Daveed Vandevoorde
daveed@vandevoorde.com

Modules in C++

(Revision 6)

1   Introduction

Modules are a mechanism to package libraries and encapsulate their implementations. They differ from the traditional 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 (somewhat similar to that of Modula-2) 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 (initialization ordering, run-time performance, etc.).

1.1   Document Overview

Section 2 first presents how modules might affect a typical command-line-driven user interface to a C++ compiler. The goal of that section is to convey how modules may or may not disrupt existing practice with regard to build systems. Section 3 then briefly introduces (mostly by example) the various language elements supporting modules. The major expected benefits are described in some detail in Section 4. Finally, Section 5 covers rather extensive technical notes, including syntactic considerations. We conclude with acknowledgments.

1.2   Changes Since Previous Version

The paper has been revised to propose standardizing *interface files* in terms of C++ source code (side-stepping the thorny issue of agreeing on an efficient representation): See subsections 2.2 and 5.1. Also, the proposed access specifier "**prohibited:**" has been dropped (since the equivalent "**= delete**" feature is now part of the language).

2   Intended User Interface

Although the C++ standard doesn't mandate or even recommend any specific user model, successful implementations of C++ share similar user interface elements. For example, they tend to rely on compiling a translation unit at a time, on name mangling, on plain-text source code, on time-stamp-based dependency checking, etc.

The introduction of modules is not expected to change this: The revised standard will continue to just describe the semantics of the program, and an implementation will continue to be free to achieve those semantics in any way. However, it is still expected that mainstream implementations will agree on the general mechanisms involved, and the intent is not to overly disrupt current software building strategies.

2.1 Interface Files vs. Header Files

The main expected change when transitioning from traditional C++ libraries to module-based libraries is that compiler-generated *interface files* will largely replace user-written header files. To make this concrete, consider a simple application consisting of one implementation file (**main.cpp**) that uses one simple library itself consisting of one implementation file (**lib.cpp**) and one header file (**lib.h**). The file **lib.h** describes the interfaces offered by the library and—as is typical—is included in both **main.cpp** and **lib.cpp**. (Let's assume for this particular case that the library does not provide any macros to client code.) The library writer might build his library with a compiler invocation like

```
$ CC −c −O lib.cpp
```

which will produce an object code file (**lib.o**, say). That object code file (in some form) is provided to the application writer along with the header file, who can then build his application using

```
$ CC −O main.cpp lib.o
```

The intent of this proposal is that the two command lines above continue to work when the code transitions to modules, but the mechanisms underneath differ. First, the header file can be dropped and its declarations moved to **lib.cpp**. Second, when **lib.cpp** is compiled, a second file is generated in addition to the object code file: A interface file (**lib.mpp**, say) that describes the public interfaces of the library[1]. Third, when **main.cpp** is compiled, the **#include** directive can be replaced by an *import directive* (which is now a core language construct instead of a preprocessor construct). The compiler will read (parts of) **lib.mpp** when it parses the import directive, and will retrieve additional information from **lib.mpp** as it encounters the need to know about the various aspects of the library (or "module") it describes.

As with header files, compilers will likely have a mechanism to describe where various interface files might be located. So if the interface file **lib.mpp** was moved to a nonstandard location, the compiler invocation might look like:

```
$ CC −M /nonstd/loc −O main.cpp lib.o
```

---

[1] The –O option in the examples is a request for optimization. It could conceivably affect the interface file by promoting to the public interface special knowledge—such as the fact that no exceptions are thrown— about the implementation.

Unlike with header files, however, a compiler might also offer an option to locate where a interface file should be written when a module is compiled. So the command to build the library could conceivably be written as follows:

```
$ CC —X /nonstd/loc —c —O lib.cpp
```

2.2   What's A Interface File Like?

Since it is intended that interface files are compiler-generated, it is tempting to allow them to be structured for maximum efficiency and therefore not human-readable. However, "maximum efficiency" means different things to different implementations, which makes standardization of a module file format difficult at the least. On the other hand, without standardization third-party tool vendors will find it impractical support modules unless their complete implementation is available in source form (often not a commercially acceptable option).

Instead, this paper proposes to standardize interface files as files containing something recognizable as C++ source code with certain restrictions and extensions. When a compiler encounters the use of such an interface file (through a module import directive; see further), it is expected to do the following:

1. Look up the file in a cache of pre-compiled interface files

2. If no match is found is found or if the cache is out of date, generate an efficient pre-compiled interface file (PIF).

3. Load information from the PIF "on demand".

The PIF is thus the maximally-efficient-but-not-human-readable representation of a module's interface. It can be optimized for efficient reading when compiling client code. In particular, it is expected that a compiler will only read the elements of a PIF that are actually needed by client code. For example, if a library offers ten independent class definitions with 5 inline member function definitions in each, and client code only uses two of those classes and six of those inline member functions, then a compiler would only load the initial "table of contents" for the module, the two class descriptions in that PIF (which include their own "table of contents", and the data defining the six inline members. Furthermore, tokenization, preprocessing, name lookup, overload resolution, and many other tasks a compiler must perform when reading a header file need not be performed when reading a PIF (it was done once when the module implementation was compiled or when the PIF was precompiled).

The scheme proposed here is similar in some ways to that of the "precompiled headers" feature available with many compilers. The important difference between a PIF and a precompiled header, however, is that the former is only dependent on the interface offered by a module, whereas the latter is specific for a particular use of header files: As a result PIFs are consistently usable for every use of the corresponding module, whereas precompiled header files are not.

2.3 Header Files May Stay Around

Although modules are meant to replace header files for interfaces expressed using the core language, they are not meant to replace header files for macro interfaces. In a C++-with-modules world, header files will therefore remain desirable. In our earlier example we may therefore re-introduce a header file **lib.h** with contents somewhat as follows:

```
#ifndef LIB_H
#define LIB_H
import Lib;  // Import the module (no guard needed)
#define LIB_MACRO(X, Y) ...
...
#endif /* ifndef LIB_H */
```

Even if a library does not provide macros it may still provide a header file (as outlined above but without the macro definitions) to maintain source-level compatibility with prior non-module-based versions of that library. (Note that import directives don't need include guards: A duplicated import is essentially just ignored.) I.e., client code that imports the library's interfaces with

```
#include "lib.h"
```

will continue to work and need not be aware of the fact that the header file is little more than a wrapper around a module import directive.

2.4 Writing Against Unimplemented Interfaces

It's not uncommon for the development of client code to start before a library has been fully implemented. The header file is written first and contains the anticipated interfaces. Client code can be compiled against that while the implementation of the library proceeds concurrently. The client code cannot be linked until enough of the library's implementation is written, but the scenario does enable compression of development schedules in practice. (Note that the C++ standard doesn't mandate that this be possible, but the compiler+linker implementation strategy nicely supports it.)

This concurrent development approach is also intended to be available in the modules world (though again outside the standard wording). This is achieved by compiling incompletely implemented modules. For example, a very simple incomplete module may look as follows:

```
export Lib:
public:
   void f();  // Not yet implemented
```

This can be compiled the usual way:

```
$ CC —c lib.cpp
```

The object file produced (if any) is not useful, but the interface file can be given to client code developers to start coding against it:

```
import Lib;
int main() {
  f();  // Will compile but not link yet.
}
```

2.5 Dependency Management

Or: How will tools like "make" work in the world of modules?

Tools like "make" typically examine the "last time modified" time-stamp of various files to decide whether a file (traditionally, an object code file or an executable file) needs to be re-built. In the header-based world, the rule for rebuilding an object file typically depends on the implementation (.cpp) file it is built from, plus any header files directly or indirectly included by that implementation file.

In the proposed module world, object files will need to depend on interface files imported by the associated implementation (.cpp) files. Specifying the imported interface files directly in the dependency descriptions could achieve this. Alternatively (for code that is transitioning from the header-based model), for modules with an associated header file as described above (i.e., one that mostly just contains a module import directive), a rule could be added to update the header file time stamp when the interface file itself is updated.

Since interface files are generated, they themselves depend on other files: the source files implementing the module (typically .cpp files, although header files are possible too) and perhaps other interface files it depends on.

As with header files today, it is relatively simple for a compiler to generate a dependency description that includes modules.

3 Module Features By Example

3.1 Import Directives

The following example shows a simple use of a **module**. In this case, the module encapsulates the standard library.

```
import 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 collection of declarations available in the translation unit. In contrast to #include preprocessor directives, module import directives are insensitive to macro expansion (except with regard to the identifiers appearing in the directive itself, of course).

The name space of modules is distinct from all other name spaces in the language. It is therefore possible to have a module and e.g. a C++ namespace share the same name. That

is assumed in the example above (where the module name **std** is identical to the main namespace being imported). It is also expected that this practice will be common in module-based libraries (but it is not a requirement; in fact a module may well contain bits of multiple C++ namespaces). So in **std::cout** the **std** does not refer to the module name **std**, but to a namespace name **std** that happens to be packaged in the **std** module.

3.2 Module Definitions

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

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

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

A module definition header must precede all declarations in a translation unit: It indicates that some of the declarations that follow may be made available for importing in other translation units.

Import directives only make visible those members of a module 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 that appear in module definitions. By default, namespace scope declarations in a module are private.

Note that the constructor of **S** is an inline function. Although its definition is separate (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).

Variables with static storage duration defined in modules are called **module variables**. Because modules[2] have a well-defined dependency relationship, it is possible to define a reliable run-time initialization order for module variables.

---

[2] This is slightly inaccurate: It is *module partitions* (subsection 3.5) that have the well-defined dependency relationship. Nonetheless, the conclusion holds.

3.3 Transitive Import Directives

Importing a module is transitive only for public import directives:

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


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


// File_3.cpp:
export MM:
public:
  import M1;  // Make exported names from M1 visible
              // here and in client code.
private:
  import M2;  // Make M2 visible here, but not in
              // client code.


// File_4.cpp:
import MM;
I1 i1;  // Okay.
I2 i2;  // Error: Declarations from M2 are invisible.
```

3.4 Private Class Members

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

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


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

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

```
export Err:
public:
  struct S { int f() {} };  // Public f.
  class 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 still exist and may in fact be known to the compiler when it imports a module. The following example illustrates a key aspect of this observation:

```
// Library file:
export Singleton:
public:
  struct Factory {
    // ...
  private:
    Factory(Factory const&);  // Disable copying.
  };
  Factory factory;


// Client file:
import Singleton;
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 is 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.

3.5 Module Partitions

A module may span multiple translation units: Each translation unit defines a **module partition**. For example:

```
// File_1.cpp:
export Lib.p1:
  struct Helper {  // Not exported.
    // ...
  };

// File_2.cpp:
export Lib.p2:
  import Lib.p1;
public:
  struct Bling: Helper {  // Okay.
    // ...
  };

// Client.cpp:
import Lib;
Bling x;
```

The example above shows that an import directive may name a module partition to make visible only part of the module, and within a module all declarations from imported partitions of that same mode are visible (i.e., not just the exported declarations).

Partitioning may also be desirable to control the import granularity for clients. For example, the standard header **`<vector>`** might be structured as follows:

```
#ifndef __STD_VECTOR_HDR

#define __STD_VECTOR_HDR

import std.vector;
    // Load definitions from std, but only those
    // from the vector partition should be made
    // visible in this translation unit.
// Definitions of macros (if any):
#define ...
#endif /* ifndef __STD_VECTOR_HDR */
```

The corresponding module partition could then be defined with following general pattern:

```
export std.vector:

public:
  import std.allocator;
    // Additional declarations and definitions...
```

The partition name is an identifier, but it must be unique among the partitions of a module (two different modules may use the same partition name, however; such partitions are unrelated). All partitions must be named, except if a module consists of just one partition.

The dependency graph of the module partitions in a program must form a directed acyclic graph. Cycles can (and should) be diagnosed. Note that this does not imply that the dependency graph of the complete modules cannot have cycles.

3.6   Nested Module Names

Module names can look like nested namespace names:

```
export Lib::Chunk:
  // ...
```

However, this is only a naming mechanism: Such names don't imply any relationship with other modules. In particular, the example above does not require the existence of a module **Lib**.

The principal motivation for this feature is to allow modules to have names matching certain namespaces. E.g.:

```
export Boost::MPL:
public:
  namespace Boost {
    namespace MPL {
      // ...
    }
  }
```

Note that unlike class and namespace names, module names cannot be used for qualification. For example:

```
// File lib.cpp:
export Lib:
  void f() {}

// File main.cpp:
import Lib;
int main() {
  Lib::f();  // Error: No class Lib or namespace Lib.
}
```

3.7   Inline Importing

When a module wants to interface to a nonmodule library, it needs to be able to declare the contents of the nonmodule library. It cannot just #include its header, because that would make each declaration of the header a member of the current module. We therefore propose an escape mechanism called "inline import":

```
export Mod:
import { // Inline import.
  extern "C" int printf(char const*, ...);
```

```
#include <stdlib.h>
}
// ...
```

Declarations appearing in an inline import are not members of any modules (and can therefore not be exported).

4   Benefits

The capabilities implied by the 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
- Global optimization properties (exceptions, side-effects, alias leaks,…)
- Dynamic library framework
- Smooth transition path from the #include world

The following subsections discuss these in more detail.

4.1   Improved (scalable) build times

Build times on typical evolving 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; a decade later, well over half the code resided in header files.) Since header files are typically included in many other files, the growth in build cycles is generally superlinear with respect to the total amount of source code. If the issue is not addressed, it is likely to become worse as the use of templates increases and more powerful declarative facilities (like concepts, contract programming, etc.) are added to the language.

Modules address this issue by replacing the textual inclusion mechanism (whose processing time is roughly proportional to the amount of code included) by a precompiled module attachment mechanism (whose processing time—when properly implemented—is roughly proportional to the number of imported declarations). The property that client translation units need not be recompiled when 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 affairs: Without it, private out-of-class member declarations become impractical to parse in the general case.

Modules 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.

Consider the following multi-translation-unit program:

```
// File X.cpp:
export X:
  import std;
public:
  struct X { X(int i) { std::cout << i << '\n'; };
  X x1(1);


// File L1.cpp:
export L.p1:
  import X; X x3(3);
```

```
// File L2.cpp:
export L.p2:
  import L.p1; X x4(4);


// File main.cpp:
import X;
X x2(2);
import L;
int main() {}
```

This program outputs:

```
1
2
3
4
```

because the location of import directives are a trigger to ensure that the imported partitions be initialized at that time. If a partition was previously initialized, it is of course not initialized a second time (i.e., the initialization code for every partition is protected by a "one time" flag).

4.5   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 whether a function will not throw an exception if it is known that the functions it calls will never throw. 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" by Walter E. Brown and Marc F. Paterno. The advantage of that 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 that approach is that without careful cooperation from the programmer, the optimizations will not occur. In particular, it is in general quite cumbersome and often impractical to manually deal with the annotations for instances of templates when these annotations may depend on the template arguments.

4.6 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.

The module concept maps naturally to dynamic libraries. Indeed, the symbol visibility/ resolution, initialization order, and general packaging aspects of modules have direct counterparts in dynamic libraries.

Modules that may be loaded and unloaded at the program's discretion are probably possible, but they are currently not discussed in this proposal.

4.7 Smooth transition path from the #include world

As proposed, modules can easily be introduced in a bottom-up fashion into an existing development environment. Nonmodule code is allowed to import modules. Top-down transitions are also possible—though likely more cumbersome—thanks to inline imports.

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.

Finally, we note that modules are a "front end" notion with no effect on traditional ABIs ("application binary interfaces"). Moving to a module-based library implementation therefore does not require breaking binary compatibility.

5 Technical Notes

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

5.1 The interface file

A module is expected to map on one or several persistent files describing its public declarations. These interface files 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 into separate object files just as they are in current implementations.

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

```
export M:
  struct S {} s;  // Private type
public:
  S f() { return s; }
```

Not every modification of the source code defining a module needs to result in updating the associated interface file. Avoiding superfluous compilations due to unnecessary interface file updates is relatively straightforward.

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 interface file: It is assumed that the implementation will use a simple convention to map module names onto file names (e.g., module name `Lib::Core` may map onto `Lib.Core.mpp`). This may be complicated somewhat by file system limitations on name length or case sensitivity.

As explained in section 2, it is desirable for the interface file to be standardized (to maintain a healthy 3rd-party tool market), and that is probably most easily achieved by having the interface file consist of C++ source code slightly augmented to simplify portability. In general, the source code of an interface file corresponding to a module partition is expected to have the contents of the corresponding partition implementation's source modified as follows:

1. The source is preprocessed.
2. Entities that are neither exported nor needed for the declaration of exported entities are omitted.
3. Names are made unique using a special suffix syntax (see below) to avoid relying on the complexities of standard C++ binding rules (lookup, deduction, overload resolution, etc.).
4. Private details that have client-visible consequences are optionally rendered opaque through attributes.

For example, the module partition

```
export M:
  class B { virtual ~B(); };
  B::~B() {}
#define X 42
```

```
public:
  class D: B {
    int f() { return X; }
  public:
    int f(int x) { return x; }
    double f(double x) { return x; }
    int g() { return f(); }
  };
```

might generate an interface file that looks as follows:

```
export M:
private:
  class [[class_layout(8, 8)]] #1 {
    ~#1();
  };
public:
  class [[class_layout(8, 8)]] D#1: #1 {
    int #2() { return 42; }
  public:
    int f#1(int x#1) { return x#1; }
    double f#2(double x#1) { return x#1; }
    int g#1() { return #2(); }
  };
```

A "unique number" scheme is not strictly needed, but it may improve portability of a generated interface file (because implementation often have small, subtle differences in lookup and overload resolution results). More work is needed in working out the details of the interface exchange source. For example, any attributes (or other extension) — like **class_layout** — used to communicate properties of the implementation to client code will require specification.

5.2   Loading a interface file

When a compiler front end encounters an import directive, it will precompile the corresponding precompiled interface file if needed, and then load that PIF. It is expected that this "loading" does not actually bring in all the declarations packaged in the module. Instead, a sort of "table of contents" is loaded (most likely into the symbol table) and if any lookup finds an entry in that table, additional declarative information is loaded as needed. For example, if the <algorithms> header is included and only one or two algorithm are used, a module-based header implementation would only load the definitions of the used algorithms.

5.3   Module dependencies

When module A imports module B (or a partition thereof) it is expected that A's interface file will not contain a copy of the contents of B's interface file. Instead it will include a reference to B's interface file. When a module is imported, a compiler first retrieves the list of modules it depends on from the interface file and loads any that have not been imported yet. To avoid undue implementation and specification complications, the following constraint is made:

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

When a partition is modified, sections of the interface 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.

5.4   Startup and termination

A 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 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 implicitly defines anonymous namespace scope variables associated with each module partition being imported. These 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).

5.5   Linkage

> **In modules, public entities cannot have internal linkage.**

5.6   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 Lib:
public:
  struct S {};  // Export complete type.
  class C;      // Export incomplete type only.
private:
  class C { ... };

// File_2.cpp:
import Lib;
int main() {
  sizeof(S); // Okay.
  sizeof(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 X:
  struct S {};  // Private by default.
public:
  S f() { return S(); }

// File_2.cpp:
import X;
int main() {
  sizeof(f());  // Allowed.
}
```

5.7 Explicit template specializations

Explicit template specializations and partial template specializations are slightly strange in that they may be packaged in a module that is other than the primary template's own module:

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

export Client:
  import Lib;
  template<> struct 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 **S<int>** it contains, while other modules might use an automatically instantiated version of **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 this proposal.

5.8 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 interface 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 Lib:
public:
  template<typename T> struct S {
    static bool flag;
  };
  ...

// File_2.cpp:
export Set:
  import Lib;
public:
  void set(bool = S<void>::flag);
  // ...

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

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

Both modules **Set** and **Reset** must instantiate **Lib::S<void>**, and in fact both expose this instantiation in their interface file. However, storing a copy of **Lib::S<void>** in both interface files can create complications similar to those encountered when implementing export templates with the existing loose ODR rules.

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 problems in the header file world).

There are several technical solutions to this problem. One possibility is to have a reference to instantiated types outside a template's module be stored in symbolic form in the client module: An implementation could then reconstruct the instantiations when they're first needed. Alternatively, references could be re-bound 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).

5.9   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 Example:
  import Friends;   // Imports 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 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 above.

5.10 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 Lib:
public:
  struct B {};
  struct D: private B {
    operator B&() { static B b; return b; }
  };
```

```
export Client:
  import 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.

5.11 Syntax considerations

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

5.11.1 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 conflicts with **import** being a keyword.

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

5.11.2 The module partition syntax

Early feedback on syntax suggested that requiring braces around a module definition was preferred:

```
export MyModule {

    ...

}
```

However, if a translation unit contains a module partition, it cannot contain anything outside that partition. That implies that requiring braces surrounding the partition's content is superfluous. Although it was not preferred by the first few reviewers, the

current brace-less syntax has since gained more traction and now appears slightly more popular than the alternative requiring braces.

5.11.3 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 makes 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 conflicts somewhat with the current syntax to introduce a module (that syntax was different in earlier revisions of this paper, however).
- 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 is now proposed) is consistent with the rules for visibility of public/private class members across module boundaries clinched the case for the current syntax.

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

5.11.4 Partition names

In earlier revisions of this paper, partition names were originally quoted strings, which allowed them to e.g. match source file names:

```
export M["m.cpp"] ...
```

However, nearly-all reviewers were surprised by that syntax and expected an identifier instead. Ultimately, simplicity and intuitiveness trumped generality and consistency.

6   Acknowledgments

Important refinements of the semantics of modules and improvements to the presentation in the various revisions of this paper were inspired by David Abrahams, Steve Adamczyk, Pete Becker, Mike Capp, Christophe De Dinechin, Peter Dimov, Lois Goldthwaite, Thorsten Ottosen, Jeremy Siek, Lally Singh, John Spicer, Bjarne Stroustrup, John Torjo, and James Widman.