The `build2` Build System

Revision `0.7`, May 2018
This revision of the document describes the `build2` build system `0.7.x` series.

Table of Contents

Preface

This document describes the `build2` build system. For the build system driver command line interface refer to the **`b(1)`** man pages.

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 regularity, the first pattern can also start with the

plus sign. 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* -foo -bar}  # Same as above.
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}}
exe{hello}: cxx{+{f* b*} -{foo bar}}
```

This is particularly useful if you would like to list the names to include or 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* b*} -{$exc}} file{$exc}
```

If we want to specify our pattern in a variable, then we have to use the explicit inclusion syntax, for example:

```
pat = 'f*'
exe{hello}: cxx{+$pat} # Pattern match.
exe{hello}: cxx{$pat}  # Literal 'f*'.

pat = '+f*'
exe{hello}: cxx{$pat}  # Literal '+f*'.

inc = 'f*'  'b*'
exc = 'f*o' 'b*r'
exe{hello}: cxx{+{$inc} -{$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 (though this is probably still a bad idea if your library is widely depended upon). 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 publishing and testing each commit.

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

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

For example:

```
1.2.3-a.1.20180319215815.26efe301f4a7
```

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 optional 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 not be longer than 16 decimal digits. 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 number and id come from? Normally from the version control system. For example, for `git`, *snapsn* is the commit date in the *YYYYMMDDhhmmss* form and UTC timezone and *snapid* is a 12-character abbreviated commit id. As discussed below, the `build2 version` module extracts and manages all this information automatically (but 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, "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
[bool]   version.snapshot_committed  # true

[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 number and id, respectively.

Otherwise (that is, the project is between commits), the `HEAD` commit date is incremented by one second and is used as the snapshot number with no id. While we can work with such uncommitted snapshots locally, we should not distribute or publish them since they are indistinguishable from each other.

Finally, if the project does not have `HEAD` (that is, the project has no commits yet), the special `19700101000000` (UNIX epoch) commit date is used.

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 number 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 installing the manifest file as well as automatically generating version headers (or other similar version-based files).

By default the project's `manifest` file is installed as documentation, just like other `doc{}` targets (thus replacing the `version` file customarily shipped in the project root directory). The manifest installation rule in the `version` module in addition patches the installed manifest file with the actual snapshot number and id, just like during the preparation of distribu-

tions.

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
```

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. Similarly, duplicate symbol issues may arise if more than one object file or library provides the same symbol.

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 modules' 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 with modules 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 provide 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 [...]. [Note: The entities are not redeclared in the translation unit containing the module import declaration. -- end note]* 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. Finally, the `import` declaration does not imply any additional visibility for names declared inside namespaces. Specifically, to access such names we must continue using the standard mechanisms, such as qualification or using declaration/directive. For example:

```
import hello.core;        // Exports hello::say().

say ();                   // Error.
hello::say ();            // Ok.

using namespace hello;
say ();                   // Ok.
```

Note also that from the consumer's perspective a module does not provide any symbols, only C++ entity names. If we use names from a module, then we may have to satisfy the corresponding symbols 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. Furthermore, a library may also have private or implementation modules that are not meant to be consumed by the library's users.

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, then perhaps now is a good opportunity to switch to one of the above. Continuing 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;       // Start of purview.

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 names belonding to a module. What about the corresponding 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 the module name.

This ownership model has an important backwards compatibility implication: a library built with modules enabled can be linked to a program that still uses headers. And even the other way around: we can build and use a module for a library that was built with headers.

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. For example, consider this module interface:

```
export module hello;

#ifndef SMALL
#define HELLO
export void say_hello (const char*);
#endif
```

And its consumer:

```
// module consumer
//
#define SMALL       // No effect.
import hello;

#ifdef HELLO        // Not defined.
...
#endif
```

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 its 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 (the consumer) 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, as discussed below):

```
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 for 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;
}
```

Besides starting a module purview, a non-exporting module declaration in the implementation unit makes non-internal linkage names declared or made visible in the *interface purview* also visible in the *implementation purview*. In this sense non-exporting module declaration acts as an extended `import`. For example:

```
import hello.impl;          // Not visible (exports impl()).

void
extra_impl ();              // Not visible.

export module hello.extra;  // Start of interface purview.

import hello.core;          // Visible (exports core()).

void
extra ();                   // Visible.

static void
extra2 ();                  // Not visible (internal linkage).
```

And this is the implementation unit:

```
module hello.extra;         // Start of implementation purview.

void
f ()
{
  impl ();        // Error.
  extra_impl ();  // Error.
  core ();        // Ok.
  extra ();       // Ok.
  extra2 ();      // Error.
}
```

In particular, this means that while the relative order of imports is not significant, the placement of imports in the module interface unit relative to the module declaration can be.

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.

Then, 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 translation units should be compiled and making sure that every compilation process 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. The latter contains object code for non-inline functions, global variables, etc., that the interface unit may define. This object file has to be linked when producing any binary (program or library) that uses this module.

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 module interface can range from a stream of preprocessed tokens to something fairly close to object code. As a result, binary interfaces can be sensitive to the compiler options and 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 also suggests that binary module interfaces are not a distribution mechanism and should probably not be installed. Instead, we should install and distribute module interface sources and build systems should be prepared to compile them, again, on the side.

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 relevant `root.build` fragment could look like for a modularized project:

```
cxx.std = experimental

using cxx

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

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

To support C++ modules the `cxx` module (build system) 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 to represent binary module interfaces targets. We normally do not need to mention them explicitly in our buildfiles except, perhaps, to specify additional, module interface-specific compile options. We will see some examples of this below.

To build a modularized executable or library we simply list the module interfaces as its prerequisites, just as we do for 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 this example, the modules support in `build2` 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 before attempting to build anything. To resolve this situation we can either adjust the interface file names or we can specify the module name explicitly with the `cxx.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}@./: cxx.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}@./: cxx.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 (don't ask how).

5.1.3 Module Symbols 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.

The 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 have a unique per-library name (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 the dual, header/module arrangement for backwards compatibility (discussed below). However, for modules-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 the matter.

5.1.4 Modules 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 provided for each library in its `.pc` (`pkg-config`) file.

Specifically, the library's `.pc` file contains the `cxx_modules` variable that lists all the exported C++ 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

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

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

```
cxx_module_symexport.hello.core = true
cxx_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.

5.1.5 Modules Design Guidelines

Modules are a physical design mechanism for structuring and organizing our code. Their explicit exportation semantics combined with the way they are built make many aspects of creating and consuming modules significantly different compared to headers. This section provides basic guidelines for designing modules. We start with the overall considerations such as module granularity and partitioning into translation units then continue with the structure of typical module interface and implementation units. The following section discusses practical approaches to modularizing existing code and providing dual, header/module interfaces for backwards-compatibility.

Unlike headers, the cost of importing modules should be negligible. As a result, it may be tempting to create "mega-modules", for example, one per library. After all, this is how the standard library is modularized with its fairly large `std.core` and `std.io` modules.

There is, however, a significant drawback to this choice: every time we make a change, all consumers of such a mega-module will have to be recompiled, whether the change affects them or not. And the bigger the module the higher the chance that any given change does not (semantically) affect a large portion of the module's consumers. Note also that this is not an issue for the standard library modules since they are not expected to change often.

Another, more subtle, issue with mega-modules (which does affect the standard library) is the inability to re-export only specific interfaces, as will be discussed below.

The other extreme in choosing module granularity is a large number of "mini-modules". Their main drawback is the tediousness of importation by the consumers.

The sensible approach is then to create modules of conceptually-related and commonly-used entities possibly complemented with aggregate modules for ease of importation. This also happens to be generally good design.

As an example, let's consider an XML library that provides support for both parsing and serialization. Since it is common for applications to only use one of the functionalities, it makes sense to provide the `xml.parser` and `xml.serializer` modules. While it is not too tedious to import both, for convenience we could also provide the `xml` module that re-exports the two.

Once we are past selecting an appropriate granularity for our modules, the next question is how to partition them into translation units. A module can consist of just the interface unit and, as discussed above, such a unit can contain anything an implementation unit can, including non-inline function definitions. Some may then view this as an opportunity to get rid of the header/source separation and have everything in a single file.

There are a number of drawbacks with this approach: Every time we change anything in the module interface unit, all its consumers have to be recompiled. If we keep everything in a single file, then every time we change the implementation we trigger recompilations that would have been avoided had the implementation been factored out into a separate unit. Note that a build system in cooperation with the compiler could theoretically avoid such unnecessary recompilations: if the compiler produces identical binary interface files when the module interface is unchanged, then the build system could detect this and skip recompiling the module's consumers.

A related issue with single-file modules is the reduction in the build parallelization opportunities. If the implementation is part of the interface unit, then the build system cannot start compiling the module's consumers until both the interface and the implementation are compiled. On the other hand, had the implementation been split into a separate file, the build system could start compiling the module's consumers (as well as the implementation unit) as soon as the module interface is compiled.

Another issues with combining the interface with the implementation is the readability of the interface which could be significantly reduced if littered with implementation details. We could keep the interface separate by moving the implementation to the bottom of the interface file but then we might as well move it into a separate file and avoid the unnecessary recompilations or parallelization issues.

The sensible guideline is then to have a separate module implementation unit except perhaps for modules with a simple implementation that is mostly inline/template. Note that more complex modules may have several implementation units, however, based on our granularity guideline, those should be rare.

Once we start writing our first real module the immediate question that normally comes up is where to put `#include` directives and `import` declarations and in what order. To recap, a module unit, both interface and implementation, is split into two parts: before the module declaration which obeys the usual or "old" translation unit rules and after the module declaration which is the module purview. Inside the module purview all non-exported declarations have module linkage which means their symbols are invisible to any other module (including the global module). With this understanding, consider the following module interface:

```
export module hello;

#include <string>
```

Do you see the problem? We have included `<string>` in the module purview which means all its names (as well as all the names in any headers it might include, recursively) are now declared as having the `hello` module linkage. The result of doing this can range from silent code blot to strange-looking unresolved symbols.

The guideline this leads to should be clear: including a header in the module purview is almost always a bad idea. There are, however, a few types of headers that may make sense to include in the module purview. The first are headers that only define preprocessor macros, for example, configuration or export headers. There are also cases where we do want the included declarations to end up in the module purview. The most common example is inline/template function implementations that have been factored out into separate files for code organization reasons. As an example, consider the following module interface that uses an export header (which presumably sets up symbols exporting macros) as well as an inline file:

```
#include <string>

export module hello;

#include <libhello/export.hxx>

export namespace hello
{
  ...
}

#include <libhello/hello.ixx>
```

A note on inline/template files: in header-based projects we could include additional headers in those files, for example, if the included declarations are only needed in the implementation. For the reasons just discussed, this does not work with modules and we have to move all the includes into the interface file, before the module purview. On the other hand, with modules, it is safe to use namespace-level using-directives (for example, `using namespace std;`) in inline/template files (and, with care, even in the interface file).

What about imports, where should we import other modules? Again, to recap, unlike a header inclusion, an `import` declaration only makes exported names visible without redeclaring them. As result, in module implementation units, it doesn't really matter where we place imports, in or out of the module purview. There are, however, two differences when it comes to module interface units: only imports in the purview are visible to implementation units and

we can only re-export an imported module from the purview.

The guideline is then for interface units to import in the module purview unless there is a good reason not to make the import visible to the implementation units. And for implementation units to always import in the purview for consistency. For example:

```
#include <cassert>

export module hello;

import std.core;

#include <libhello/export.hxx>

export namespace hello
{
  ...
}

#include <libhello/hello.ixx>
```

By putting all these guidelines together we can then create a module interface unit template:

```
// Module interface unit.

<header includes>

export module <name>;      // Start of module purview.

<module imports>

<special header includes>  // Configuration, export, etc.

<module interface>

<inline/template includes>
```

As well as the module implementation unit template:

```
// Module implementation unit.

<header includes>

module <name>;             // Start of module purview.

<extra module imports>     // Only additional to interface.

<module implementation>
```

Let’s now discuss module naming. Module names are in a separate "name plane" and do not collide with namespace, type, or function names. Also, as mentioned earlier, the standard does not assign a hierarchical meaning to module names though it is customary to assume module `hello.core` is a submodule of `hello` and importing the latter also imports the former.

It is important to choose good names for public modules (that is, modules packaged into libraries and used by a wide range of consumers) since changing them later can be costly. We have more leeway with naming private modules (that is, the ones used by programs or internal to libraries) though it's worth coming up with a consistent naming scheme here as well.

The general guideline is to start names of public modules with the library's namespace name followed by a name describing the module's functionality. In particular, if a module is dedicated to a single class (or, more generally, has a single primary entity), then it makes sense to use its name as the module name's last component.

As a concrete example, consider `libbutl` (the `build2` utility library): All its components are in the `butl` namespace so all its module names start with `butl.` One of its components is the `small_vector` class template which resides in its own module called `butl.small_vector`. Another component is a collection of string parsing utilities that are grouped into the `butl::string_parser` namespace with the corresponding module called `butl.string_parser`.

When is it a good idea to re-export a module? The two straightforward cases are when we are building an aggregate module out of submodules, for example, `xml` out of `xml.parser` and `xml.serializer`, or when one module extends or supersedes another, for example, as `std.core` extends `std.fundamental`. It is also clear that there is no need to re-export a module that we only use in the implementation. The case when we use a module in our interface is, however, a lot less clear cut.

But before considering the last case in more detail, let's understand the issue with re-export. In other words, why not simply re-export any module we import in our interface? In essence, re-export implicitly injects another module import anywhere our module is imported. If we re-export `std.core` then consumers of our module will also automatically "see" all the names exported by `std.core`. They can then start using names from `std` without explicitly importing `std.core` and everything will compile until one day they no longer need to import our module or we no longer need to import `std.core`. In a sense, re-export becomes part of our interface and it is generally good design to keep interfaces minimal.

And so, at the outset, the guideline is then to only re-export the minimum necessary. This, by the way, is the reason why it may make sense to divide `std.core` into submodules such as `std.core.string`, `std.core.vector`, etc.

Let's now discuss a few concrete examples to get a sense of when re-export might or might not be appropriate. Unfortunately, there does not seem to be a hard and fast rule and instead one has to rely on their good sense of design.

To start, let's consider a simple module that uses `std::string` in its interface:

```
export module hello;

import std.core;

export namespace hello
{
  void say (const std::string&);
}
```

Should we re-export `std.core` (or, `std.core.string`) in this case? Most likely not. If consumers of our module want to use `std::string` in order to pass an argument to our function, then it is natural to expect them to explicitly import the necessary module. In a sense, this is analogous to scoping: nobody expects to be able to use just `string` (without `std::`) because of `using namespace hello;`.

So it seems that a mere usage of a name in an interface does not generally warrant a re-export. The fact that a consumer may not even use this part of our interface further supports this conclusion.

Let's now consider a more interesting case (inspired by real events):

```
export module small_vector;

import std.core;

template <typename T, std::size_t N>
export class small_vector: public std::vector<T, ...>
{
  ...
};
```

Here we have the `small_vector` container implemented in terms of `std::vector` by providing a custom allocator and with most of the functions derived as is. Consider now this innocent-looking consumer code:

```
import small_vector;

small_vector<int, 1> a, b;

if (a == b) // Error.
  ...
```

We don't reference `std::vector` directly so presumably we shouldn't need to import its module. However, the comparison won't compile: our `small_vector` implementation re-uses the comparison operators provided by `std::vector` (via implicit to-base conversion) but they aren't visible.

There is a palpable difference between the two cases: the first merely uses `std.core` interface while the second is *based on* and, in a sense, *extends* it which feels like a stronger relationship. Re-exporting `std.core` (or, better yet, `std.core.vector`, should it become available) does not seem unreasonable.

Note also that there is no re-export of headers nor header inclusion visibility in the implementation units. Specifically, in the previous example, if the standard library is not modularized and we have to use it via headers, then the consumers of our `small_vector` will always have to explicitly include `<vector>`. This suggest that modularizing a codebase that still consumes substantial components (like the standard library) via headers can incur some development overhead compared to the old, headers-only approach.

5.1.6 Modularizing Existing Code

The aim of this section is to provide practical guidelines to modularizing existing codebases as well as supporting the dual, header/module interface for backwards-compatibility.

Predictably, a well modularized (in the general sense) set of headers makes conversion to C++ modules easier. Inclusion cycles will be particularly hard to deal with (C++ modules do not allow circular interface dependencies). Furthermore, as we will see below, if you plan to provide the dual header/module interface, then having a one-to-one header to module mapping will simplify this task. As a result, it may make sense to spend some time cleaning and re-organizing your headers prior to attempting modularization.

Let's first discuss why the modularization approach illustrated by the following example does not generally work:

```
export module hello;

export
{
#include "hello.hxx"
}
```

There are several issue that usually make this unworkable. Firstly, the header we are trying to export most likely includes other headers. For example, our `hello.hxx` may include `<string>` and we have already discussed why including it in the module purview, let alone exporting its names, is a bad idea. Secondly, the included header may declare more names than what should be exported, for example, some implementation details. In fact, it may declare names with internal linkage (uncommon for headers but not impossible) which are illegal to export. Finally, the header may define macros which will no longer be visible to the consumers.

Sometimes, however, this can be the only approach available (for example, if trying to non-intrusively modularize a third-party library). It is possible to work around the first issue by *pre-including* outside of the module purview headers that should not be exported. Here we rely on the fact that the second inclusion of the same header will be ignored. For example:

```
#include <string> // Pre-include to suppress inclusion below.

export module hello;

export
{
#include "hello.hxx"
}
```

Needless to say this approach is very brittle and usually requires that you place all the inter-related headers into a single module. As a result, its use is best limited to exploratory modularization and early prototyping.

When starting modularization of a codebase there are two decisions we have to make at the outset: the level of the C++ modules support we can assume and the level of backwards compatibility we need to provide.

The two modules support levels we distinguish are just modules and modules with the modularized standard library. The choice we have to make then is whether to support the standard library only as headers, only as modules, or both. Note that some compiler/standard library combinations may not be usable in some of these modes.

The possible backwards compatibility levels are *modules-only* (consumption via headers is no longer supported), *modules-or-headers* (consumption either via headers or modules), and *modules-and-headers* (as the previous case but with support for consuming a library built with modules via headers and vice versa).

What kind of situations call for the last level? We may need to continue offering the library as headers if we have a large number of existing consumers that cannot possibly be all modularized at once (or even ever). So the situation we may end up in is a mixture of consumers trying to use the same build of our library with some of them using modules and some – headers. The case where we may want to consume a library built with headers via modules is not as far fetched as it may seem: the library might have been built with an older version of the compiler (for example, it was installed from a distribution's package) while the consumer is being built with a compiler version that supports modules. Note also that as discussed earlier the modules ownership semantics supports both kinds of such "cross-usage".

Generally, compiler implementations do not support mixing inclusion and importation of the same entities in the same translation unit. This makes migration tricky if you plan to use the modularized standard library because of its pervasive use. There are two plausible strategies to handling this aspect of migration: If you are planning to consume the standard library exclusively as modules, then it may make sense to first change your entire codebase to do that. Simply replace all the standard library header inclusions with importation of the relevant `std.*` modules.

The alternative strategy is to first complete the modularization of our entire project (as discussed next) while continuing consuming the standard library as headers. Once this is done, we can normally switch to using the modularized standard library quite easily. The reason for waiting until the complete modularization is to eliminate header inclusions between components which would often result in conflicting styles of the standard library consumption.

Note also that due to the lack of header re-export and include visibility support discussed earlier, it may make perfect sense to only support the modularized standard library when modules are enabled even when providing backwards compatibility with headers. In fact, if all the compiler/standard library implementations that your project caters to support the modular-

ized standard library, then there is little sense not to impose such a restriction.

The overall strategy for modularizing our own components is to identify and modularize inter-dependent sets of headers one at a time starting from the lower-level components. This way any newly modularized set will only depend on the already modularized ones. After converting each set we can switch its consumers to using imports keeping our entire project buildable and usable.

While ideally we would want to be able to modularize just a single component at a time, this does not seem to work in practice because we will have to continue consuming some of the components as headers. Since such headers can only be imported out of the module purview, it becomes hard to reason (both for us and often the compiler) what is imported/included and where. For example, it's not uncommon to end up importing the module in its implementation unit which is not something that all the compilers can handle gracefully.

Let's now explore how we can provide the various levels of backwards compatibility discussed above. Here we rely on two feature test macros to determine the available modules support level: `__cpp_modules` (modules are available) and `__cpp_lib_modules` (standard library modules are available, assumes `__cpp_modules` is also defined).

If backwards compatibility is not necessary (the *modules-only* level), then we can use the module interface and implementation unit templates presented earlier and follow the above guidelines. If we continue consuming the standard library as headers, then we don't need to change anything in this area. If we only want to support the modularized standard library, then we simply replace the standard library header inclusions with the corresponding module imports. If we want to support both ways, then we can use the following templates. The module interface unit template:

```
// C includes, if any.

#ifndef __cpp_lib_modules
<std includes>
#endif

// Other includes, if any.

export module <name>;

#ifdef __cpp_lib_modules
<std imports>
#endif

<module interface>
```

The module implementation unit template:

```
// C includes, if any.

#ifndef __cpp_lib_modules
<std includes>

<extra std includes>
#endif
```

```
// Other includes, if any.

module <name>;

#ifdef __cpp_lib_modules
<extra std imports>        // Only additional to interface.
#endif

<module implementation>
```

For example:

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

#ifndef __cpp_lib_modules
#include <string>
#endif

export module hello;

#ifdef __cpp_lib_modules
import std.core;
#endif

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

// hello.cxx (module implementation)

#ifndef __cpp_lib_modules
#include <string>

#include <iostream>
#endif

module hello;

#ifdef __cpp_lib_modules
import std.io;
#endif

using namespace std;

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

If we need support for symbol exporting in this setup (that is, we are building a library and need to support Windows), then we can use the `__symexport` mechanism discussed earlier, for example:

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

...

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

The consumer code in the *modules-only* setup is straightforward: they simply import the desired modules.

To support consumption via headers when modules are unavailable (the *modules-or-headers* level) we can use the following setup. Here we also support the dual header/modules consumption for the standard library (if this is not required, replace `#ifndef __cpp_lib_modules` with `#ifndef __cpp_modules` and remove `#ifdef __cpp_lib_modules`). The module interface unit template:

```
#ifndef __cpp_modules
#pragma once
#endif

// C includes, if any.

#ifndef __cpp_lib_modules
<std includes>
#endif

// Other includes, if any.

#ifdef __cpp_modules
export module <name>;

#ifdef __cpp_lib_modules
<std imports>
#endif
#endif

<module interface>
```

The module implementation unit template:

```
#ifndef __cpp_modules
#include <module interface file>
#endif

// C includes, if any.

#ifndef __cpp_lib_modules
<std includes>

<extra std includes>
#endif

// Other includes, if any

#ifdef __cpp_modules
module <name>;

#ifdef __cpp_lib_modules
<extra std imports>        // Only additional to interface.
#endif
#endif

<module implementation>
```

Notice the need to repeat `<std includes>` in the implementation file due to the lack of include visibility discussed above. This is necessary when modules are enabled but the standard library is not modularized since in this case the implementation does not "see" any of the headers included in the interface.

Besides these templates we will most likely also need an export header that appropriately defines a module export macro depending on whether modules are used or not. This is also the place where we can handle symbol exporting. For example, here is what it could look like for our `libhello` library:

```
// export.hxx (module and symbol export)

#pragma once

#ifdef __cpp_modules
#  define LIBHELLO_MODEXPORT export
#else
#  define LIBHELLO_MODEXPORT
#endif

#if   defined(LIBHELLO_SHARED_BUILD)
#  ifdef _WIN32
#    define LIBHELLO_SYMEXPORT __declspec(dllexport)
#  else
#    define LIBHELLO_SYMEXPORT
#  endif
#elif defined(LIBHELLO_SHARED)
#  ifdef _WIN32
#    define LIBHELLO_SYMEXPORT __declspec(dllimport)
#  else
#    define LIBHELLO_SYMEXPORT
#  endif
#else
#  define LIBHELLO_SYMEXPORT
#endif
```

And this is the module that uses it and provides the dual header/module support:

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

#ifndef __cpp_modules
#pragma once
#endif

#ifndef __cpp_lib_modules
#include <string>
#endif

#ifdef __cpp_modules
export module hello;

#ifdef __cpp_lib_modules
import std.core;
#endif
#endif

#include <libhello/export.hxx>
```

```
LIBHELLO_MODEXPORT namespace hello
{
  LIBHELLO_SYMEXPORT void say (const std::string& name);
}

// hello.cxx (module implementation)

#ifndef __cpp_modules
#include <libhello/hello.mxx>
#endif

#ifndef __cpp_lib_modules
#include <string>

#include <iostream>
#endif

#ifdef __cpp_modules
module hello;

#ifdef __cpp_lib_modules
import std.io;
#endif
#endif

using namespace std;

namespace hello
{
  void say (const string& n)
  {
    cout << "Hello, " << n << '!' << endl;
  }
}
```

The consumer code in the *modules-or-headers* setup has to use either inclusion or importation depending on the modules support availability, for example:

```
#ifdef __cpp_modules
import hello;
#else
#include <libhello/hello.mxx>
#endif
```

Predictably, the final backwards compatibility level (*modules-and-headers*) is the most onerous to support. Here existing consumers have to continue working with the modularized version of our library which means we have to retain all the existing header files. We also cannot assume that just because modules are available they are used (a consumer may still prefer headers), which means we cannot rely on (only) the `__cpp_modules` and `__cpp_lib_modules` macros to make the decisions.

One way to arrange this is to retain the headers and adjust them according to the *modules-or-headers* template but with one important difference: instead of using the standard module macros we use our custom ones (and we can also have unconditional `#pragma once`). For example:

```
// hello.hxx (module header)

#pragma once

#ifndef LIBHELLO_LIB_MODULES
#include <string>
#endif

#ifdef LIBHELLO_MODULES
export module hello;

#ifdef LIBHELLO_LIB_MODULES
import std.core;
#endif
#endif

#include <libhello/export.hxx>

LIBHELLO_MODEXPORT namespace hello
{
  LIBHELLO_SYMEXPORT void say (const std::string& name);
}
```

Now if this header is included (for example, by an existing consumer) then none of the `LIBHELLO_*MODULES` macros will be defined and the header will act as, well, a plain old header. Note that we will also need to make the equivalent change in the export header.

We also provide the module interface files which appropriately define the two custom macros and then simply includes the corresponding headers:

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

#ifdef __cpp_modules
#define LIBHELLO_MODULES
#endif

#ifdef __cpp_lib_modules
#define LIBHELLO_LIB_MODULES
#endif

#include <libhello/hello.hxx>
```

The module implementation unit can remain unchanged. In particular, we continue including `hello.mxx` if modules support is unavailable. However, if you find the use of different macros in the header and source files confusing, then instead it can be adjusted as follows (note also that now we are including `hello.hxx`):

```
// hello.cxx (module implementation)

#ifdef __cpp_modules
#define LIBHELLO_MODULES
#endif

#ifdef __cpp_lib_modules
#define LIBHELLO_LIB_MODULES
#endif

#ifndef LIBHELLO_MODULES
```

```
#include <libhello/hello.hxx>
#endif

#ifndef LIBHELLO_LIB_MODULES
#include <string>

#include <iostream>
#endif

#ifdef LIBHELLO_MODULES
module hello;

#ifdef LIBHELLO_LIB_MODULES
import std.io;
#endif
#endif

...
```

In this case it may also make sense to factor the `LIBHELLO_*MODULES` macro definitions into a common header.

In the *modules-and-headers* setup the existing consumers that would like to continue using headers don't require any changes. And for those that would like to use modules if available the arrangement is the same as for the *modules-or-headers* compatibility level.

If our module needs to "export" macros then the recommended approach is to simply provide an additional header that the consumer includes. While it might be tempting to also wrap the module import into this header, some may prefer to explicitly import the module and include the header, especially if the macros may not be needed by all consumers. This way we can also keep the header macro-only which means it can be included freely, in or out of module purviews.