C++ Modules Are a Tooling Opportunity

Gabriel Dos Reis
Microsoft

It must be considered that there is nothing more difficult to carry out, nor more doubtful of success, nor more dangerous to handle, than to initiate a new order of things. For the reformer has enemies in all those who profit by the old order, and only lukewarm defenders in all those who would profit by the new order, this lukewarmness arising partly from fear of their adversaries, who have the laws in their favour; and partly from the incredulity of men, who do not truly believe in anything new until they have had the actual experience of it. – Niccolò Machiavelli

---

Introduction

The advent of C++ modules represents a unique and unprecedented opportunity for the C++ community to nurture a vibrant and robust tooling ecosystem, from build systems to packaging, semantics-aware development tools, etc.

Traditionally, the C++ standards definition does not directly deal with how to build a C++ program, leaving such matters to implementations for various reasons. The existing standards text deals mostly with one translation unit at a time – rarely the typical scenario for building programs or components at scale. On the other hand, modules - by necessity - put emphasis on the aggregation and the coherence of sets of translation units. And this is for the better, because they reflect how we organize code. Still, how to build and how to distribute components is left outside the standards text. Build systems are far more varied in nature, and C++ implementations vary dramatically in how they integrate into development environments for the WG21 committee to legislate. There is no reason to attempt to severely restrict how C++ should be implemented, nor are there compelling reasons to limit or to stifle innovations in this space. In fact, according to ISO [1],

> *Through its members, it brings together experts to share knowledge and develop voluntary, consensus-based, market relevant International Standards that support innovation and provide solutions to global challenges.*

By and large, the diversity in the C++ implementation landscape and the avoidance of a monoculture are immensely beneficial to the development of C++ itself, programming ideas, and implementation techniques. This diversity also comes at a (small) price as evidenced by the occasional fragmentation of "implementation user interfaces" (e.g. compiler switches) or development environments. This is unavoidable because C++ is used is a very diverse environment.

However, to facilitate interoperability, free circulation of ideas, and tooling, I would like to encourage C++ implementers to work together to resolve implementation techniques that cannot be legislated in the C++ standards definition without also stifling innovation. The C++ Modules TS presents such an opportunity.

Processing Module Uses

Modules are a set of translation units, with one distinguished translation unit called the *module interface unit*; all others are called *module implementation units*. The module interface unit contains a module declaration that establishes the purview of a module, and the set of declarations that are made visible to the consumers of that module. Those declarations are said to be exported. A consumer of a module M executes an import-declaration, e.g. "`import M;`", to get access to all exported declarations of M which are hosted in M's interface unit. How the source of the interface unit of M is stored, or how the C++ implementation finds that source when it translates that import-declaration is not dictated by the Module TS, just like the C++ standards text does not dictate how the "header files" in '`#include`' directives are found. That non-prescription of "header files" mapping is necessary for innovation, and indeed is used by various C++ implementations to (1) just open a file based on certain file search protocol; (2) map to a pre-compiled header (PCH) to avoid reprocessing the same content over and over; (3) internally make available certain (builtin) declarations; etc. As an aside, the C++ standards text: (a) makes a distinction between standard "headers" and "header files", as standard headers need not be stored in files; (b) does not require the standard library to be written purely in ISO C++, and in fact most (all?) aren't.

So much for the abstract semantics. Concretely, how does a C++ implementation go from "`import M;`" to finding the exported declarations of M? Let's note that because of the ambiguities in the C++ grammar, it is often necessary for the C++ implementation to know which names are exported by M before further processing – worse, it is even necessary for correct tokenization purposes. Consequently, there is a logical necessity for the source of the module interface unit of M to have been processed in some form before further processing after the import-declaration. This isn't new requirement, and it is similar to how standard headers and header files are handled today. One can imagine at least four scenarios for processing import-declarations:

1. Treat the interface unit as a **glorified header file**: by an implementation-defined manner (much like for header files), locate the source file containing the module interface unit and process it in a way that preserves the standard required observable behavior. Do this repeatedly for any non-redundant import declaration nominating that module. Clearly, this may not be the most efficient translation method, but it is valid and has its uses.
2. Require the **module interface unit be translated prior to the importing translation unit**: this the scenario where the module M's interface is "compiled" before compiling any consumer – currently implemented by all three major compilers implementing the Module TS. This approach exposes the dependency graph of a component without duplication. The result of prior compilation of the module interface is then used to expose the exported declarations of M.
3. Treat the interface unit as an **on-demand pre-compiled glorified header file**: this is a combination of (1) and (2) where the result of prior processing is "cached" in an implementation-defined manner.
4. Duplicate and **compile-as-you-go**: this is a variation of (3), but with some redundancy allowed.

These aren't the only possible translation techniques, but they are likely common ones. The Clang compiler has facilities to do (2), (3), and possibly (4). The current (experimental) implementation in Microsoft Visual C++ offers (2). It is unclear where other compilers, in particular GCC, stands in this spectrum. One trend though is to avoid turning the C++ implementation into a full-fledged build system for the C++ ecosystem supports far more build systems than one could realistically expect a single

compiler to adequately duplicate and support. Thankfully, the Module TS does not require a C++ implementation to become a build system – it is an option if an implementer chooses, but not one that I would recommend.

Build Systems

Any of the translation techniques (2)-(4) implies an update to the build definition to express the dependencies of components on module interface units. Furthermore, modules can be consumed in either source form, or in binary form (e.g. equivalent of conventional header+binary library). Build systems need to accommodate for either forms, and the existence of translation artifacts related to modules.

Packaging

Modules provide linguistic support for expressing software architecture, boundary, dependencies at scale. But they are not a distribution medium, by design. With modules' emphasis on set of translation units that make up a logical component, there is an opportunity for beefing up packaging support taking advantage of characteristics offered by modules. Again, this is an opportunity for the C++ community to develop common packaging systems or packaging protocols.

Binary Module Interface Formats

C++ implementations using any of the translation techniques (2)-(4) will store the results of compiling a module interface in some form. It would be a serious case of missed opportunity for the C++ community if N compilers targeting the same platform ended up using N different binary formats. This is an opportunity for tool vendors to coalesce around well-defined binary formats when targeting a given platform. Better yet, it is an opportunity for the C++ ecosystem to define a binary format and/or APIs to manipulate the compilation artifacts associated with module compilation. The binary format does not need to be in one-to-one correspondence with the internal data structures used by a given compiler. As a community effort, there would be no need for N-to-N converter, at most there would be an 1-to-N converter. The cost of developing such format and converter would be much less (quite insignificant) when shouldered by a community than if it is the endeavor of individuals.

For example, the Microsoft Visual C++ compiler is developing a format called IFC, inspired by the IPR [2] data structures, to store binary module interfaces. These data structures aren't what the Visual C++ compiler internally uses for normal processing of C++ source code. However, the IFC contains logical description of the abstract semantics graph resulting from compiling a module interface unit. The IPR is general, efficient, and flexible enough to describe the entire C++ and more. Microsoft is willing to work with other vendors to refine the IFC, and is fully committed to publication of the specification of the IFC binary format and associated tools to the larger C++ community, a commitment to nurturing a robust C++ tools ecosystem for all.

Because the abstract semantics graph stored in the IFC file captures the full semantics, and is simple to parse so you do not need another C++ compiler (or front-end) to reprocess a module interface, an IFC file offers more information (e.g. the set of configuration parameters that were used to build a component, the calling convention in place, etc.) to tools such as SWIG (mentioned in P0804R0) than the conventional unprocessed headers methodology provides. Further, it offers the ability to construct "binding interfaces"

that use the higher levels of C++ abstractions instead of the lower "C level" constructs that routinely lead to bugs and other vulnerabilities that higher level C++ constructs can and do catch.

P0804R0

Tom Honermann shared an early draft of his paper P0804R0 titled "*Impacts of the module TS on the C++ tools ecosystem*" with a few people from the C++ community (including myself) touching upon some of the topics discussed above, and also as a rationale for the PDTS ballot comment US/001. The comments below are based on that early draft. The concerns raised in P0804R0 appear to fall roughly in two categories, and are based on the experimental implementation of the Modules TS in Microsoft's Visual C++ compiler.

Interaction with build systems, and consumption of binary module interfaces

As discussed in earlier sections, the advent of modules represents an opportunity to beef up the C++ tooling ecosystem, not reason to stay forever in the '70s technologies. It is an opportunity for the C++ community to build shared knowledge and tools around platforms, or universally. The specific translation technique that Visual C++ is using is technique (2) but it is hardly the final state. This translation technique isn't required by the Module TS in any form; it is one of the allowed and the standards text must make room for innovation, while supporting robust tooling.

Microsoft is a producer and a **huge consumer** of software and software development tools, including notably software and tools produced outside the company. So, it is affected as well by Modules, which is why it has been at the front of suggesting more (open) tooling - not less - especially semantics-aware development tools. Microsoft is fully committed to opening up its IFC format and associated tooling for the C++ community. The Visual C++ team has been working on related efforts, such as build system support. There are numerous build systems in used in Microsoft, in addition to the fact as a C++ toolset vendor it has to supports build systems outside its control. Consequently, this is an area where Microsoft is most interested in solutions that work for the larger C++ community, and the community can learn from practices from other software communities.

Customer service support

A concern in P0804R0 is that "At least one large company is in the process of modularizing its source code (not currently using the Modules TS) in order to reduce overhead in its distributed build system by distributing module artifacts in lieu of source code."

As of this writing, we know of only one C++ tool vendor who is making available its experimental implementation of the Standard Library modules to collect early feedback, as WG21 usually asks. That is the Microsoft's Visual C++ compiler. If that is indeed the "one large company" of concern, the situation is easily clarified:

1. There is no actual modularization of the source code. The module artifacts (e.g. '`std.core`', etc.) were produced by just writing '`#include`' of the existing standard headers unmodified in a source file and running the compiler with suitable switches (e.g. `/module:export /module:name std.core`). That is it. Microsoft generally distributes its C++ standard library source files (even though the standard library is not required to be written in C++), and these source files just fell through the crack during packaging.

2. When WG21 finally settles on the modular decomposition of the Standard Library, that will be time to invest into actual source-level implementation, and as ever, the source will be available.

Conclusion

While C++ Modules are new language constructs, and therefore requires tools support and understanding, they represent an opportunity for the C++ community to finally improve the tools support ecosystem. This will most certainly require cooperation between tool vendors to define aspects that are by necessity outside the purview of the language definition itself.

References

[1] ISO, "ISO: All about ISO," [Online]. Available: https://www.iso.org/about-us.html.

[2] G. Dos Reis and B. Stroustrup, "A Principled, Complete, and Efficient Representation of C++," *Mathematics in Computer Science; Springer,* vol. 3, no. 3, pp. 335-356, 2011.