N1778=05-0038
2005-01-18
Daveed Vandevoorde
daveed@vandevoorde.com

Modules in C++

(Revision 2)

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 as an extension to namespaces with three primary goals:

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

Before delving in the detailed benefits and issues of the proposal, this paper offers some examples illustrating general principles and common use cases. Along the way some nomenclature is introduced to enable concise discussion.

Example 1

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

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

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

Nonmodule namespaces are henceforth called **open namespaces**. The unqualified term "namespace" then refers to both modules and open namespaces.

Example 2

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

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

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

Import directives only make visible those members of a module namespace that were declared with the keyword export.

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

Example 3

Importing a module is not transitive by default, except for exported import directives:

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

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

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

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

Example 4

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

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

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

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

```
namespace >> Err {
  export struct S { int f() {} };  // Public f.
  export 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.
    }
  };
}
```

Example 5

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

```
// File_1.cpp:
namespace >> Lib {
  namespace << std;
  void main() { std::cout << “Hello “; }
  void ~main() { std::cout << “World\n”; }
}

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

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

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

Example 6

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

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

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

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

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

Example 7

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

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

The corresponding definition has the following general form:

```
namespace >> std["vector_hdr"] {
  export namespace << std["allocator_hdr"];
  // Additional declarations and definitions
}
```

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

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

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

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

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

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

Example 8

Nested modules must be declared (but not defined) in their enclosing module:

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

A nested module can import its enclosing module.

```
namespace >> Lib::Nested {  // Only valid if declared
                            // in module Lib.
  namespace << Lib;  // Okay.
}
```

The converse is true too:

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

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

```
namespace >> Lib["part 3"] {
  namespace >> Lib::Nest;
  namespace << Lib::Nest;  // Error.
}
```

Note that modules can also contain nested open namespaces that are not so constrained:

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

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

Example 9

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

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

This facility is primarily meant to enable a binary compatible transition from pre-module C++: Global module members can be code-generated as if in the global namespace, but their visibility is limited to the module namespace.

Benefits

The capabilities implied in the introductory examples 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.

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: Over the last decade it has gone from about 10 to about 1.) Since header files are typically included in many other files, the growth in build cycles is generally superlinear wrt. to the total amount of source code.

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

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

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 and N1625).

Shielding from private members

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

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

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 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.

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.

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 unacceptable. 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. The advantage of this alternative is that the properties can be associated with function types and not just functions. In turn that allows indirect calls to still take advantage of the related optimizations (at a cost in type system constraints). A practical downside of this approach is that without careful cooperation from the programmer, the optimizations will not occur. In particular, it is in general quite difficult and cumbersome to manually deal with the annotations for instances of templates when these annotations may depend on the template arguments.

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.

It has been suggested that the module namespace concepts may map naturally to dynamic libraries and that this may be sufficient to address the issue in the next standard.

Smooth transition path from the #include world

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

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

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

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

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 be reused when implementing support for export templates in open namespace scopes (as already specified in the standard).

Options

This section explores some additional possible features for module namespaces.

Auto-loading

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

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

Opinions on whether this simplifies introductory teaching seem to vary, but there appears to be 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).

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:

```
namespace >> Macros {
#define P A   // Invisible?
#export 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.

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:
namespace >> Lib["core"] {
  // ...
}

// File_2.cpp:
namespace >> Lib["utils"] {
  namespace << Lib["core"];
  // ...
}

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

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

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

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

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:
namespace >> M1 {
  // ...
}
namespace >> M2 {
  // ...
}
```

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

Program-directed module loading

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

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

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

It may also be desirable to have such modules generate additional RTTI information that could be used to synthesize safe calls to components not originally linked into the application.

Self-importing module

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

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

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

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 probably desirable that the module file format be partially standardized, so that third party tools can portably load exported 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. */
namespace >> Simple {
  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++).

Technical Notes

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

The module file

A module is expected to map on one or several persistent files describing its exported 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 exported 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 nonexported entities may still need to be stored in the module file because they are (directly or indirectly) referred to by exported declarations, inline function definitions, or private member declarations.

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

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

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

Module dependencies

When module A imports module B 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.

> **If a translation unit contains a module partition definition, it must contain no declarations outside that partition definition.**

This rule reflects the fact that module partitions are the "translation units" of the module world. The rule only helps fitting the proposed module system in existing compilers. It could be relaxed if deemed necessary.

Nested namespaces

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

> **Nonexported namespaces cannot contain exported members.**

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

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

Startup and termination

> **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 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).

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 can be indicated using a trailing ellipsis token:

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

The static keyword 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 exported.**

Note that only namespace scope declaration can be declared (or defined) with the keyword "export". Public and protected class members are exported only as a consequence of their outermost enclosing class being exported.

Significantly more thought needs to go into the concept of linkage specification in modules.

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 exported.

For example:

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

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

Note that this does not answer the question of whether non-exported types accessible through exported declarations are complete. For example:

```
// File_1.cpp:
namespace >> X {
  struct S {};
  export S f() { return S(); }
}

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

It is believed that this should indeed be allowed (in part because the invisible type may have applicable conversion functions).

Explicit template specializations

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

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

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

There are however no known technical problems with this situation.

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

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.

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

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

// File_2.cpp:
namespace >> Set {
  namespace << Lib;
  export void set(bool = Lib::S<void>::flag);
  // ...
}

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

```
// File_4.cpp:
namespace >> App {
  namespace << Set;
  namespace << Reset;
  // ...
}
```

Both modules "Set" and "Reset" must instantiate Lib::S<void>, and in fact both expose this instantiation in their module file. However, storing a copy of Lib::S<void> in both module files can create complications of the same kind that the loose ODR rules create in the context of open namespace export templates.

Specifically, in module App, which of those two instantiations should be imported? In theory, the two are equivalent, 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 possible technical solutions to this problem. Most of them rely on having references to instantiated types outside the template's module to be stored in symbolic form in the client module. This could allow (for example) an implementation to temporarily reconstruct the instantiations every time they're needed. Alternatively, references could be rebound to a single randomly chosen instance (this is similar to the COMDAT section approach used in many implementations of the greedy instantiation strategy). Yet another alternative, might involve keeping a pseudo-module of instantiations associated with every module containing exported templates (that could resemble queried instantiation).

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:

```
namespace >> Example {
  namespace << Friends;
  void p() { /* ... */ };
  export 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 present 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:

```
namespace >> Example2 {
  export template<typename T> struct S {
    void f() {}
  };
  export template<typename T> 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.

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:

```
namespace >> Lib {
  export struct B {};
  export struct D: private B {
    operator B&() { static B b; return b; }
  };
}
```

```
namespace["program"] >> Prog {
  namespace << Lib;
  void main() {
    B b;
    D d;
    b = d;  // Should invoke user-defined conversion.
  }
}
```

If B were known to be a base class of D in the Prog 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.

Syntax considerations

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

Import and export syntax

The notation chosen for this proposal is meant to be reminiscent of the streaming concept and parallels preprocessor inclusion in that "a file is being read". However, it is not the only option. An obvious alternative may be to introduce new keywords "module" and "import". This might for example allow:

```
module Lib {
  import module Lib2;
  export import module Lib3;  // Ugly?
}
```

Replacing the keyword "import" by the keyword "using" seems undesirable: The latter keyword is currently used to indicate name aliasing, which is both sufficiently similar to and sufficiently different from the notion of importing that confusion is likely to ensue.

Another direction wrt. syntax is the introduction of **export blocks**:

```
namespace >> Lib {
  export {
    typedef int I;   // Exported.
    typedef char C;  // Exported.
  }
}
```

There are no known technical problems with this export block syntax, but they're not preferred by this author.

Yet another alternative, is the separation of interface and implementation is separate sections or files. (This is the approach used in languages like Modula-2 and Object Pascal.) For example:

```
namespace >> Lib {
export:
  // Exported declarations.
implementation:
  // Implementation of exported interfaces.
}
```

It is tempting to reuse the "public" and "private" keywords for this purpose. Note that if there is a strict limitation to two sections (as is the case in some Pascal dialects), it becomes impractical to include exported interfaces that include non-exported entities.

Yet another alternative is to rely on scoping to determine visibility: All namespace scope declarations not enclosed in an unnamed namespace could be deemed exported. A programmer would then use unnamed namespaces to render various declarations invisible. One disadvantage of this alternative is that it can force subtle binary compatibility issues when implementation details are moved these unnamed namespaces. Specifically, unnamed namespaces have external linkage and affect (e.g.) the name mangling of their members. So the transformation of the pre-module code

```
namespace Lib {
  struct Priv {};
  void f(Priv v = Priv());
}
```

to something like

```
namespace >> Lib {
  namespace { struct Priv {}; }
  void f(Priv v = Priv());
}
```

would require the recompilation of client code. The other alternatives discussed do not raise this issue.

Namespace attributes

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

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

- The form of each individual attribute

With regards to the relative position of the attributes, an interesting alternative is to place the attributes first. The advantage is that it more easily generalizes to other constructs that may not have a keyword to attach the attributes to. In the same vein, it has been suggested to use a attribute list delimiter that is more unique (since single brackets are already used for array types and for subscripting). Specifically, one could envisage the use of a double bracket ("[[...]]").

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

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

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

Taken together, all of the above alternatives may lead to the following example syntax:

```
[[program]] namespace >> First {
  namespace << std;
  void main() { std::cout << "Hello World!\n" }
}
```

Partition names

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

Acknowledgments

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