Document Number: P2577R0
Date: 2022-04-11
Reply-to: Daniel Ruoso <druoso@bloomberg.net>
Audience: SG15

C++ Modules Discovery in Prebuilt Library Releases

Abstract

This paper presents a mechanism that allows for publishers and consumers of prebuilt C++ Module libraries to communicate on which modules are provided by a given library.

It builds on the existing convergence around the existence of “linker arguments” in C++ projects, and the fact that C++ Libraries today already frequently have to communicate which linker arguments their consumers need.

The convention is that the metadata files describing C++ modules will be located alongside the files that are inputs to the linker, with a naming convention deterministically defined for that particular environment.

The paper also evaluates Benefits, Limitations and Requirements for that approach, and includes an example on how this could be applied in the GNU/Linux environment.

Introduction

This paper presents an alternative solution to what was presented in P2473R1[1]. Specifically, the “Goals” and “Non-goals” described in the introduction of that paper can be understood as the starting point for this paper.

The approach taken in this paper, however, is to build a convention on top of a surface of convergence that exists today amongst most implementations, even if surrounded by implementation-defined behavior.

It is the understanding of the author that a longer-term solution to this problem necessarily involves a much more extended convergence in the general area of package management in the C++ ecosystem. However, this paper proposes a solution that works within the limited convergence that exists today.

The expectation is that by building on the existing status-quo, we should be able to discourage the creation of build-system-specific and package-manager-specific solutions to the problem of discovering C++ modules in prebuilt libraries, even before we are able to establish a wider convergence on the area of package management as a whole.

[1] Ruoso, Daniel (2021). Distributing C++ Module Libraries. https://wg21.link/p2473r1

It is also the understanding of the author that a solution specific for C++ modules that involves stepping into areas that are in the scope of package managers would have the potential of making the future work of a wider convergence in the package management space harder.

The expectation is that the convention proposed in this paper will eventually be superseded by the work to create convergence in the package management space, although this will likely remain a reliable lowest-common-denominator mechanism even then.

Study Group Feedback on P2473R1

The work on P2473R1 started from the principle that we didn't have enough convergence in the package management space that would allow us to build on concepts such as "package" or "library", and for that reason it considered introducing an entirely new search mechanism that was indexed by the module name. This allowed the cost of the discoverability to be proportional to the number of modules the system depended upon, and not proportional to the number of modules present on the system.

However, the idea of making that discoverability indexed through the module name, particularly assuming that the module name would be encoded in the filesystem, failed to reach consensus. There was a clear indication that we needed a higher-level of abstraction, that allowed for reading metadata files more closely aligned with libraries themselves (even if we don't have a consensual definition of "library"), as well as keeping the module names out of file system entries.

This, however, depended on being able to identify what sets of metadata files we needed to consume, as well as how to identify dependencies between those metadata files, such that a build system wouldn't need to explicitly identify the complete transitive closure of all the metadata files. There is currently no consensus on how to achieve this.

"Linker Arguments" as Convergent Concept

While almost the entirety of the behavior of build systems and package managers are implementation-defined, there is once concept that is common to all of them: At some point, the build system and the package manager, when consuming a library as a prebuilt artifact, need to communicate enough to identify what are the linker arguments that a consuming build needs in order to successfully use that library[2].

While the way in which those arguments are discovered is fully implementation-defined, while the format and semantics of those arguments are also fully implementation-defined, the concept

[2] "Header-only" libraries are an exception to that rule. This is an important caveat which will be handled in a separate section of this document.

that when you consume a prebuilt library you will need to use additional linker arguments is a point of convergence.

It is also a point of convergence that those arguments will be translatable to files on disk, through implementation-defined translations. This is a requirement for C++ libraries being distributed as prebuilt artifacts today.

Quality of Implementation Issue: Translating Linker Arguments Into Input Files

While the format and arguments of the linker are specified in an implementation-defined way, the current reality is that build systems, package managers, static analysis tools, etc that want to interact with different linkers consistently need to re-implement the parsing of the arguments and trying to emulate the behavior of the linker.

This paper recommends that a toolchain should provide a mechanism to translate linker arguments, even if just a fragment of a complete linker invocation, into the input files that are going to be used by the linker.

Third-party implementations of that translation can be provided in case vendors do not provide it directly, but there is room for better interoperability if a tool that performs that translation exists one way or another.

Header-Only Libraries

Header-only libraries have been a common practice in the C++ ecosystem. It makes a specific trade-off on how to use specific C++ language constructs in order to avoid the complexities of the lack of convergence on the package management for various different environments.

It also allows a library to avoid ABI-compatibility questions, and therefore support virtually any standards-conforming toolchain without having to provide a prebuilt artifact to any of them.

It is often the case that package managers will still provide a release of those header-only libraries within their ecosystem, such that they can be addressed for dependency management as well as to manage required compiler arguments, such as include directories and compile definitions, even when linker arguments are not required.

As we transition to C++ modules, it is reasonable to presume that the same approach can be translated, we can call them “interface-unit-only module libraries”. As with header-only libraries, there are specific constraints on how the code has to be written, but, in principle, any library that could be implemented as a header-only library could also be implemented as an “interface-unit-only module library”.

However, C++ modules create additional requirements for the parsing of the consumers of a library. Therefore even if a library doesn't require linker arguments, it needs to be able to specify how to parse those interface units coherently. In practice, that means there is distinctively more metadata that needs to be provided with an "interface-unit-only module library" than the comparable "header-only library".

Specifically, this will mean much more importance to the work done by the maintainers of the package management metadata for those libraries in the various package management systems. It is fair to say that it will not be practical, beyond illustrative cases, to consume C++ module libraries in the absence of some amount of package management infrastructure.

Solving the consumption of "interface-unit-only module libraries" in an interoperable way will require a wider convergence on package management in general. Therefore **this paper recommends that package managers produce library artifacts, even if empty, and include linker arguments for interface-unit-only module libraries**.

This will allow the linker arguments to be a consistent point of convergence for all cases, which is enough ground to stand a convention on how to discover C++ modules in pre-built library releases. Future work for the convergence in the area of package management may make this recommendation unnecessary.

Finding Module Metadata from Linker Input Files

This paper proposes a convention where metadata files describing the modules that are part of a pre-built library are distributed alongside the library files that will be used as input files by the linker.

The mechanism by which the linker arguments are communicated from the producer of the pre-built library artifact to the build system consuming that library is implementation defined. But the expectation is those mechanisms are already in place, since that's already a requirement for C++ libraries to be consumed.

The naming convention for the input files is also implementation-defined, therefore the specific translations will also be implementation-defined. There are, however, a number of requirements to be fulfilled by any such implementation.

Over the following subsections, we will discuss specific benefits, requirements, as well as work through examples on some specific environments.

Document Number: P2577R0
Date: 2022-04-11
Reply-to: Daniel Ruoso <druoso@bloomberg.net>
Audience: SG15

Summary

In order to consume a pre-built library, a build system currently already has to interact with the package manager in order to discover how to consume that library. In most cases, that will include discovering fragments of linker arguments that need to be added to the target build.

The proposed convention is that metadata files will be deterministically named alongside the files that are used as inputs to the linker, such that the build system can parse those to create a complete understanding of all modules relevant to this project.

Benefits

Better Mitigation Against ODR Violations

The separate passes for compilation and linking in C++ is a persistent source of problems caused by ODR violations. It's frequently the case that a header is parsed by the consumer of a library in a way that is different from the way it was parsed by the code in the library as it will be used at link time.

C++ Modules create an opportunity to reduce that risk, since the parsing of the module interface unit is decoupled from the parsing of the caller code. However, matching the compile-time arguments with the correct link-time arguments is not automatically solved by modules, in fact, the same asymmetry that existed between headers and libraries could be reproduced in the modules ecosystem.

What this proposal does, by making the module metadata tightly coupled with the library artifact, is to ensure that the build system has a mechanism to establish a strong coupling between the way that module interface units are parsed and the linker input files that will be used in the final program.

This will provide a level of guarantees for ODR coherency that was not possible in the C++ ecosystem before.

Using Only Existing Semantics

This proposal can be implemented entirely on top of existing implementation-defined behavior. That means no required changes in package management infrastructure is required at this time.

The only changes required on the producing side are to compose and install a new metadata file alongside the library artifact. The only change required from the consuming side is to evaluate the fragment of linker arguments required for an external library in order to find the metadata files and merge that information into the general module dependency graph.

Limitations

Requires “Linker arguments”

As discussed in the “Header-only Libraries” section of this paper, it is not true that C++ Libraries always have linker arguments. Finding a better solution for those will remain as future work that builds better interoperability among package managers.

Build systems may need changes to make sure the linker arguments fragment required from external libraries are available early to allow the building of the entire module graph.

Emulating the Argument Parser from the Linker

While this paper recommends that toolchains provide a mechanism to translate linker arguments into a list of input files, it is true that this doesn’t exist today. There is an opportunity for third-party providers to ship tools that perform that task.

Linker Scripts and Other Indirections

Some linkers – the GNU Linker being a notable example – provide support for “linker scripts”, which allow the indirection of input files and more advanced logic to be processed by the linker itself. This paper does not specify if the translation of the linker arguments should interpret those scripts or not, although it recommends that this should be interoperable to any library within the same scope of ABI compatibility.

C++ Module libraries shipping linker input files that use linker scripts or other indiections should be aware of the conventions of that specific environment and provide the metadata files accordingly.

Relationship Between Parse-Time and Link-Time Encapsulation

Since we are using the link line to identify all the modules that need to be considered for the build, any mechanisms that provide link-time encapsulation will carry that encapsulation to the visibility of modules.

For instance, when you produce a shared object, it’s possible to encapsulate which symbols your interfaces depend on, in such a way that the linker arguments only need to reference the direct shared object being depended on.

C++ Module Libraries using this technique will need to make sure that they only import, in their module interface units, other modules provided by libraries that also appear in the linker arguments.

Document Number: P2577R0
Date: 2022-04-11
Reply-to: Daniel Ruoso <druoso@bloomberg.net>
Audience: SG15

Requirements

Match a Specific Flavor When Available

In the Microsoft Windows operating system, when using the native ABI, it's frequently the case that you will need to parse a module interface unit differently if you are linking against a static library versus when you are linking against a dynamic library. This is typically implemented in terms of preprocessor macros that propagate through the build system.

Therefore the translation of linker input file to metadata file should allow for preferring a metadata file that specifically matches that particular flavor of the library before searching a flavor-agnostic metadata file.

Archives with Members of Different ISA/ABI

Toolchains that support archives with members that implement different ISA/ABI stored in a single archive will need to encode in that translation a mechanism to find the ISA/ABI-specific metadata.

Example

While this paper is not trying to establish the precise convention that will be used in any particular environment, it's useful as an illustration of how it can be used, and it can also be considered as a starting point.

For the purpose of this example, we will consider the GNU/Linux environment, using the GNU Linker with GCC as the linker driver, in an environment that currently uses pkg-config to identify how to consume a pre-built library.

In this particular environment, you will identify "packages" you depend on in terms of their pkg-config names. And then you can obtain linker argument fragments either for each of those dependencies individually, or for the whole set of dependencies at once.

The pkg-config tool is then responsible for identifying the transitive dependencies, and is then able to produce compiler and linker arguments:

- `pkg-config --cflags foo bar baz`: produces the compiler flags necessary to consume all three libraries. This is typically used to identify include directories and preprocessor arguments. This will still be used for finding headers exposed by those libraries, but it's not used to identify module dependencies.
- `pkg-config --libs foo bar baz`: produces the fragments of linker arguments that are necessary to consume all three libraries, including any transitive dependency

> that is required on the link line. This is usually expressed in terms of "`-L/directory`" arguments and "`-llibrary`" arguments.

For the sake of argument, let's imagine that the output of "`pkg-config --libs foo bar baz`" is "`-L/opt/foobar/lib64 -lfoo -lbaz -lbar -lqux -lquxx`".

That represents the linker arguments for the libraries external to this build system. The goal of the build system is to then identify which C++ modules are provided by each one of them. The first step of that process is to convert the linker arguments into a list of input files. Fortunately, in the case of the GNU Linker on GNU/Linux, this is a fairly straightforward process.

Since linker scripts are considered "input files" to the linker, the translation stops at the direct interpretation of the arguments, and does not attempt to interpret those scripts to emulate the entire linking process. That means that any library that is shipped with linker scripts knows it needs to provide the module metadata alongside the linker script.

After translating the arguments (which involves introspecting the default system library paths), the following files are determined to be the relevant linker input files:

- `/opt/foobar/lib64/libfoo.a`
- `/opt/foobar/lib64/libbaz.a`
- `/opt/foobar/lib64/libbar.so`
- `/usr/lib64/libqux.so`
- `/lib64/libquxx.so`

Since on GNU/Linux there isn't a requirement that the code needs to be parsed differently depending on whether it will be linked against a static or a shared library, and likewise since it's not common practice to add objects for a different ISA/ABI in the same archives, on this environment the rule can be a simple matter of replacing the extension of the library file, such as .a or .so, and replace by .meta-ixx-info to find the metadata files, resulting in the following files being found:

- `/opt/foobar/lib64/libfoo.meta-ixx-info`
- `/opt/foobar/lib64/libbar.meta-ixx-info`

Since not every library will provide C++ modules, it is not an error to have libraries without those metadata files.

Once those files are found, then the union of the metadata for all of them should contain the entire dependency graph of the modules involved in this build, with enough instructions on how to correctly parse them, to match the build of the library artifact being used.

Document Number: P2577R0
Date: 2022-04-11
Reply-to: Daniel Ruoso <druoso@bloomberg.net>
Audience: SG15

Future Work

Format of the Metadata File

This paper is only trying to specify how metadata files are found to identify the complete module dependency graph for a C++ project. Since that file doesn't exist today, we have an opportunity to specify it in a way that is fully interoperable across the entire C++ ecosystem.

Conventions for Each Environment

The scope for interoperability in this proposal should match the overall scope of ABI compatibility. Any environment where pre-built libraries can interoperate will benefit from the ability to interoperate on this metadata.

Therefore, each of those environments will need a specification of how the translation is made from the linker arguments to the metadata file, including all implementation-specific considerations.

Convergence in Package Management

This specification may become obsolete by a wider scale convergence in the area of package management in the C++ ecosystem. However, it is possible that the benefits presented in this paper will justify that the metadata file remains hosted alongside the library files. It is also possible this may remain viable as a lowest-common-denominator approach to this problem.