The `build2` Build System

Revision `0.6`, August 2017
This revision of the document describes the `build2` build system `0.6.x` series.

Table of Contents

Preface

This is the preface.

1 Name Patterns

For convenience, in certain contexts, names can be generated with shell-like wildcard patterns. A name is a *name pattern* if its value contains one or more unquoted wildcard characters or character sequences. For example:

```
./: */                      # All (immediate) subdirectories
exe{hello}: {hxx cxx}{**}   # All C++ header/source files.
pattern = '*.txt'           # Literal '*.txt'.
```

Pattern-based name generation is not performed in certain contexts. Specifically, it is not performed in target names where it is interpreted as a pattern for target type/pattern-specific variable assignments. For example.

```
s = *.txt              # Variable assignment (performed).
./: cxx{*}             # Prerequisite names (performed).
cxx{*}: dist = false   # Target pattern (not performed).
```

In contexts where it is performed, it can be inhibited with quoting, for example:

```
pat = 'foo*bar'
./: cxx{'foo*bar'}
```

The following characters are wildcards:

```
*  - match any number of characters (including zero)
?  - match any single character
```

If a pattern ends with a directory separator, then it only matches directories. Otherwise, it only matches files. Matches that start with a dot (`.`) are automatically ignored unless the pattern itself also starts with this character.

In addition to the above wildcard characters, `**` and `***` are recognized as wildcard character sequences. If a pattern contains `**`, then it is matched just like `*` but in all the subdirectories, recursively. The `***` wildcard behaves like `**` but also matches the start directory itself. For example:

```
exe{hello}: cxx{**}  # All C++ source files recursively.
```

A group-enclosed (`{}`) pattern value may be followed by inclusion/exclusion patterns/matches. A subsequent value is treated as an inclusion or exclusion if it starts with a literal, unquoted plus (+) or minus (−) sign, respectively. In this case the remaining group values, if any, must all be inclusions or exclusions. If the second value doesn't start with a plus or minus, then all the group values are considered independent with leading pluses and minuses not having any special meaning. For example:

```
exe{hello}: cxx{f* -foo}           # Exclude foo if present.
exe{hello}: cxx{f* +foo}           # Include foo if not present.
exe{hello}: cxx{f* -fo?}           # Exclude foo and fox if present.
exe{hello}: cxx{f* +b* -foo -bar}  # Exclude foo and bar if present.
exe{hello}: cxx{f* b* -z*}         # Names matching three patterns.
```

Inclusions and exclusions are applied in the order specified and only to the result produced up to that point. The order of names in the result is unspecified. However, it is guaranteed not to contain duplicates. The pattern and the following inclusions/exclusions must be consistent with regards to the type of filesystem entry they match. That is, they should all match either files or directories. For example:

```
exe{hello}: cxx{f* -foo +*oo}  # Exclusion has no effect.
exe{hello}: cxx{f* +*oo}       # Ok, no duplicates.
./: {*/ -build}                # Error: exclusion not a directory.
```

As a more realistic example, let's say we want to exclude source files that reside in the `test/` directories (and their subdirectories) anywhere in the tree. This can be achieved with the following pattern:

```
exe{hello}: cxx{** -***/test/**}
```

Similarly, if we wanted to exclude all source files that have the `-test` suffix:

```
exe{hello}: cxx{** -**-test}
```

In contrast, the following pattern only excludes such files from the top directory:

```
exe{hello}: cxx{** -*-test}
```

If many inclusions or exclusions need to be specified, then an inclusion/exclusion group can be used. For example:

```
exe{hello}: cxx{f* -{foo bar}}  # Exclude foo and bar if present.
```

This is particularly useful if you would like to list the names to exclude in a variable. For example, this is how we can exclude certain files from compilation but still include them as ordinary file prerequisites (so that they are still included into the distribution):

```
exc = foo.cxx bar.cxx
exe{hello}: cxx{f* -{$exc}} file{$exc}
```

One common situation that calls for exclusions is auto-generated source code. Let's say we have auto-generated command line parser in `options.hxx` and `options.cxx`. Because of the in-tree builds, our name pattern may or may not find these files. Note, however, that we cannot just include them as non-pattern prerequisites. We also have to exclude them from the pattern match since otherwise we may end up with duplicate prerequisites. As a result, this is how we have to handle this case provided we want to continue using patterns to find other, non-generated source files:

```
exe{hello}: {hxx cxx}{* -options} {hxx cxx}{options}
```

If the name pattern includes an absolute directory, then the pattern match is performed in that directory and the generated names include absolute directories as well. Otherwise, the pattern match is performed in the *pattern base* directory. In buildfiles this is `src_base` while on the command line – the current working directory. In this case the generated names are relative to the base directory. For example, assuming we have the `foo.cxx` and `b/bar.cxx` source files:

```
exe{hello}: $src_base/cxx{**}  # $src_base/cxx{foo} $src_base/b/cxx{bar}
exe{hello}:           cxx{**}  #           cxx{foo}           b/cxx{bar}
```

Pattern matching as well as inclusion/exclusion logic is target type-specific. If the name pattern does not contain a type, then the `dir{}` type is assumed if the pattern ends with a directory separator and `file{}` otherwise.

For the `dir{}` target type the trailing directory separator is added to the pattern and all the inclusion/exclusion patterns/matches that do not already end with one. Then the filesystem search is performed for matching directories. For example:

```
./: dir{* -build}  # Search for */, exclude build/.
```

For the `file{}` and `file{}`-based target types the default extension (if any) is added to the pattern and all the inclusion/exclusion patterns/matches that do not already contain an extension. Then the filesystem search is performed for matching files.

For example, the `cxx{}` target type obtains the default extension from the `extension` variable. Assuming we have the following line in our `root.build`:

```
cxx{*}: extension = cxx
```

And the following in our `buildfile`:

```
exe{hello}: {cxx}{* -foo -bar.cxx}
```

The pattern match will first search for all the files matching the `*.cxx` pattern in `src_base` and then exclude `foo.cxx` and `bar.cxx` from the result. Note also that target type-specific decorations are removed from the result. So in the above example if the pattern match produces `baz.cxx`, then the prerequisite name is `cxx{baz}`, not `cxx{baz.cxx}`.

If the name generation cannot be performed because the base directory is unknown, target type is unknown, or the target type is not directory or file-based, then the name pattern is returned as is (that is, as an ordinary name). Project-qualified names are never considered to be patterns.

2 Grammar

```
eval:         '(' (eval-comma | eval-qual)? ')'
eval-comma:   eval-ternary (',' eval-ternary)*
eval-ternary: eval-or ('?' eval-ternary ':' eval-ternary)?
eval-or:      eval-and ('||' eval-and)*
eval-and:     eval-comp ('&&' eval-comp)*
eval-comp:    eval-value (('=='|'!='|'<'|'>'|'<='|'>=') eval-value)*
eval-value:   value-attributes? (<value> | eval | '!' eval-value)
eval-qual:    <name> ':' <name>

value-attributes: '[' <key-value-pairs> ']'
```

Note that `?:` (ternary operator) and `!` (logical not) are right-associative. Unlike C++, all the comparison operators have the same precedence. A qualified name cannot be combined with any other operator (including ternary) unless enclosed in parentheses. The `eval` option in the `eval-value` production shall contain single value only (no commas).

3 Test Module

The targets to be tested as well as the tests/groups from testscripts to be run can be narrowed down using the `config.test` variable. While this value is normally specified as a command line override (for example, to quickly re-run a previously failed test), it can also be persisted in `config.build` in order to create a configuration that will only run a subset of tests by default. For example:

```
b test config.test=foo/exe{driver} # Only test foo/exe{driver} target.
b test config.test=bar/baz         # Only run bar/baz testscript test.
```

The `config.test` variable contains a list of `@`-separated pairs with the left hand side being the target and the right hand side being the testscript id path. Either can be omitted (along with `@`). If the value contains a target type or ends with a directory separator, then it is treated as a target name. Otherwise – an id path. The targets are resolved relative to the root scope where the `config.test` value is set. For example:

```
b test config.test=foo/exe{driver}@bar
```

To specify multiple id paths for the same target we can use the pair generation syntax:

```
b test config.test=foo/exe{driver}@{bar baz}
```

If no targets are specified (only id paths), then all the targets are tested (with the testscript tests to be run limited to the specified id paths). If no id paths are specified (only targets), then all the testscript tests are run (with the targets to be tested limited to the specified targets). An id path without a target applies to all the targets being considered.

A directory target without an explicit target type (for example, `foo/`) is treated specially. It enables all the tests at and under its directory. This special treatment can be inhibited by specifying the target type explicitly (for example, `dir{foo/}`).

4 Version Module

A project can use any version format as long as it meets the package version requirements. The toolchain also provides additional functionality for managing projects that conform to the `build2` *standard version* format. If you are starting a new project that uses `build2`, you are strongly encouraged to use this versioning scheme. It is based on much thought and, often painful, experience. If you decide not to follow this advice, you are essentially on your own when version management is concerned.

The standard `build2` project version conforms to Semantic Versioning and has the following form:

```
<major>.<minor>.<patch>[-<prerel>]
```

For example:

```
1.2.3
1.2.3-a.1
1.2.3-b.2
```

The `build2` package version that uses the standard project version will then have the following form (*epoch* is the versioning scheme version and *revision* is the package revision):

```
[<epoch>~]<major>.<minor>.<patch>[-<prerel>][+<revision>]
```

For example:

```
1.2.3
1.2.3+1
1~1.2.3-a.1+2
```

The *major*, *minor*, and *patch* should be numeric values between 0 and 999 and all three cannot be zero at the same time. For initial development it is recommended to use 0 for *major*, start with version `0.1.0`, and change to `1.0.0` once things stabilize.

In the context of C and C++ (or other compiled languages), you should increment *patch* when making binary-compatible changes, *minor* when making source-compatible changes, and *major* when making breaking changes. While the binary compatibility must be set in stone, the source compatibility rules can sometimes be bent. For example, you may decide to make a breaking change in a rarely used interface as part of a minor release. Note also that in the context of C++ deciding whether a change is binary-compatible is a non-trivial task. There are resources that list the rules but no automated tooling yet. If unsure, increment *minor*.

If present, the *prerel* component signifies a pre-release. Two types of pre-releases are supported by the standard versioning scheme: *final* and *snapshot* (non-pre-release versions are naturally always final). For final pre-releases the *prerel* component has the following form:

```
(a|b).<num>
```

For example:

```
1.2.3-a.1
1.2.3-b.2
```

The letter 'a' signifies an alpha release and 'b' – beta. The alpha/beta numbers (*num*) should be between 1 and 499.

Note that there is no support for release candidates. Instead, it is recommended that you use later-stage beta releases for this purpose (and, if you wish, call them "release candidates" in announcements, etc).

What version should be used during development? The common approach is to increment to the next version and use that until the release. This has one major drawback: if we publish intermediate snapshots (for example, for testing) they will all be indistinguishable both between each other and, even worse, from the final release. One way to remedy this is to increment the pre-release number before each publication. However, unless automated, this will be burdensome and error-prone. Also, there is a real possibility of running out of version numbers if, for example, we do continuous integration by testing (and therefore publishing) each commit.

To address this, the standard versioning scheme supports *snapshot pre-releases* with the *prerel* component having the following form:

```
(a|b).<num>.<snapsn>[.<snapid>]
```

For example:

```
1.2.3-a.1.1422564055.340c0a26a5efed1f
```

In essence, a snapshot pre-release is after the previous final release but before the next (`a.1` and, perhaps, `a.2` in the above example) and is uniquely identified by the snapshot sequence number (*snapsn*) and snapshot id (*snapid*).

The *num* component has the same semantics as in the final pre-releases except that it can be 0. The *snapsn* component should be either the special value 'z' or a numeric, non-zero value that increases for each subsequent snapshot. It must fit into an unsigned 64-bit integer. The *snapid* component, if present, should be an alpha-numeric value that uniquely identifies the snapshot. It is not required for version comparison (*snapsn* should be sufficient) and is included for reference. It must not be longer than 16 characters.

Where do the snapshot sn and id come from? Normally from the version control system. For example, for `git`, *snapsn* is the commit date (as UNIX timestamp in the UTC timezone) and *snapid* is a 16-character abbreviated commit id. As discussed below, the `build2 version` module extracts and manages all this information automatically (the use of `git` commit dates is not without limitations; see below for details).

The special 'z' *snapsn* value identifies the *latest* or *uncommitted* snapshot. It is chosen to be greater than any other possible *snapsn* value and its use is discussed further below.

As an illustration of this approach, let's examine how versions change during the lifetime of a project:

```
0.1.0-a.0.z     # development after a.0
0.1.0-a.1       # pre-release
0.1.0-a.1.z     # development after a.1
0.1.0-a.2       # pre-release
0.1.0-a.2.z     # development after a.2
0.1.0-b.1       # pre-release
0.1.0-b.1.z     # development after b.1
0.1.0           # release
0.1.1-b.0.z     # development after b.0 (bugfix)
0.2.0-a.0.z     # development after a.0
0.1.1           # release (bugfix)
1.0.0           # release (jumped straight to 1.0.0)
...
```

As shown in the above example, there is nothing wrong with "jumping" to a further version (for example, from alpha to beta, or from beta to release, or even from alpha to release). We cannot, however, jump backwards (for example, from beta back to alpha). As a result, a sensible strategy is to start with `a.0` since it can always be upgraded (but not downgrade) at a later stage.

When it comes to the version control systems, the recommended workflow is as follows: The change to the final version should be the last commit in the (pre-)release. It is also a good idea to tag this commit with the project version. A commit immediately after that should change the version to a snapshot essentially "opening" the repository for development.

The project version without the snapshot part can be represented as a 64-bit decimal value comparable as integers (for example, in preprocessor directives). The integer representation has the following form:

```
AAABBBCCCDDDE

AAA - major
BBB - minor
CCC - patch
DDD - alpha / beta (DDD + 500)
E   - final (0) / snapshot (1)
```

If the *DDDE* value is not zero, then it signifies a pre-release. In this case one is subtracted from the *AAABBBCCC* value. An alpha number is stored in *DDD* as is while beta – incremented by 500. If *E* is 1, then this is a snapshot after *DDD*.

For example:

```
              AAABBBCCCDDDE
0.1.0         0000010000000
0.1.2         0000010010000
1.2.3         0010020030000
2.2.0-a.1     0020019990010
3.0.0-b.2     0029999995020
2.2.0-a.1.z   0020019990011
```

A project that uses standard versioning can rely on the `build2 version` module to simplify and automate version managements. The `version` module has two primary functions: eliminate the need to change the version anywhere except in the project's manifest file and automatically extract and propagate the snapshot information (serial number and id).

The `version` module must be loaded in the project's `bootstrap.build`. While being loaded, it reads the project's manifest and extracts its version (which must be in the standard form). The version is then parsed and presented as the following build system variables (which can be used in the buildfiles):

```
[string] version                    # 2~1.2.3-b.4.1234567.deadbeef+3

[string] version.project            # 1.2.3-b.4.1234567.deadbeef
[uint64] version.project_number     # 0010020025041
[string] version.project_id         # 1.2.3-b.4.deadbeef

[bool]   version.stub               # false (true for 0[+<revision>])

[uint64] version.epoch              # 2

[uint64] version.major              # 1
[uint64] version.minor              # 2
[uint64] version.patch              # 3

[bool]   version.alpha              # false
[bool]   version.beta               # true
[bool]   version.pre_release        # true
[string] version.pre_release_string # b.4
[uint64] version.pre_release_number # 4

[bool]   version.snapshot           # true
[uint64] version.snapshot_sn        # 1234567
[string] version.snapshot_id        # deadbeef
[string] version.snapshot_string    # 1234567.deadbeef

[uint64] version.revision           # 3
```

As a convenience, the `version` module also extract the `summary` and `url` manifest values and sets them as the following build system variables (this additional information is used, for example, when generating the `pkg-config` files):

```
[string] project.summary
[string] project.url
```

If the version is the latest snapshot (that is, it's in the `.z` form), then the `version` module extracts the snapshot information from the version control system used by the project. Currently only `git` is supported with the following semantics.

If the project's source directory (`src_root`) is clean (that is, it does not have any changed or untracked files), then the `HEAD` commit date and id are used as the snapshot sn and id, respectively. Otherwise, the snapshot is left in the `.z` form (which signals the latest/uncommitted snapshot). While we can work with such a `.z` snapshot locally, preparing a distribution of such an uncommitted snapshot is an error.

The use of `git` commit dates for snapshot ordering has its limitations: they have one second resolution which means it is possible to create two commits with the same date (but not the same commit id and thus snapshot id). We also need all the committers to have a reasonably accurate clock. Note, however, that in case of a commit date clash/ordering issue, we still end up with distinct versions (because of the commit id) – they are just not ordered correctly. As a result, we feel that the risks are justified when the only alternative is manual version management (which is always an option, nevertheless).

When we prepare a distribution of a snapshot, the `version` module automatically adjusts the package name to include the snapshot information as well as patches the manifest file in the distribution with the snapshot sn and id (that is, replacing `.z` in the version value with the actual snapshot information). The result is a package that is specific to this commit.

Besides extracting the version information and making it available as individual components, the `version` module also provide rules for automatically generating the `version` (or `Version`/`VERSION`) file that is customarily found in the root of a project as well as the version headers (or other similar version-based files).

The `version` file rule matches a `doc` target that contains the `version` substring in its name (comparison is case-insensitive) and that depends on the project's `manifest` file. To utilize this rule you would normally have something along these lines to your project's root `buildfile`:

```
./: ... doc{version}

doc{version}: file{manifest} # Generated by the version module.
doc{version}: dist = true    # Include into the distribution.
```

The `version` header rule pre-processes a template file (which means it can be used to generate any kinds of files, not just C/C++ headers). It matches a `file`-based target that has a corresponding `in` prerequisite and also depends on the project's `manifest` file. As an example, let's assume we want to auto-generate a header called `version.hxx` for our `libhello` library. To acomplish this we add the `version.hxx.in` template as well as something along these lines to our `buildfile`:

```
lib{hello}: ... hxx{version}

hxx{version}: in{version} $src_root/file{manifest}
hxx{version}: dist = true
```

The header rule is a line-based pre-processor that substitutes fragments enclosed between (and including) a pair of dollar signs (`$`) with `$$` being the escape sequence. As an example, let's assume our `version.hxx.in` contains the following lines:

```
#ifndef LIBHELLO_VERSION

#define LIBHELLO_VERSION     $libhello.version.project_number$ULL
#define LIBHELLO_VERSION_STR "$libhello.version.project$"

#endif
```

If our `libhello` is at version `1.2.3`, then the generated `version.hxx` will look like this:

```
#ifndef LIBHELLO_VERSION

#define LIBHELLO_VERSION     10020030000ULL
#define LIBHELLO_VERSION_STR "1.2.3"

#endif
```

The first component after the opening `$` should be either the name of the project itself (like `libhello` above) or a name of one of its dependencies as listed in the manifest. If it is the project itself, then the rest can refer to one of the `version.*` variables that we discussed earlier (in reality it can be any variable visible from the project's root scope).

If the name refers to one of the dependecies (that is, projects listed with `depends:` in the manifest), then the following special substitutions are recognized:

```
$<name>.version$                          - textual version constraint
$<name>.condition(<VERSION>[,<SNAPSHOT>])$ - numeric satisfaction condition
$<name>.check(<VERSION>[,<SNAPSHOT>])$     - numeric satisfaction check
```

Here *VERSION* is the version number macro and the optional *SNAPSHOT* is the snapshot number macro. The snapshot is only required if you plan to include snapshot information in your dependency constraints.

As an example, let's assume our `libhello` depends on `libprint` which is reflected with the following line in our manifest:

```
depends: libprint >= 2.3.4
```

We also assume that `libprint` provides its version information in the `libprint/version.hxx` header and uses analogous-named macros. Here is how we can add a version check to our `version.hxx.in`:

```
#ifndef LIBHELLO_VERSION

#define LIBHELLO_VERSION     $libhello.version.project_number$ULL
#define LIBHELLO_VERSION_STR "$libhello.version.project$"

#include <libprint/version.hxx>

$libprint.check(LIBPRINT_VERSION)$

#endif
```

After the substitution our `version.hxx` header will look like this:

```
#ifndef LIBHELLO_VERSION

#define LIBHELLO_VERSION     10020030000ULL
#define LIBHELLO_VERSION_STR "1.2.3"

#include <libprint/version.hxx>

#ifdef LIBPRINT_VERSION
#  if !(LIBPRINT_VERSION >= 20030040000ULL)
#    error incompatible libprint version, libprint >= 2.3.4 is required
#  endif
#endif

#endif
```

The `version` and `condition` substitutions are the building blocks of the `check` substitution. For example, here is how we can implement a check with a customized error message:

```
#if !($libprint.condition(LIBPRINT_VERSION)$)
#  error bad libprint, need libprint $libprint.version$
#endif
```

The `version` module also treats one dependency in a special way: if you specify the required version of the build system in your manifest, then the module will automatically check it for you. For example, if we have the following line in our manifest:

```
depends: * build2 >= 0.5.0
```

And someone tries to build our project with `build2 0.4.0`, then they will see an error like this:

```
build/bootstrap.build:3:1: error: incompatible build2 version
  info: running 0.4.0
  info: required 0.5.0
```

What version constraints should be use when depending on other project. We start with a simple case where we depend on a release. Let's say `libprint 2.3.0` added a feature that we need in our `libhello`. If `libprint` follows the source/binary compatibility guidelines discussed above, then any `2.X.Y` version should work provided `X >= 3`. And this how we can specify it in the manifest:

```
depends: libprint [2.3.0 3.0.0-)
```

Let's say we are now working on `libhello 2.0.0` and would like to start using features from `libprint 3.0.0`. However, currently, only pre-releases of `3.0.0` are available. If you would like to add a dependency on a pre-release (most likely from your own pre-release), then the recommendation is to only allow a specific version, essentially "expiring" the combination as soon as newer versions become available. For example:

```
version: 2.0.0-b.1
depends: libprint == 3.0.0-b.2
```

Finally, let's assume we are feeling adventerous and would like to test development snapshots of `libprint` (most likey from our own snapshots). In this case the recommendation is to only allow a snapshot range for a specific pre-release with the understanding and a warning that no compatibility between snapshot versions is guaranteed. For example:

```
version: 2.0.0-b.1.z
depends: libprint [3.0.0-b.2.1 3.0.0-b.3)
```

5 `cxx` (C++) Module

This chapter describes the `cxx` build system module which provides the C++ compilation and linking support. Most of its functionality, however, is provided by the `cc` module, a common implementation for the C-family languages.

5.1 C++ Modules Support

This section describes the build system support for C++ modules.

5.1.1 Modules Introduction

The goal of this section is to provide a practical introduction to C++ Modules and to establish key concepts and terminology.

A pre-modules C++ program or library consists of one or more *translation units* which are customarily referred to as C++ source files. Translation units are compiled to *object files* which are then linked together to form a program or library.

Let's also recap the difference between an *external name* and a *symbol*: External names refer to language entities, for example classes, functions, and so on. The *external* qualifier means they are visible across translation units.

Symbols are derived from external names for use inside object files. They are the cross-referencing mechanism for linking a program from multiple, separately-compiled translation units. Not all external names end up becoming symbols and symbols are often *decorated* with additional information, for example, a namespace. We often talk about a symbol having to be satisfied by linking an object file or a library that provides it.

What is a C++ module? It is hard to give a single but intuitive answer to this question. So we will try to answer it from three different perspectives: that of a module consumer, a module producer, and a build system that tries to make those two play nice. But we can make one thing clear at the outset: modules are a *language-level* not a preprocessor-level mechanism; it is `import`, not `#import`.

One may also wonder why C++ modules, what are the benefits? Modules offer isolation, both from preprocessor macros and other module's symbols. Unlike headers, modules require explicit exportation of entities that will be visible to the consumers. In this sense they are a *physical design mechanism* that forces us to think how we structure our code. Modules promise significant build speedups since importing a module, unlike including a header, should be essentially free. Modules are also the first step to not needing the preprocessor in most translation units. Finally, modules have a chance of bringing to mainstream reliable and easy to setup distributed C++ compilation, since now build systems can make sure compilers on the local and remote hosts are provided with identical inputs.

To refer to a module we use a *module name*, a sequence of dot-separated identifiers, for example `hello.core`. While the specification does not assign any hierarchical semantics to this sequence, it is customary to refer to `hello.core` as a submodule of `hello`. We discuss submodules and the module naming guidelines below.

From a consumer's perspective, a module is a collection of external names, called *module interface*, that become *visible* once the module is imported:

```
import hello.core
```

What exactly does *visible* mean? To quote the standard: *An import-declaration makes exported declarations [...] visible to name lookup in the current translation unit, in the same namespaces and contexts [...].* One intuitive way to think about this visibility is *as-if* there were only a single translation unit for the entire program that contained all the modules as well as all their consumers. In such a translation unit all the names would be visible to everyone in exactly the same way and no entity would be redeclared.

This visibility semantics suggests that modules are not a name scoping mechanism and are orthogonal to namespaces. Specifically, a module can export names from any number of namespaces, including the global namespace. While the module name and its namespace names need not be related, it usually makes sense to have a parallel naming scheme, as discussed below.

Note also that from the consumer's perspective a module does not provide any symbols, only C++ entity names. If we use a name from a module, then we may have to satisfy the corresponding symbol(s) using the usual mechanisms: link an object file or a library that provides them. In this respect, modules are similar to headers and as with headers module's use is not limited to libraries; they make perfect sense when structuring programs.

The producer perspective on modules is predictably more complex. In pre-modules C++ we only had one kind of translation unit (or source file). With modules there are three kinds: *module interface unit*, *module implementation unit*, and the original kind which we will call a *non-module translation unit.*

From the producer's perspective, a module is a collection of module translation units: one interface unit and zero or more implementation units. A simple module may consist of just the interface unit that includes implementations of all its functions (not necessarily inline). A more complex module may span multiple implementation units.

A translation unit is a module interface unit if it contains an *exporting module declaration*:

```
export module hello.core;
```

A translation unit is a module implementation unit if it contains a *non-exporting module declaration*:

```
module hello.core;
```

While module interface units may use the same file extension as normal source files, we recommend that a different extension be used to distinguish them as such, similar to header files. While the compiler vendors suggest various (and predictably different) extensions, our recommendation is `.mxx` for the `.hxx/.cxx` source file naming and `.mpp` for `.hpp/.cpp`. And if you are using some other naming scheme, perhaps now is a good opportunity to switch to one of the above. Using the source file extension for module implementation units appears reasonable and that's what we recommend.

A module declaration (exporting or non-exporting) starts a *module purview* that extends until the end of the module translation unit. Any name declared in a module's purview *belongs* to said module. For example:

```
#include <string>               // Not in purview.

export module hello.core;

void
say_hello (const std::string&);  // In purview.
```

A name that belongs to a module is *invisible* to the module's consumers unless it is *exported*. A name can be declared exported only in a module interface unit, only in the module's purview, and there are several syntactic ways to accomplish this. We can start the declaration with the `export` specifier, for example:

```
export module hello.core;

export enum class volume {quiet, normal, loud};

export void
say_hello (const char*, volume);
```

Alternatively, we can enclose one or more declarations into an *exported group*, for example:

```
export module hello.core;

export
{
  enum class volume {quiet, normal, loud};

  void
  say_hello (const char*, volume);
}
```

Finally, if a namespace definition is declared exported, then every name in its body is exported, for example:

```
export module hello.core;

export namespace hello
{
  enum class volume {quiet, normal, loud};

  void
  say (const char*, volume);
}

namespace hello
{
  void
  impl (const char*, volume); // Not exported.
}
```

Up until now we've only been talking about module's names. What about module's symbols? For exported names, the resulting symbols would be the same as if those names were declared outside of a module's purview (or as if no modules were used at all). Non-exported names, on the other hand, have *module linkage*: their symbols can be resolved from this module's units but not from other translation units. They also cannot clash with symbols for identical names from other modules (and non-modules). This is usually achieved by decorating the non-exported symbols with a module name.

This ownership model has one important backwards-compatibility implication: a library built with modules enabled can be linked to a program that still uses headers. And vice versa: we can build a module for a library that only uses headers. For example, if our compiler does not provide a module for the standard library, we should be able to build our own:

```
export module std.core;

export
{
  #include <string>
  //...
}
```

What about the preprocessor? Modules do not export preprocessor macros, only C++ names. A macro defined in the module interface unit cannot affect the module's consumers. And macros defined by the module's consumers cannot affect the module interface they are importing. In other words, module producers and consumers are isolated from each other when the preprocessor is concerned. This is not to say that the preprocessor cannot be used by either, it just doesn't "leak" through the module interface. One practical implication of this model is the insignificance of the import order.

If a module imports another module in its purview, the imported module's names are not made automatically visible to the consumers of the importing module. This is unlike headers and can be surprising. Consider this module interface as an example:

```
export module hello;

import std.core;

export void
say_hello (const std::string&);
```

And this module consumer:

```
import hello;

int
main ()
{
  say_hello ("World");
}
```

This example will result in a compile error and the diagnostics may confusingly indicate that there is no known conversion from a C string to "something" called `std::string`. But with the understanding of the difference between `import` and `#include` the reason should be clear: while the module interface "sees" `std::string` (because it imported its module), we do not (since we did not). So the fix is to explicitly import `std.core`:

```
import std.core;
import hello;

int
main ()
{
  say_hello ("World");
}
```

A module, however, can choose to re-export a module it imports. In this case, all the names from the imported module will also be visible to the importing module's consumers. For example, with this change to the module interface the first version of our consumer will compile without errors (note that whether this is a good design choice is debatable):

```
export module hello;

export import std.core;

export void
say_hello (const std::string&);
```

One way to think of a re-export is *as-if* an import of a module also "injects" all the imports said module re-exports, recursively. That's essentially how most compilers implement it.

Module re-export is the mechanism of assembling bigger modules out of submodules. As an example, let's say we had the `hello.core`, `hello.basic`, and `hello.extra` modules. To make life easier for users that want to import all of them we can create the `hello` module that re-exports the three:

```
export module hello;

export
{
  import hello.core;
  import hello.basic;
  import hello.extra;
}
```

The final perspective that we consider is that of the build system. From its point of view the central piece of the module infrastructure is the *binary module interface*: a binary file that is produced by compiling the module interface unit and that is required when compiling any translation unit that imports this module (as well as the module's implementation units).

So, in a nutshell, the main functionality of a build system when it comes to modules support is figuring out the order in which all the units should be compiled and making sure that every compilation is able to find the binary module interfaces it needs.

Predictably, the details are more complex. Compiling a module interface unit produces two outputs: the binary module interface and the object file. Also, all the compilers currently implement module re-export as a shallow reference to the re-exported module name which means that their binary interfaces must be discoverable as well, recursively. In fact, currently, all the imports are handled like this, though a different implementation is at least plausible, if unlikely.

While the details vary between compilers, the contents of the binary interfaces are generally sensible to the compiler options. If the options used to produce the binary interface (for example, when building a library) are sufficiently different compared to the ones used when compiling the module consumers, the binary interface may be unusable. So while a build system should strive to reuse existing binary interfaces, it should also be prepared to compile its own versions "on the side". This suggests that modules are not a distribution mechanism, that binary module interfaces should probably not be installed, and that instead we should install and distribute module interface sources.

5.1.2 Building Modules

Compiler support for C++ Modules is still experimental. As a result, it is currently only enabled if the C++ standard is set to `experimental`. After loading the `cxx` module we can check if modules are enabled using the `cxx.features.modules` boolean variable. This is what the corresponding `root.build` fragment could look like for a modularized project:

```
cxx.std = experimental

using cxx

assert $cxx.features.modules 'c++ compiler does not support modules'

mxx{*}: extension = mxx
cxx{*}: extension = cxx
```

To support C++ modules the `cxx` (build system) module defines several additional target types. The `mxx{}` target is a module interface unit. As you can see from the above `root.build` fragment, in this project we are using the `.mxx` extension for our module interface files. While you can use the same extension as for `cxx{}` (source files), this is not recommended since some functionality, such as wildcard patterns, will become unusable.

The `bmi{}` group and its `bmie{}`, `bmia{}`, and `bmis{}` members are used for binary module interfaces targets. We normally do not need to mention them explicitly in our build-files except, perhaps, to specify additional, module interface-specific compile options. We will see some example of this below.

To build a modularized executable or library we simply list the module interfaces as its prerequisites, just as we do source files. As an example, let's build the `hello` program that we have started in the introduction (you can find the complete project in the Hello Repository under `mhello`). Specifically, we assume our project contains the following files:

```
// file: hello.mxx (module interface)

export module hello;

import std.core;

export void
say_hello (const std::string&);

// file: hello.cxx (module implementation)

module hello;

import std.io;

using namespace std;

void
say_hello (const string& name)
{
  cout << "Hello, " << name << '!' << endl;
}

// file: driver.cxx

import std.core;
import hello;

int
main ()
{
  say_hello ("World");
}
```

To build a `hello` executable from these files we can write the following `buildfile`:

```
exe{hello}: cxx{driver} {mxx cxx}{hello}
```

Or, if you prefer to use wildcard patterns:

```
exe{hello}: {mxx cxx}{*}
```

Alternatively, we can package the module into a library and then link the library to the executable:

```
exe{hello}: cxx{driver} lib{hello}
lib{hello}: {mxx cxx}{hello}
```

As you might have surmised from the above, the modules support implementation automatically resolves imports to module interface units that are specified either as direct prerequisites or as prerequisites of library prerequisites.

To perform this resolution without a significant overhead, the implementation delays the extraction of the actual module name from module interface units (since not all available module interfaces are necessarily imported by all the translation units). Instead, the implementation tries to guess which interface unit implements each module being imported based on the interface file path. Or, more precisely, a two-step resolution process is performed: first a best match between the desired module name and the file path is sought and then the actual module name is extracted and the correctness of the initial guess is verified.

The practical implication of this implementation detail is that our module interface files must embed a portion of a module name, or, more precisely, a sufficient amount of "module name tail" to unambiguously resolve all the modules used in a project. Note also that this guesswork is only performed for direct module interface prerequisites; for those that come from libraries the module names are known and are therefore matched exactly.

As an example, let's assume our `hello` project had two modules: `hello.core` and `hello.extra`. While we could call our interface files `hello.core.mxx` and `hello.extra.mxx`, respectively, this doesn't look particularly good and may be contrary to the file naming scheme used in our project. To resolve this issue the match of module names to file names is made "fuzzy": it is case-insensitive, it treats all separators (dots, dashes, underscores, etc) as equal, and it treats a case change as an imaginary separator. As a result, the following naming schemes will all match the `hello.core` module name:

```
hello-core.mxx
hello_core.mxx
HelloCore.mxx
hello/core.mxx
```

We also don't have to embed the full module name. In our case, for example, it would be most natural to call the files `core.mxx` and `extra.mxx` since they are already in the project directory called `hello/`. This will work since our module names can still be guessed correctly and unambiguously.

If a guess turns out to be incorrect, the implementation issues diagnostics and exits with an error. To resolve this situation we can either adjust the interface file names or we can specify the module name explicitly with the `cc.module_name` variable. The latter approach can be used with interface file names that have nothing in common with module names, for example:

```
mxx{foobar}@./: cc.module_name = hello
```

Note also that standard library modules (`std` and `std.*`) are treated specially: they are not fuzzy-matched and they need not be resolvable to the corresponding `mxx{}` or `bmi{}` in which case it is assumed they will be resolved in an ad hoc way by the compiler. This means that if you want to build your own standard library module (for example, because your compiler doesn't yet ship one; note that this may not be supported by all compilers), then you have to specify the module name explicitly. For example:

```
exe{hello}: cxx{driver} {mxx cxx}{hello} mxx{std-core}

mxx{std-core}@./: cc.module_name = std.core
```

When C++ modules are enabled and available, the build system makes sure the `__cpp_modules` feature test macro is defined. Currently, its value is 201703 for VC and 201704 for GCC and Clang but this will most likely change in the future.

One major difference between the current C++ modules implementation in VC and the other two compilers is the use of the `export module` syntax to identify the interface units. While both GCC and Clang have adopted this new syntax, VC is still using the old one without the `export` keyword. We can use the `__cpp_modules` macro to provide a portable declaration:

```
#if __cpp_modules >= 201704
export
#endif
module hello;
```

Note, however, that the modules support in `build2` provides temporary "magic" that allows us to use the new syntax even with VC.

5.1.3 Symbol Exporting

When building a shared library, some platforms (notably Windows) require that we explicitly export symbols that must be accessible to the library users. If you don't need to support such platforms, you can thank your lucky stars and skip this section.

When using headers, the traditional way of achieving this is via an "export macro" that is used to mark exported APIs, for example:

```
LIBHELLO_EXPORT void
say_hello (const string&);
```

This macro is then appropriately defined (often in a separate "export header") to export symbols when building the shared library and to import them when building the library's users.

Introduction of modules changes this in a number of ways, at least as implemented by VC (hopefully other compilers will follow suit). While we still have to explicitly mark exported symbols in our module interface unit, there is no need (and, in fact, no way) to do the same

when said module is imported. Instead, the compiler automatically treats all such explicitly exported symbols (note: symbols, not names) as imported.

One notable aspect of this new model is the locality of the export macro: it is only defined when compiling the module interface unit and is not visible to the consumers of the module. This is unlike headers where the macro has to be unique per-library (that `LIBHELLO_` prefix) because a header from one library can be included while building another library.

We can continue using the same export macro and header with modules and, in fact, that's the recommended approach when maintaining dual, header/module arrangements for backwards compatibility (discussed below). However, for module-only codebases, we have an opportunity to improve the situation in two ways: we can use a single, keyword-like macro instead of a library-specific one and we can make the build system manage it for us thus getting rid of the export header.

To enable this functionality in `build2` we set the `cxx.features.symexport` boolean variable to `true` before loading the `cxx` module. For example:

```
cxx.std = experimental

cxx.features.symexport = true

using cxx

...
```

Once enabled, `build2` automatically defines the `__symexport` macro to the appropriate value depending on the platform and the type of library being built. As library authors all we have to do is use it in appropriate places in our module interface units, for example:

```
export module hello;

import std.core;

export __symexport void
say_hello (const std::string&);
```

As an aside, you may be wondering why can't a module export automatically mean a symbol export? While you will normally want to export symbols of all your module-exported names, you may also need to do so for some non-module-exported ones. For example:

```
export module foo;

__symexport void
f_impl ();

export __symexport inline void
f ()
{
  f_impl ();
}
```

Furthermore, symbol exporting is a murky area with many limitations and pitfalls (such as auto-exporting of base classes). As a result, it would not be unreasonable to expect such an automatic module exporting to only further muddy matters.

5.1.4 Module Installation

As discussed in the introduction, binary module interfaces are not a distribution mechanism and installing module interface sources appears to be the preferred approach.

Module interface units are by default installed in the same location as headers (for example, `/usr/include`). However, instead of relying on a header-like search mechanism (`-I` paths, etc.), an explicit list of exported modules is listed for each library in its `.pc` (`pkg-config`) file.

Specifically, the library's `.pc` file contains the `modules` variable that lists all the exported modules in the `<name>=<path>` form with `<name>` being the module's C++ name and `<path>` – the module interface file's absolute path. For example:

```
Name: libhello
Version: 1.0.0
Cflags:
Libs: -L/usr/lib -lhello

modules = hello.core=/usr/include/hello/core.mxx hello.extra=/usr/include/hello/extra.mxx
```

Additional module properties are specified with variables in the `module_<property>.<name>` form, for example:

```
module_symexport.hello.core = true
module_preprocessed.hello.core = all
```

Currently, two properties are defined. The `symexport` property with the boolean value signals whether the module uses the `__symexport` support discussed above.

The `preprocessed` property indicates the degree of preprocessing the module unit requires and is used to optimize module compilation. Valid values are `none` (not preprocessed), `includes` (no `#include` directives in the source), `modules` (as above plus no module declarations depend on the preprocessor, for example, `#ifdef`, etc.), and `all` (the source is fully preprocessed). Note that for `all` the source may still contain comments and line continuations.