Document number: P1484R1
Date: *2019-03-11*
Project: Programming Language C++, SG15 Tooling
Reply-to: *Peter Bindels <dascandy at gmail dot com>*

I. Table of Contents

II. Introduction

In compiling software users currently work with include paths and complicated build system driven include path orders to find the relevant headers for compiling their software. In a post-modules world, there would need to be an equivalent lookup from module name to module source or binary module include file done. This paper explores a direction already taken by many existing modules implementations; having a deterministic mapping from module name to source name and location.

III. Motivation and Scope

In the current paper on modules (P1103) the question on how to map from a given module import to the appropriate module export statement is not specified. Multiple suggestions have been made to fix this problem in a roundabout way (P1184, P1302) but these solutions do not tackle the full holistic problem of mapping an import to a pre-compiled BMI file resulting in a feasible build tree that a tool can realistically retrieve from the input files (P1427).

In this paper we take a step beyond the proposal suggested in P1302 - having a fixed method for finding the module imports referenced. This answers the question how to resolve a given import to a relevant binary module import. The solution proposed is far from a new idea - GCC and Clang already implement a form of it.

As the C++ standard itself does not define how compilers are implemented; this paper targets the SG15 TR slated to be created for this purpose. That is actually the desired outcome - to have a standard described way of doing something, while leaving the implementers free to do something when it is not applicable or when something else with clear benefits seems to exist.

As of yet, similar to Clang -fimplicit_module_maps [https://clang.llvm.org/docs/Modules.html#module-maps] and GCC's default lookup start ("The GCC modules implementation began with a fixed mapping of module name to BMI filename, and a search path to look for them." [P1184]).

IV. Impact On the Standard

As the change is around how compilers perform a lookup it would be a non-normative addition to the proposed modules implementation (P1103). The intended wording would be similar to P1302 but specifying the module lookup that P1302 explicitly does not propose.

V. Design Decisions

How does this mapping compare to the one introduced by P1302?

The mapping is intentionally specified as one that matches the results from P1302, while explicitly choosing the alternative it spells out as “**We do not do this**” in its Design section. The design is 100% compatible with P1302 and includes it wholly.

Is the mapping a forced requirement? What if the system in question is unable to use it?
In order for the mapping to be a useful standard, its use is strongly recommended if the system is able to support it. Three mappings are given as examples and if any of these is implementable, using them has preference over a platform specific choice.

If none of the three are supportable, the suggestion is to make a platform-specific documented fixed transformation from module to originating file, publicly shared to avoid conflicting standards on platforms and to encourage tool developers to support these transformations on the given platforms.

What if a collision occurs between a module and its partition, versus a different module and (potentially) its partition?
The paper proposes this to be malformed code. For example, having a io partition in std, and a std.io module would be considered ill-formed, no diagnostic required (but very welcome regardless).

VI. Technical Specifications

Given a module import for “std.io”, the tool or compiler will know this comes from one of the following specific files:

1. *“std/io/module.cxx”,*
2. *“std/io.cxx”,*
3. *“std.io.cxx”*

in that order. When none of these are found, the compiler will not resolve the module. Module partitions are mapped equivalently, treating the partition separator as equivalent to the dot. An example for the “std.io:stream”:

1. *“std/io/stream/module.cxx”,*
2. *“std/io/stream.cxx”,*
3. *“std.io.stream.cxx”*

As a more formal specification,

1. *The name is transformed by replacing all dots with ‘/’, and appending ‘/module.cxx’.*
2. *The name is transformed by replacing all dots with ‘/’ and appending ‘.cxx’.*
3. *The name is transformed by appending ‘.cxx’.*

VII. Acknowledgements

Thank you to Nathan Sidwell for his help with implementing modules, Gabriel Dos Reis for the original Modules TS and all other contributors to the existing Visual Studio, Clang and GCC implementations. An additional thank you is extended to the people working on build system and related tools, for inspiring this paper. A final thank you goes to Bryce Adelstein Lelbach for providing the impetus to actually write the paper.

VIII. References

[P1302] Implicit module partition lookup, Isabella Muerte, Richard Smith, 2019-01-21
[P1103] Merging modules, Richard Smith, 2018-11-26
[P1184] A Module Mapper, Nathan Sidwell, 2018-11-12
[P1427] Concerns about module toolability, Peter Bindels, Ben Craig et al., 2018-11-20