Document Number: P2409R0
Date: 2021-07-12
Reply-to: Daniel Ruoso <druoso@bloomberg.net>
Audience: SG15

Requirements for Usage of C++ Modules at Bloomberg

Abstract

This paper explores the status quo of how C++ code is reused in organizations that don't have a monorepo, evaluates the impact C++ Modules have on this status quo, and finally formulates a set of requirements to be fulfilled to enable the effective adoption of C++ Modules in those organizations.

Introduction

In this document we will explore requirements for the successful implementation of the C++ Modules standard from the context of Bloomberg.

The goal of this document is to raise awareness of technical requirements and to start a conversation on how we can address those requirements. It is not our expectation that this will require a change to the language itself, but rather that this will motivate the creation of a Technical Report to drive interoperability across tools in the scope where there is already ABI interoperability, particularly in the scope of GNU/Linux but that likely can be extended to all POSIX systems.

Bloomberg has a code base with tens of thousands of independent C++ projects, which are integrated together with a package-manager approach with aggressive dependency rebuilds to ensure coherence of all those projects into what we call a "distribution snapshot".

That distribution snapshot contains prebuilt artifacts. Most developers will create a build context that contains only the source code they're expecting to change, and build against an installed location where artifacts (headers, archives, pkg-config files) are deployed.

This stands in contrast with the practice of a "monorepo". At Bloomberg we explain that distinction by the categories "Source-to-Source Builds" (short for "Resolve Source dependencies by consuming Other Sources") versus "Source-to-Binary Builds" (short for "Resolve Source dependencies by consuming Binary artifacts''). We will start by exploring those concepts, and how they reflect real-world practice.

It is our understanding that Bloomberg's experience is not dissimilar to most Free/Libre Open Source Software communities, so while this document is heavily influenced by our particular experience, this document tries to frame it in a way that is not Bloomberg-specific.

After we describe the state of the world before the adoption of modules, we will break down different requirements for the successful adoption of modules for organizations that follow similar patterns. These requirements, explained more in depth below, are as follows:

> *R1: It should be possible to test the existence of a module outside of the current build by performing a fixed IO cost, regardless of the number of modules present in the system (e.g.: testing for the presence of a file with a name deterministically defined by the module name), without the need to open, read, and parse files containing C++ source code.*
>
> *R2: It should be possible to discover how to consume a module outside of the current build with a fixed IO cost, regardless of the number of modules present in the system (e.g.: by reading a file with a name deterministically defined by the module name).*
>
> *R3: It should be possible to discover the dependencies of modules outside of the current build without incurring the cost of parsing that module's source code (e.g.: by reading a simplified file with a name deterministically defined by the module name).*
>
> *R4: Module discoverability should be interoperable for different compilers and static analysis tools running on the same platform.*
>
> *R5: Module discoverability should include sufficient instructions to parse the module interface.*
>
> *R6: The compilation command, in addition to files on disk being discoverable and parseable in an interoperable way, should be sufficient to correctly reproduce the semantics of the translation unit.*
>
> *R7: Module discoverability should include an interoperable format to reduce the cost of parsing module files external to the build system.*

Review of Status-quo

Source-to-Source Builds

The defining characteristic of what we call a "Source-to-Source build" is that the build system understands how any other part of the codebase is built. It has access to any compilation commands and code generation steps it needs access to for the purposes of parsing code that is not part of the standard library.

In this case the intermediate build artifacts are all internal implementation details of the build system, and the only significant product is the final executable. With the prevalence of static linking, that executable becomes a self-contained artifact, completely decoupled from build-time concerns.

This means that any arguments required by the compiler are also an implementation detail for the build system, since any dependency outside of the standard library is known by the build system itself.

Source-to-Binary Builds

When operating on this style, different projects will not have visibility into each other's build processes. This is implemented by making intermediate artifacts the interfaces by which different projects consume each other's build outputs.

This is often achieved in conjunction with a package management system (e.g.: dpkg, rpm, pacman, Conan), where library projects will produce a “devel” package that can be consumed by downstream systems.

This is the prevalent way in which GNU/Linux distributions are built, and it is particularly useful to manage the heterogeneity of build systems in the FLOSS C++ Community. But it also fits well in organizations that have teams operating more independently, instead of in a monorepo and a common build system.

The Interfaces of a Source-to-Binary Build

The particular interfaces exposed in a Source-to-Binary build are going to be heavily influenced by the particular package management infrastructure. For the purposes of this discussion we will focus on the conventions adopted by the Debian project, as Bloomberg's experience is heavily influenced by it.

For the purposes of this document, we want to focus on library projects. There are many other forms of dependencies in terms of C++ (build tools, code generation, etc), but when attempting to adopt C++ Modules, library projects are most significantly impacted by the specification.

POSIX standard for compilers/linkers

It is important to note that a lot of interoperability is possible on the Linux/Unix World due to a few standards that are implemented by compilers and linkers. Those interfaces allow projects that are built with different build systems -- even different compiler versions or even compiler products -- to safely interoperate.

Even when not using the actual standardized `cc` command[1], compilers that target those architectures still preserve a compatible interface for the relevant options, such as `-I/path`, `-DTOKEN=VALUE`, `-L/path`, `-llibrary`.

This standardization has played a significant role in the landscape of code reuse on POSIX compliant, or even POSIX-inspired platforms, and it is a cornerstone of the “Source-to-Binary” approach to code reuse in C++.

ABI Uniformity in Linux/Unix systems

Additionally, it is a standard practice to assume that a given version of the operating system has a uniform Application Binary Interface for C++, and that compilers and linkers provided for that system will adhere to that.

This allows far more flexibility, as it is usually fair to assume that it is safe to use a library artifact that is delivered with the Operating System. As an example, Red Hat provides an ABI Compatibility Guide[2] that specifies what level of compatibility you can expect for library artifacts.

Driving the linker

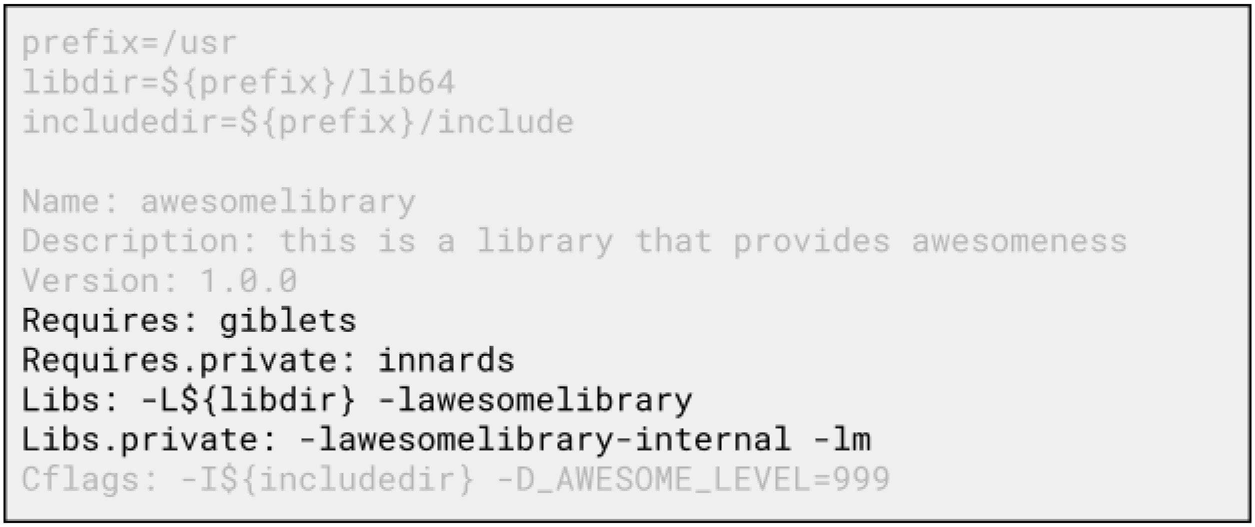
There are many complexities that arise when you try to invoke the linker consuming different projects in C++. The lack of a standardized package management system resulted in a proliferation of alternatives, be it specific to a particular build system (e.g.: CMake Config modules), or a more generic metadata (e.g.: pkg-config files).

The C++ Modules specification, as it stands now, doesn’t change the requirements for the invocation of the linker in a significant way, so we will not do a deep dive into extra requirements later in the document, but for completeness sake, we will include a brief summary of how this interface works.

That metadata for the linker also needs to be able to correctly distinguish between static or dynamic linking in most cases, which is also going to be heavily architecture-dependent.

For the purpose of this document, we will focus on the pkg-config file format, as that provides a concrete, yet simple, metadata format. This document will not cover the whole specification for that metadata, rather in this section we will cover how the format handles the linkage interface.

[1] https://pubs.opengroup.org/onlinepubs/7908799/xcu/cc.html
[2] https://access.redhat.com/articles/rhel-abi-compatibility#Appendix



```
prefix=/usr
libdir=${prefix}/lib64
includedir=${prefix}/include

Name: awesomelibrary
Description: this is a library that provides awesomeness
Version: 1.0.0
Requires: giblets
Requires.private: innards
Libs: -L${libdir} -lawesomelibrary
Libs.private: -lawesomelibrary-internal -lm
Cflags: -I${includedir} -D_AWESOME_LEVEL=999
```

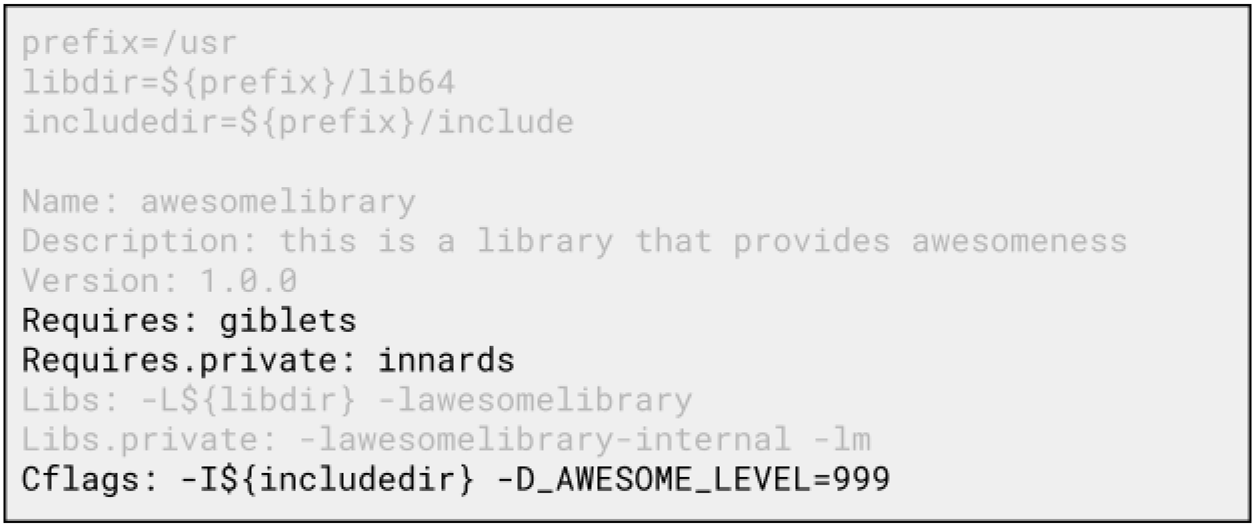
The relevant parts of the snippet above are the `Requires` and `Libs` properties, with their `.private` counterparts.

The distinction between the bare property and the “`.private`” counterpart is whether you are linking it statically or dynamically, represented by the `--static` argument to `pkg-config`. When linking dynamically only the bare property will be followed; when linking statically, both the bare property and the “`.private`” counterpart will be followed. This is because a dynamic library is expected to either embed or reference its dependencies directly, while static linking requires the full transitive set of dependencies to be included in the final linker invocation.

The `Requires` and `Requires.private` properties allow you to reference other dependencies that have a `pkg-config` file themselves. The `Libs` and `Libs.private` properties specify what library arguments need to be added, either for this library package, or for any library dependency that doesn’t have an equivalent `pkg-config` file.

Translation Interface

As is the case for the linkage interface, the lack of an industry standard package management framework on the C++ language has led to a lot of heterogeneity in this space. For the purposes of this document we will continue focusing on the `pkg-config` format.



```
prefix=/usr
libdir=${prefix}/lib64
includedir=${prefix}/include

Name: awesomelibrary
Description: this is a library that provides awesomeness
Version: 1.0.0
Requires: giblets
Requires.private: innards
Libs: -L${libdir} -lawesomelibrary
Libs.private: -lawesomelibrary-internal -lm
Cflags: -I${includedir} -D_AWESOME_LEVEL=999
```

As with the linkage interface, you can specify dependencies on libraries that offer pkg-config files, but you can also specify the `Cflags` property, which is used to compose the compilation command.

Note that this is not a set of instructions that were used to compile the code in the library itself, but rather it is the set of arguments expected from users of those libraries in order to be able to reuse that code correctly.

Files on disk

Ultimately, package management systems represent the intermediate build artifacts of C++ library projects as a set of files, where at least some of those files are meant to be in a fixed place for discoverability.

Projects using `pkg-config`, for instance, need to deploy their metadata files in the path hard-coded in the pkg-config executable for them to be discoverable. That being said, `pkg-config` also supports that to be customized via the `PKG_CONFIG_PATH` environment variable, so it is realistic for an organization to set a different convention for how to discover those metadata files.

Once the metadata files can be found, everything else can be derived from there, so an organization can have their own conventions on how to deploy their library artifacts. However, it is also common to follow the Filesystem Hierarchy Standards[3] and use standard directories for headers and libraries.

[3] https://refspecs.linuxfoundation.org/FHS_3.0/fhs/index.html

Compilation Database

The compilation database[4] was first introduced by the clang project as a way to allow an easier integration of "tools based on the C++ Abstract Syntax Tree". This became a de-facto standard that allows various build systems to provide a single entry-point for tools to be able to correctly parse the C++ code without the need to be tightly coupled with the particular build system.

This has also become a significant enabler for IDEs to offer interactive feedback to the users by either running the compiler directly, or via the Language Server Protocol.

Requirements Statement

This section will now focus on the requirements that arise from the use of "Source-to-Binary Builds" at large-scale. It is our expectation that this applies to any organization that performs large-scale C++ integration without a monorepo. This is the case at Bloomberg, but it also includes the C++ Libraries distributed by most GNU/Linux distributions.

Cheap Module Discoverability

One significant gap in the C++ Modules as specified in C++20 is the definition of a cheap way of identifying which modules exist and where they are defined. The standard allows for any translation unit to export a module of any name, without any expectation for correlation with the file system.

This works reasonably well for the "Source-to-Source" use case, as there's no expectation that this will represent additional parsing beyond what would be necessary for building the project anyway, since the build systems in general have a higher-level concept of "targets" which allows reducing the scope of what needs to be scanned.

However, in "Source-to-Binary" builds, it is frequently the case that the system will have a much larger number of dependencies available when compared to what this particular project may need. This problem gets further exacerbated by transitive dependencies, or the reuse of the system installation across multiple projects. And since there's not an uniform concept of "package", the build system would end up having to parse all modules available in the system.

Therefore having to parse all existing module files to build a mapping is unrealistic for organizations using the "Source-to-Binary" approach.

> *R1: It should be possible to test the existence of a module outside of the current build by performing a fixed IO cost, regardless of the number of modules present in the system*

[4] https://clang.llvm.org/docs/JSONCompilationDatabase.html

> *(e.g.: testing for the presence of a file with a name deterministically defined by the module name), without the need to open, read, and parse files containing C++ source code.*
>
> *R2: It should be possible to discover how to consume a module outside of the current build with a fixed IO cost, regardless of the number of modules present in the system (e.g.: by reading a file with a name deterministically defined by the module name).*

Dependency Graph of Modules External to the Build

Having to perform the discoverability of module dependencies topologically for modules outside of the current build will result in an undesirable performance penalty. Therefore it’s important that the module dependency graph should be exposed in a way that is cheaper to parse.

Additionally, building the dependency graph is a precondition to being able to parse the module files in the first place, therefore making the process of building that dependency graph cheaper will have a significant impact on the overall cost.

In a Source-to-Source Build, all module files already need to be scanned for the full parsing, therefore the additional cost to build the dependency graph is not relevant. However, in a Source-to-Binary Build, we don’t necessarily expect to parse all module files, therefore having a simplified interface to discover the dependency graph of modules external to the build will provide a significant improvement in build times.

Identifying the dependency graph for the module happens as a step before the module can be parsed, since the parsing of this module may depend on other modules. Therefore if the tool needs to parse the original source to identify dependencies, it may have to read a lot more content before having access to the dependency information.

While the standard limits this in a very concrete way[5], It doesn’t forbid - and in fact, it explicitly allows[6] - the use of a preprocessing macro in an import statement, so you can't just use a simple lexer to identify dependencies, you need to evaluate with a complete preprocessor, including the possibility of having to open included files in order to be able to evaluate macros.

Therefore, it would be beneficial to have a file with a simpler syntax, which is less likely to contain a significant extra amount of data to be read and parsed.

[5] https://eel.is/c++draft/cpp#import-3 -- "If a pp-import is produced by source file inclusion (including by the rewrite produced when a #include directive names an importable header) while processing the group of a module-file, the program is ill-formed."
[6] https://eel.is/c++draft/cpp#import-2 -- "The preprocessing tokens after the import preprocessing token in the import control-line are processed just as in normal text (i.e., each identifier currently defined as a macro name is replaced by its replacement list of preprocessing tokens)."

> *R3: It should be possible to discover the dependencies of modules outside of the current build without incurring the cost of parsing that module's source code (e.g.: by reading a simplified file with a name deterministically defined by the module name).*

Compiler-independent Module Discoverability

In an environment with ABI uniformity, it is safe to have a library that was compiled with g++-7, another library that was compiled with g++-9 and an application that is compiled with clang++, as long as they all agree to the uniform ABI.

Likewise, Static Analysis tools will need to be able to consume modules that are both inside and outside of the current build, therefore that discoverability process needs to be made interoperable.

Therefore it is important that we have a compiler-independent way of identifying which modules exist on a system and where they are provided from.

> *R4: Module discoverability should be interoperable for different compilers and static analysis tools running on the same platform.*

Compiler-independent Module Parsing

Most initial implementations of the module specification utilize an implementation-specific binary module interface file, which is an implementation detail for the compiler. This is a reasonable approach for a monorepo. However, in the case of "Source-to-Binary Builds", this will prevent reuse across compiler versions or products that would otherwise be compatible, as well as static analysis tools.

It is possible to understand the implementation-specific binary interface file to be a simple optimization for what should be available for any other compiler, but in that case, it's necessary that a compiler should be able to know how to parse those additional modules without knowing how those modules were built in the first place.

There should be an interoperable way of specifying any preprocessor or additional instructions (e.g.: `-D`, `-I`, `-isystem` arguments) that are required to correctly parse the module files. The alternative is to rely on the current heterogeneous solutions, such as `pkg-config`.

The relevant difference is that modules, unlike include statements, have their own independent parsing context, and therefore it would be entirely reasonable to expect that arguments to parsing a module would be different from the arguments to parse the consumer of a module.

> *R5: Module discoverability should include sufficient instructions to parse the module interface.*

Observability Outside of the Build System

There is a de-facto standard that has been established over the past few years on how to observe a C++ build system in order to perform static analysis. This allows any tool to know how to parse a given translation unit without the need to be tightly coupled with the build system.

This is currently used by IDE integrations, either via the Language Server Protocol or by just invoking the compiler directly to offer immediate feedback to the user.

In order to preserve those characteristics, it’s important that a tool should be able to observe the build system from the outside and be able to analyze the source code.

> *R6: The compilation command, in addition to files on disk being discoverable and parseable in an interoperable way, should be sufficient to correctly reproduce the semantics of the translation unit.*

Cheap Parsing of Modules External to the Build

In a monorepo, the full parsing of all modules is already expected, therefore there is no concern on the requirement that the build system must fully parse the module interface from its original source code when using a different compiler or a static analysis tool.

However, when you are consuming a significant amount of modules from outside your build system, having to parse the original source code for all those modules can easily become a scalability problem.

Ideally what is currently implementation-defined for the module interface file would become standardized, such that different compiler products and versions would be able to reuse the same files, but in the absence of that, having an interoperable format that is cheaper to parse would significantly improve the parsing costs.

The difference that Modules introduce is that each module has now a fully-independent parsing context, which means that things like the optimizations around include guards will no longer be valid, and the code in a header may need to be parsed multiple times if we need to parse unprocessed module interface files.

> *R7: Module discoverability should include an interoperable format to reduce the cost of parsing module files external to the build system.*

Summary

We believe that the requirements presented here will be relevant for any organization that is not working in a monorepo environment, including GNU/Linux distributions. At the same time, we recognize that it may not be realistic to solve those requirements with a single universal specification across all platforms.

We believe SG15 is in the position to drive interoperability even if with a limited scope. In particular, we believe there would be substantial benefit for interoperability for the usage of C++ Modules anywhere where we already have an expectation of ABI interoperability. We believe a Technical Report can be used to establish a de-facto standard with that more limited scope.