Document Number: D2597R0
Date: 2022-05-28
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 formatting.

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 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.

3.1.1 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 TODO: modules-ecosystem-tr#5

3.1.2 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

3.1.3 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

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

1 TODO: modules-ecosystem-tr#8

3.1.5 Compilation [build.steps.compilation]

1 TODO: modules-ecosystem-tr#9

3.1.6 Linking [build.steps.linking]

1 TODO: modules-ecosystem-tr#10

3.2 Coherency Requirements [build.coherency]

1 TODO: modules-ecosystem-tr#11

3.3 Header Units [build.header-units]

1 TODO: modules-ecosystem-tr#12

3.4 Relationship with the Standard Library [build.stdlib]

1 TODO: modules-ecosystem-tr#13

3.5 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

Index