P1427R0

Date: 2018-11-20

Programming Language C++

Audience: SG-15, WG21

Reply to

Peter Bindels - dascandy@gmail.com - TomTom Global Content BV
Ben Craig - ben.craig@ni.com - National Instruments
Steve Downey - sdowney2@bloomberg.net - Bloomberg
Rene Rivera - grafikrobot@gmail.com - B2 Maintainer
Tom Honermann - tom@honermann.net - Engineer at Synopsys
Corentin Jabot - corentin.jabot@gmail.com
Stephen Kelly - steveire@gmail.com - CMake

Concerns about module toolability

While we are looking forward to the standardization of modules in C++ which will allow cleaner, more manageable, sandboxed interfaces, we would like to raise some concerns we have with modules from a tooling and tool-ability stand-point. Many of the concerns raised here have been raised by papers in the past, but the issues are not strictly within the purview of any particular working group, other than WG21 as a whole.

What do we mean by tooling?

Tooling here means 2 things:

- Build Systems
- Source processing tools such as IDEs, code indexer, static analysis and refactoring tools

Both of these categories face different sets of challenges to support C++ modules, and these sets of challenges need to be considered independently.

A Primer On Build Systems

What do build systems actually do?

For our purpose, a build system is a tool that transforms a set of inputs (header and source files) into machine executable for, commonly a program. This is done by invoking a compiler on each translation unit (TU) so that inputs are transformed to object code and ultimately linked together.

To that purpose, build systems maintain a dependency graph such that the operations leading to the creation of the program are executed in an order which corresponds to what is described by the source code and build system-dependant information.

An important characteristic of a build system is that modifying a file that contributes to a program should trigger the minimal and sufficient set of operations to regenerate that program.

Therefore, the dependency graph maintained by build systems needs to remain accurate.

If a build system fails to provide that guarantee, they expose developers to a wide range of hard to debug issues from spurious build failure to silent ODR violation (link or run time errors).

How does dependency extraction work?

For our purposes, a program depends on object files, which each depend on a source file and the set of headers that are included directly and transitively from the source file.
And so, for each translation unit, a build system needs to extract and maintain a list of the transitively included headers such that modifying any of these headers triggers a rebuild of the translation units that include them.

Traditionally this was done by build systems preprocessing each file and extracting a list of headers as a separate step.
However, this approach suffered from several fatal issues:

- If the dependency step was run as a distinct build system target, the dependency tree for an object file was often incorrect, leading to inconsistent builds, causing at best link errors, and at worst runtime errors (ie Ill-Formed-No-Diagnostic-Required).
- Each source file and header needed to be read at least twice upon each modification at a time when computers had little memory for caching, slow disk access and relatively quick compilation:
    - Once by the build system to extract a list of header dependency

    - Once by the compiler during the actual compilation of the Translation Unit(s)
- The build system needed to run a preprocessor on each source file and header, however, preprocessors are surprisingly non-trivial to implement and are riddled with vendor-specific extensions, such as the order in which system directories are searched, what the local directory is, and the flags by which the search path for headers is changed, which means build systems were often not able to extract an accurate list of header dependencies.

This scheme proved to be incredibly bug-prone, and unstable and is less often in use today.

Nowadays, to remediate these issues, header dependencies are collected **after** the build of each TU by means of a communication channel between the compiler and the build system (typically a file). This solves both of these issues and works because:

- A TU is required to be built at least once
- TUs do not have a dependency on each other
- Header files do not have dependencies
- Header files are usually not the result of a transformation
- Header files do not usually need to be transformed (Unless generated)

This system is not perfect. If files are moved, headers added with the same name as external headers, or new header search paths are added, the build can be inconsistent. However, assuming a stable environment and build rules, if the object file does not exist, it will be built and the exact files the object file depends on is produced by the compiler. After that point, the only way the set of dependencies of the object file can change is if one of current dependencies change, which will trigger a recompilation and updated dependency information. The modern standard scheme for unix-like systems was invented by Tom Tromey <tromey@cygnus.com>, and is described in detail at http://make.mad-scientist.net/papers/advanced-auto-dependency-generation/

Why do modules affect build systems?

The modules proposal makes significant changes that affects how build systems will need to determine the dependency graph and orchestrate the building of modularized code. Object files are not dependent on a simple list of dependencies easily determined by a compiler pass, but potentially on a complex directed acyclic graph of translation units that may not be resolved by examining a single TU in isolation.

To be clear, modules will not be successful in any meaningful way without build system support. Build systems will need to be able to handle modularized code.

Indeed, modules are not expected to be imported as text files, although that would be a valid implementation, but are instead expected to be transformed into a “Binary Module Interface” (BMI) once and then reused throughout the build.

This transformation needs to happen before a module is imported, or indeed before that same module is itself transformed.
Furthermore, for a module to be transformed into a binary module interface, all its imported modules must also be transformed transitively.

This introduces dependencies between translations units and BMI as well between BMI themselves.
This makes the dependency graph of a modularized code base much more interdependent and precludes extracting the list of dependency of a TU after it is built but instead needs to be done beforehand - Or let the compiler build module interface on demand which is a strategy that requires that a compiler become even more of a build system itself, which is usually poorly received.
This characteristic of modules poses a number of concerns and questions that we think need to be addressed.

Issues with The Modules Proposal

Extracting modules dependencies

To the best of our understanding, in a modularized world, a TU may depend on

- Included header files
- Imported modules in the preamble of a module
- Imported legacy header units, which implementations may or may not elect to convert to BMI.
- Included headers that also correspond to legacy headers.
- Imported modules anywhere in legacy headers and non-modularized code.

Furthermore, accurate extraction of dependencies in modularized and non-modularized code alike require an accurate preprocessing of the file.
While EWG tried to mitigate this particular point, we believe preprocessing files still requires running the same preprocessing steps as the compiler would.

These rules stem from the necessity of offering a way for projects to be modularized incrementally. Specifically, there has been strong support for legacy header units by the committee. However, this legacy header support comes at the cost of increased complexity for build system implementers.

The best strategy for accurate dependency extraction is probably to invoke the compiler in a special mode on each file such that the dependency extraction is left to the compiler.

There are performance implications to this strategy. Primarily, it resorts back to a system where every modified file is required to be scanned at least twice; before and for the compilation. More critically, deferring to the compiler to extract the list of dependency incur the cost of a process call which on some system is non-negligible: Invoking the compiler as a separate step in order to extract the dependencies is very expensive on operating systems where process launch is expensive. Even on systems where process launch is less expensive, it is not zero which is one of the reasons folding dependency regeneration into the compiler was seen as a large win.

This was explored in more detail in P1300, Remember the FORTRAN.

Extracting modules names

Headers are identified by their path. That is a header, such as “foo/bar.h”, corresponds to a specific file “foo/bar.h” which is located by the compiler by searching in a well defined set of locations supplied to the compiler by the build system. While this is technically implementation-defined, we are unaware of compilers supporting a scheme that differs in significant ways. Much of the behavior of the compiler and preprocessor in this respect is standardised by The Open Group, in IEEE Std 1003, aka POSIX, in order to allow portable compilation across different operating system implementations.

Modules, however, are identified not by the name of the file that declares them, but rather by a purely logical name used within the source of the importer, exporter, and implementation. This creates another dimension of complexity in the dependency graph that build-systems, especially meta-build systems, will have a hard time handling correctly.
We are also concerned that it pessimizes the performance of build system aka the time it takes for the build system to launch the first compilation and the time it takes for the build system to create and update the dependency graph.

The lack of immediate mapping between a module name and its file is considered to be the main issue with the use and packaging of Fortran 90 modules. There is no agreement about how the interface should be delivered to consumers, and there has been little adoption of modules in open source Fortran packages.

Most programming languages with a module system tie a module name to either a file name or a file path. This tie is defined unambiguously such that mapping a module to a file or inversely can always be done directly by looking in a limited and well defined number of places.

Build parallelization performances

Compiling C++ is historically slow but embarrassingly parallelizable. Some companies distribute C++ builds over a large number of cores such that many TUs may be built simultaneously. A second solution is to preprocess files only on the master machine and distribute the compilation task to secondary machines.

Modules introduce binary module interface files, speeding up the build by only parsing and preprocessing the module once. They inhibit local parallelism by causing all compilation depending on a BMI to be delayed until it is generated, while speeding up the part of the build that is parallelizable. For a remote build it causes even more trouble, as the original solution was to fully preprocess the source file locally and then send the otherwise fully independent intermediate to a secondary computer. With modules, this secondary computer will now need to be able to find and load the BMI files for the modules, complicating the process significantly. By having these added dependencies between generated artifacts it makes the compilation of C++ less parallelizable.

While the advertised reduction of compile time for a single TU is a more than welcome change, there is significant concern that modules have the potential to increase compile times within significantly large modularized projects (aka composed of a large number of modules). For some limited data measurements in this area see P1441 (Are modules fast?).

Further experimentation in this area is required to determine when a tipping point exists. And how significant the cost benefit trade offs are in this respect.

The legacy header fallacy.

We do understand the need for facilities that make adopting modules easier and welcome these facilities.
However, as currently specified, legacy headers pessimize both modularized and non-modularized TU:

- Module units require a non-trivially-parsable preamble to support the global module fragment
- Non-modularized code can import either modules or legacy header anywhere which again means the dependencies cannot be trivially extracted.

The build system has no way to determine that a given non-modular TU contains import declarations or #include directives that should be translated to legacy header units. To be accurate it would have no choice but to assume it does and therefore incurs a pre-compilation file scanning.

We would like to see solutions or guidance to mitigate these issues.

Lack of implementation and experience

Because modules affect tooling in unprecedented ways, it is important to be clear of what is meant by experience.

Various iterations of the modules TS have been partially implemented in multiple compilers and the merged proposal is being implemented. Each implementation helped refine and guide the visibility and exporting rules at the language level. As such, they were greatly beneficial towards creating the current merged proposal.

Some companies deployed modules, by declaring a small number of modules with manually defined interdependencies. This is enough to validate the design of modules as a language feature but is insufficient to validate their toolability and usability.

Modules have further been integrated into build2 by Boris Kolpackov and some experimental work is in progress in other build systems including CMake.

However, we are not aware of any projects that have validated the modules design with a large number of modules, certainly not at the scale of replacing all headers files with modules, which is often mentioned as a desired end state.

With no complete compiler implementation of the merged proposal yet, it will take a while before build systems support modules. We realize there is a bit of a chicken-egg scenario going on, but we think collecting more tooling experience is necessary to ascertain the soundness of the module proposal from a toolability perspective.

While modules have been proposed in some form or another since 2007 (!), the proposal changed quite a lot over the past year and we feel it would benefit from more baking time.

Getting Modules wrong is not an option

The current proposal goes to great length to ensure modularized and non-modularized code integrate well with each other, offer a smooth migration story, and do not create dialects.

Yet in effect, projects depending on build systems that do not support modules will not be able to consume modularized libraries, regardless if these libraries use modules or legacy headers.

Hence even though the language does not introduce a dialect with modules, lack of module support in tooling will likely lead to the same result.

Projects will not be able to opt-out of modules if some of their dependencies elect to modularize. Seen another way, projects will not be able to migrate to modules if their downstreams are not ready to use them. This is very similar to the Python 2 to 3 migration, but without a way to write code such that it works with both. This may limit the rate of adoption of modules. Therefore, it is important to ensure that implementing modules is doable and that doing so is not an unreasonable effort. It has been stated as a requirement that modules can be adopted into a codebase in a purely additive manner, that is by only writing new code, leaving the existing code unchanged. (see p0678r0)

We should further ensure that working on modularized codebases is, at the very least, not an increased burden for C++ developers.

A lot of uncertainty regarding, modules and non-build-systems tools.

Over the past decade, C++ gained many accurate tools and IDE support for accurate code completion, indexing and to some extent refactoring (albeit the later is still hindered by macros). While these tools require some configuration such as a list of include paths and macros, they do not further rely on the build system or a specific compiler. Notably, they do operate on sources such that they don't usually depend on or generate object files.

In a modularized world, this poses a number of interesting challenges:

- In the absence of efficient mapping between a module name and its source file, it is unclear how tools would be able to index symbols pertaining to the current file without including a significant overhead compared to today's tools.
- Tools would benefit from being able to read BMI as they could potentially be much more efficiently parsable, however, BMIs are compiler specific and tools and compilers often rely on different frontends, which would force tools such as clangd to rely on different sets of BMI than the compilers

Distribution and packaging of modular code

Applications will often be built using second and third party libraries, not built within the build system. That is, an application will incorporate code generated and distributed by other developers within a company or project, or entirely external to a company or project. Binary only distribution, without full source, only headers, is normal and frequent, even in open source systems.

It is unclear what should be distributed with modular code, and how that package should be consumed. Gcc, for example, will only consume the generated bmi from the same exact build of the compiler. This suggests that it is intended as an intra build system artifact, and should not be shipped in any fashion. Even if the bmi could be consumed by the system compiler, it is unlikely that it could be consumed by a different, but otherwise compatible compiler. Compiler vendor lock in is a concern for C++ developers. However, there is also an object file created. If the object file is included in a library for consumption by the build system, there are potential ODR issues with the compilation of the interface. Also, including an object file vs a library is very different from most linkers point of view. Not to mention the issues with recreating the correct build environment for the source module interface.

Module name and ABI

We are concerned that the current proposal may encourage vendors to tie symbols with module linkage to the name of the module in the ABI. This would mean that projects that concern themselves with ABI stability, which include the standard library, will not be able to move declarations from a module to another imported module.

We suggest for example updating SD-8 to clarify WG21 position in this regard.

Scope and stated goals of Modules

One of the original module proposals (P0142) focused on 4 design goals

- Componentization;
- Isolation from macros;
- Scalable build;
- Support for modern semantics-aware developer tools.

Noble goals indeed.

But it is unclear how modules are successful at reaching some of these goals

- While non-legacy modules do in fact protect from macros which we believe is an essential benefit of modules, the preprocessor is still an irreplaceable tool and further work is needed to decrease the reliance on the preprocessor and increase modularization, code safety and toolability at scale.
- Single-machine build speed are increased with modules, but it is unclear how modules behave on a large number of distributed machines.
- Any effort to provide a portable efficient representation of C++ source code or the necessary interfaces to consume modules in their compiled form seem to have been abandoned.

More importantly, C++ faces new challenges.

“Package Management”, is one of the most pressing interest of the C++ community - which was made clear by a survey conducted in 2018 on isocpp.org.

A “Simple mechanisms for packaging, distribution, and installation for libraries and programs” is also identified by the Direction Group as a necessity to make the teaching of C++ effective.

We agree and we think that modules, if they are designed with tooling concerns and this objective in mind, would contribute to make this ambitious goal a success.

A note about the Tooling ecosystem.

The C++ committee is all too aware that backward compatibility is a major concern for a lot of projects which are deployed in environments where they cannot easily be rebuilt or modified either for financial or technical reasons.

Unfortunately, the same applies to tools.

Due to a wide variety of constraints and a lack of standards, a great many build systems have been developed either as internal company tools, commercial products or open source projects over the past 3 decades. Some of these tools have been originally developed for C.

There are a lot of build systems that may not have the financial or human resources to support C++ modules if the cost is deemed too great.

Nonetheless, projects with thousands of translations units and complex build procedures rely on these tools as part of their infrastructure.

In this ecosystem, migrating a code base to a different, perhaps more modern build system is a costly and complicated endeavor that a lot of projects are unable or unwilling to undertake.

So while it is laudable to expect that modules will lead to the creation of better tools, perceived benefits of such tools have to be weighted against the cost of adopting them.

We do agree that C++ would benefit from better tools. And a lot of great tools have been created in the past decade.

We also agree with the authors of the module proposals that modules can help create better, easier to use tools.

Yet, it is unrealistic to expect modules alone will reshape the ecosystem.

Therefore it is important that modules can be implemented within the constraints of existing tools.

Modules adoption across the industry will take time. It will be a long and costly process.
We agree that it is a worthy effort.

Yet it is important that the committee ensures that this effort is not greater than it needs to be nor expect the impossible from tools whose evolution is constrained by resources, environments and backward-compatibilities concerns.

Suggestions and recommendations to improve module toolability.

SG-15 review

SG-15 (the Tooling Study group), was not able to meet in a daytime binding session yet.

But we think this group should review this paper from a toolability standpoint and offer remediation to any issue they would identify. The overlap between WG21 and non-compiler tool experts is not considerable, but we think more tool experts and providers should review this paper. Indeed, tools and language design have different concerns and requirements.

Notably, SG-15 should determine the viability of implementing the proposal in existing tools, the performance impact of modules in regards to compile times in different scenarios.

SG-15 should also study the impact of modules in regard to their stated mission and long terms goals.

Make module identifier to filename mapping deterministic and immediate

We consider the lack of specified mapping between a module name the biggest issue with the module proposal as far as toolability is concerned. Among the tools that need an efficient way to map a module name to a module interface unit source file are:

- Compilers
- Build Systems
- Code Indexers
- IDEs

There are a few ways to solve this problem:

- Encode the name of the module in the name of the module interface source file

This allow a tool to find a module interface source file quickly, deterministically and accurately in a list of directories without having to open any file or have any knowledge of C++ grammar.

It also makes build system easier to implement as a module can then not be renamed without the file being renamed which simplifies greatly dependency management.

- Encode the name of the module in the path of the module interface source file

This is also a satisfying solution for tools. We fear however that rules may be harder to agree upon as existing projects already have layouts that may not be easily adapted.

- A module mapping file

The idea would be that each project would maintain a file that maps every module name to the path of its module interface source file. The issue with this approach is that it would be harder to maintain as the same information would be duplicated in the module interface file, this mapping file, the module interface file name (which need to have a name), and in the build system files. These external files would further complicate the distribution of “module interface unit only” library, which will no doubt remain popular until better package management exist. And we want to reiterate that we think that using modules should be as transparent and straightforward as possible for C++ developers.

We understand that WG21 might feel that these concerns fall outside of the scope of “The C++ Programming language”, and that the tools will figure these things for themselves. However, without fully specified and compiler enforced rules, every tool vendor will be forced to provide their own set of subtlety broken, non portable heuristics and limitations that contributes to making C++ libraries harder to distribute.

We want to stress that these problems have been encountered and solved in numerous languages over the past 4 decades. In fact, we know of no language beside Fortran 90 which does not specify and enforces rules to map modules and their files.

Several papers were proposed to raise and address this issue, notably the very detailed [p0778R0] (circa 2017), unfortunately this issue Is perduring.

Simplify the lexing rules of import declarations

To make extracting dependencies in non-modularized code simpler, it could be envisaged to

1. Prohibit import declarations from being the result of a macro expansion.
2. Prohibit comments within an import declaration
3. Force import declarations to be the first and only statement in a preprocessed line of code

We believe these limitations would have little bearing on expressiveness while permitting more manageable and stable preprocessing implementations tailored for dependency extractions within non-modular translations units.

This would notably make it easier for compiler implementers to ship hardware-accelerated libraries that implement dependencies extraction facilities matching the behavior of their compiler and can be used by build systems.

We further think that prohibiting import declarations from being the result of a macro expansion in the preamble will reconcile the objective not to import macros before the end of the preamble with the necessity of simplifying the preprocessor implementation in regard to module preamble.

Let the Merged Proposal mature while the Tooling community works with WG21 to provide viable long term solution and high-quality module support in build systems and non-build systems tools.

We rejoice of the progress made by the module proposal, and that EWG has validated its design from a language standpoint.

However, because the proposal evolved so dramatically over the past year and because there is no compiler implementation of the latest iteration of the proposal, the tooling community was not given the opportunity to provide its own implementation experience.

Because of that, and because of the concerns collected in this document, we believe it would be premature to standardize modules.

We believe the concerns outlined here can be addressed within the C++20 release cycle, but fear that landing Modules now will make it too difficult to make changes and the feature will not be a success in practice

As Modules seem nearing readiness from a language standpoint, this extra time will give the opportunity for tool writers to work with compiler vendors and WG21 to make sure modules are usable in a wide range of environment and contribute to C++ continued success for decades to come.

It would also give the committee the time to modularize the STL as we think it is important the Standard Library gets modularized along the standardization of modules.

Finally, proposals such as `std::embed` (which would make asset bundling more efficient) and module level attributes (which could simplify and standardize some compilations flags) are of interest to the tooling community and would benefit from being well integrated in the module design.

Work with Build System vendors

As The C++ Standard works on proposals that have the potential to impact the ecosystem significantly, we think it is critical to find way to work with tools implementers to the same extent that compilers implementers are implicated in the standardisation process.

The direction group mentions:

> The latter [NdE: Package Management] is beyond the current scope of WG21, but it is essential for the continuing success of C++; maybe the new Tools SG can help. If WG21 cannot do something official, maybe its members can help establish widely accepted de facto standards (note that languages and systems competing with C++ tend not to have formal standards).

We understand that the C++ Standard has a narrow scope, but unfortunately the C++ ecosystem does not have a good track record of establishing good de-facto standards. This can be attributed partly to the lack of cooperation between WG21 and tools writers. It may be critical that the committee finds ways to involve itself more directly in the C++ tools ecosystem.

Acknowledgment

We would like to thank the people who contributed their time and expertise notably Mathias Stearn (MongoDB/Evergreen), JeanHeyd Meneide (std::embed proposal), Dan Kalowsky (Software Engineer at ESRI), Colby Pike (CMake Tools), Rob Maynard (CMake Maintainer), Dmitry Kozhevnikov (CLion Developer), Jussi Pakkanen (Creator and Maintainer of the Meson Build System).

References

A Module System for C++ (Revision 4), Gabriel Dos Reis, Mark Hall, Gor Nishanov, 2016-02-15
https://wg21.link/p0142r0

Business Requirements for Modules, John Lakos, 2017-06-16
https://wg21.link/p0678r0.pdf

Module Names, Nathan Sidwell. 2017-10-10
https://wg21.link/p0778r0

Impact of the Modules TS on the C++ tools ecosystem, Tom Honermann, 2017-10-15
https://wg21.link/p0778r0

C++ Modules Are a Tooling Opportunity, Gabriel Dos Reis, 2017-10-16
https://wg21.link/p0822r0

Retire Pernicious Language Constructs in Module Contexts, Nathan Myers (MayStreet), Adam Martin (MongoDB), Eric Keane (Intel), 2018-10-09
https://wg21.link/p0997r0

Modules, Macros, and Build Systems, Boris Kolpackov, 2018-05-02
https://wg21.link/p1052r0

Merging Modules, Richard Smith, 2018-11-26
https://wg21.link/p1103r2

A Module Mapper, Nathan Sidwell, 2018-11-12
https://wg21.link/p1184r1

Merged Modules and Tooling, Boris Kolpackov, 2018-10-04
https://wg21.link/p1156r0

Global Module Fragment Is Unnecessary, Nathan Sidwell, 2018-11-12
https://wg21.link/p1213r1

Remember the FORTRAN, Jussi Pakkanen, Isabella Muerte, Peter Bindels, 2018-10-8
https://wg21.link/p1300r0

Implicit Module Partition Lookup, Isabella Muerte, Richard Smith, 2018-10-07
https://wg21.link/p1302r0

A Principled, Complete, and Efficient Representation of C++. Gabriel Dos Reis , Bjarne Stroustrup
http://www.stroustrup.com/macis09.pdf

IsoCpp.org 2018 Survey Results -
https://isocpp.org/files/papers/CppDevSurvey-2018-02-summary.pdf

Directions for ISO C++ (by H. Hinnant, R. Orr, B. Stroustrup, D. Vandevoorde, M. Wong) (2018-10-08)
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0939r1.pdf

Package Ecosystem Plan, Rene Rivera
https://wg21.link/p1177r1

Build2 Documentation on C++ Modules
https://build2.org/build2/doc/build2-build-system-manual.xhtml#cxx-modules

Fortran Module Unsupported by Debian - Debian Bug report https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=714730

Fortran 90 Module Dependencies, Matthew West
http://lagrange.mechse.illinois.edu/f90_mod_deps/

Are modules fast? Rene Rivera
https://wg21.link/p1441r0