Document Number: P2473R1
Date: 2021-12-02
Reply-to: Daniel Ruoso <druoso@bloomberg.net>
Audience: SG15

Distributing C++ Module Libraries

Abstract

This paper proposes a format for interoperability between build tools, compilers, and static analysis tools that facilitates the adoption of C++ Modules where libraries are distributed as pre-built artifacts as opposed to the build system having access to the entirety of the source code. This proposal aims to address requirements R1 to R5 of the paper P2409R0[1].

Changes

- Changes in Revision 1
    - Clarified wording that the meta-ixx-info file is not necessarily in the same location as the ixx file, but instead is subject to an independent lookup with the same relative path.
    - Moved the concept of Variable Substitution to be a responsibility of the package manager, meaning any variables must be substituted before the files are presented to the build system.
    - Introduced a section related to alternative parsing options for the same interface file. Additionally, the bmi filename now includes a fragment to identify which of the alternative parsings are being used.
    - Clarified the "Search order" section to explain it in better detail including the reason for it.
    - Made it a requirement to provide a meta-ixx-info file.
    - Proof-of-concept implementation is now required to generate recipes for building each individual bmi file, since names are now dependent on the contents of the meta-ixx-info file.

Introduction

In environments where package managers are used to distribute pre-built library artifacts, the general expectation is that the contents of the library can be represented as files on disk. Those files are deployed into either standard locations on disk[2] or per-package locations[3].

[1]Ruoso, Daniel (2021). Requirements for Usage of C++ Modules at Bloomberg. https://wg21.link/P2409R0
[2] Most GNU/Linux distributions follow the Filesystem Hierarchy Standards: https://refspecs.linuxfoundation.org/FHS_3.0/fhs-3.0.pdf
[3] Those are usually relocatable locations, however, some package managers, such as Nix, use the location as an addressing mechanism to uniquely identify the versions of dependencies.

Document Number: P2473R1
Date: 2021-12-02
Reply-to: Daniel Ruoso <druoso@bloomberg.net>
Audience: SG15

This proposal aims to provide a convention by which C++ Module Libraries can be distributed as pre-built artifacts. However, a number of related topics are out of scope for this proposal.

Non-Goals

- **Dependency Resolution**: This proposal presumes that the libraries available in the system are made available coherently (i.e.: no ODR violations) and completely (i.e.: no missing modules or headers).
- **Package Distribution Methods**: This proposal will not handle how packages are addressed or distributed.
- **Package Format**: This proposal handles the conventions for how the files will be laid out on disk, not how they are transported or the mechanisms by which they are unpacked.
- **Linker Arguments**: This proposal assumes the build system will work in conjunction with existing package managers in order to resolve the required arguments to be used when invoking the linker.
- **Compiler Arguments for the Translation Unit consuming the modules from the library**: As with discovering the linker arguments, the arguments required for the Translation Unit calling the library are presumed to be resolved between the existing build systems and package managers.
- **Internal implementation of the build system**: This convention in no way tries to define how a build system may decide to produce and reuse modules that are part of the build system or even how to produce and consume intermediates that may be needed when consuming a module library.

Goals

- **Module Discoverability**: Given a configuration of module search paths decided between the build system and the package manager, establish a convention to discover modules available in the system.
- **Instructions to Parse the Module Interface**: Given that C++ Modules introduce a distinction between the parsing of the module and the translation unit consuming it, combined with the fact that the intermediate Binary Module Interface files are not interoperable, this proposal defines a mechanism for instructing how to parse a module interface source file.
- **Convention for optimizing dependency discovery**: Avoiding the need to do a preprocessor pass in the step of building the module dependency graph is a desirable optimization in some situations. While this is not a requirement, this proposal sets a convention for those that decide to implement it.

Assumptions

- **Module search path**: The concept of a module search path is common in other languages, such as in gfortran[4] and javac[5], and it allows the system to work without a full manifest of all modules available in the system. The build system and the package managers should be able to define an ordered list of paths on disk by which lookups are made with the understanding that every new lookup starts from the beginning of the ordered list in order to identify the module.
- **Relocatability**: All path-related conventions described here should be located from the top-level module path as configured by the build system and the package manager; it should not assume that absolute paths can be referenced outside of those paths.
- **Path Substitutions**: Package managers may use variables to perform substitutions in the paths described by the metadata to decouple dependencies in cases where package managers use deployments with versions in the paths, such as Nix. Those substitutions are expected to be made prior to the information being given to the build system.
- **Explicit installation step**: This decouples the layout of the source code in its origin repository, and allows any rewriting to be made in paths in order to comply with this convention.

Convention for Distributing C++ Module Libraries as Files on Disk

This convention defines a simple lookup method for modules within a particular item of the ordered search list, with the assumption that the ordered list will be traversed for each required module lookup.

Module name to file names

The module name, as used in the import statement, will be translated to a file name by using the following convention:

- The dot character represents hierarchy, so it's converted to a nested directory structure
- The partitions of a module are moved to a directory named after the module with the suffix ".part", with the file named after the partition name
- The module interface units will have the ".ixx" extension[6]

Examples, assuming a POSIX file system:

[4] https://gcc.gnu.org/onlinedocs/gfortran/Directory-Options.html
[5] https://docs.oracle.com/javase/9/tools/javac.htm
[6] Assuming that's the convention we expect, this paper defers that decision to match what the general expectation for the extensions for interface units.

Module name	Path relative to root of module search path
foo	foo.ixx
foo.bar	foo/bar.ixx
foo.bar:baz	foo/bar.part/baz.ixx

The general expectation is that the installation process for a module library will make the translation on behalf of the user, which allows for a deterministic lookup when consuming a module from outside of the build system.

Caveats

- Case-insensitive file systems may lead to conflicts for modules that only differ in case.
- Any character that would be invalid in a file name on a given system becomes invalid as a module name on that system.
- Unicode codepoints in the module name may result in less portable projects due to incoherent expectations on the encoding of file names in different file systems and operating systems.

Instructions for consuming the module

The instructions for consuming the module will be searched through the module search path in the same relative path from the root as the module interface file, with the extension "`.meta-ixx-info`".

This file will be encoded as a JSON object, with the following keys:

- **`include_path`**: Ordered list of paths required for the preprocessor pass on the interface unit file. This does not include the paths required for standard or system headers implicitly provided by the compiler. Defaults to empty.
- **`definitions`**: Object with key-value pairs of preprocessor definitions. Defaults to empty.
- **`imports`**: List of module names imported by this module interface unit. This is an optional field that allows the build system to avoid parsing module units[7] to deduce the same information. An empty list specifies that no modules are imported, if this key is missing or the value is nil, the implementation should perform the dependency discovery.
- **`_VENDOR_extension`**: Vendors may use this format for specifying extensions for the metadata that may be used by the build system.

[7] The standard currently optimizes for detecting whether a source file imports or exports modules, but the dependency discovery still requires a fully capable preprocessor.

It is an assumption from this convention that any other option beyond the ones that would apply to the preprocessor must be applied uniformly between the translation unit consuming the module and the parsing of the module interface unit. Those are flags such as the language standard version, or other flags that influence ABI compatibility.

It is also assumed that the include paths and definitions from the translation unit consuming the module should not be used when parsing the imported module.

Variable Substitution

When substitution variables are required for include paths or definitions, the package manager should perform those substitutions by itself and it should present to the build system with a fully resolved set of meta-ixx-info files. As a suggestion to implementers, those can be added as an overlay in front of the search path.

Alternative parsing of the same Module Interface files

There are significant prior examples of changes in semantics and behavior depending on particular preprocessor arguments. While they are strictly considered a violation of the One-Definition-Rule, in some cases, such as conditional compilation of assertions, those are in general considered "benign" violations.

In addition, we should also consider the scenario where the build system is configured to use an alternative parsing of the interface in coordination with selecting an alternative build of the library to link against, avoiding violations of the One-Definition-Rule.

This is frequently expressed in terms of "build flavors", such as "Debug" versus "Release". For the purposes of identifying how the module is parsed, the build system should consider the sha1 checksum of the contents of the meta-ixx-info file.

Distributing Binary Module Interface files

While the scope of interoperability for binary module interface (bmi) files is quite limited[8], it is a significant optimization for environments where the code is mostly produced with the same compiler. That is the case for most GNU/Linux distributions, for instance.

However, it's also important to avoid any ambiguity on whether the bmi file is applicable to a given compiler. Therefore this convention requires compilers to provide an identifier that is as unique as the compatibility it supports.

[8] At the point of this writing, the interoperability of Binary Module Interface files is the same as that of precompiled headers, which is limited to the same exact version of the compiler, or in some cases even only the same build.

Likewise, given that different instructions for parsing the module interface file can result in different interpretations for that module, the sha1 checksum of the contents of the meta-ixx-info file that was parsed should be included in the name of the bmi file, thus preventing ambiguities between the meta-ixx-info lookup and the bmi lookup.

Those identifiers should then be used on the name of the file being distributed, following the pattern:

```
${module_path}.bmi.${vendor}.${compat_uuid}.${meta_ixx_info_sha1sum}
```

For instance, if g++ uses a uuid-v4 to identify the build of the compiler that can reuse a bmi, and if the lookup finds a meta-ixx-file with the contents of '`{}`', the file for the module foo.bar would be named:

`foo/bar.bmi.g++.20734238-4fc7-4725-bf22-be9700326774.bf21a9e8fbc5a3846fb05b4fa0859e0917b2202f`.

Note that even though the example above uses a uuid-v4, a compiler that decides to version its compatibility format could use any identifier that would be a valid extension to the file name. The compiler is free to assign any identifier it wants. Implementations following this convention should treat it as an opaque identifier.

While it would be possible for the compiler to support different BMI formats at the same time, adding a provisioning for that in the convention would significantly increase the complexity of the lookup. In order to limit the complexity, this convention sets the expectation that a compiler expects only one input format.

Search order

When searching for the files in the module search path, the search should always be started from the beginning for each file that needs to be found. This is important to allow overlays that contain only the meta-ixx-info file, or overlays that contain only the bmi file.

For instance, if the module search path is, in order, `/path/a/`, `/path/b/`, `/path/c/` then the search order for a module `foo.bar` should be:

1. Lookup for ixx file, in order:
    a. `/path/a/foo/bar.ixx`
    b. `/path/b/foo/bar.ixx`
    c. `/path/c/foo/bar.ixx`
2. Lookup for meta-ixx-info file, in order:
    a. `/path/a/foo/bar.meta-ixx-info`
    b. `/path/b/foo/bar.meta-ixx-info`
    c. `/path/c/foo/bar.meta-ixx-info`

3. Lookup for the bmi file, assuming `g++.20734238-4fc7-4725-bf22-be9700326774` as the compiler compatibility identifier and `bf21a9e8fbc5a3846fb05b4fa0859e0917b2202f` as the checksum of the meta-ixx-info file.
    a. `/path/a/foo/bar.bmi.g++.20734238-4fc7-4725-bf22-be9700326774.bf21a9e8fbc5a3846fb05b4fa0859e0917b2202f`
    b. `/path/b/foo/bar.bmi.g++.20734238-4fc7-4725-bf22-be9700326774.bf21a9e8fbc5a3846fb05b4fa0859e0917b2202f`
    c. `/path/c/foo/bar.bmi.g++.20734238-4fc7-4725-bf22-be9700326774.bf21a9e8fbc5a3846fb05b4fa0859e0917b2202f`

Missing Files

While the build system should be able to react to a missing bmi file by scheduling its parsing, both the ixx and the meta-ixx-info files must always be present for a module to be considered valid when distributed. While it would be possible to assume the absence of a meta-ixx-info file as having an empty object by default, this would be error prone. Making the meta-ixx-info file mandatory allows the detection of common configuration errors by failing fast, rather than having a potentially incorrect default behavior.

De-optimizing the presence of bmi files

When a build system recognizes the presence of a compatible bmi file for the current build, it would be a valid optimization to stop the traversal of dependencies and just consume that directly. However, when that happens, the build system hides details on how to reproduce the parsing of those module interfaces from any tooling that would integrate with the system.

Therefore, build systems that want to preserve maximum interoperability will need to continue traversing module interfaces, even when they see no reason to parse them, and make that information available as it is the case today with the compilation database, otherwise the observability of that build will be greatly reduced.

Discovery tooling

Even with the convention established here, there's still an additional piece of tooling that becomes a requirement for the consumption of a module library: the construction of a plan to parse all required modules.

The main problem happens when dealing with transitive dependencies of modules being provided as libraries.

This paper proposes a tool called `c++-modules-config`, that takes as input a search path of module libraries available to be consumed, the compatibility identifier for bmi files, and a list of modules that need to be resolved.

This tool will do the recursive traversal to return a complete description that is sufficient for the build system to plan the parsing of all modules provided by the system that need to be consumed. This will exclude modules that may be available but that are not going to be consumed.

The output of this tool will be a complete listing of the modules involved in this build, the source files with their interfaces, the instructions required to parse those files, as well as module dependencies. This information should be made compatible with the format documented in P1689R4[9].

This paper proposes an extension to that format to store the contents of the meta-ixx-info file under the `meta-ixx-info` key. It can be used by anyone who needs to produce a different bmi file.

This tool also needs to be able to incrementally update this file such that whenever the list of external module dependencies changes, the re-calculation should be fast.

For better observability of the build, the output of this tool should be stored in a file named `module_config.json` in the same location where `compile_commands.json` is currently saved.

Could that be shipped by libraries?

In the presence of a standardized package manager, it would be reasonable to assume that the package manager could also play the role of providing the information that would be otherwise discovered by the `c++-modules-config` tool for all the modules that are included in each of the packages available in the system. The package-to-package dependency could be used to limit the amount of data that would need to be ingested.

In the absence of a standard package manager, however, we don’t have a mechanism to identify a logical unit at a higher level than the module itself. An alternative approach would be to have a standard location where the metadata for all available modules would be read from, however, on a system with many more dependencies available than consumed, this could result in a substantially higher amount of unnecessary I/O.

[9] Boeckel, Ben & King, Brad (2021). Format for describing dependencies of source files. wg21.link/p1689r4

Enabling the build with plain Makefiles

C++ Modules created a new layer of indirection on the compilation process. Prior to modules, a simple Makefile would be able to produce all objects in an embarrassingly parallel way. With modules, however, it is necessary to create the bmi files before each consumer of a module can be parsed.

Additionally, the lack of correlation between file names and module names means that the rules for the build depend on a mapping between source files and module names to be extracted. Compounding that to the lack of a standardized package management system, results in the fact that there was no build-system-agnostic way of identifying which modules were present on the system, short of parsing all the source files that could be found.

By adopting this convention, we enable the generation of make-compatible dependency instructions for the parsing of modules external to the system by a simple tool invocation that parses as few files as necessary, including the generation of rules for locally building the bmi files for modules external to the current build system that didn't ship a compatible bmi.

Moreover, If the source code of the project itself has the physical layout described in this paper, it will be possible to automate the generation of Make rules for all bmi files for code internal to the project. The process would be not unlike how object files, archives and executables are produced in a project without modules. A Proof-of-Concept example of how a build system would be able to achieve that accompanies this paper.

While it's not the intent of this paper to advocate for a simple Makefile as a desirable build system, it is an important illustration of how much simpler the semantics of the build become when we can count on a convention between file names and module names.

Integration Scenarios

This section is meant to illustrate how this convention could be used to improve interoperability in various scenarios.

Consuming Module Libraries from CMake

CMake would be able to discover modules from outside of the build system and configure a target for each of those modules by identifying the ixx file by name, as well as the instructions on how to parse those modules. It may also be able to detect when the system already provides the bmi for the particular compiler being used.

This removes the need for CMake-specific "find module" implementations for arbitrary module libraries, as long as a module search path can be defined between the package manager, CMake, and the user.

Consuming Module Libraries from Plain Makefiles

The use of well-defined search paths on disk enables an implementation with vanilla Makefiles, where the rule for producing intermediate bmi files uses vpath[10] to find the sources for a module. This supports cases where the bmi for that compiler is already present or where it needs to be produced. See the Proof-of-Concept that accompanies this paper.

Building Module Projects with Plain Makefiles

If the compiler supports looking up modules according to this convention, a project with the source layout matching this convention, will be able to have its build driven by a plain Makefile by simply including an additional scan step that generates module dependency information from a translation unit.

Transparent Compiler Support

When compilers and static analysis tools support the same convention, it is possible for the single-translation-unit case to work transparently where a translation unit depends on modules. It would even be possible for independent invocations of the compiler to use the filesystem alone to synchronize[11] the production of bmi files and avoid duplicate work between different compiler invocations.

Deploying internally in the build system

If a build system were to create an internal deployment of the module units within the build system itself, it would make it trivial for static analysis tools completely external to the build system to reproduce the semantics of the build without any additional integration beyond the compile command.

Proof-of-Concept Implementation Example

Accompanying this paper, we are publishing proof-of-concept code exercising the lookup mechanism, exercising the interaction between the proposed c++-modules-config tool with a simple GNU Make build system and a mock compiler. The example uses vpath to parse modules internal or external to the local build, as well as can coherently consume a bmi file in case it's shipped with the library.

[10] https://www.gnu.org/software/make/manual/html_node/General-Search.html
[11] In POSIX systems, for instance, creating a hard link and renaming a file are atomic operations. By combining those two operations it is possible for a compiler to create a "leader election" mechanism allowing one process to know not to try and produce a BMI file because it's already being parsed by another process.

Document Number: P2473R1
Date: 2021-12-02
Reply-to: Daniel Ruoso <druoso@bloomberg.net>
Audience: SG15

See the proof-of-concept at https://github.com/bloomberg/P2473