N2073=06-0143
2006-09-07
Daveed Vandevoorde
daveed@vandevoorde.com

Modules in C++

(Revision 4)

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 (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 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 and an analysis of the benefits offered by these basic features is presented in Section 3.

Section 4 then presents the "premium features": Ideas that naturally fit the module formulation (e.g., an enhanced model for dynamic libraries). Section 5 covers rather extensive technical notes, including syntactic considerations. Section 6 is a repository of ideas that have been considered and rejected. We conclude with acknowledgments.

1.2 Modules vs. Namespaces

Previous revisions on this paper introduced modules as an extension to namespaces. While that approach still seems viable, this revision breaks with that notion and makes the two concepts orthogonal. The resulting feature is conceptually somewhat simpler (the change reduced the length of this paper by 15%), but most of all it eases the transition from traditional header-based libraries to compatible module-based libraries.

2 Basic Features By Example

2.1 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 wrt. 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).

2.2 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:
  struct S {
    S() { std::cout << "S()\n"; }
  };
} // End of module definition (last token).

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

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

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 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).

Module definitions can only make limited use of nonmodule declarations. The limitations — detailed elsewhere in this document — are driven by implementation considerations, and imply that modules can only be transitioned to in a bottom-up fashion. For example, a standard library implementation must be converted to a module (or a few modules) 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.

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 or module partition. This module file is the counterpart of header files in the nonmodule world. Section 5 delves more deeply into the details of how compilers can effectively implement a module file policy.

2.3 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
              // visible here and in code that imports
              // module MM.
private:
  import M2;  // Make M2 visible here, but not in
              // not in clients.
}

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

2.4 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 nonmodule namespaces 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.
  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:

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

// Client file:
import 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 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. Subsection 4.4 proposes an additional construct to handle a less common access-based technique that would otherwise not so easily translate into modularized code.

2.5 Module Partitions

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

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

// File_2.cpp:
export Lib["part 2"] {
  import Lib["part 1"];
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 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:

```
import 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 module partition could then be defined with following general pattern:

```
export std["vector_hdr"] {
public:
 import 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 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.

3 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.

3.1 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; a decade later, 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. the total amount of source code. If the issue is not addressed, it is only 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 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 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.

3.2 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).

3.3 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.

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.)

3.4 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.

3.5 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.

3.6 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 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 impossible to manually deal with the annotations for instances of templates when these annotations may depend on the template arguments.

3.7 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 concept 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 4.5 briefly discusses the possibility of modules that may be loaded and unloaded at the program's discretion.

3.8 Smooth transition path from the #include world

As proposed, modules 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.

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.

3.9 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 nonmodule contexts (i.e., as already specified in the standard).

4 Premium Options

This section explores some additional possible features for modules.

4.1 Startup and Shutdown Functions

Modules could be equipped with a startup and/or a shutdown function. For example, the identifier **`main`** could be reserved for that purpose:

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

// File_2.cpp:
import 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 Lib {
  import std;
  struct Init {
    Init() { std::cout << “Hello “; }
    ~Init() { std::cout << "World\n"; }
  } init;
}
```

4.2 Program Modules

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

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

The square bracket construct preceding the keyword **`export`** in the preceding example is an **attribute list**. (Other potential attributes are mentioned in the remainder of this paper.)

Note that this example does not assume 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 modules 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.

4.3 Nested Modules

Nested modules could be allowed.

```
export Lib::Nest {
  // ...
}
```

Some arguments can be made that such nested modules be declared in their enclosing module:

```
export Lib["part 1"] {
public:
  export Lib::Nest; // Nested module declaration.
  // ...
}
```

However, this forces a dependency between the two that is possibly artificial. We therefore propose that the nesting is solely a lexical property of the module name, and in fact we propose that the "enclosing module" need not exist. E.g., a module may be named **`Lib::Nest`** even without the existence of a module named **`Lib`**.

4.4 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 nonmodule example:

```
struct S {
    void f(double);
  private:
    void f(int);
  };
}
void g(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 struct **`S`** is moved to a module, the code might become:

```
import M;
void g(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[1], 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 considered part of the public interface and

[1] For example, a public member template could be added that would trigger an instantiation error is selected.

therefore not made invisible by a module boundary. The example above would thus be rewritten as follows:

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

4.5 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). I propose a module-based design for such a facility in N2074 "Plugins in C++".

4.6 Module registration

It is common practice to declare the interfaces of a library in header files well before the implementation of that library is written. The development of client code can then more easily proceed in parallel with the implementation of the library.

The same is true with modules: A compiler will likely be able to compile a module whose members are not defined to produce output that can be imported but not linked.

A step further with that idea may be for the compiler (or some related tool) to be able to compare two "compiled module" files. That would enable the implementation of a library to be checked against its original formal specification.

Although such a capacity doesn't need any extension beyond what has been presented so far, it may be desirable to formalize the distinction between a module specification and its implementation. The following example illustrates some possible syntax:

```
// Specification (= module registration):
register export M {
  struct S { virtual ~S(); };
}
```

```
// Implementation:
export for M {
  struct S {
    ~S() {}  // Error: Should be virtual.
  };
}
```

N2074 "Plugins in C++" explains why this capability may be considerably more important if modules are to be given the ability to be loaded under program control.

4.7 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 module 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:

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

4.8 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 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++).

5 Technical Notes

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

5.1 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 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 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.

5.2 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.

5.3 Namespaces

The concept of a namespace embedded in a module presents few problems: It is treated as other declarative entities. If such a 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.

5.4 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 feature") 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).

5.5 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 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.

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 different from 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 the 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 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 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 module file. However, storing a copy of **Lib::S<void>** in both module 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 problem 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 temporarily reconstruct the instantiations every time they're 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 previously.

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 conflict with **`import`** being a keyword.

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

5.11.2 The module partition syntax

If a translation unit contains a module partition, it cannot contain anything outside that partition. That implies that the braces surrounding the partition's content are superfluous. A partition could instead be defined with something like:

```
export MyModule:
  // ... declarations
```

Indeed, from an aesthetical perspective such an option (possibly with the colon replaced by a semicolon or left out altogether) I prefer than syntax. However, early feedback suggests I'm in a small minority in this respect. The syntax with braces also has the advantage that if there were a need to be able to declare a module name without defining one, the syntax for that would be obvious (replace the brace-enclosed member list by a semicolon).

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 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 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 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.

5.11.4 Module attributes

There are at least three different aspects of the module 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 **`export`** (or **`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.

Finally, whether the individual attributes should be delimited by quotation marks can be argued either way.

Since the earlier revisions of this paper, the need for module attributes has been drastically reduced, and it is expected that they may no longer be needed in the final proposal (they are already no longer needed in the basic feature set).

5.11.5 Partition names

As with module 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 my opinion) in favor of requiring the quotation marks: In the module world module 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.

5.12 Known Technical Issues

5.12.1 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 realistic. 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:

```
export Lib {
#include "unistd.h"
  // 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).

The proposed solution to this technical issue is to exempt declarations that are private to a module and that are not involved in any module interface from the strict ODR requirements otherwise imposed on module declarations.

5.12.2 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).

6 Rejected Features

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

6.1 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 Lib["core"] {
  // ...
}

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

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

// File_4.cpp:
export Lib["helpers"] {
  import 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 Lib["main"] = ["main", "utils", "core"] {
  import Lib["utils"];
  // Partitions "main", "utils", and "core"
  // form the complete module Lib.
}
```

6.2 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 M1 {
  // ...
}
export 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.

6.3 Auto-loading

In earlier revisions of this paper, modules were tied to namespaces. With such an approach, 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). (And now that namespaces and modules are orthogonal concepts, auto-loading is no longer an option.)

6.4 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 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.

7 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, Lally Singh, John Spicer, Bjarne Stroustrup, and John Torjo.