Remember the FORTRAN

Document number	P1300
Audience	SG15, EWG
Date	2018-10-8

Authors	Jussi Pakkanen, Isabella Muerte, Peter Bindels
Reply-to	`jpakkane@gmail.com`

A brief summary of the issue

Suppose you have two source files called `hello.cc` and `main.cc`. The first one defines a module called first and the second file would use that. Using the most current GCC this would be compiled in two phases. First you'd compile the first one with this command [1]:

```
g++ -fmodules-ts hello.cc
```

This generates two files: an object file and a file called `hello.nsm` containing the compiled binary representation of the module. Once this is done you'd compile the second file using the module like this:

```
g++ -fmodules-ts main.cc
```

This would pick up the `hello.nsm` module file generated on the previous step and use it during the compilation instead of a classical header file.

The way this works is, roughly, the way FORTRAN's module system works. It also suffers from the exact same problem: it makes writing a reliable and performant build system extremely difficult.

The problem in a nutshell

To understand the scope of the problem it is illustrative to look at the history of Fortran modules. The concept of a module as discussed above was added to the language in the Fortran-90 revision [2]. This means that the feature has existed for almost 30 years. This can be considered sufficient time for tooling and workflows to stabilize. However if we look at compilation instructions as provided by Intel, we find the following sentence [3]:

> You need to make sure that the module files are created before they are referenced by another program or subprogram.

This statement is vacuous. It provides no information or help. It does not provide you with any information on how you *should* do things. There are no recommended pieces of software to use, no links, no instructions, nothing. The documentation does have a Makefile snippet elsewhere but it requires running `depend` targets manually and thus is not reliable.

We thus find that one of the biggest corporations in the world with ample time and resources have basically deemed this too hard of a problem. Instead they provide their paying customers a poor experience that reduces, roughly, down to "good luck with that".

This is such a problem that sometimes even people working on the compilers can't get things to work reliably [1]:

> Last year I asked Jakub [who knew] how the GNU Fortran module thing went and he said that the way he gets GNU Fortran modules to work is that he executes Make sufficiently many times for it to succeed.

The core problem

Let's assume you have a C++ project with many libraries and executables. If the project has standard header files, then the build system can, given only a list of which sources make up each target, start compiling sources immediately with maximal parallelism so as to saturate the CPU. This is also fairly simple to implement: a simple handwritten Makefile does fully parallel builds automatically. This is even possible to do when using precompiled headers, though it does require a bit more work.

With modules this is not possible, because the compilation model adds an extra dependency. You can not compile any source file using a module *until* you have built the source file that provides the corresponding module definition. This causes a big problem to the build system because i*t does not know this ordering beforehand*.

This implies that when building C++ code that uses modules as defined currently, the build system *must first parse the contents of all source files*, extract the module information from them, build a DAG and then execute it in the correct order. This turns out to be incredibly difficult to implement.

Dependency graph dynamism

A simple approach to this problem would be to do a two phase build. First you scan all the files and serialise the information out to disk. Once this is done you can keep invoking the build system (such as Make or Ninja) which would then use the stored data and execute compilation jobs optimally. Unfortunately this does not work.

Because the build order is in the source files, it means that the dependency graph can be altered in arbitrary ways just by editing the code. Suppose you have an application A that uses modules defined in files B and C. Suppose further that B also uses the module defined in C. This implies a build order of C ⇒ B ⇒ A. If you edit the code by changing the contents of files B and C (but not changing them in any way) then the order changes to B ⇒ C ⇒ A. This implies that *every* build step must be started by scanning *all* changed files.

It would seem sufficient, then, to always build in two phases: a scanning phase followed by a compilation phase. Unfortunately this does not work either when we add source code generation to the mix. If we generate C++ sources from other files (such as Protobuf definitions, DBus service files and others) then this approach falls apart. Doing a full scan of the source dependencies up front is not possible, because some of the source files needed for the scan only come into existence after parts of the project have been built. This means that a build system handling modules must be able to switch between the scanning phase and build phase an arbitrary number of times during the build.

Bringing all this information together we find that the big change here is one of dynamism. The build graph of classical C++ programs is *static*. In contrast the graph of C++ builds with modules is

dynamic. Not only can it change when changing code, it is in some circumstances unknowable, meaning the graph will change while it is being iterated on. This is a fundamentally more difficult problem to solve.

Impact on existing build systems

The vast majority of build systems in use today will use either Make or Ninja to do the actual build. There are other approaches as well, for example SCons invokes the compiler directly, but they are in the minority. Whatever the module system of C++ ends up being like, it should be easy to invoke via these two backends.

The most thorough and reliable build system for Fortran that we know of is the one provided by CMake when using its Make backend. Its implementation[4] consists of 3000 lines of C++ as well as 250 lines of Bison just for the parsing part. There is also a buch of code needed in the backend, but its size is more difficult to estimate since most of it is shared with other language backends. This implies a level of complexity that is way beyond even most experienced users of Make. There are many people who build their C++ code with hand written makefiles. This is especially common in simple experiments and legacy projects. If C++ were to adopt a Fortran style module system it would mean that, in practice, compiling C++ with simple Makefiles would become infeasible.

For Ninja the situation is even worse. CMake's Fortran support described above can not be made to work with standard Ninja. Instead it requires a forked version of Ninja with custom patches. This version is not currently packaged by any known Linux distribution. Since Ninja does not have the semantics necessary to express Fortran style modules, any build system that only supports Ninja can **not** provide C++ module support.

Overall we find that Fortran style modules will significantly increase a build system’s implementation complexity. For legacy projects it may even completely prevent the usage of modules, since many legacy projects build with Makefiles and rewriting the build system from scratch might not be feasible due to e.g. budgetary limitations.

An estimate of the scanning overhead

A mandatory requirement of Fortran type modules is that the build system must scan all source files to determine where modules come from and how they can be used. If the build system is using a tool such as Make or Ninja to do this, it must invoke an external binary to do the scan. In practice this has to be the compiler, since reliable scanning requires knowledge of the entire C++ language including macros.

Invoking processes is fast on Unix-like systems but fairly slow on Windows. We tested this performance by invoking the Visual Studio 2017 compiler 1000 times in a row with the `/?` command line switch, meaning it only printed command line help and then exited. Using a Windows 7 VM under macOS, the test took 160 seconds to run.

Let's assume a project with 10 000 files and a development machine with 8 cores. This means that, with 100% CPU utilization on all cores and ignoring the time it takes to actually scan the files and write the results to a file, the scan takes more than three minutes of wall clock time. Source code compilation can not begin until the scan has fully completed.

If compilers were to provide a batch mode for scanning this may be sped up. On the other hand the scan time may actually be slower on compilers such as MinGW that are not as optimized on Windows as they are on Unix.

Conclusions

C++ is considering a module model where the module information is stored in the source files and which is then turned into a binary file during compilation. We feel that this is fundamentally the wrong solution. This presupposition is not based on theoretical assumptions but rather on the fact that a similar, though simpler, module system for Fortran has existed for almost thirty years and still causing problems to developers on a day-to-day basis. Even the solutions that can be considered usable harbor massive implementation complexity.

C++ is sometimes called the language that will eventually turn even the simplest of things into complex, expert-only beasts. This statement is not without merit. We urge everyone not to go down this route with build system requirements. Optimally the future module system should be one that:

1. Is easy to understand.
2. Is easy to drive from an external build system
3. Affords maximal build parallelism
4. Does not require that the build system to parse the contents of all sources (especially with an external tool) before it can start compiling.

References

1: Nathan Sidwell, C++ Modules and Tooling, GNU Tools Cauldron, 2018
2: Wikipedia authors, Fortran, 2018, https://en.wikipedia.org/wiki/Fortran
3: Intel, Intel® Fortran Compiler 16.0 User and Reference Guide, 2016
4: Bill Hoffman, Kenneth Martin, Brad King et al, CMake source code, ,
https://gitlab.kitware.com/cmake/cmake/tree/master