Clang 16.0.0git documentation

MODULES

Modules

- Introduction
  - Problems with the current model
  - Semantic import
  - Problems modules do not solve
- Using Modules
  - Standard C++ Modules
  - Objective-C Import declaration
  - Includes as imports
  - Module maps
  - Compilation model
  - Command-line parameters
    - -cc1 Options
  - Using Prebuilt Modules
- Module Semantics
  - Macros
- Module Map Language
  - Lexical structure
  - Module map file
  - Module declaration
    - Requires declaration
    - Header declaration
    - Umbrella directory declaration
    - Submodule declaration
    - Export declaration
    - Re-export Declaration
    - Use declaration
    - Link declaration
    - Configuration macros declaration
    - Conflict declarations
  - Attributes
  - Private Module Map Files
- Modularizing a Platform
- Future Directions
- Where To Learn More About Modules

Introduction

Most software is built using a number of software libraries, including libraries supplied by the platform, internal libraries built as part of the software itself to provide structure, and third-party libraries. For each library, one needs to access both its interface (API) and its implementation. In the C family of languages, the interface to a library is accessed by including the appropriate header files(s):

```
#include <SomeLib.h>
```

The implementation is handled separately by linking against the appropriate library. For example, by passing `-lSomeLib` to the linker.

Modules provide an alternative, simpler way to use software libraries that provides better compile-time scalability and eliminates many of the problems inherent to using the C preprocessor to access the API of a library.

Problems with the current model

The `#include` mechanism provided by the C preprocessor is a very poor way to access the API of a library, for a number of reasons:

- **Compile-time scalability**: Each time a header is included, the compiler must preprocess and parse the text in that header and every header it includes, transitively. This process must be repeated for every translation unit in the application, which involves a huge amount of redundant work. In a project with *N* translation units and *M* headers included in each translation unit, the compiler is performing *M* x *N* work even though most of the *M* headers are shared among multiple translation units. C++ is particularly bad, because the compilation model for templates forces a huge amount of code into headers.

- **Fragility**: `#include` directives are treated as textual inclusion by the preprocessor, and are therefore subject to any active macro definitions at the time of inclusion. If any of the active macro definitions happens to collide with a name in the library, it can break the library API or cause compilation failures in the library header itself. For an extreme example, `#define std "The C++ Standard"` and then include a standard library header: the result is a horrific cascade of failures in the C++ Standard Library's implementation. More subtle real-world problems occur when the headers for two different libraries interact due to macro collisions, and users are forced to reorder `#include` directives or introduce `#undef` directives to break the (unintended) dependency.
- **Conventional workarounds**: C programmers have adopted a number of conventions to work around the fragility of the C preprocessor model. Include guards, for example, are required for the vast majority of headers to ensure that multiple inclusion doesn't break the compile. Macro names are written with `LONG_PREFIXED_UPPERCASE_IDENTIFIERS` to avoid collisions, and some library/framework developers even use `__underscored` names in headers to avoid collisions with "normal" names that (by convention) shouldn't even be macros. These conventions are a barrier to entry for developers coming from non-C languages, are boilerplate for more experienced developers, and make our headers far uglier than they should be.
- **Tool confusion**: In a C-based language, it is hard to build tools that work well with software libraries, because the boundaries of the libraries are not clear. Which headers belong to a particular library, and in what order should those headers be included to guarantee that they compile correctly? Are the headers C, C++, Objective-C++, or one of the variants of these languages? What declarations in those headers are actually meant to be part of the API, and what declarations are present only because they had to be written as part of the header file?

Semantic import

Modules improve access to the API of software libraries by replacing the textual preprocessor inclusion model with a more robust, more efficient semantic model. From the user's perspective, the code looks only slightly different, because one uses an `import` declaration rather than a `#include` preprocessor directive:

```
import std.io; // pseudo-code; see below for syntax discussion
```

However, this module import behaves quite differently from the corresponding `#include <stdio.h>`: when the compiler sees the module import above, it loads a binary representation of the `std.io` module and makes its API available to the application directly. Preprocessor definitions that precede the import declaration have no impact on the API provided by `std.io`, because the module itself was compiled as a separate, standalone module. Additionally, any linker flags required to use the `std.io` module will automatically be provided when the module is imported [1] This semantic import model addresses many of the problems of the preprocessor inclusion model:

- **Compile-time scalability**: The `std.io` module is only compiled once, and importing the module into a translation unit is a constant-time operation (independent of module system). Thus, the API of each software library is only parsed once, reducing the *M x N* compilation problem to an *M + N* problem.
- **Fragility**: Each module is parsed as a standalone entity, so it has a consistent preprocessor environment. This completely eliminates the need for `__underscored` names and similarly defensive tricks. Moreover, the current preprocessor definitions when an import declaration is encountered are ignored, so one software library can not affect how another software library is compiled, eliminating include-order dependencies.
- **Tool confusion**: Modules describe the API of software libraries, and tools can reason about and present a module as a representation of that API. Because modules can only be built standalone, tools can rely on the module definition to ensure that they get the complete API for the library. Moreover, modules can specify which languages they work with, so, e.g., one can not accidentally attempt to load a C++ module into a C program.

Problems modules do not solve

Many programming languages have a module or package system, and because of the variety of features provided by these languages it is important to define what modules do *not* do. In particular, all of the following are considered out-of-scope for modules:

- **Rewrite the world's code**: It is not realistic to require applications or software libraries to make drastic or non-backward-compatible changes, nor is it feasible to completely eliminate headers. Modules must interoperate with existing software libraries and allow a gradual transition.
- **Versioning**: Modules have no notion of version information. Programmers must still rely on the existing versioning mechanisms of the underlying language (if any exist) to version software libraries.
- **Namespaces**: Unlike in some languages, modules do not imply any notion of namespaces. Thus, a struct declared in one module will still conflict with a struct of the same name declared in a different module, just as they would if declared in two different headers. This aspect is important for backward compatibility, because (for example) the mangled names of entities in software libraries must not change when introducing modules.
- **Binary distribution of modules**: Headers (particularly C++ headers) expose the full complexity of the language. Maintaining a stable binary module format across architectures, compiler versions, and compiler vendors is technically infeasible.

Using Modules

To enable modules, pass the command-line flag `-fmodules`. This will make any modules-enabled software libraries available as modules as well as introducing any modules-specific syntax. Additional **command-line parameters** are described in a separate section later.

Standard C++ Modules

Note

Modules are adopted into C++20 Standard. And its semantic and command line interface are very different from the Clang C++ modules. See **StandardCPlusPlusModules** for details.

Objective-C Import declaration

Objective-C provides syntax for importing a module via an *@import declaration*, which imports the named module:

```
@import std;
```

The `@import` declaration above imports the entire contents of the `std` module (which would contain, e.g., the entire C or C++ standard library) and make its API available within the current translation unit. To import only part of a module, one may use dot syntax to specific a particular submodule, e.g.,

```
@import std.io;
```

Redundant import declarations are ignored, and one is free to import modules at any point within the translation unit, so long as the import declaration is at global scope.

At present, there is no C or C++ syntax for import declarations. Clang will track the modules proposal in the C++ committee. See the section **Includes as imports** to see how modules get imported today.

Includes as imports

The primary user-level feature of modules is the import operation, which provides access to the API of software libraries. However, today's programs make extensive use of `#include`, and it is unrealistic to assume that all of this code will change overnight. Instead, modules automatically translate `#include` directives into the corresponding module import. For example, the include directive

```
#include <stdio.h>
```

will be automatically mapped to an import of the module `std.io`. Even with specific `import` syntax in the language, this particular feature is important for both adoption and backward compatibility: automatic translation of `#include` to `import` allows an application to get the benefits of modules (for all modules-enabled libraries) without any changes to the application itself. Thus, users can easily use modules with one compiler while falling back to the preprocessor-inclusion mechanism with other compilers.

Note

The automatic mapping of `#include` to `import` also solves an implementation problem: importing a module with a definition of some entity (say, a `struct Point`) and then parsing a header containing another definition of `struct Point` would cause a redefinition error, even if it is the same `struct Point`. By mapping `#include` to `import`, the compiler can guarantee that it always sees just the already-parsed definition from the module.

While building a module, `#include_next` is also supported, with one caveat. The usual behavior of `#include_next` is to search for the specified filename in the list of include paths, starting from the path *after* the one in which the current file was found. Because files listed in module maps are not found through include paths, a different strategy is used for `#include_next` directives in such files: the list of include paths is searched for the specified header name, to find the first include path that would refer to the current file. `#include_next` is interpreted as if the current file had been found in that path. If this search finds a file named by a module map, the `#include_next` directive is translated into an import, just like for a `#include` directive.``

Module maps

The crucial link between modules and headers is described by a *module map*, which describes how a collection of existing headers maps on to the (logical) structure of a module. For example, one could imagine a module `std` covering the C standard library. Each of the C standard library headers (`<stdio.h>`, `<stdlib.h>`, `<math.h>`, etc.) would contribute to the `std` module, by placing their respective APIs into the corresponding submodule (`std.io`, `std.lib`, `std.math`, etc.). Having a list of the headers that are part of the `std` module allows the compiler to build the `std` module as a standalone entity, and having the mapping from header names to (sub)modules allows the automatic translation of `#include` directives to module imports.

Module maps are specified as separate files (each named `module.modulemap`) alongside the headers they describe, which allows them to be added to existing software libraries without having to change the library headers themselves (in most cases **[2]**). The actual **Module map language** is described in a later section.

Note

To actually see any benefits from modules, one first has to introduce module maps for the underlying C standard library and the libraries and headers on which it depends. The section **Modularizing a Platform** describes the steps one must take to write these module maps.

One can use module maps without modules to check the integrity of the use of header files. To do this, use the `-fimplicit-module-maps` option instead of the `-fmodules` option, or use `-fmodule-map-file=` option to explicitly specify the module map files to load.

Compilation model

The binary representation of modules is automatically generated by the compiler on an as-needed basis. When a module is imported (e.g., by an `#include` of one of the module's headers), the compiler will spawn a second instance of itself **[3]**, with a fresh preprocessing context **[4]**, to parse just the headers in that module. The resulting Abstract Syntax Tree (AST) is then persisted into the binary representation of the module that is then loaded into translation unit where the module import was encountered.

The binary representation of modules is persisted in the *module cache*. Imports of a module will first query the module cache and, if a binary representation of the required module is already available, will load that representation directly. Thus, a module's headers will only be parsed once per language configuration, rather than once per translation unit that uses the module.

Modules maintain references to each of the headers that were part of the module build. If any of those headers changes, or if any of the modules on which a module depends change, then the module will be (automatically) recompiled. The process should never require any user intervention.

Command-line parameters

`-fmodules`
: Enable the modules feature.

`-fbuiltin-module-map`
: Load the Clang builtins module map file. (Equivalent to `-fmodule-map-file=<resource dir>/include/module.modulemap`)

`-fimplicit-module-maps`
: Enable implicit search for module map files named `module.modulemap` and similar. This option is implied by `-fmodules`. If this is disabled with `-fno-implicit-module-maps`, module map files will only be loaded if they are explicitly specified via `-fmodule-map-file` or transitively used by another module map file.

`-fmodules-cache-path=<directory>`
: Specify the path to the modules cache. If not provided, Clang will select a system-appropriate default.

`-fno-autolink`
: Disable automatic linking against the libraries associated with imported modules.

`-fmodules-ignore-macro=macroname`
: Instruct modules to ignore the named macro when selecting an appropriate module variant. Use this for macros defined on the command line that don't affect how modules are built, to improve sharing of compiled module files.

`-fmodules-prune-interval=seconds`
: Specify the minimum delay (in seconds) between attempts to prune the module cache. Module cache pruning attempts to clear out old, unused module files so that the module cache itself does not grow without bound. The default delay is large (604,800 seconds, or 7 days) because this is an expensive operation. Set this value to 0 to turn off pruning.

`-fmodules-prune-after=seconds`
: Specify the minimum time (in seconds) for which a file in the module cache must be unused (according to access time) before module pruning will remove it. The default delay is large (2,678,400 seconds, or 31 days) to avoid excessive module rebuilding.

`-module-file-info <module file name>`
: Debugging aid that prints information about a given module file (with a `.pcm` extension), including the language and preprocessor options that particular module variant was built with.

`-fmodules-decluse`
: Enable checking of module `use` declarations.

`-fmodule-name=module-id`
: Consider a source file as a part of the given module.

`-fmodule-map-file=<file>`
: Load the given module map file if a header from its directory or one of its subdirectories is loaded.

`-fmodules-search-all`
: If a symbol is not found, search modules referenced in the current module maps but not imported for symbols, so the error message can reference the module by name. Note that if the global module index has not been built before, this might take some time as it needs to build all the modules. Note that this option doesn't apply in module builds, to avoid the recursion.

`-fno-implicit-modules`
: All modules used by the build must be specified with `-fmodule-file`.

`-fmodule-file=[<name>=]<file>`
: Specify the mapping of module names to precompiled module files. If the name is omitted, then the module file is loaded whether actually required or not. If the name is specified, then the mapping is treated as another prebuilt module search mechanism (in addition to `-fprebuilt-module-path`) and the module is only loaded if required. Note that in this case the specified file also overrides this module's paths that might be embedded in other precompiled module files.

`-fprebuilt-module-path=<directory>`
: Specify the path to the prebuilt modules. If specified, we will look for modules in this directory for a given top-level module name. We don't need a module map for loading prebuilt modules in this directory and the compiler will not try to rebuild these modules.

This can be specified multiple times.

`-fprebuilt-implicit-modules`

Enable prebuilt implicit modules. If a prebuilt module is not found in the prebuilt modules paths (specified via `-fprebuilt-module-path`), we will look for a matching implicit module in the prebuilt modules paths.

-cc1 Options

`-fmodules-strict-context-hash`

Enables hashing of all compiler options that could impact the semantics of a module in an implicit build. This includes things such as header search paths and diagnostics. Using this option may lead to an excessive number of modules being built if the command line arguments are not homogeneous across your build.

Using Prebuilt Modules

Below are a few examples illustrating uses of prebuilt modules via the different options.

First, let's set up files for our examples.

```
/* A.h */
#ifdef ENABLE_A
void a() {}
#endif
```

```
/* B.h */
#include "A.h"
```

```
/* use.c */
#include "B.h"
void use() {
#ifdef ENABLE_A
  a();
#endif
}
```

```
/* module.modulemap */
module A {
  header "A.h"
}
module B {
  header "B.h"
  export *
}
```

In the examples below, the compilation of `use.c` can be done without `-cc1`, but the commands used to prebuild the modules would need to be updated to take into account the default options passed to `clang -cc1`. (See `clang use.c -v`) Note also that, since we use `-cc1`, we specify the `-fmodule-map-file=` or `-fimplicit-module-maps` options explicitly. When using the clang driver, `-fimplicit-module-maps` is implied by `-fmodules`.

First let us use an explicit mapping from modules to files.

```
rm -rf prebuilt ; mkdir prebuilt
clang -cc1 -emit-module -o prebuilt/A.pcm -fmodules module.modulemap -fmodule-name=A
clang -cc1 -emit-module -o prebuilt/B.pcm -fmodules module.modulemap -fmodule-name=B -fmodule-file=A=prebuil
clang -cc1 -emit-obj use.c -fmodules -fmodule-map-file=module.modulemap -fmodule-file=A=prebuilt/A.pcm -fmod
```

Instead of of specifying the mappings manually, it can be convenient to use the `-fprebuilt-module-path` option. Let's also use `-fimplicit-module-maps` instead of manually pointing to our module map.

```
rm -rf prebuilt; mkdir prebuilt
clang -cc1 -emit-module -o prebuilt/A.pcm -fmodules module.modulemap -fmodule-name=A
clang -cc1 -emit-module -o prebuilt/B.pcm -fmodules module.modulemap -fmodule-name=B -fprebuilt-module-path=
clang -cc1 -emit-obj use.c -fmodules -fimplicit-module-maps -fprebuilt-module-path=prebuilt
```

A trick to prebuild all modules required for our source file in one command is to generate implicit modules while using the `-fdisable-module-hash` option.

```
rm -rf prebuilt ; mkdir prebuilt
clang -cc1 -emit-obj use.c -fmodules -fimplicit-module-maps -fmodules-cache-path=prebuilt -fdisable-module-h
ls prebuilt/*.pcm
# prebuilt/A.pcm  prebuilt/B.pcm
```

Note that with explicit or prebuilt modules, we are responsible for, and should be particularly careful about the compatibility of our modules. Using mismatching compilation options and modules may lead to issues.

```
clang -cc1 -emit-obj use.c -fmodules -fimplicit-module-maps -fprebuilt-module-path=prebuilt -DENABLE_A
# use.c:4:10: warning: implicit declaration of function 'a' is invalid in C99 [-Wimplicit-function-declarati
#   return a(x);
#           ^
# 1 warning generated.
```

So we need to maintain multiple versions of prebuilt modules. We can do so using a manual module mapping, or pointing to a different prebuilt module cache path. For example:

```
rm -rf prebuilt ; mkdir prebuilt ; rm -rf prebuilt_a ; mkdir prebuilt_a
clang -cc1 -emit-obj use.c -fmodules -fimplicit-module-maps -fmodules-cache-path=prebuilt -fdisable-module-h
clang -cc1 -emit-obj use.c -fmodules -fimplicit-module-maps -fmodules-cache-path=prebuilt_a -fdisable-module
clang -cc1 -emit-obj use.c -fmodules -fimplicit-module-maps -fprebuilt-module-path=prebuilt
clang -cc1 -emit-obj use.c -fmodules -fimplicit-module-maps -fprebuilt-module-path=prebuilt_a -DENABLE_A
```

Instead of managing the different module versions manually, we can build implicit modules in a given cache path (using `-fmodules-cache-path`), and reuse them as prebuilt implicit modules by passing `-fprebuilt-module-path` and `-fprebuilt-implicit-modules`.

```
rm -rf prebuilt; mkdir prebuilt
clang -cc1 -emit-obj -o use.o use.c -fmodules -fimplicit-module-maps -fmodules-cache-path=prebuilt
clang -cc1 -emit-obj -o use.o use.c -fmodules -fimplicit-module-maps -fmodules-cache-path=prebuilt -DENABLE_
find prebuilt -name "*.pcm"
# prebuilt/1AYBIGPM8R2GA/A-3L1K4LUA6O31.pcm
# prebuilt/1AYBIGPM8R2GA/B-3L1K4LUA6O31.pcm
# prebuilt/VH0YZMF1OIRK/A-3L1K4LUA6O31.pcm
# prebuilt/VH0YZMF1OIRK/B-3L1K4LUA6O31.pcm
clang -cc1 -emit-obj -o use.o use.c -fmodules -fimplicit-module-maps -fprebuilt-module-path=prebuilt -fprebu
clang -cc1 -emit-obj -o use.o use.c -fmodules -fimplicit-module-maps -fprebuilt-module-path=prebuilt -fprebu
```

Finally we want to allow implicit modules for configurations that were not prebuilt. When using the clang driver a module cache path is implicitly selected. Using `-cc1`, we simply add use the `-fmodules-cache-path` option.

```
clang -cc1 -emit-obj -o use.o use.c -fmodules -fimplicit-module-maps -fprebuilt-module-path=prebuilt -fprebu
clang -cc1 -emit-obj -o use.o use.c -fmodules -fimplicit-module-maps -fprebuilt-module-path=prebuilt -fprebu
clang -cc1 -emit-obj -o use.o use.c -fmodules -fimplicit-module-maps -fprebuilt-module-path=prebuilt -fprebu
```

This way, a single directory containing multiple variants of modules can be prepared and reused. The options configuring the module cache are independent of other options.

Module Semantics

Modules are modeled as if each submodule were a separate translation unit, and a module import makes names from the other translation unit visible. Each submodule starts with a new preprocessor state and an empty translation unit.

Note

This behavior is currently only approximated when building a module with submodules. Entities within a submodule that has already been built are visible when building later submodules in that module. This can lead to fragile modules that depend on the build order used for the submodules of the module, and should not be relied upon. This behavior is subject to change.

As an example, in C, this implies that if two structs are defined in different submodules with the same name, those two types are distinct types (but may be *compatible* types if their definitions match). In C++, two structs defined with the same name in different submodules are the *same* type, and must be equivalent under C++'s One Definition Rule.

Note

Clang currently only performs minimal checking for violations of the One Definition Rule.

If any submodule of a module is imported into any part of a program, the entire top-level module is considered to be part of the program. As a consequence of this, Clang may diagnose conflicts between an entity declared in an unimported submodule and an entity declared in the current translation unit, and Clang may inline or devirtualize based on knowledge from unimported submodules.

Macros

The C and C++ preprocessor assumes that the input text is a single linear buffer, but with modules this is not the case. It is possible to import two modules that have conflicting definitions for a macro (or where one `#define`s a macro and the other `#undef`ines it). The rules for handling macro definitions in the presence of modules are as follows:

- Each definition and undefinition of a macro is considered to be a distinct entity.
- Such entities are *visible* if they are from the current submodule or translation unit, or if they were exported from a submodule that has been imported.
- A `#define X` or `#undef X` directive *overrides* all definitions of `X` that are visible at the point of the directive.
- A `#define` or `#undef` directive is *active* if it is visible and no visible directive overrides it.

- A set of macro directives is *consistent* if it consists of only `#undef` directives, or if all `#define` directives in the set define the macro name to the same sequence of tokens (following the usual rules for macro redefinitions).
- If a macro name is used and the set of active directives is not consistent, the program is ill-formed. Otherwise, the (unique) meaning of the macro name is used.

For example, suppose:

- `<stdio.h>` defines a macro `getc` (and exports its `#define`)
- `<cstdio>` imports the `<stdio.h>` module and undefines the macro (and exports its `#undef`)

The `#undef` overrides the `#define`, and a source file that imports both modules *in any order* will not see `getc` defined as a macro.

Module Map Language

Warning

The module map language is not currently guaranteed to be stable between major revisions of Clang.

The module map language describes the mapping from header files to the logical structure of modules. To enable support for using a library as a module, one must write a `module.modulemap` file for that library. The `module.modulemap` file is placed alongside the header files themselves, and is written in the module map language described below.

Note

For compatibility with previous releases, if a module map file named `module.modulemap` is not found, Clang will also search for a file named `module.map`. This behavior is deprecated and we plan to eventually remove it.

As an example, the module map file for the C standard library might look a bit like this:

```
module std [system] [extern_c] {
  module assert {
    textual header "assert.h"
    header "bits/assert-decls.h"
    export *
  }

  module complex {
    header "complex.h"
    export *
  }

  module ctype {
    header "ctype.h"
    export *
  }

  module errno {
    header "errno.h"
    header "sys/errno.h"
    export *
  }

  module fenv {
    header "fenv.h"
    export *
  }

  // ...more headers follow...
}
```

Here, the top-level module `std` encompasses the whole C standard library. It has a number of submodules containing different parts of the standard library: `complex` for complex numbers, `ctype` for character types, etc. Each submodule lists one of more headers that provide the contents for that submodule. Finally, the `export *` command specifies that anything included by that submodule will be automatically re-exported.

Lexical structure

Module map files use a simplified form of the C99 lexer, with the same rules for identifiers, tokens, string literals, `/* */` and `//` comments. The module map language has the following reserved words; all other C identifiers are valid identifiers.

```
config_macros export_as   private
conflict       framework  requires
exclude        header      textual
explicit       link        umbrella
extern         module      use
export
```

Module map file

A module map file consists of a series of module declarations:

```
module-map-file:
  module-declaration*
```

Within a module map file, modules are referred to by a *module-id*, which uses periods to separate each part of a module's name:

```
module-id:
  identifier ('.' identifier)*
```

Module declaration

A module declaration describes a module, including the headers that contribute to that module, its submodules, and other aspects of the module.

```
module-declaration:
  explicitopt frameworkopt module module-id attributesopt '{' module-member* '}'
  extern module module-id string-literal
```

The *module-id* should consist of only a single *identifier*, which provides the name of the module being defined. Each module shall have a single definition.

The `explicit` qualifier can only be applied to a submodule, i.e., a module that is nested within another module. The contents of explicit submodules are only made available when the submodule itself was explicitly named in an import declaration or was re-exported from an imported module.

The `framework` qualifier specifies that this module corresponds to a Darwin-style framework. A Darwin-style framework (used primarily on macOS and iOS) is contained entirely in directory `Name.framework`, where `Name` is the name of the framework (and, therefore, the name of the module). That directory has the following layout:

```
Name.framework/
  Modules/module.modulemap  Module map for the framework
  Headers/                  Subdirectory containing framework headers
  PrivateHeaders/           Subdirectory containing framework private headers
  Frameworks/               Subdirectory containing embedded frameworks
  Resources/                Subdirectory containing additional resources
  Name                      Symbolic link to the shared library for the framework
```

The `system` attribute specifies that the module is a system module. When a system module is rebuilt, all of the module's headers will be considered system headers, which suppresses warnings. This is equivalent to placing `#pragma GCC system_header` in each of the module's headers. The form of attributes is described in the section **Attributes**, below.

The `extern_c` attribute specifies that the module contains C code that can be used from within C++. When such a module is built for use in C++ code, all of the module's headers will be treated as if they were contained within an implicit `extern "C"` block. An import for a module with this attribute can appear within an `extern "C"` block. No other restrictions are lifted, however: the module currently cannot be imported within an `extern "C"` block in a namespace.

The `no_undeclared_includes` attribute specifies that the module can only reach non-modular headers and headers from used modules. Since some headers could be present in more than one search path and map to different modules in each path, this mechanism helps clang to find the right header, i.e., prefer the one for the current module or in a submodule instead of the first usual match in the search paths.

Modules can have a number of different kinds of members, each of which is described below:

```
module-member:
  requires-declaration
  header-declaration
  umbrella-dir-declaration
  submodule-declaration
  export-declaration
  export-as-declaration
  use-declaration
  link-declaration
  config-macros-declaration
  conflict-declaration
```

An extern module references a module defined by the *module-id* in a file given by the *string-literal*. The file can be referenced either by an absolute path or by a path relative to the current map file.

Requires declaration

A *requires-declaration* specifies the requirements that an importing translation unit must satisfy to use the module.

```
requires-declaration:
  requires feature-list

feature-list:
  feature (',' feature)*

feature:
  !opt identifier
```

The requirements clause allows specific modules or submodules to specify that they are only accessible with certain language dialects, platforms, environments and target specific features. The feature list is a set of identifiers, defined below. If any of the features is not available in a given translation unit, that translation unit shall not import the module. When building a module for use by a compilation, submodules requiring unavailable features are ignored. The optional `!` indicates that a feature is incompatible with the module.

The following features are defined:

altivec
: The target supports AltiVec.

blocks
: The “blocks” language feature is available.

coroutines
: Support for the coroutines TS is available.

cplusplus
: C++ support is available.

cplusplus11
: C++11 support is available.

cplusplus14
: C++14 support is available.

cplusplus17
: C++17 support is available.

c99
: C99 support is available.

c11
: C11 support is available.

c17
: C17 support is available.

freestanding
: A freestanding environment is available.

gnuinlineasm
: GNU inline ASM is available.

objc
: Objective-C support is available.

objc_arc
: Objective-C Automatic Reference Counting (ARC) is available

opencl
: OpenCL is available

tls
: Thread local storage is available.

target feature
: A specific target feature (e.g., `sse4`, `avx`, `neon`) is available.

platform/os
: A os/platform variant (e.g. `freebsd`, `win32`, `windows`, `linux`, `ios`, `macos`, `iossimulator`) is available.

environment
: A environment variant (e.g. `gnu`, `gnueabi`, `android`, `msvc`) is available.

Example: The `std` module can be extended to also include C++ and C++11 headers using a *requires-declaration*:

```
module std {
   // C standard library...

   module vector {
     requires cplusplus
     header "vector"
   }

   module type_traits {
     requires cplusplus11
     header "type_traits"
   }
 }
```

Header declaration

A header declaration specifies that a particular header is associated with the enclosing module.

```
header-declaration:
  privateopt textualopt header string-literal header-attrsopt
  umbrella header string-literal header-attrsopt
  exclude header string-literal header-attrsopt

header-attrs:
  '{' header-attr* '}'

header-attr:
  size integer-literal
  mtime integer-literal
```

A header declaration that does not contain `exclude` nor `textual` specifies a header that contributes to the enclosing module. Specifically, when the module is built, the named header will be parsed and its declarations will be (logically) placed into the enclosing submodule.

A header with the `umbrella` specifier is called an umbrella header. An umbrella header includes all of the headers within its directory (and any subdirectories), and is typically used (in the `#include` world) to easily access the full API provided by a particular library. With modules, an umbrella header is a convenient shortcut that eliminates the need to write out `header` declarations for every library header. A given directory can only contain a single umbrella header.

Note

Any headers not included by the umbrella header should have explicit `header` declarations. Use the `-Wincomplete-umbrella` warning option to ask Clang to complain about headers not covered by the umbrella header or the module map.

A header with the `private` specifier may not be included from outside the module itself.

A header with the `textual` specifier will not be compiled when the module is built, and will be textually included if it is named by a `#include` directive. However, it is considered to be part of the module for the purpose of checking *use-declaration*s, and must still be a lexically-valid header file. In the future, we intend to pre-tokenize such headers and include the token sequence within the prebuilt module representation.

A header with the `exclude` specifier is excluded from the module. It will not be included when the module is built, nor will it be considered to be part of the module, even if an `umbrella` header or directory would otherwise make it part of the module.

Example: The C header `assert.h` is an excellent candidate for a textual header, because it is meant to be included multiple times (possibly with different `NDEBUG` settings). However, declarations within it should typically be split into a separate modular header.

```
module std [system] {
  textual header "assert.h"
}
```

A given header shall not be referenced by more than one *header-declaration*.

Two *header-declaration*s, or a *header-declaration* and a `#include`, are considered to refer to the same file if the paths resolve to the same file and the specified *header-attr*s (if any) match the attributes of that file, even if the file is named differently (for instance, by a relative path or via symlinks).

Note

The use of *header-attr*s avoids the need for Clang to speculatively `stat` every header referenced by a module map. It is recommended that *header-attr*s only be used in machine-generated module maps, to avoid mismatches between attribute values and the corresponding files.

Umbrella directory declaration

An umbrella directory declaration specifies that all of the headers in the specified directory should be included within the module.

```
umbrella-dir-declaration:
  umbrella string-literal
```

The *string-literal* refers to a directory. When the module is built, all of the header files in that directory (and its subdirectories) are included in the module.

An *umbrella-dir-declaration* shall not refer to the same directory as the location of an umbrella *header-declaration*. In other words, only a single kind of umbrella can be specified for a given directory.

Note

Umbrella directories are useful for libraries that have a large number of headers but do not have an umbrella header.

Submodule declaration

Submodule declarations describe modules that are nested within their enclosing module.

```
submodule-declaration:
  module-declaration
  inferred-submodule-declaration
```

A *submodule-declaration* that is a *module-declaration* is a nested module. If the *module-declaration* has a `framework` specifier, the enclosing module shall have a `framework` specifier; the submodule's contents shall be contained within the subdirectory `Frameworks/SubName.framework`, where `SubName` is the name of the submodule.

A *submodule-declaration* that is an *inferred-submodule-declaration* describes a set of submodules that correspond to any headers that are part of the module but are not explicitly described by a *header-declaration*.

```
inferred-submodule-declaration:
  explicitopt frameworkopt module '*' attributesopt '{' inferred-submodule-member* '}'

inferred-submodule-member:
  export '*'
```

A module containing an *inferred-submodule-declaration* shall have either an umbrella header or an umbrella directory. The headers to which the *inferred-submodule-declaration* applies are exactly those headers included by the umbrella header (transitively) or included in the module because they reside within the umbrella directory (or its subdirectories).

For each header included by the umbrella header or in the umbrella directory that is not named by a *header-declaration*, a module declaration is implicitly generated from the *inferred-submodule-declaration*. The module will:

- Have the same name as the header (without the file extension)
- Have the `explicit` specifier, if the *inferred-submodule-declaration* has the `explicit` specifier
- Have the `framework` specifier, if the *inferred-submodule-declaration* has the `framework` specifier
- Have the attributes specified by the *inferred-submodule-declaration*
- Contain a single *header-declaration* naming that header
- Contain a single *export-declaration* `export *`, if the *inferred-submodule-declaration* contains the *inferred-submodule-member* `export *`

Example: If the subdirectory "MyLib" contains the headers `A.h` and `B.h`, then the following module map:

```
module MyLib {
  umbrella "MyLib"
  explicit module * {
    export *
  }
}
```

is equivalent to the (more verbose) module map:

```
module MyLib {
  explicit module A {
    header "A.h"
    export *
  }

  explicit module B {
    header "B.h"
    export *
  }
}
```

Export declaration

An *export-declaration* specifies which imported modules will automatically be re-exported as part of a given module's API.

```
export-declaration:
  export wildcard-module-id

wildcard-module-id:
  identifier
  '*'
  identifier '.' wildcard-module-id
```

The *export-declaration* names a module or a set of modules that will be re-exported to any translation unit that imports the enclosing module. Each imported module that matches the *wildcard-module-id* up to, but not including, the first `*` will be re-exported.

Example: In the following example, importing `MyLib.Derived` also provides the API for `MyLib.Base`:

```
module MyLib {
  module Base {
    header "Base.h"
  }

  module Derived {
    header "Derived.h"
    export Base
  }
}
```

Note that, if `Derived.h` includes `Base.h`, one can simply use a wildcard export to re-export everything `Derived.h` includes:

```
module MyLib {
  module Base {
    header "Base.h"
  }

  module Derived {
    header "Derived.h"
    export *
  }
}
```

Note

The wildcard export syntax `export *` re-exports all of the modules that were imported in the actual header file. Because `#include` directives are automatically mapped to module imports, `export *` provides the same transitive-inclusion behavior provided by the C preprocessor, e.g., importing a given module implicitly imports all of the modules on which it depends. Therefore, liberal use of `export *` provides excellent backward compatibility for programs that rely on transitive inclusion (i.e., all of them).

Re-export Declaration

An *export-as-declaration* specifies that the current module will have its interface re-exported by the named module.

```
export-as-declaration:
  export_as identifier
```

The *export-as-declaration* names the module that the current module will be re-exported through. Only top-level modules can be re-exported, and any given module may only be re-exported through a single module.

Example: In the following example, the module `MyFrameworkCore` will be re-exported via the module `MyFramework`:

```
module MyFrameworkCore {
  export_as MyFramework
}
```

Use declaration

A *use-declaration* specifies another module that the current top-level module intends to use. When the option *-fmodules-decluse* is specified, a module can only use other modules that are explicitly specified in this way.

```
use-declaration:
  use module-id
```

Example: In the following example, use of A from C is not declared, so will trigger a warning.

```
module A {
  header "a.h"
}

module B {
  header "b.h"
}

module C {
  header "c.h"
  use B
}
```

When compiling a source file that implements a module, use the option `-fmodule-name=module-id` to indicate that the source file is logically part of that module.

The compiler at present only applies restrictions to the module directly being built.

Link declaration

A *link-declaration* specifies a library or framework against which a program should be linked if the enclosing module is imported in any translation unit in that program.

```
link-declaration:
  link frameworkopt string-literal
```

The *string-literal* specifies the name of the library or framework against which the program should be linked. For example, specifying "clangBasic" would instruct the linker to link with `-lclangBasic` for a Unix-style linker.

A *link-declaration* with the `framework` specifies that the linker should link against the named framework, e.g., with `-framework MyFramework`.

Note

Automatic linking with the `link` directive is not yet widely implemented, because it requires support from both the object file format and the linker. The notion is similar to Microsoft Visual Studio's `#pragma comment(lib...)`.

Configuration macros declaration

The *config-macros-declaration* specifies the set of configuration macros that have an effect on the API of the enclosing module.

```
config-macros-declaration:
  config_macros attributesopt config-macro-listopt

config-macro-list:
  identifier (',' identifier)*
```

Each *identifier* in the *config-macro-list* specifies the name of a macro. The compiler is required to maintain different variants of the given module for differing definitions of any of the named macros.

A *config-macros-declaration* shall only be present on a top-level module, i.e., a module that is not nested within an enclosing module.

The `exhaustive` attribute specifies that the list of macros in the *config-macros-declaration* is exhaustive, meaning that no other macro definition is intended to have an effect on the API of that module.

Note

The `exhaustive` attribute implies that any macro definitions for macros not listed as configuration macros should be ignored completely when building the module. As an optimization, the compiler could reduce the number of unique module variants by not considering these non-configuration macros. This optimization is not yet implemented in Clang.

A translation unit shall not import the same module under different definitions of the configuration macros.

Note

Clang implements a weak form of this requirement: the definitions used for configuration macros are fixed based on the definitions provided by the command line. If an import occurs and the definition of any configuration macro has changed, the compiler will produce a warning (under the control of `-Wconfig-macros`).

Example: A logging library might provide different API (e.g., in the form of different definitions for a logging macro) based on the `NDEBUG` macro setting:

```
module MyLogger {
  umbrella header "MyLogger.h"
  config_macros [exhaustive] NDEBUG
}
```

Conflict declarations

A *conflict-declaration* describes a case where the presence of two different modules in the same translation unit is likely to cause a problem. For example, two modules may provide similar-but-incompatible functionality.

```
conflict-declaration:
  conflict module-id ',' string-literal
```

The *module-id* of the *conflict-declaration* specifies the module with which the enclosing module conflicts. The specified module shall not have been imported in the translation unit when the enclosing module is imported.

The *string-literal* provides a message to be provided as part of the compiler diagnostic when two modules conflict.

Note

Clang emits a warning (under the control of `-Wmodule-conflict`) when a module conflict is discovered.

Example:

```
module Conflicts {
  explicit module A {
    header "conflict_a.h"
    conflict B, "we just don't like B"
  }

  module B {
    header "conflict_b.h"
  }
}
```

Attributes

Attributes are used in a number of places in the grammar to describe specific behavior of other declarations. The format of attributes is fairly simple.

```
attributes:
  attribute attributesopt

attribute:
  '[' identifier ']'
```

Any *identifier* can be used as an attribute, and each declaration specifies what attributes can be applied to it.

Private Module Map Files

Module map files are typically named `module.modulemap` and live either alongside the headers they describe or in a parent directory of the headers they describe. These module maps typically describe all of the API for the library.

However, in some cases, the presence or absence of particular headers is used to distinguish between the “public” and “private” APIs of a particular library. For example, a library may contain the headers `Foo.h` and `Foo_Private.h`, providing public and private APIs, respectively. Additionally, `Foo_Private.h` may only be available on some versions of library, and absent in others. One cannot easily express this with a single module map file in the library:

```
module Foo {
  header "Foo.h"
  ...
}

module Foo_Private {
  header "Foo_Private.h"
  ...
}
```

because the header `Foo_Private.h` won’t always be available. The module map file could be customized based on whether `Foo_Private.h` is available or not, but doing so requires custom build machinery.

Private module map files, which are named `module.private.modulemap` (or, for backward compatibility, `module_private.map`), allow one to augment the primary module map file with an additional modules. For example, we would split the module map file above into two module map files:

```
/* module.modulemap */
module Foo {
  header "Foo.h"
}

/* module.private.modulemap */
module Foo_Private {
  header "Foo_Private.h"
}
```

When a `module.private.modulemap` file is found alongside a `module.modulemap` file, it is loaded after the `module.modulemap` file. In our example library, the `module.private.modulemap` file would be available when `Foo_Private.h` is available, making it easier to split a library’s public and private APIs along header boundaries.

When writing a private module as part of a *framework*, it’s recommended that:

- Headers for this module are present in the `PrivateHeaders` framework subdirectory.
- The private module is defined as a *top level module* with the name of the public framework prefixed, like `Foo_Private` above. Clang has extra logic to work with this naming, using `FooPrivate` or `Foo.Private` (submodule) trigger warnings and might not work as expected.

Modularizing a Platform

To get any benefit out of modules, one needs to introduce module maps for software libraries starting at the bottom of the stack. This typically means introducing a module map covering the operating system’s headers and the C standard library headers (in `/usr/include`, for a Unix system).

The module maps will be written using the **module map language**, which provides the tools necessary to describe the mapping between headers and modules. Because the set of headers differs from one system to the next, the module map will likely have to be somewhat customized for, e.g., a particular distribution and version of the operating system. Moreover, the system headers themselves may require some modification, if they exhibit any anti-patterns that break modules. Such common patterns are described below.

Macro-guarded copy-and-pasted definitions

System headers vend core types such as `size_t` for users. These types are often needed in a number of system headers, and are almost trivial to write. Hence, it is fairly common to see a definition such as the following copy-and-pasted throughout the headers:

```
#ifndef _SIZE_T
#define _SIZE_T
typedef __SIZE_TYPE__ size_t;
#endif
```

Unfortunately, when modules compiles all of the C library headers together into a single module, only the first actual type definition of `size_t` will be visible, and then only in the submodule corresponding to the lucky first header. Any other headers that have copy-and-pasted versions of this pattern will *not* have a definition of `size_t`. Importing the submodule corresponding to one of those headers will therefore not yield `size_t` as part of the API, because it wasn't there when the header was parsed. The fix for this problem is either to pull the copied declarations into a common header that gets included everywhere `size_t` is part of the API, or to eliminate the `#ifndef` and redefine the `size_t` type. The latter works for C++ headers and C11, but will cause an error for non-modules C90/C99, where redefinition of `typedefs` is not permitted.

Conflicting definitions

Different system headers may provide conflicting definitions for various macros, functions, or types. These conflicting definitions don't tend to cause problems in a pre-modules world unless someone happens to include both headers in one translation unit. Since the fix is often simply "don't do that", such problems persist. Modules requires that the conflicting definitions be eliminated or that they be placed in separate modules (the former is generally the better answer).

Missing includes

Headers are often missing `#include` directives for headers that they actually depend on. As with the problem of conflicting definitions, this only affects unlucky users who don't happen to include headers in the right order. With modules, the headers of a particular module will be parsed in isolation, so the module may fail to build if there are missing includes.

Headers that vend multiple APIs at different times

Some systems have headers that contain a number of different kinds of API definitions, only some of which are made available with a given include. For example, the header may vend `size_t` only when the macro `__need_size_t` is defined before that header is included, and also vend `wchar_t` only when the macro `__need_wchar_t` is defined. Such headers are often included many times in a single translation unit, and will have no include guards. There is no sane way to map this header to a submodule. One can either eliminate the header (e.g., by splitting it into separate headers, one per actual API) or simply `exclude` it in the module map.

To detect and help address some of these problems, the `clang-tools-extra` repository contains a `modularize` tool that parses a set of given headers and attempts to detect these problems and produce a report. See the tool's in-source documentation for information on how to check your system or library headers.

Future Directions

Modules support is under active development, and there are many opportunities remaining to improve it. Here are a few ideas:

Detect unused module imports

Unlike with `#include` directives, it should be fairly simple to track whether a directly-imported module has ever been used. By doing so, Clang can emit `unused import` or `unused #include` diagnostics, including Fix-Its to remove the useless imports/includes.

Fix-Its for missing imports

It's fairly common for one to make use of some API while writing code, only to get a compiler error about "unknown type" or "no function named" because the corresponding header has not been included. Clang can detect such cases and auto-import the required module, but should provide a Fix-It to add the import.

Improve modularize

The modularize tool is both extremely important (for deployment) and extremely crude. It needs better UI, better detection of problems (especially for C++), and perhaps an assistant mode to help write module maps for you.

Where To Learn More About Modules

The Clang source code provides additional information about modules:

`clang/lib/Headers/module.modulemap`

Module map for Clang's compiler-specific header files.

`clang/test/Modules/`

Tests specifically related to modules functionality.

`clang/include/clang/Basic/Module.h`

The `Module` class in this header describes a module, and is used throughout the compiler to implement modules.

`clang/include/clang/Lex/ModuleMap.h`

The `ModuleMap` class in this header describes the full module map, consisting of all of the module map files that have been parsed, and providing facilities for looking up module maps and mapping between modules and headers (in both directions).

PCHInternals

Information about the serialized AST format used for precompiled headers and modules. The actual implementation is in the `clangSerialization` library.

[1] Automatic linking against the libraries of modules requires specific linker support, which is not widely available.

[2] There are certain anti-patterns that occur in headers, particularly system headers, that cause problems for modules. The section **Modularizing a Platform** describes some of them.

[3] The second instance is actually a new thread within the current process, not a separate process. However, the original compiler instance is blocked on the execution of this thread.

[4] The preprocessing context in which the modules are parsed is actually dependent on the command-line options provided to the compiler, including the language dialect and any `-D` options. However, the compiled modules for different command-line options are kept distinct, and any preprocessor directives that occur within the translation unit are ignored. See the section on the **Configuration macros declaration** for more information.

« **Standard C++ Modules** :: Contents :: **MSVC compatibility** »