests that modules therefore effectively solve the issue of excessive build times.

4.2 Shielding from macro interference

The possibility that macros inadvertently change the meaning of code from an unrelated module is averted. Indeed, macros cannot "reach into" a module. They only affect identifiers in the current translation unit.

This proposal may therefore obviate the need for a dedicated preprocessor facility for this specific purpose (for example as suggested in N1614 "#scope: A simple scoping mechanism for the C/C++ preprocessor" by Bjarne Stroustrup).

4.3 Shielding from private members

The fact that private members are inaccessible but not invisible regularly surprises incidental programmers. Like macros, seemingly unrelated declarations interfere with subsequent code. Unfortunately, there are good reasons for this state of affair: Without it, private out-of-class member declarations become impractical to parse in the general case.

Module namespaces appear to be an ideal boundary for making the private member fully invisible: Within the module the implementer has full control over naming conventions and can therefore easily avoid interference, while outside the module the client will never have to implement private members. (Note that this also addresses the concerns of N1602 "Class Scope Using Declarations & private Members" by Francis Glassborow; the extension proposed therein is then no longer needed.)

4.4 Improved initialization order guarantees

A long-standing practical problem in C++ is that the order of dynamic initialization of namespace scope objects is not defined across translation unit boundaries. The module partition dependency graph defines a natural partial ordering for the initialization of module variables that ensures that implementation data is ready by the time client code relies on it. I.e., the initialization run-time can ensure that the entities defined in an imported module partition are initialized before the initialization of the entities in any client module partition.

4.5 Avoidance of undiagnosed ODR problems

The one-definition rule (ODR) has a number of requirements that are difficult to diagnose because they involve declarations in different translation units. For example:

```
// File_1.cpp:
int global_cost;

// File_2.cpp:
extern unsigned global_cost;
```

Such problems are fortunately avoided with a reasonable header file discipline, but they nonetheless show up occasionally. When they do, they go undiagnosed and are typically expensive to track down.

Modules avoid the problem altogether because entities can only be declared in one module.

4.6 Global optimization properties (exceptions, side-effects, alias leaks, ...)

Certain properties of a function can be established relatively easily if these properties are known for all the functions called by the first function. For example, it is relatively easy to determine that a function will not throw an exception if it is known that the functions it calls will never throw either. Such knowledge can greatly increase the optimization opportunities available to the lower-level code generators. In a world where interfaces can only be communicated through header files containing source code, consistently applying such optimizations requires that the optimizer see all code. This leads to build times and resource requirements that are often (usually?) impractical. Historically such optimizers have also been less reliable, further decreasing the willingness of developers to take advantage of them.

Since the interface specification of a module is generated from its definition, a compiler can be free to add any interface information it can distill from the implementation. That means that various simple properties (such as a function not having side-effects or not throwing exceptions) can be affordably determined and taken advantage of.

An alternative solution is to add declaration syntax for this purpose as proposed for example in N1664 "Toward Improved Optimization Opportunities in C++0X" byWalter E. Brown and Marc F. Paterno. The advantage of this alternative is that the properties can be associated with function types and not just functions. In turn that allows indirect calls to still take advantage of the related optimizations (at a cost in type system constraints). A practical downside of this approach is that without careful cooperation from the programmer, the optimizations will not occur. In particular, it is in general quite difficult and cumbersome to manually deal with the annotations for instances of templates when these annotations may depend on the template arguments.

4.7 Possible dynamic library framework

C++ currently does not include a concept of dynamic libraries (aka. shared libraries, dynamically linked libraries (DLLs), etc.). This has led to a proliferation of vendor-specific, ad-hoc constructs to indicate that certain definitions can be dynamically linked to. N1400 "Toward standardization of dynamic libraries" by Matt Austern offers a good first overview of some of the issues in this area.

It has been suggested that the module namespace concepts may map naturally to dynamic libraries and that this may be sufficient to address the issue in the next standard. Indeed, the symbol visibility/resolution, initialization order, and general packaging aspects of modules have direct counterparts in dynamic libraries.

Section 5.6 briefly discusses the possibility of modules that may be loaded and unloaded at the program's discretion.

4.8 Smooth transition path from the #include world

As proposed, module namespaces can be introduced in a bottom-up fashion into an existing development environment. This is a consequence of nonmodule code being allowed to import modules while the reverse cannot be done.

The provision for module partitions allows for existing file organizations to be retained in most cases. Cyclic declaration dependencies between translation units are the only exception. Such cycles are fortunately uncommon and can easily be worked around by moving declarations to separate partitions.

The "global" module attribute enables a binary-compatible transition from a global namespace library to a module namespace library. This is particularly needed for some standard library facilities not declared in namespace `std`.

Finally, we note that modules are just a special kind of namespace. Moving a library from an open namespace to a module namespace does therefore not require a binary incompatible transition.

4.9 Halfway point to full exported template support

Perhaps unsurprisingly, from an implementer's perspective, templates are expected to be the most delicate language feature to integrate in the module world. However, the stricter ODR requirements in modules considerably reduce the difficulty in supporting separately compiled templates (the loose nonmodule ODR constraints turned out to be perhaps the major hurdle during the development of export templates by EDG). Furthermore, it is expected that the work to allow module templates to be exported can contribute to the implementation of export templates in open namespace scopes (as already specified in the standard).

5 Premium Options

This section explores some additional possible features for module namespaces.

5.1 Startup and Shutdown Functions

Module namespaces could be equipped with a startup and/or a shutdown function (always using the identifier `main`).

```
// File_1.cpp:
export namespace Lib {
  import namespace std;
  void main() { std::cout << "Hello "; }
  void ~main() { std::cout << "World\n"; }
}

// File_2.cpp:
import namespace Lib;
int main() {}
```

This program outputs “Hello World”. Clearly, this is mostly syntactic convenience since the same could be achieved through a global variable of a special-purpose class type with a default constructor and destructor as follows:

```
export namespace Lib {
  import namespace std;
  struct Init {
    Init() { std::cout << “Hello “; }
    ~Init() { std::cout << "World\n"; }
  } init;
}
```

5.2 Program Modules

A module could be designated as a program entry point by making it a **program module**:

```
export [["program"]] namespace P {
  import namespace std;
  void main() {
    std::cout << “Hello World\n”;
    std::exit(1);
  }
}
```

The square bracket construct preceding the first namespace keyword in the preceding example is a **namespace attribute list**. Additional attributes are discussed in the remainder of this paper.

Note that this proposal does not provide an option to pass command-line arguments through a parameter list of `main()`, nor does it allow for `main()` to return an integer. Instead, it is assumed that the standard library will be equipped with a facility to access command-line argument information (the function `std::exit()` already returns an integer to the execution environment).

The ability to write a program entirely in terms of module namespaces may be desirable, not only out of a concern for elegance, but also to clarify initialization order and to enable a new class of tools (which would not have to worry about ODR violations).

It is also possible to designate a module (and function) as an entry point through implementation-defined means (e.g., a compilation option), although that is probably of limited use.

5.3 Nested Modules

Nested modules could be allowed. They'd have to be declared (but not defined) in their enclosing module:

```
export namespace Lib["part 1"] {
public:
  export namespace Lib::Nest; // Nested module
                              // declaration.
}
export namespace Lib::Nested { // Only valid if
                               // declared in Lib.
  import namespace Lib;  // Okay.
}
```

The example above shows that a nested module can import its enclosing module. The converse is true too:

```
export namespace Lib["part 2"] {
  import namespace Lib::Nested;  // Okay.
}
```

However, a single partition cannot both declare and import a nested module since that would amount to a cyclic dependency in that partition's dependency graph:

```
export namespace Lib["part 3"] {
  export namespace Lib::Bad;
  import namespace Lib::Bad;  // Error.
}
```

Nested open namespaces that are not so constrained, of course:

```
export namespace Outer {
public:
  namespace Inner {
    typedef char C;
  }
  Inner::C flag;
}
```

Namespace scope variables and static data members appearing inside modules (e.g., "flag" in this example) are called **module variables**. The example above makes both the nested open namespace `Inner` and its member type `C` visible to code that imports the module namespace `Outer`.

5.4 Prohibited (Precluded?) Members

The fact that private namespace members become invisible when imported from a module may change the overload set obtained in such cases when compared with the pre-module situation. Consider the following example:

```
namespace N {
  struct S {
    void f(double);
  private:
    void f(int);
  };
}
void g(N::S &s) {
  s.f(0);  // Access error
}
```

The overload set for the call **`s.f(0)`** contains two candidates, but the private member is preferred. An access error ensues.

If namespace **`N`** is transitioned to a module, the code might become:

```
import namespace N;
void g(N::S &s) {
  s.f(0);  // Selects S::f(double)
}
```

In the transformed code, the overload set for **`s.f(0)`** only contains the public member **`S::f,`** which is therefore selected. In this case, the programmer of **`S`** may have opted to deliberately introduce the private member to diagnose unintended uses at compile time.

There exist alternative techniques to achieve a similar effect without relying on private members, but none are as direct and effective as the approach shown above. It may therefore be desirable to introduce a new access specifier **`prohibited`** to indicate that a member cannot be called; this property is to considered part of the public interface and therefore not made invisible by a module boundary. The example above would thus be rewritten as follows:

```
export namespace N {
  struct S {
    void f(double);
  prohibited:
    void f(int);  // Visible but not callable
  };
}
```

Note that this parallels the "**`not default`**" functionality proposed in N1445 "Class defaults" by Francis Glassborow. The access label "**`prohibited:`**" could conceivably also be extended to namespace scope module members. For example:

```
export namespace P {
public:
  void f(double) { ... }
prohibited:
  void f(int);
}
```

5.5 Mixed Namespaces

Transitioning large library namespaces to a module-based implementation can represent a fair amount of work. To allow for a more gradual transition process, it is possible to allow namespaces to be partially implemented in modules and partially in open namespaces. For example:

```
// File n_basics.cpp:
export namespace N["basics.cpp"] {
  // Various declarations and definitions...
}


// File n.h:
import namespace N["basics.cpp"];
namespace N {  // Open namespace extension
  // Additional declarations...
}


// File n_remainder.cpp:
#include "n.h"
namespace N { // Open namespace extension
  // Additional declarations and definitions...
}
```

Clearly, the benefits of modules are not maximally exploited by such a partial transition, but it is technically very feasible and may be on the most practical transition path for various large projects.

5.6 Program-directed module loading

If modules can be compiled to dynamic libraries, it is natural to ask whether they could be loaded and unloaded under program control (as can be done with nonstandard APIs today).

Loading probably presents few problems other than agreeing on a standard library API to do so. Unloading presumably requires slightly different termination semantics: All the static lifetime variables must be destroyed at that point (instead of in strict reverse construction order). If this is desirable, the alternative termination semantics could be indicated with a namespace attribute. For example:

```
export [["dynamic"]] namespace Component {
  // ...
}
```

It may also be desirable to have such modules generate additional RTTI information that could be used to synthesize safe calls to components not originally linked into the application (as an alternative to the more traditional pointer-casting approach).

5.7 Self-importing module

Occasionally it is useful to have initializers from a certain translation unit run without any function in that translation unit explicitly being called from another translation unit. This would require an implementation-specific mechanism to indicate that that translation unit is part of the program, but even so the current standard does not guarantee that the initializers for namespace scope objects will execute.

Perhaps another namespace attribute could be used to indicate that if a module were somehow deemed part of the program, its initialization would occur before the initialization of the program module. For example:

```
export [["selfregister"]] namespace SpecialAlloc {
  // ...
}
```

5.8 Standard module file format

Probably the major drawback of modules compared to header files is that the interface description of a library may end up being obfuscated in a proprietary module file format. This is particularly concerning for third-party tool vendors who until now could assume plaintext header files.

It is therefore desirable that the module file format be partially standardized, so that third party tools can portably load public declarations (at the very least). This can be done in a manner that would still allow plenty of flexibility for proprietary information. For example, vendors could agree to store two offsets indicating a section of the file in which the potentially visible declarations appear using plain C++ syntax, extended with a concise notation for the invisible private sections that may affect visible properties. E.g.:

```
/* Conventional plain-source module description. */
export namespace Simple {
public:
  struct S {
    private[offset 0, size 4, alignment 4];
    public:
    // ...
  };
}
```

A native compiler would presumably ignore this textual form in favor of the more complete and more efficient proprietary representation, but tool vendors would have access to sufficient information to perform their function (and the use of quasi-C++ syntax offers an affordable transition path for existing tools that already parse standard C++).

6 Technical Notes

This section collects some thoughts about specific constraints and semantics, as well as practical implementation considerations.

6.1 The module file

A module is expected to map on one or several persistent files describing its public declarations. This module file (we will use the singular form in what follows, but it is understood that a multi-file approach may have its own benefits) will also contain any public definitions except for definitions of noninline functions, namespace scope variables, and nontemplate static data members, which can all be compiled to a separate object file just as they are in current implementations.

Some private entities may still need to be stored in the module file because they are (directly or indirectly) referred to by public declarations, inline function definitions, or private member declarations.

Not every modification of the source code defining a module namespace needs to result in updating the associated module file. Avoiding superfluous compilations due to unnecessary module file updates is relatively straightforward. One algorithm is as follows: A module file is initially produced in a temporary location and is subsequently compared to any existing file for the same module; if the two files are equivalent, the newer one can be discarded.

As mentioned before, an implementation may store interface information that is not explicit in the source. For example, it may determine that a function won't throw any exceptions, that it won't read or write persistent state, or that it won't leak the address of its parameters.

In its current form, the syntax does not allow for the explicit naming of the module file: It is assumed that the implementation will use a simple convention to map module namespace names onto file names (e.g., module name "Lib::Core" may map onto "Lib.Core.mf"). This may be complicated somewhat by file system limitations on name length or case sensitivity.

6.2 Module dependencies

When module A imports module B (or a partition thereof) it is expected that A's module file will not contain a copy of the contents of B's module file. Instead it will include a reference to B's module file. When a module is imported, a compiler first retrieves the list of modules it depends on from the module file and loads any that have not been imported yet. When this process is complete, symbolic names can be resolved much the same way linkers currently tackle the issue. Such a two-stage approach allows for cycles in the module dependency graph.

> **The dependencies among partitions within a module must form a directed acyclic graph.**

When a partition is modified, sections of the module file on which it depends need not be updated. Similarly, sections of partitions that do not depend on the modified partition do not need to be updated. Initialization order among partitions is only defined up to the partial ordering of the partitions.

6.3 Nested namespaces

The concept of open namespaces nested in a module namespace presents few problems: It is treated as other declarative entities. If such an open namespace is anonymous, it cannot be public.

> **Private namespaces cannot contain public members.**

This rule could be relaxed since it is in fact possible to access members of a namespace without naming the namespace, either through argument-dependent lookup, or through a public namespace alias.

Nested module namespaces must be declared in their enclosing module on the grounds that all namespace members should be declared in that namespace. However, that creates an implicit dependence of the nested module on its enclosing module, which in turn creates a dependency cycle if the enclosing module imports the nested module. Our way out of this conundrum is to specify the dependency constraints in terms of module partitions and not in terms of modules.

6.4 Startup and termination

Assuming program modules and module startup/termination functions are part of the feature set, the following assertion is useful:

> **A program can contain at most one program module. If it does contain such a module it cannot declare `::main()` and the program's execution amounts to the initialization of the program module's module variables.**

We'll cast the execution of a module `main()` function (a "premium features") in terms of variable initializations.

> **The module function `main()` is executed as if it were the default constructor of a module variable defined in a synthesized partition dependent on all other partitions. Similarly, the module function `~main()` is executed as if it were the destructor that same module variable.**

This fits the notion that `main()` and `~main()` are essentially a syntactic convenience that could be replaced by special-purpose singleton class types. (The notion of a synthesized dependent module partition is just to ensure that `main()` runs after all the module variables have been initialized.) Like `::main()` these functions are subject to some restrictions (see also [basic.start.main] §3.6.1):

> **The module functions `main()` and `~main()` cannot be called explicitly. Their address cannot be taken and they cannot be bound to a reference. They cannot be exported and they cannot be declared without being defined.**

A fairly natural initialization order can be achieved within modules and module partitions.

> **Within a module partition the module variables are initialized in the order currently specified for a translation unit (see [basic.start.init] §3.6.2). The initialization of module variables in one module partition must complete before the initializations of module variables in another partition that has a dependency on the first partition. The module variables and local static variables of a *program* are destroyed in reverse order of initialization (see [basic.start.term] §3.6.3).**

As with the current translation unit rules, it is the point of definition and not the point of declaration that determines initialization order.

The initialization order between module partitions is determined as follows:

> **Every import directive defines anonymous namespace scope variables associated with each module partition being imported. Theses variables require dynamic initialization. The first of such variables associated with a partition to be initialized triggers by its initialization the initialization of the associated partition; the initialization of the other variables associated with the same partition is without effect.**

This essentially means that the initialization of a module partition must be guarded by Boolean flags much like the dynamic initialization of local static variables. Also like those local static variables, the Boolean flags will likely need to be protected by the compiler if concurrency is a possibility (e.g., thread-based programming).

6.5 Linkage

> **Namespace scope declarations cannot be declared `extern` or `static` in modules. The `extern` keyword can only be used for linkage specifications (see [dcl.link] §7.5) in module definitions.**

Module namespaces and the import/export mechanisms make the storage specifiers **`extern`** and **`static`** mostly redundant in namespace scopes. The only case that is not trivially covered appears to be the forward declaration of module variables. Consider the following non-module example:

```
void f(int*);

extern int i;  // Forward declaration.

int p = &i;

int i = f(p);
```

It may be desirable to allow such constructs in modules, but the keyword **`extern`** does not convey the right semantics. Instead, forward declarations could be indicated using a trailing ellipsis token:

```
export namespace Lib {
  int f(int*) ...;  // Ellipsis optional.
  int i ...;         // Forward declaration.
  int p = &i;
  int i = f(p);      // Definition.
}
```

The keyword `static` can still be used in class scopes and local scopes (and the semantics are similar in both cases).

In modules, names have external linkage if and only if they are public.

6.6 Exporting incomplete types

It is somewhat common practice to declare a class type in a header file without defining that type. The definition is then considered an implementation detail. To preserve this ability in the module world, the following rule is stated:

An imported class type is incomplete unless its definition was public or a public declaration requires the type to be complete.

For example:

```
// File_1.cpp:
export namespace Lib {
public:
  struct S {};  // Export complete type.
  class C;      // Export incomplete type only.
private:
  class C { ... }
}

// File_2.cpp:
import namespace Lib;
int main() {
  sizeof(lib::S); // Okay.
  sizeof(Lib::C); // Error: Incomplete type.
}
```

The following example illustrates how even when the type is not public, it may need to be considered complete in client code:

```
// File_1.cpp:
export namespace X {
  struct S {};  // Private by default.
public:
  S f() { return S(); }
}

// File_2.cpp:
import namespace X;
```

```
int main() {
  sizeof(X::f());  // Allowed.
}
```

6.7 Explicit template specializations

Explicit template specializations and partial template specializations are slightly strange in that they may be module namespace members not packaged in their own module:

```
export namespace Lib {
public:
  template<typename T> struct S { ... };
}

export namespace Client {
  import namespace Lib;
  template<> struct Lib::S<int>;
}
```

There are however no known major technical problems with this situation.

It has been suggested that modules might allow "private specialization" of templates. In the example above this might mean that module `Client` will use the specialization of `Lib::S<int>` it contains, while other modules might use an automatically instantiated version of `Lib::S<int>` or perhaps another explicit specialization. The consequences of such a possibility have not been considered in depth at this point. (For example, can such a private specialization be an argument to an exported specialization?) Private specializations are not currently part of the proposal.

6.8 Automatic template instantiations

The instantiations of noninline function templates and static data members of class templates can be handled as they are today using any of the common instantiation strategies (greedy, queried, or iterated). Such instantiations do not go into the module file (they may go into an associated object file).

However instances of class templates present a difficulty. Consider the following small multimodule example:

```
// File_1.cpp:
export namespace Lib {
public:
  template<typename T> struct S {
    static bool flag;
  };
  ...
}

// File_2.cpp:
```

```
export namespace Set {
  import namespace Lib;
public:
  void set(bool = Lib::S<void>::flag);
  // ...
}

// File_3.cpp:
export namespace Reset {
  import namespace Lib;
public:
  void reset(bool = Lib::S<void>::flag);
  // ...
}

// File_4.cpp:
export namespace App {
  import namespace Set;
  import namespace Reset;
  // ...
}
```

Both modules **`Set`** and **`Reset`** must instantiate **`Lib::S<void>`**, and in fact both expose this instantiation in their module file. However, storing a copy of **`Lib::S<void>`** in both module files can create complications related to the problems caused by the loose ODR rules in the context of open namespace export templates.

Specifically, in module **`App`**, which of those two instantiations should be imported? In theory, the two are equivalent (unlike the header file world, there can ultimately be only one source of the constituent components), but an implementation cannot ignore the possibility that some user error caused the two to be different. Ideally, such discrepancies ought to be diagnosed (although current implementation often do not diagnose similar problem in the header file world).

There are several technical solutions to this problem. One possibility is to have reference to instantiated types outside a template's module be stored in symbolic form in the client module: An implementation could then temporarily reconstruct the instantiations every time they're needed. Alternatively, references could be rebound to a single randomly chosen instance (this is similar to the COMDAT section approach used in many implementations of the greedy instantiation strategy). Yet another alternative, might involve keeping a pseudo-module of instantiations associated with every module containing public templates (that could resemble queried instantiation).

6.9 Friend declarations

Friend declarations present an interesting challenge to the module implementation when the nominated friend is not guaranteed to be an entity of the same module. Consider the following example illustrating three distinct situations:

```
export namespace Example {
  import namespace Friends;
  void p() { /* ... */ };
public:
  template<typename T> class C {
    friend void p();
    friend Friends::F;
    friend T;
    // ...
  };
}
```

The first friend declaration is the most common kind: Friendship is granted to another member of the module. This scenario presents no special problems: Within the module private members are always visible.

The second friend declaration is expected to be uncommon, but must probably be allowed nonetheless. Although private members of a class are normally not visible outside the module in which they are declared, an exception must be made to out-of-module friends. This implies that an implementation must fully export the symbolic information of private members of a class containing friend declarations nominating nonlocal entities. On the importing side, the implementation must then make this symbolic information visible to the friend entities, but not elsewhere. The third declaration is similar to the second one in that the friend entity isn't known until instantiation time and at that time it may turn out to be a member of another module.

For the sake of completeness, the following example is included:

```
export namespace Example2 {
public:
  template<typename T> struct S {
    void f() {}
  };
  class C {
    friend void S<int>::f();
  };
}
```

The possibility of **S<int>** being specialized in another module means that the friend declaration in this latter example also requires the special treatment discussed previously.

6.10 Base classes

Private members can be made entirely harmless by deeming them "invisible" outside their enclosing module. Base classes, on the other hand, are not typically accessed through name lookup, but through type conversion. Nonetheless, it is desirable to make private base classes truly private outside their module. Consider the following example:

```
export namespace Lib {
public:
  struct B {};
  struct D: private B {
    operator B&() { static B b; return b; }
  };
}
export namespace Client {
  import namespace Lib;
  void f() {
    B b;
    D d;
    b = d;  // Should invoke user-defined conversion.
  }
}
```

If **B** were known to be a base class of **D** in the **Client** module (i.e., considered for derived-to-base conversions), then the assignment **b = d** would fail because the (inaccessible) derived-to-base conversion is preferred over the user-defined conversion operator.

> **Outside the module containing a derived class, its private base classes are not considered for derived-to-base or base-to-derived conversions.**

Although idioms taking advantage of the different outcomes of this issue are uncommon, it seems preferable to also do "the right thing" in this case.

6.11 Syntax considerations

The following notes summarize some of the alternatives and conclusions considered for module-related syntax.

6.11.1 Avoiding new keywords

Previous incarnations of this proposal presented modules using syntax that does not require any new keywords. That original syntax uses the tokens "<<" and ">>" in a way that is meant to be reminiscent of the streaming concept (somewhat paralleling the preprocessor file inclusion mechanism in the sense that "a file is being read to import interfaces"). A short example illustrates the syntax:

```
namespace >> App {   // Module definition
  namespace << Lib;  // Import directive
  ...
}
```

In addition to avoiding new keywords, this notation also has the benefit of being slightly more compact than the syntax used throughout this paper. However, many early reviewers of this proposal also reported that deciding which way the "streaming symbol" should point is non-intuitive. Although most seem to agree that it is not likely to remain a problem after using the feature for a while, it is likely to be a (minor) teaching hurdle.

The introduction of the keyword **`import`** could also be avoided by just dropping that keyword in the proposed syntax. For example:

```
export namespace App {  // Module definition
  namespace Lib;        // Import directive
  ...
}
```

This currently presents no technical difficulties but this import directive syntax is unsatisfactory in two major ways. First, from a programmers' point of view it seems asymmetric and it is inconvenient to search for (both when eyeballing source code and when using simple text searching tools). Second, it is the syntax one would expect for a "forward namespace declaration". There is currently no such concept in C++, but it's probably unwise to use such syntax for an alternative purpose.

Other ways to avoid the introduction of a new keyword for import directives are possible, but don't seem to fit in the overall "C++ syntactic look". Examples include:

```
::namespace Lib;
+ namespace Lib;
* namespace Lib;
```

and worse. Perhaps more likely is

```
public namespace Lib;
```

although it has the downside of suggesting there might be such a thing as "**`private namespace Lib;`**". Both with this last alternative, and with the syntax used for the main examples in this paper, the keyword **`namespace`** is unnecessary for parsing. However, it reads rather well, and it leaves open more alternative uses of the keyword **`import`** (or **`public`**) should.

6.11.2 Is a keyword `import` viable?

The word "import" is fairly common, and hence the notion of making it a new keyword gives one pause. The introduction of the keyword **`export`** might however have been the true bullet that needed to be bitten: The two words usually go hand in hand, and reserving one makes alternative uses of the other far less likely. Various Google searches of "import" combined with other search terms likely to produce C or C++ code (like "#define", "extern", etc.) did not find use of "import" as an identifier. Of note however, are preprocessor extensions spelled "**`#import`**" both in Microsoft C++ and in Objective-C++, but neither of those uses conflict with **`import`** being a keyword.

Overall, a new keyword **`import`** appears to be a viable choice.

6.11.3 Module definition syntax

The syntax "**`export namespace`** ***`module-name`*** **`{ ... }`**" is meaningful in at least two ways:

- it indicates that a namespace of the given name is being defined, and
- it indicates that the contents of that namespace are in some way made available elsewhere (perhaps more actively so than what e.g. would be implied by the keyword **`extern`**).

However, it doesn't quite convey the "packaging" function of modules. (The same can be said of the earlier proposed syntax "**`namespace >> Lib { ... }`**". A straightforward alternative might be

```
module Lib {
  ...
}
```

Unfortunately, "module" appears to be somewhat in common use as an identifier in C code (specifically, in Linux kernel modules), and it is therefore expected to also appear in a fair number of C++ source code files.

6.11.4 Public module members

Earlier revisions on this paper made all module declarations "private" by default, and required the use of the keyword **`export`** on those declarations meant to be visible to client code. Advantages of that choice include:

- it make explicit (both in source and in thought) which entities are exported, and which are not, and
- the existing meaning of export (for templates) matches the general meaning of this syntactical use.

There are also some disadvantages:

- it disables the "**`export namespace`**" option for nested module declarations, and severely degrades the viability of "**`export namespace`**" for module definitions in general, and

- the requirement to repeat **`export`** on every public declaration can be unwieldy.

Peter Dimov's observation that the use of "public:" and "private:" for namespace scope declarations (as proposed in this revision of the paper) is consistent with the rules for visibility of public/private class members across module boundaries clinched the case to propose that particular alternative for this aspect of the syntax.

Other alternatives have been considered, but do not seem as effective as the ones discussed.

6.11.5 Namespace attributes

There are at least three different aspects of the namespace attribute syntax that can reasonably be varied:

- The relative position of the attributes
- The introduction/delimitation of the attribute list
- The form of each individual attribute

With regards to the relative position of the attributes, an interesting alternative is to place the attributes just after the keyword **`namespace`** (as was the case in earlier revisions of the proposal). In at least some implementations this slightly simplifies parsing and diagnosis. However, the advantage of the current proposal is that it more easily generalizes to other constructs that may not have a keyword to attach the attributes to. The use of doubled square brackets similarly is more general than e.g. single square brackets, although with the currently proposed uses single brackets would be sufficient.

Perhaps the most common suggestion for namespace attributes is that of not requiring quotation marks (i.e., use identifiers instead of string literals). This author's preference for the string literal option is motivated by three minor considerations:

- Every first use of a non-keyword identifier in C++ is currently a declaration; If attributes were identifiers, they would form an exception to that observation.
- String literals allow for a wider variety of attribute spellings including the use of dashes and spaces. This may be desirable for future standardization or for proprietary extensions.
- String literals are expressions. Should attributes be extended to allow for more general expressions in the future, the extension would be more general.

At this point in time, however, none of these arguments seem compelling.

6.11.6 Partition names

As with namespace attributes, the most commonly suggested alternative for partition names is not to require quotation marks. In this case however, there is a rather compelling argument (in this author's opinion) in favor of requiring the quotation marks: In the module world namespace partitions are the logical counterpart of "source files" and hence it will likely prove convenient and natural to name partitions after the file in which they are defined. The use of quotation marks is reminiscent of the #include syntax and allows for most names permitted as file names by modern operating systems.

6.12 Known Technical Issues

6.12.1 Declarations outside module definitions

The current model for C++ modules assumes a bottom-up transition from a header-based program to a module-based program. Making this a strict requirement, however, is unlikely to be realisting. In particular, C-based libraries might never be able to make a transition to modules. It is therefore likely that code structured like the following example must be accepted:

```
#include "unistd.h"
export namespace Lib {
  // Make use of "unistd.h"
}
```

The problem with code like that is that a module may end up referring to entities that are not defined in just one place (e.g., a type in "**`unistd.h`**" which may be defined elsewhere too).

6.12.2 Open module file format

From a programmer's perspective, a module-based development model presents the danger of hiding interface specifications in a proprietary module file format. (Header files, while hard to parse, present no such problem.) Such a situation would discourage the development of third-party software analysis tools (e.g., Lint). It is therefore highly desirable that some kind of implementation-independent interface description be part of module file formats.

As mentioned earlier, this could take the form of a section of the module file representing a module's interfaces as quasi-C++ code (very similar to what would be found in a header file today). This section should be easy to locate (e.g., through a file offset stored at the beginning of the file). The C++ standard is unlikely to be the right place to specify such a (partial) standard format. A technical report might be a more appropriate vehicle to do so, or it could be left to implementers and tool vendors themselves (which might treat is like an aspect of the ABI).

7 Rejected Features

A few features were considered at one point, but rejected as unlikely to be desirable. This section collects those ideas.

7.1 Module seals

With the rules so far, third parties may in principle add partitions to existing multi-partition modules. This may be deemed undesirable.

One way to address this is to assume implementation-specific mechanisms (e.g., compiler options) will allow modules to be "sealed" in some fashion.

Alternatively, a language-based sealing mechanism could be devised. A possibility is a namespace attribute to indicate that a given partition and all the partitions it imports (directly or indirectly) from the same module form a complete module. For example:

```
// File_1.cpp:
export namespace Lib["core"] {
  // ...
}

// File_2.cpp:
export namespace Lib["utils"] {
  import namespace Lib["core"];
  // ...
}

// File_3.cpp:
export [["complete"]] namespace Lib["main"] {
  import namespace Lib["utils"];
  // Partitions "main", "utils", and "core"
  // form the complete module Lib.
}

// File_4.cpp:
export namespace Lib["helpers"] {
  import namespace Lib["core"];
  // Error: "helpers" not imported into sealing
  // partition "main".
}
```

A somewhat more explicit alternative is to require a sealing directive listing all the partitions in one (any) of these partitions. The third file in the example above might for example be rewritten as:

```
// File_3.cpp:
export namespace Lib["main"] {
  import namespace Lib["utils"];
  namespace = ["main", "utils", "core"];
  // Partitions "main", "utils", and "core"
  // form the complete module Lib.
}
```

7.2 More than one partition per translation unit

It may be possible to specify that multiple modules or partitions be allowed in a single translation unit. For example:

```
// File_1.cpp:
export namespace M1 {
  // ...
}
export namespace M2 {
  // ...
}
```

However doing so may require extra specification to define visibility rules between such modules and is also likely to be an extra burden for many existing implementations.

7.3 Auto-loading

It is possible to automatically import a module when its first use is encountered, without requiring an explicit import directive. This would for example simplify Hello World to the following:

```
int main() {
  std::cout << "Hello World!\n";
}
```

Opinions on whether this simplifies introductory teaching appear to vary, but there is a general agreement that it could be harmful to code quality in practice. It also has slightly subtle implications for initialization order (since a module's import directives determine when it may be initialized).

7.4 Exported macros

It may be possible to export macro definitions. However, this forces a C++ compiler to integrate its preprocessor and it raises various subtleties wrt. dependent macros. For example:

```
export namespace Macros {
#define P A   // Invisible?
public:
#define X P   // Expansion of X will not pick up P?
}
```

Exported macros are therefore probably undesirable. If needed, an import directive can be wrapped in a header to package macros with modules.

8 Acknowledgments

Important refinements of the semantics of modules and improvements to the presentation in this paper were inspired by David Abrahams, Pete Becker, Mike Capp, Christophe De Dinechin, Peter Dimov, Thorsten Ottosen, Jeremy Siek, John Spicer, Bjarne Stroustrup, and John Torjo.