Document Number: D2597R0
Date: 2022-11-17
Reply to: Daniel Ruoso
Bloomberg LP
druoso@bloomberg.net

C++ Modules Ecosystem Technical Report, Version 1

Note: this is an early draft. It's known to be incomplet and incorrekt, and it has lots of bad fomatting.

Contents

1 Introduction [introduction]

1.1 Scope [introduction.scope]

1 This aim of this technical report is to describe a model and best practices for how the C++ software development ecosystem should adopt modules.

1.2 Normative references [introduction.references]

1 The following referenced documents are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

(1.1) — ISO/IEC 14882:2020, *Programming languages — C++*

2 ISO/IEC 14882:2020 is hereafter called the *C++ Standard*.

1.3 Structure of this document [introduction.structure]

1 Clause 2 discusses how tooling relates for the different types of C++ source files, the terminology used by tools when referring to those and how they relate to each other.

2 Clause 3 discusses requirements for tooling related to the build process of code declaring and/or consuming C++ modules.

3 Clause 4 discusses the interpretation of specific features described in the C++ Standard in relationship to modules.

4 Clause 5 discusses interoperability for the distribution of C++ libraries with modules across different compilers, build systems and package managers.

2 Source Files [source]

1 For tooling purposes, there are two main categories of source files: the primary source file, and included files. The primary source file of a translation unit is the one used to start the first phase of the translation. The included files contribute to translation unit by methods of “Source file inclusion”.

2 With the exception of header units, the translation unit type can be defined unambiguously by the contents of the primary source file because of preprocessing restrictions established by the C++ Standard. Header units require additional information at the tooling level.

2.1 Translation Unit Types [source.types]

2.1.1 [source.types.importable]

importable units
Translation units that must be translated before other translation units (usually via the encoding of a binary module interface file), such that the usage of “import” statements can be correctly interpreted.

2.1.2 [source.types.interface]

module interface units
Importable units that contribute to the “external interface” of a named module with declaration and definitions within the purview of a named module.

2.1.3 [source.types.primary-interface]

primary module interface unit
Module interface unit without a partition, it can be unambiguosly identified by the presence of the export-keyword and the absense of a module-partition. There can be only one primary unit for each module.

2.1.4 [source.types.interface-partition]

module interface partition unit
Module interface unit with a module-partiton. It can be unambiguously identified by the presence of both export-keyword and the presence of module-partition. There can only be one unit for a given partition name.

2.1.5 [source.types.internal-partition]

internal module partition unit
Importable translation units with declarations and definitions in the purview of a given named module, but that do not contribute to its “external interface”. It can be unambiguously identified by the absense of the export-keyword and the presence of module-partition. An internal partition may or may not be reachable by the module interface units. An internal partition that is not reachable by the interface units is not required to be made available to translation units outside the module.

2.1.6 [source.types.header-unit]

header unit
Header units are the independent translation of what would otherwise be available via source inclusion. They are not identifiable directly by the contents of the primary source file. Additional information is needed at the tooling level to identify them. Implementations are not required to accept any header or header file that could be included to be also importable.

2.1.7 [source.types.implementation]

module implementation unit
Module implementation units contain definitions in the module purview that were omitted on the interface units and internal partitions units. It can be unambiguously identified by the absense of the export-keyword and the absence of module-partition. It may also contain other declarations in the module purview, but those are not reachable from the module interface or internal partitions.

2.1.8 [source.types.non-module]

non-module unit
Translation units without a module-declaration that are not header units.

2.2 Filename Extensions [source.file-extensions]

1 TODO: modules-ecosystem-tr#20

3 Build Process [build]

3.1 Import relationship between translation units [build.import]

1 The name of a module used in the “import” and “module” keywords is the “logical name” of a module.

2 When a translation unit imports a named module, the translation unit doing the import has an import dependency on the translation unit providing the importable unit identified via the “logical name”.

3 The translation of an importable unit is externalized into an intermediate artifact, the Built Module Interface file (BMI). That file is then consumed when importing a named module.

4 BMI files can be constructed in a way to optimize for the efficiency of the reuse in the import operation, which may result in a limitation on the number of scenarios where that file can be used.

3.2 Compiler arguments and the import relationship [build.arguments]

1 “ABI coherency” is the expected level of coherency in the compiler arguments for the translation of the code in the purview of a module and all translation units importing that module. When that level of coherency is not met, the final program may fail to link or it may link and have undefined behavior at runtime.

2 “BMI coherency” is the expected level of coherency in the compiler arguments for a BMI file to be used when importing a module. When that level of coherency is not met, the translation will fail because the BMI cannot be loaded.

3 While all translation units importing a module must have “ABI coherency”, different translation units importing the same module may not meet the requirements for “BMI coherency”. It must be valid for those to consume different BMI files of the same importable unit, and to be linked into the same program.

4 “Local Preprocessor Arguments” are the arguments that are expected to be different in the translation of an importable unit and the translation of a unit importing that named module.

5 The decision on whether an argument is a “Local Preprocessor Argument” cannot be made by introspection on the code or the command line, and should be be provided by the author of the importable units and of the unit importing a module.

6 To assemble a “BMI coherent” command line in order to produce a BMI file that is usable in the context of a translation unit importing that module, the starting point should be the compilation command of the translation unit doing the import, then the “Local Preprocessor Arguments” for the translation unit doing the import are removed, and the ones for the importable unit are integrated into the command line.

7 Compilers should offer an interface to generate an identifier for the command line that allows matching the translation unit importing a module with a BMI that can be loaded in that context. That interface should not require the primary source file of the translation unit to be identified, nor the output names to be produced. Any preprocessor argument given to the compiler in this mode should be considered for the identifier.

8 Build systems should be able to deduplicate the rules to generate BMI files to be used when importing the same named module from different translation units by getting the identifier for the command line of those translation units with the “Local Preprocessor Arguments” filtered out.

3.3 Steps of the Build [build.steps]

1 When building C++ code that produces or consumes modules, the build process needs to be broken down in different steps. The steps described here represent the semantic organization, not the interface presented to the C++ developer, the organization in steps does not preclude parts of those steps to be executed in parallel.

3.3.1 Identify External Importable Units [build.steps.external-importable]

1 As the C++ standard library itself will offer a modular interface as well as importable headers, it will be very rarely the case that a build system will be able to consider only modules declared inside the same build context.

2 External importable units will be described in metadata files made available to the build system. While this report recommends a specific format for that metadata file, the mechanism by which the set of metadata files to be considered is identified will be implementation specific.

3 Annex A.1 specifies a convention for environments where the build system and the package manager can resolve linker argument fragments early enough such that it can be used to identify the set of module metadata files that need to be considered.

4 TODO: Format of the metadata file.

5 TODO: modules-ecosystem-tr#5

3.3.2 Identify Importable Headers [build.steps.importable-headers]

1 While external importable units may include header units, it's also necessary to identify any importable header internal to the project itself.

2 The C++ standard describes "importable headers" and "non-importable headers", however those cannot be differentiated from their content alone, therefore they need to identified at the tooling level.

3 The C++ standard also allows (15.3.7) the implementation to optimize away source file inclusion when a header is known to be importable.

4 Since an include directive may be replaced by an import directive, a change in the identification of header units may change the dependency information of any translation unit.

5 TODO: modules-ecosystem-tr#6

3.3.3 Dependency Scanning [build.steps.dependency-scan]

1 Once the list of importable headers is defined, the dependency scanning phase can be performed. The dependency scanning is invoked by the build system on all translation units (including header units).

2 The dependency scanning of each translation unit is independent of the dependency scanning of any other translation unit.

3 TODO: modules-ecosystem-tr#7

3.3.4 Generation of the Binary Module Inteface [build.steps.bmi-generation]

1 TODO: modules-ecosystem-tr#8

3.3.5 Compilation [build.steps.compilation]

1 TODO: modules-ecosystem-tr#9

3.3.6 Linking [build.steps.linking]

1 TODO: modules-ecosystem-tr#10

3.4 Coherency Requirements [build.coherency]

1 TODO: modules-ecosystem-tr#11

3.5 Header Units [build.header-units]

1 TODO: modules-ecosystem-tr#12

3.6 Relationship with the Standard Library [build.stdlib]

1 TODO: modules-ecosystem-tr#13

3.7 Trivial Cases [build.trivial]

1 TODO: modules-ecosystem-tr#14

4 Language Semantics [language]

4.1 Propagation of Attributes [language.attributes]

1 TODO: modules-ecosystem-tr#15

4.2 Command-line compilation definitions [language.command-line-defines]

1 TODO: modules-ecosystem-tr#16

5 Distribution [distribution]

5.1 Source Distribution [distribution.source]

1 TODO: modules-ecosystem-tr#17

5.1.1 Binary Distribution [distribution.binary]

1 TODO: modules-ecosystem-tr#18

5.1.2 Importable-Unit-Only Libraries [distribution.importable-unit-only]

1 TODO: modules-ecosystem-tr#19

Annex A (informative) C++ Modules Discovery in Prebuilt Library Releases [discovery-prebuilt]

A.1 "Linker Arguments" as a mechanism to discover C++ module metadata files [discovery-prebuilt.linker-arguments]

1 This section describes a framework for the discovery of module metadata files on environments where the linker arguments for the library can be resolved early enough. This is not a requirement for all implementations, as some build systems need to defer the resolution of linker arguments to a step that is later than the translation of the module interfaces.

2 While almost the entirety of the behavior of build systems and package managers are implementation-defined, there is one concept that is common to most 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.

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

4 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 pre-built artifacts today.

5 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.

6 This report 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.

7 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.

8 The convention estalishes 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.

9 The specific conversions from the paths to the input to the linkers to the paths of the metadata files are implementation defined. Different sections of this report will describe the convention for specific environments.

A.1.1 Header-only Libraries [discovery-prebuilt.header-only]

1 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.

2 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.

3 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.

4 As we transition to C++ modules, it is reasonable to presume that the same approach can be translated, we can call them "importable-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 "importable-unit-only module library".

5 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 "importable-unit-only module library" than the comparable "header-only library".

6 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.

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

8 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.

A.1.2 Conventions for translating linker input files to module metadata files [discovery-prebuilt.conventions]

1 TODO: This section should document the different conventions in place, there was an example in the original paper, so we need a new paper where that is discussed in more details.

Index