Document Number: P0778r0
Date: 2017-10-10
To: SC22/WG21 CWG/EWG
Reply to: Nathan Sidwell
nathan@acm.org / nathans@fb.com
Re: Working Draft, Extensions to C ++ for Modules, n4681

Module Names

Nathan Sidwell

The current wording of n4681 specifies that a module name is a dotted sequence of identifiers. It specifies no particular mapping between module names and the location of corresponding source code.

1 Background

The use of a dotted identifier sequence for module naming is attractive, at first glance. However, it leaves a number of behaviours unspecified. Implementations will have to define those behaviours, which can easily lead to implementation divergence. These are not minor items, but impact the development toolchain, module authors and module users. Authors and users will have to understand these implementation-defined behaviours for portability. It does no service to leave these unspecified.

1.1 Name Space Confusion

The dotted identifier nature causes new users to immediately make incorrect assumptions:

- That the module name is a namespace component, whereas it is instead orthogonal. One cannot explicitly look in a module '`foo`' by using a name prefixed with '`foo`'. One can also name a namespace '`foo`' in a module '`foo`', or a module importing '`foo`'.
- There is a hierarchical and sibling relationships between '`foo`', '`foo.bar`' and '`foo.baz`'. While the module developer may design these with such relationships, the language provides no such guarantee or requirement – module names are flattened.

I have been asked these questions by new users multiple times.

1.2 Locating Build Components

The fundamental problems are, given a module-name:

- how is the corresponding module interface source file located?
- how is the internal representation of that module interface located?

First, let us define some terms:

- *Module interface*. This is the source expressing a module interface definition.
- *Module implementation*. These are the sources expressing implementation components of a module.
- *Module interface binary*. This is the internally generated representation of a module interface. It contains the processed representation of the module interface. While the modules-ts does not describe this artifact, it is clearly an intended output of module system, as it is one main mechanism to achieve compilation performance.

The *module interface binary* is treated a little differently by in-development implementations:

- It may be a new, independent, output of compilation.
- It may be a new part of the existing assembler or object file output.
- It may be a new stage in the compilation pipeline.
- It may be a now non-transitory stage in an existing compilation pipeline.

These differences are not significant to this discussion. Regardless of its form, implementations have to solve the same questions discussed here. For brevity, but without loss of generality, this document presumes the first case.

The *module interface binary* is not intended as a distributable artifact. Firstly implementations are too immature to expect any kind of stability in format.[1] Secondly, compiler internals are vastly different, and enforcing some kind of common format (beyond the source code itself) is likely to be expensive to implement – and limit experimentation.

1.3 Header Files, Existing Practice

The current standard specifies that program source is provided in *source files:*

> 1 The text of the program is kept in units called *source files* in this International Standard
> [lex.separate,5.1]

Header inclusion is via '`#include`' directives, which name either a *header*, or a *source file*:

> 1 A `#include` directive shall identify a header or source file that can be processed by the implementation. … [cpp.include,19.2]

1 The GCC implementation locks the binary's version to the modification of the compiler's source. I understand the Microsoft compiler has a similarly volatile version numbering.

A `#include` directive of the form '`<name>`' names a header, and one of the form '`"name"`' names a source file. The standard presents little guidance on how the sequence of characters forming '`name`' are converted into a header name:

> 4 … The method by which a sequence of preprocessing tokens between a `<` and a `>` preprocessing token pair or a pair of `"` characters is combined into a single header name preprocessing token is implementation-defined. [cpp.include,19.2]

> 5 The implementation shall provide unique mappings for sequences consisting of one or more nondigits or digits (5.10) followed by a period (`.`) and a single nondigit. The first character shall not be a digit. The implementation may ignore distinctions of alphabetical case. [cpp.include.19.2]

However, implementations generally directly treat the sequence as a file name, without mapping. Thus the name cannot contain characters prohibited by the file system.

The set of places searched for the header or source file is implementation defined:

> 3 … The named source file is searched for in an implementation-defined manner. … [cpp.include,19.2]/

Implementations have converged on the concept of a *header search path*, a list of directories that are searched, in order, for an included header or source file. These search paths might be distinct for the two cases with a *user include path* and a *system include path*.

Include file names sometimes contain sub-directory components:

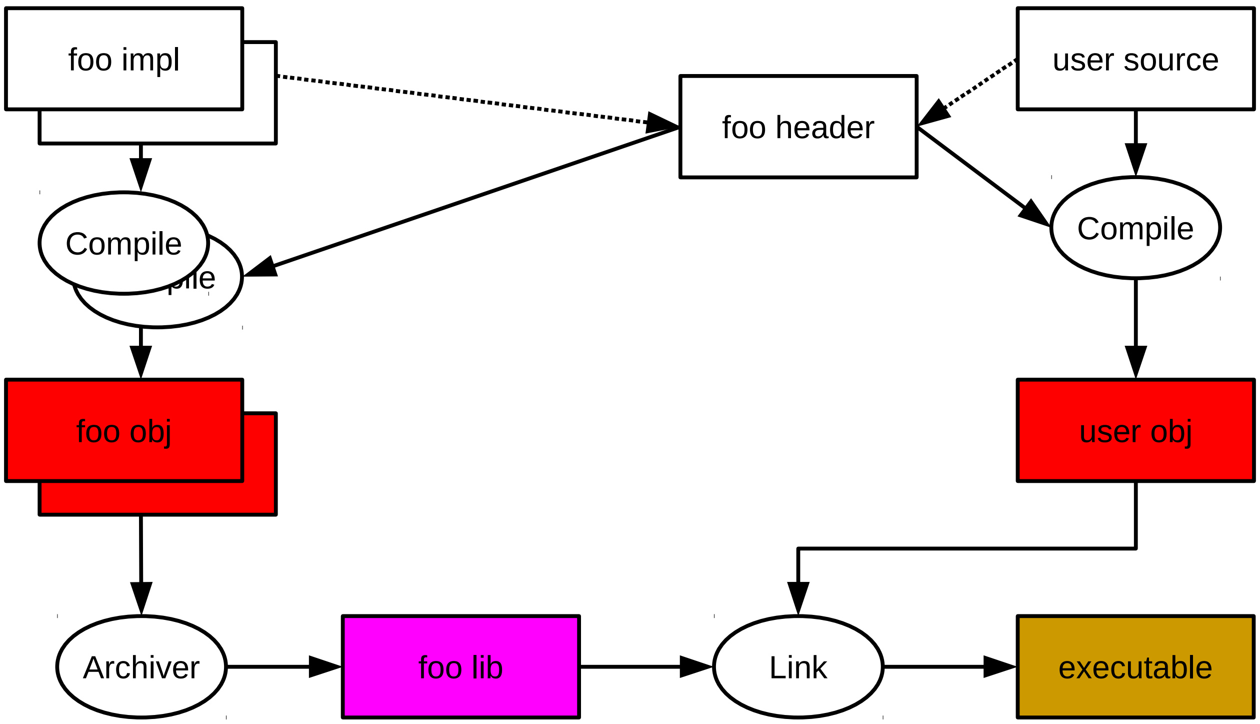
- Naming an internal implementation header file – for instance `"bits/errno.h"` – only intended to be included from the containing directory.
- Naming (components of) an optional library – for instance `"X11/Intrinsic.h"`.[2]

Semantics of such sub-director includes are well defined by the implementation.

Developers, by convention, refer to either case as *header inclusion*, and are generally aware that the `"name"` form first searches relative to the file containing the `#include`, and then both search using an include path. Compilers may have small differences in behaviour, but generally have a default system include path, and support command line options ('`-I`' or '`/I`') to extend the include path(s).

As files have names, developers have become used to equating the header file name with the header file itself – we do not see the abstraction and believe *ceci est une pipe*.

2 Neither practice is universal, internal headers may reside in the same directory, and sometimes the include path needs extending to mention the optional library directly.



Header-file Library

The diagram shows a conventional header-file style library. The library and user sources reference a header file, which is located via the compiler's include path. The library author distributes the compiled library '`libfoo.a`'[3] (unless they are providing complete library source, of course) and the header file. The user installs both into appropriate places in the file system and, perhaps, amends her compilation commands to extend the include (and library) paths.

1.4 Mapping Between Module Names and File Names

Module interfaces and implementations are provided in source files. The module interface binary will also be a file. These files have names.

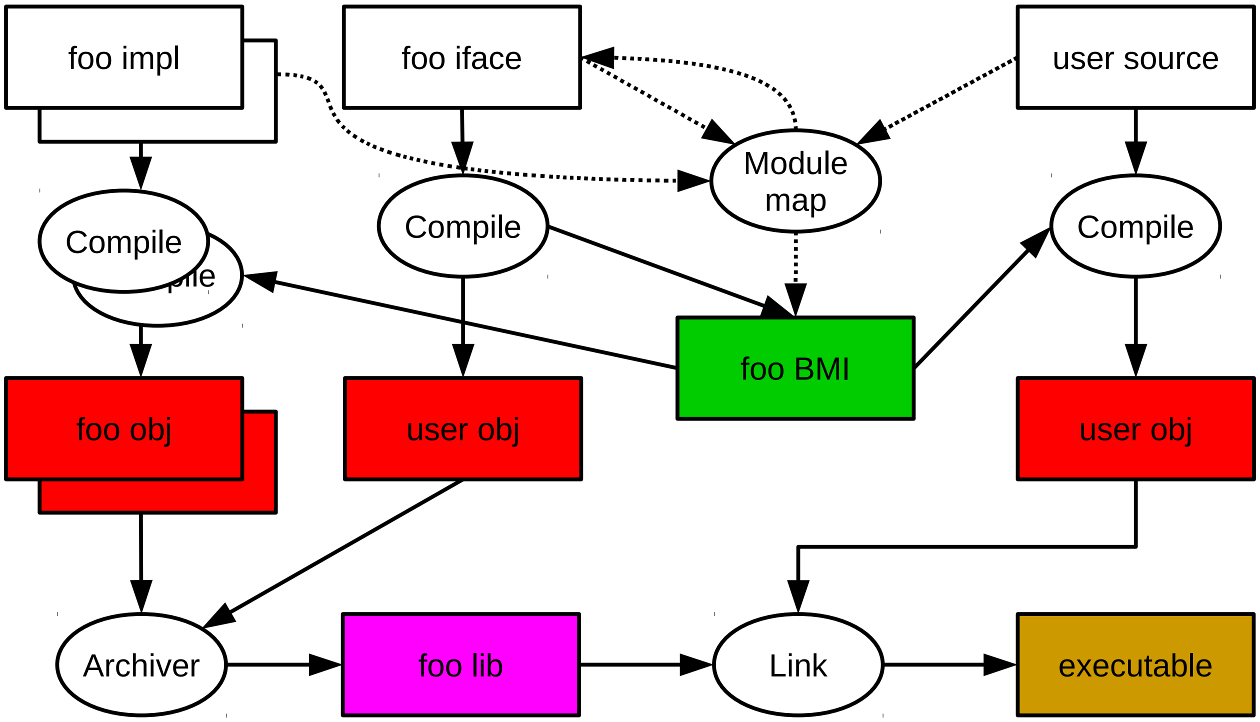
Compilation systems will need a way to map between module names and file names in the following cases:

- Compiling a module interface '`foo`', what is the file name to write the module interface binary to?
- Compiling source that imports a module '`foo`', what is the file name to search for the module interface binary?
- If there is no such binary available, what is the module interface source file name? (So that the module interface binary can be generated.)

3 Or `libfoo.so`. Note that libraries are conventionally named '`lib$name.$suffix`', and linkers use a library search path to find them, in a similar manner to how header files are found.

- Compiling a module implementation '`foo`' degenerates to the same questions.

Compilation systems already have a similar set of tasks. Often a library or executable is specified in a *makefile*,[4] and a set of required object file names are listed. As object files are compilation artifacts, the compilation system has to locate the corresponding source file for each required object file. The default behaviour is typically to search for a source file name of the form '`foo.$suffix`' when trying to build an object file named '`foo.o`'. The set of suffixes searched may be something like `{c,cc,C,cxx,f90}` and various more esoteric choices. The particular suffix located determines which language compiler to invoke. For brevity, let us just consider just '`cc`' suffix denoting a C++ source file. The mapping from object file name to corresponding source file name could clearly be more esoteric, perhaps adding or removing prefix characters, or converting characters such as capitalization. Some build systems do add additional prefixes or suffixes, but they do not alter the file name beyond optionally removing the suffix. We do not expect a source file '`FOO.cc`' to generate an object file '`foo.cc`'. File suffixes denote the type of a file and file base names denote the particular instance.



Module library

The diagram shows how a library might be provided as a module. Instead of a header file we have an interface file. Compiling the interface produces a regular object file, added to the library archive, and a binary module interface. A module map component informs the compilation system where the binary

4 I am describing a typical unix-like makefile environment with that platform's idioms for file suffixes. This does not restrict the generality of discussion. Other build environments are available. I will discuss more complicated distributed build systems in Section 1.4.1.

module interface is located. Notice too, that the module map has a mapping to the interface file. This is needed so that the binary module interface can be created on demand.

The module map component is not specified in the modules TS. Implementations shall have to define it.

Using dotted sequences of identifiers for module names presents challenges to the base-name & suffix idiom. It will be confusing to simply use that dotted sequence as a file base name. Dots are already used to separate the base-name from the suffix. File names with multiple dots are unusual – one cannot, in general, tell if the name has been stripped of a suffix. A case where multiple dots occur is when one container type wraps another container type, for instance '`foo.tar.gz`'. Such wrapping does not led to confusion, because the final suffix is removed when unwrapping from that particular container. Thus it is very likely that compilation systems will remap '`.`' characters in a module name to some other character. Which character to map to is not obvious – perhaps '`-`', perhaps '`/`'. A mapping to '`/`' will presumably create a directory hierarchy. But if the intent is for sub-directories, why is a directory separator not used directly? Whatever mapping is chosen, it is unlikely to satisfy all users, and to be portable all implementations will have to choose the same mapping.

The mapping from a module name to its interface source file name is similarly difficult. The modules-ts provides no guidance or restrictions on how the source file name might be related to the module name.

The difficulty in this mapping is so great that initial implementations eschew[5] it entirely and make it the user's problem. When compiling a module interface, the user provides a '`--module-name=-somename.suffix`' option, completely at their discretion. In order to import modules, the user provides a set of mappings from module name to module interface binary file name. This set of mappings must be the transitive set of imports, which can get very large. No help is given to the user in determining what this set might be, and it must be known before compilation begins. The problem of mapping a missing module interface binary file to a module interface source file is left entirely to the build system outside of the compiler.

This is will not help user adoption. Users will have to decide on their desired mappings, compilers (and other tools) will have different ways of describing these mappings and module authors may well diverge on their approach.

Finally users will have to internalize the mapping, because when importing a module '`foo`', they are likely to want to look at `foo`'s module interface source file to learn its API. While a build system could generate a mapping table, this is going to be awkward for the user to use without it being incorporated into their editing environment. It will be a tough requirement to meet, if modules are practically unusable until all development tools are updated.

5 The in-development GNU implementation currently punts with a deliberately stupid default choice and a FIXME comment.

The standard mechanism we have to map between different forms of the same entity is file name suffix replacement and the mechanism we choose for locating files is a search path. It would benefit modules, if such mechanisms were applicable.

1.4.1 Dependency Generation

Build systems are able to automatically generate dependency information, so that:

1. Artifacts are rebuilt whenever their inputs change
2. Input artifacts are built before items that depend on them are

With header file inclusion, the first problem is common, but the second problem is rare – synthesized header files are uncommon.

There is a fundamental difference between the two cases. For case 1, the first build doesn't need to know whether inputs have changed – we're building because the output artifact is missing. This rebuild will succeed, and as a side-effect could generate dependency information. Compilers often have options to emit such dependency information in Make or other compatible forms.

Case 2 is not the same, the input artifact must be built first. If it is missing dependency compilation will fail. Because it is rare, the programmer typically has to explicitly specify the dependency to the build system. This dependency does not need to be complete – remaining dependency information can be generated during the build in a similar manner to case 1.

Imports (and module implementations) are case 2 – the build fails if the binary module interface is unavailable. This dramatically changes the balance, and it will be unacceptable for the user to have to explicitly tell the build system of the dependencies. After all, the dependencies are already noted in the source file, just not visible outside the compiler.

Proposed techniques suggest the build system pre-process the source code to extract import declarations and construct dependency information from that. Double parsing the source code is sub-optimal. Complicated distributed build systems may have no choice but to do this already, but we should not force all build systems to do this if at all possible.

Perhaps it would be nice if a hook was provided so that the compiler could interrogate the build system when an import was needed. Depending on design, this might only be needed on first compilation (or on discovering a new import). That would allow rebuilds to be parallelized, even if the initial build had serialization points.

While not directly addressed by this paper, there clearly needs to be interaction between the compiler and the build system in general. Such an interface would be clearer, if it could use already understood entities, such as file names.

1.5 Example Hello World

Users new to modules are going to write something like a hello-world module. Here is an example:

```
// hello.cc
export module hello;
import std.ostream; // assume modularized STL
export void hello () {
  std::cout << "hello world!" << std::endl;
}

// main.cc
import hello;
int main () {
  hello ();
  return 0;
}
```

The new user will be put off if compilation and execution of this example is more complicated than something approximating:[6]

```
> cc-compiler -c -fmodules hello.cc
> cc-compiler -c -fmodules main.cc
> cc-compiler hello.o main.o
> ./a.out
hello world!
>
```

The mappings and options described above add complexity.

1.6 Other Languages

The modules-ts has been guided by other languages' module systems. Let us examine their approaches:[7]

Language	Module name form	Definition vs Implementation	Notes
ADA	identifier	Optionally multiple separate files	Module compilation is monolithic
Assembler	"filename"	Multiple implementation files	Flat namespace
C, C++	"filename" or <filename>	Multiple implementation	Via preprocessing, flat

6 The equivalent `#include` version would not need special flags for `main.cc` to find `hello.h`.
7 This is clearly not exhaustive, nor is it detailed.

Language	Module name form	Definition vs Implementation	Notes
		files	namespace
Fortran	identifier	Same file	
Java	dotted identifier sequence	Same file	Package name, co-opts DNS
Lisp	identifier	Same file	Package system
Modula-2	identifier	Pair of files	
Perl	identifier	Same file	Packages & modules
Python	dotted identifier sequence	Same file	Packages & module extend namespace hierarchy

I have included C, C++ and asssembler, because, although modules-ts intends add a module scheme to C++, we have an existing system for modular software. The table shows that the majority of languages that have module systems use either a single identifier, or a dotted sequence of them. The implementation column attempts to describe how an interface description is or is not separated from the implementation of that interface.

Let us examine this closer. For some (Java, Python), the module name is part of the regular naming hierarchy. One must use the module name to access the contents of the module (or use the equivalent of using directives and declarations). That is not true of the modules-ts.

Some of them (Java, Modula-2, Python, Perl at least), had a single initial implementation. That implementation set direction for later implementations, and effectively defined the mapping between module names and file names. The modules-ts does not define a mapping, and has multiple initial implementations in development.

The mapping between source & artifact files and module names is fixed in several implementation instances (Ada, Java, Modula-2, Perl, Python). The mapping is commonly that dots separate directory components and the final identifier is the basename. Several of these abstract the file system to a much greater degree than C++ has, and consequently implementations have different mechanisms mapping modules to files. The GNAT ADA compiler has a tight coupling of module and sub-component names to file names.

Many of these languages do not have a wide and varied software ecosystem with many different compilation tools, some do not have portable ABIs. Software repositories may be effectively singular, and an entire code base must be built by a single compilation system. Neither of these is true for C++ software, separate compilation permits compiled artifact distribution, and C++ has ABI stability that makes it practical. There are many C++ compilers available.

Thus, although the majority of other languages use identifiers for module names, the module-ts does not leverage the naming possibilities that permits. The existing system for C++ modular development does not use identifiers.

Several of these languages either enforce, or have the idiom of naming the source file from the module name. It might be worth considering enforcing such a scheme by eliding the name in locations where there is no ambiguity – rather than have to specify the same thing twice.

Many of the other languages either do not have a separate implementation source file, or have a 1:1 correspondence between interface description and implementation. Neither is true for the modules-ts. One idiom I have seen suggested is naming the implementation '`foo.mxx`' and its implementation '`foo.cxx`'. I think this shows a header-file mentality, copying the idiom '`foo.h`'/'`foo.cxx`'. However modules do not require that. The implementation can go directly into the interface file – it would be a QoI issue as to whether the module interface binary contained all the information in the interface file, or restricted it to just that needed for imports & module implementation units.[8] Larger modules may well want to split the implementation across several translation units, perhaps '`foo.mxx`', '`foo-engine.cxx`' & '`foo-user.cxx`'. This is an idiom that I do not think used in other module systems.[9]

2 Proposal

This paper proposes that the module name be a string constant, rather than a dotted identifier sequence.

The discussion below raises some questions is does not completely answer. But these are already implicitly being asked and at best unsatisfactorily answered as 'implementation issues'.

Such a change will dispel the confusion described in Section 1.1. There can be no ambiguity that a string-literal is part of name lookup, or denotes any compilation hierarchy relationship with other string literals.

A string module name will also make it clear that the intended use is as a base name, and suffix replacement becomes the mechanism to determine module interface source binary file names. A search path mechanism can be posited, and remain implementation defined. The questions arising in Section 1.2 have clear answers.

A user importing module `"foo"` can use her experience with header files, as described in Section 1.3 to locate module interface source file named `foo.cc`[10] using the module search path.

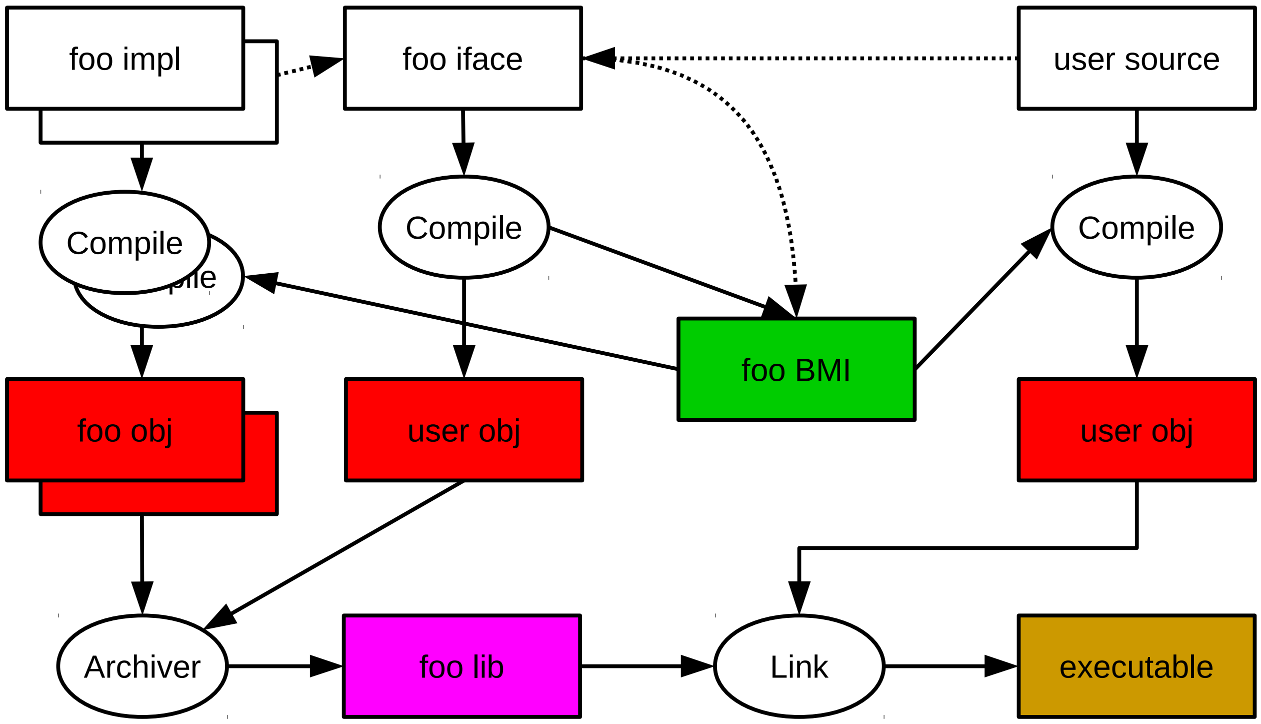
The requirement for the user to provide a transitive set of module name→file mappings is lost. The build system integration described in Section 1.4 becomes clearer using existing concepts.

8 Or, equally good, had sufficient indexing so that imports etc can skip data they do not require.
9 Beyond the trivial case of an include mechanism to concatenate the implementation components.
10 I use a '`.cc`' suffix as an example. As with a choice of suffixes for C++ source files, I imagine implementations will provide a set of suffixes (possibly the same set) for module interface source files.

The example in Section 1.5 should just work, no module mapping needs providing.

2.1 Module Names are Base Names

A clear restriction this adds is that the '`foo`' module interface source must reside in a file with base name '`foo`'. But, without this identity mapping, we are back to providing module maps. Providing a consistency removes doubt, and besides, why would you not give the file providing the '`foo`' interface a name unrelated to '`foo`'?



String-literal module names

With the use of strings as file base names, the module mapping becomes direct, and the mapping unit is no longer required.[11]

2.2 Directory Separators

Systems vary in their canonical directory separator character. Common choices are '`/`' and '`\`'. Systems can use plain '`\`' in `#include` directive names because the grammar gives implementations sufficient leeway about how to interpret the sequence of characters. They are h-char-sequences or q-char-sequences:

> 2 The appearance of either of the characters `'` or `\` or of either of the character sequences `/*` or `//` in a *q-char-sequence* or an *h-char-sequence* is conditionally-supported with

11 Pedantically there is still a mapping, it is 1:1 with suffix replacement using a search path – marginally more complex than existing header files.

> implementation-defined semantics, as is the appearance of the character `"` in an *h-char-sequence*. [lex.header,5.8]

Thus ‘`#include "X11\Intrinsic.h"`’ can be well formed on a suitable implementation.

If regular string-literals are used for module names, it will be disconcerting on such systems to have to write ‘`\\`’ as a separator. Some alternatives:

1. A special lexing grammar for string-like literals used as module names.
2. Specify a canonical directory separator, which is mapped to the system's separator.
3. Prohibit sub-directories in module names.

Option 1 requires context-sensitive lexing. A string literal after a module or import keyword would be lexed differently to other string-literals. Option 2 requires picking the canonical character, which may prove contentious. Option 3 seems to be giving up useful functionality.

2.3 Are Sub-modules Sub-directories?

As mentioned in Section 1.1, dotted identifier sequences are permitted and are intended for sub-modules. However, there is no hierarchy enforced – the sequences are in effect flat names permitted to contain an additional character, ‘`.`’.

If module names are strings, a different sub-classing character suggests itself – ‘`/`’. This would map to a directory separator in the underlying file system. Directories are a well known hierarchy, and easily navigable with conventional tools. Because the final component of a module name is a base-name, we can also use it as a directory component – the source file ‘`foo.cc`’ and the sub-directory ‘`foo`’ can reside in the same directory.

There is a potential point of confusion with using directory separators, and that is given an arbitrary path name, where does the module name component start? For instance:

```
> cc-compiler -c ../../src/path/foo/bob.cc
```

Is that a compilation of a module called ‘`bob`’, ‘`foo/bob`’, ‘`path/foo/bob`’ or any of the other possible trailing directories of the specified path name?[12] Also, by default, the usual output location of any artifacts is the current directory. If the module name has directory components, which is less surprising – emit artifacts in an sub-directory of current or emitting output in the current directory?

Related to this is the general difficulty toolchains can have with identically named files in different directories. For instance it can be impossible to specify breakpoints by file name and line number, if the

12 Java might prove instructive here, as it maps dotted package names to a directory tree. The compilation has the concept of ‘root’ directory.

file name is ambiguous.[13] Unfortunately there are likely to be many sub-module names ending with '`/internal`' or '`/core`'.

There is a conflict between the nicety of mapping sub-directories to sub-module and existing practice of outputting to the current directory and tooling defects for identically named files. I believe additional experimentation is needed here.

2.4 Implementation Note

Some in-development implementations use the module name as part of the mangling of module-linkage symbols. This is simplified by the use of identifiers, as they already have a mangling. This needs wrapping into a new mangling component denoting the module. The only extraneous character is '`.`', which Clang & G++ elide, by concatenating the manglings of the identifiers.[14]

String literals contain a much greater range of characters, but as the intent is for them to be used as file names, they should reside in the set applicable to file names. Typically that means they cannot have embedded `NUL` characters.

However, an encoding escape mechanism can easily be conceived, such that characters outside those permitted in mangled symbols are converted to a sequence that is conforming. Possibly as simple as reserving '`_`' as an escape, and encoding characters as '`_<enc>`' with a suitable definition of the `<enc>` encoding.

Thus, naming via string literal does not appear an implementation impediment.

3 Changes to Modules-TS Draft

The following changes make string-literals the naming scheme for modules, rather than dotted identifier sequences.

In [basic,6]/8 modify the new 6th bullet:

> – they are *module-names* whose *string-literals* are the same kind and encode ~~composed~~ of the same character sequence~~dotted sequence of *identifiers*~~.

Modify the grammar changes in [basic.link,6.5/1:

> *module-name*:
>   *string-literal*
>   ~~*module-name-qualifier-seq*$_{opt}$ *identifier*~~

13 DWARF line information provides 'directory' and 'file name' tables, which entrenches such confusion.
14 To be specific a module name '`foo.bar`' is mangled as '`W3foo3barE`', in the yet-to-be-formalized scheme.

~~*module-name-qualifier-seq*~~
~~*module-name-qualifier* .~~
~~*module-name-qualifier-seq identifier* .~~

~~*module-name-qualifier*~~
~~*identifier*~~

Modify [dcl.module,10.7] as follows:

1 A *module unit* is a translation unit that contains a *module-declaration*. A *named module* is the collection of module units with the same *module-name*. A translation unit may not contain more than one *module-declaration*. ~~A *module-name* has external linkage but cannot be found by name lookup.~~

Add the following paragraphs to [dcl.module,10.7]

7 A *module-name* is used to locate files (source or compilation artifacts) in an implementation-specific manner. [*Note:* It is expected that *module-names* are file base-names, with an optional directory component. With file suffix replacement used to locate the desired type of file (either source file or compilation artifact), using a search-path mechanism similar to that used for header file location. – *end note*]

Finally modify all examples to enclose the module name in double quotes (`"`) and replace any dots (`.`) with dashes (-) or directory separators (`/`):

- [basic.def,6.1]/2
- [basic.scope.namespace,6.3.6]/1
- [basic.lookup.argdep,6.4.2]/2
- [basic.namespace,10.3]/1
- [dcl.module.interface,10.7.1]/2
- [dcl.module.import,10.7.2]/1, 3, 4
- [temp.dep.res.,17.6.4]/2, 3