Document Number: P0678R0
Date: 2017-06-16
Audience: Evolution Working Group
Reply to John Lakos (jlakos@bloomberg.net)

Business Requirements for Modules

By: John Lakos, Bloomberg L.P.

1. Introduction and Purpose

Modules are considered to be a critically needed language feature by many C++ developers, but the reasons for the urgency vary considerably from one engineer to the next. Some are looking, primarily, to reduce protracted build times for template-ladened header files. Others want to use modules as a vehicle to clean up impure vestiges of the language, such as macros, that leak out into client code. Still others are looking to "modernize" the way we view C++ rendering completely – even if it means forking the language. These are all very different motivations, and they may or may not be entirely compatible, but if the agreed-upon implementation of modules does not take into account established code bases, such as Bloomberg's, they will surely fall far short of wide-spread adoption by industry.

The primary purpose of this paper is to serve as a proxy for discussion regarding critically important requirements for substantial software organizations, such as Bloomberg, that have very specific architectural needs, yet also have vast amounts of legacy source code that cannot reasonably be migrated to a new syntax in any bounded amount of time.

2. Current Situation

Some of the strategies require existing code bases to change before they can take advantage of modules. Significant work has gone into tooling that converts existing code bases to become "modularized", replacing conventional '.h'/'.cpp' pairs with the equivalent in module syntax, import statements in place of #include directives, etc. For companies, like Bloomberg, that have an enormous sprawling code base along with numerous disparate clients at every level of the software's physical hierarchy, any approach that requires transforming the entire codebase along with all the clients is a non-starter.

Don Knuth asserted that premature optimization is the root of all evil. Any sensible implementation of modules will enable the kind of compile-time optimizations we are all looking for, but the converse is not true. If we come up with an optimization-oriented implementation of modules and release it first, it will be impossible to graft on the necessary architecture-oriented features that would make modules realize their potential value for large-scale C++ software designers and architects. If we are to be truly successful, we must start with a fully-baked design; only after that should we attempt to optimize it.

In order for any new module technology to have a plausibly successful path to adoption, its integration must be (purely) additive, hierarchical, incremental, and interoperable, but not necessarily backward compatible with traditional rendering (e.g., '.h'/'.cpp' pairs). By (purely) additive, we mean that providing a module-style interface to existing code does not require that code to be modified (in any way whatsoever). By hierarchical, we mean that what we add to an existing code base to provide module interfaces depends on that code base (and never vice versa). By incremental, we mean that adding a module interface to one part of the code base never implies adding it to some other, disparate part of the code base. Finally, by interoperable, we mean that a C++ construct consumed through both a module interface and a (conventional) header-file interface is understood by the client's compiler to be the same construct without violating the ODR.

3. High-Level Requirements

Modules will realize their full potential as an important new feature of C++ only if:

I. Modules deliver effective support for a larger, more powerful unit of logical and physical architectural abstraction, beyond what is currently realizable using conventional .h/.cpp pairs to form components compiled as separate translation units.

a. Logical versus physical encapsulation. Today, if I have a private data member, my client needs to see the definition of that data member. Modules should allow that definition to be exported to the client's compiler, but not to the client, for arbitrary reuse. In this way, modules fix an important and pervasive problem: that of transitive includes.

b. Modules should be atomic with respect to compilation for all of the elements they comprise. That is, if I build a module containing templates and inline functions at a given level of contract assertions, the client will see that level, rather than the level at which the client was build. While this is just an example, it should apply to any and all build options.

c. Modules can be used as views on existing software subsystems consisting of arbitrary numbers of '.h' and '.cpp' files. That is, without changing an existing, conventionally implemented subsystem, one can create a module interface (purely additively) that provides an arbitrary subset of the logical entities that the module comprises. Ideally, but not necessarily initially, the level of filtering will enable one to drop below global entities to incorporate (or not) nested entities such as individual member functions. In this way a module does not encapsulate the original definition of the legacy code, but rather its use through this module interface. Finally it should be possible for multiple modules to wrap the same conventional software as views aimed ad distinct clients that converge to a single main. All of the entities exported should be known to be the same with no ODR violation.

d. Modules that act as views should behave similarly to C procedural interfaces. (See Lakos'96, section 6.5.1, pp. 425-445.) What I mean by that is that if a conventional TU is exposed in parallel with a modular view of that TU, then a client importing entities from both will get the union of access, and overlapping entities will be considered by the client's compiler as being the same entity (without violating the ODR).

II. There exists a well-considered, viable adoption strategy that does NOT require existing software to be altered in any way in order to begin to make use of the new features to allow new clients to consume legacy software.

a. Let's take a look at a real-world scenario. Suppose we have a library, 'L1_h', implemented as '.h'/'.cpp' pairs. Suppose further that we have a subsystem, 'S1', that depends on 'L1_h', and traffics in types defined in 'L1_h' in its interface. Now suppose we want to add, hierarchically, a module interface for 'L1_h', which we'll call L1_m. The current state of affairs now looks roughly like this:

```
[L1_m]    [S1_h]
     \    /
     [L1_h]
```

b. Now suppose that we get another client subsystem written entirely in module speak, ‘S2_m’. This client has no legacy implementation and none of its sub-components are consumable by conventional renderings (which is "fine" because it is new code and no old code currently depends on it):

```
[S2_m]
  |
[L1_m]   [S1_h]
    \    /
    [L1_h]
```

c. Finally a client, ‘C1_m’ comes along and wants to use both ‘S2_m’ and ‘S1_h’, both of which make use in their respective interfaces of types defined in ‘L1_h’’:

```
      [C1_m]
      / |  |
[S2_m]  |  |
   |    |  |
[L1_m]  | [S1_h]
     \  | /
     [L1_h]
```

d. Types defined in ‘L1_h’ and consumed from both ‘S2_m’ and ‘S1_h’ need to refer to the same entities. In this way, we can keep our current code base while continuously evolving towards the "more modern" module only approach. At some later point, ‘S1_m’ may be created at which point ‘C1_m’ may or may not may want to convert to use it instead, but now all new code will benefit from using the more powerful, more modern, more efficient ‘S1_m’ rendering.

III. The implementation chosen does not require centralized repositories or other known-to-be brittle techniques that would render important software processes such as distributed development or interaction with source-code control systems significantly more problematics than they already are.

a. The Google approach seems to me to rely heavily on a module cache which, from what I recall with template repositories from the 1990s was sufficiently problematic that it ushered in the current linker technology where template instantiations are duplicated locally in each translation unit in which they are used. (By “repository” here, I mean a cache of binary template instantiations that can be reused across translation units.)

IV. Once we have addressed I, II, and III, it is assumed and expected that compile-times -- especially for template-ladened interfaces -- will realize dramatic improvements over always fully reparsing source text in every translation unit.

4. Conclusion

There are many different competing ideas surrounding the design and implementation of modules in C++. There are many ways to realize modules in ways that address the requirements elucidated in this paper. It is hard for me to know, from what I have read, if and to what extent all of these requirements are addressed by the current proposal. It is my intention that this paper serve as a proxy for a discussion to learn more about where are currently, and where we need to be to move forward.