The `build2` Build System

Revision `0.11`, June 2019
This revision of the document describes the `build2` build system `0.11.x` series.

Table of Contents

Preface 1
1 Introduction 1
1.1 Hello, World 2
1.2 Project Structure 7
1.3 Output Directories and Scopes 15
1.4 Operations 26
1.4.1 Configuration 27
1.4.2 Testing 30
1.4.3 Installation 35
1.4.4 Distribution 38
1.5 Target Importation 40
1.6 Library Exportation and Versioning 43
1.7 Subprojects and Amalgamations 49
1.8 Buildfile Language 53
1.8.1 Expansion and Quoting 55
1.8.2 Conditions (`if-else`) 59
1.8.3 Repetitions (`for`) 61
1.9 Implementing Unit Testing 62
1.10 Diagnostics and Debugging 65
2 Name Patterns 70
3 Variables 73
4 `test` Module 74
5 `version` Module 74
6 `bin` Module 82
7 `cxx` Module 82
7.1 C++ Modules Support 83
7.1.1 Modules Introduction 83
7.1.2 Building Modules 90
7.1.3 Module Symbols Exporting 93
7.1.4 Modules Installation 94
7.1.5 Modules Design Guidelines 95
7.1.6 Modularizing Existing Code 101
8 `in` Module 110
9 `bash` Module 112

Preface

This document describes the `build2` build system. For the build system driver command line interface refer to the **`b(1)`** man pages. For other tools in the `build2` toolchain (package and project managers, etc) see the Documentation index.

1 Introduction

The `build2` build system is a native, cross-platform build system with a terse, mostly declarative description language, a conceptual model of build, and a uniform interface with consistent behavior across platforms and compilers.

Those familiar with `make` will see many similarities, though mostly conceptual rather than syntactic. This is not by accident since `build2` borrows the fundamental DAG-based build model from original `make` and many of its conceptual extensions from GNU `make`. We believe, paraphrasing a famous quote, that *those who do not understand `make` are condemned to reinvent it, poorly.* So our goal with `build2` was to reinvent `make` *well* while handling the demands and complexity of modern cross-platform software development.

Like `make`, `build2` is an *"honest"* build system without magic or black boxes. You can expect to understand what's going on underneath and be able to customize most of its behavior to suit your needs. This is not to say that it's not an *opinionated* build system and if you find yourself "fighting" some of its fundamental design choices, it would probably be wiser to look for alternatives.

We believe the importance and complexity of the problem warranted the design of a new purpose-built language and will hopefully justify the time it takes for you to master it. In the end we hope `build2` will make creating and maintaining build infrastructure for your projects a pleasant task.

Also note that `build2` is not specific to C/C++ or even to compiled languages; its build model is general enough to handle any DAG-based operations. See the `bash` Module for a good example.

While the build system is part of a larger, well-integrated build toolchain that includes the package and project dependency managers, it does not depend on them and its standalone usage is the only subject of this manual.

We begin with a tutorial introduction that aims to show the essential elements of the build system on real examples but without getting into too much detail. Specifically, we want to quickly get to the point where we can build useful executable and library projects.

1.1 Hello, World

Let's start with the customary *"Hello, World"* example: a single source file from which we would like to build an executable:

```
$ tree hello/
hello/
`-- hello.cxx

$ cat hello/hello.cxx

#include <iostream>

int main ()
{
  std::cout << "Hello, World!" << std::endl;
}
```

While this very basic program hardly resembles what most software projects look like today, it is useful for introducing key build system concepts without getting overwhelmed. In this spirit we will also use the `build2` *simple project* structure, which, similarly, should not be used for anything but quick sketches.

To turn our `hello/` directory into a simple project all we need to do is add a `buildfile`:

```
$ tree hello/
hello/
|-- hello.cxx
`-- buildfile

$ cat hello/buildfile

using cxx

exe{hello}: cxx{hello.cxx}
```

Let's start from the bottom: the second line is a *dependency declaration*. On the left hand side of `:` we have a *target*, the `hello` executable, and on the right hand side – a *prerequisite*, the `hello.cxx` source file. Those `exe` and `cxx` in `exe{...}` and `cxx{...}` are called *target types*. In fact, for clarity, target type names are always mentioned with trailing `{}`, for example, "the `exe{}` target type denotes an executable".

Notice that the dependency declaration does not specify *how* to build an executable from a C++ source file – this is the job of a *rule*. When the build system needs to update a target, it tries to *match* a suitable rule based on the types of the target and its prerequisites. The `build2` core has a number of predefined fundamental rules with the rest coming from *build system modules*. For example, the `cxx` module defines a number of rules for compiling C++ source code as well as linking executables and libraries.

It should now be easy to guess what the first line of our `buildfile` does: it loads the `cxx` module which defines the rules necessary to build our program (it also registers the `cxx{}` target type).

Let's now try to build and run our program (`b` is the build system driver):

```
$ cd hello/  # Change to project root.

$ b
c++ cxx{hello}
ld exe{hello}

$ ls -1
buildfile
hello.cxx
hello
hello.d
hello.o
hello.o.d

$ ./hello
Hello, World!
```

Or, if we are on Windows and using Visual Studio, from the Visual Studio development command prompt:

```
> cd hello

> b
c++ cxx{hello}
ld exe{hello}

> dir /b
buildfile
hello.cxx
hello.exe
hello.exe.d
hello.exe.obj
hello.exe.obj.d

> .\hello.exe
Hello, World!
```

By default `build2` uses the same C++ compiler it was built with and without passing any extra options, such as debug or optimization. To change these defaults we use *configuration variables*. For example, to specify a different C++ compiler we use `config.cxx`:

```
$ b config.cxx=clang++
```

And for additional compile options, such as debug information or optimization level, there is `config.cxx.coptions`. For example:

```
$ b config.cxx=clang++ config.cxx.coptions=-g
```

These and other configuration variables will be discussed in more detail later. We will also learn how to make our configuration persistent so that we don't have to repeat such long command lines on every build system invocation.

Similar to `config.cxx`, there is also `config.c` for specifying the C compiler. Note, however, that if your project uses both C and C++, then you normally only need to specify one of them – `build2` will determine the other automatically.

Let's discuss a few points about the build output. Firstly, to reduce the noise, the commands being executed are by default shown abbreviated and with the same target type notation as we used in the `buildfile`. For example:

```
c++ cxx{hello}
ld exe{hello}
```

If, however, you would like to see the actual command lines, you can pass `-v` (to see even more, there is the `-V` as well as `--verbose` options; see **`b(1)`** for details). For example:

```
$ b -v
g++ -o hello.o -c hello.cxx
g++ -o hello hello.o
```

Most of the files produced by the build system should be self-explanatory: we have the object file (`hello.o`, `hello.obj`) and executable (`hello`, `hello.exe`). For each of them we also have the corresponding `.d` files which store the *auxiliary dependency information*, things like compile options, header dependencies, etc.

To remove the build system output we use the `clean` *operation* (if no operation is specified, the default is `update`):

```
$ b clean
rm exe{hello}
rm obje{hello}

$ ls -1
buildfile
hello.cxx
```

One of the main reasons behind the *target type* concept is the platform/compiler-specified variances in file names as illustrated by the above listings. In our `buildfile` we refer to the executable target as `exe{hello}`, not as `hello.exe` or `hello$EXT`. The actual file extension, if any, will be determined based on the compiler's target platform by the rule doing the linking. In this sense, target types are a platform-independent replacement of file extensions (though they do have other benefits, such as allowing non-file targets as well as being hierarchical).

Let's revisit the dependency declaration line from our `buildfile`:

```
exe{hello}: cxx{hello.cxx}
```

In light of target types replacing file extensions this looks tautological: why do we need to specify both the `cxx{}` target type *and* the `.cxx` file extension? In fact, we don't have to if we specify the default file extension for the `cxx{}` target type. Here is our updated `buildfile` in its entirety:

```
using cxx

cxx{*}: extension = cxx

exe{hello}: cxx{hello}
```

Let's unpack the new line. What we have here is a *target type/pattern-specific variable*. It only applies to targets of the `cxx{}` type whose names match the `*` wildcard pattern. The `extension` variable name is reserved by the `build2` core for specifying target type extensions.

Let's see how all these pieces fit together. When the build system needs to update `exe{hello}`, it searches for a suitable rule. A rule from the `cxx` module matches since it knows how to build a target of type `exe{}` from a prerequisite of type `cxx{}`. When the matched rule is *applied*, it searches for a target for the `cxx{hello}` prerequisite. During this search, the `extension` variable is looked up and its value is used to end up with the `hello.cxx` file.

Here is our new dependency declaration again:

```
exe{hello}: cxx{hello}
```

It has the canonical form: no extensions, only target types. Sometimes explicit extension specification is still necessary, for example, if your project uses multiple extensions for the same file type. But if unnecessary, it should be omitted for brevity.

If you prefer the `.cpp` file extension and your source file is called `hello.cpp`, then the only line in our `buildfile` that needs changing is the `extension` variable assignment:

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

Let's say our `hello` program got complicated enough to warrant moving some functionality into a separate source/header module (or a real C++ module). For example:

```
$ tree hello/
hello/
|-- hello.cxx
|-- utility.hxx
|-- utility.cxx
`-- buildfile
```

This is what our updated `buildfile` could look like:

```
using cxx

hxx{*}: extension = hxx
cxx{*}: extension = cxx

exe{hello}: cxx{hello} hxx{utility} cxx{utility}
```

Nothing really new here: we've specified the default extension for the `hxx{}` target type and listed the new header and source files as prerequisites. If you have experience with other build systems, then explicitly listing headers might seem strange to you. As will be discussed later, in `build2` we have to explicitly list all the prerequisites of a target that should end up in a distribution of our project.

You don't have to list *all* headers that you include, only the ones belonging to your project. Like all modern C/C++ build systems, `build2` performs automatic header dependency extraction.

In real projects with a substantial number of source files, repeating target types and names will quickly become noisy. To tidy things up we can use *name generation*. Here are a few examples of dependency declarations equivalent to the above:

```
exe{hello}: cxx{hello utility} hxx{utility}
exe{hello}: cxx{hello} {hxx cxx}{utility}
```

The last form is probably the best choice if your project contains a large number of header/source pairs. Here is a more realistic example:

```
exe{hello}: {    cxx}{hello}               \
            {hxx    }{forward types}       \
            {hxx cxx}{format print utility}
```

Manually listing a prerequisite every time we add a new source file to our project is both tedious and error prone. Instead, we can automate our dependency declarations with *wildcard name patterns*. For example:

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

Based on the previous discussion of default extensions, you can probably guess how this works: for each target type the value of the `extension` variable is added to the pattern and files matching the result become prerequisites. So, in our case, we will end up with files matching the `*.hxx` and `*.cxx` wildcard patterns.

In more complex projects it is often convenient to organize source code into subdirectories. To handle such projects we can use the recursive wildcard:

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

Using wildcards is somewhat controversial. Patterns definitely make development more pleasant and less error prone: you don't need to update your `buildfile` every time you add, remove, or rename a source file and you won't forget to explicitly list headers, a mistake that is often only detected when trying to build a distribution of a project. On the other hand, there is the possibility of including stray source files into your build without noticing. And, for more complex projects, name patterns can become fairly complex (see Name Patterns for details). Note also that on modern hardware the performance of wildcard searches hardly warrants a consideration.

In our experience, when combined with modern version control systems like `git(1)`, stray source files are rarely an issue and generally the benefits of wildcards outweigh their drawbacks. But, in the end, whether to use them or not is a personal choice and, as shown above, `build2` supports both approaches.

And that's about all there is to our `hello` example. To summarize, we've seen that to build a simple project we need a single `buildfile` which itself doesn't contain much more than a dependency declaration for what we want to build. But we've also mentioned that simple projects are only really meant for quick sketches. So let's convert our `hello` example to the *standard project* structure which is what we will be using for most of our real development.

Simple projects have so many restrictions and limitations that they are hardly usable for anything but, well, *really* simple projects. Specifically, such projects cannot be imported by other projects nor can they use build system modules that require bootstrapping. This includes `test`, `install`, `dist`, and `config` modules. And without the `config` module there is no support for persistent configurations. As a result, you should only use a simple project if you are happy to always build in the source directory and with the default build configuration or willing to specify the output directory and/or custom configuration on every invocation.

1.2 Project Structure

A `build2` *standard project* has the following overall layout:

```
hello/
|-- build/
|   |-- bootstrap.build
|   `-- root.build
|-- ...
`-- buildfile
```

Specifically, the project's root directory should contain the `build/` subdirectory as well as the root `buildfile`. The `build/` subdirectory contains project-wide build system information.

The **`bdep-new(1)`** command is an easy way to create the standard layout executable (`-t exe`) and library (`-t lib`) projects. To change the C++ file extensions to `.hpp/.cpp`, pass `-l c++,cpp`. For example:

```
$ bdep new --no-init -t exe -l c++,cpp hello
```

It is also possible to use an alternative build file/directory naming scheme where every instance of the word *build* is replaced with *build2*, for example:

```
hello/
|-- build2/
|   |-- bootstrap.build2
|   `-- root.build2
|-- ...
`-- build2file
```

Note that the naming must be consistent within a project with all the filesystem entries either following *build* or *build2* scheme. In other words, we cannot call the directory `build2/` while still using `buildfile`.

The alternative naming scheme is primarily useful when adding `build2` support to an existing project along with other build systems. In this case, the fairly generic standard names might already be in use. For example, it is customary to have `build/` in `.gitignore`. Plus more specific naming will make it easier to identify files and directories as belonging to the `build2` support. For new projects as well as for existing projects that are switching exclusively to `build2` the standard naming scheme is recommended.

To create a project with the alternative naming using **`bdep-new(1)`** pass the `alt-naming` project type sub-option. For example:

```
$ bdep new -t exe,alt-naming ...
```

To support lazy loading of subprojects (discussed later), reading of the project's build information is split into two phases: bootstrapping and loading. During bootstrapping the project's `build/bootstrap.build` file is read. Then, when (and if) the project is loaded completely, its `build/root.build` file is read followed by the `buildfile` (normally from the project root but possibly from a subdirectory).

The `bootstrap.build` file is required. Let's see what it would look like for a typical project using our `hello` as an example:

```
project = hello

using version
using config
using test
using install
using dist
```

The first non-comment line in `bootstrap.build` should be the assignment of the project name to the `project` variable. After that, a typical `bootstrap.build` file loads a number of build system modules. While most modules can be loaded during the project load phase in `root.build`, certain modules have to be loaded early, while bootstrapping (for example, because they define new operations).

Let's examine briefly the modules loaded by our `bootstrap.build`: The `version` module helps with managing our project versioning. With this module we only maintain the version in a single place (project's `manifest` file) and it is automatically made available in various convenient forms throughout our project (`buildfiles`, header files, etc). The `version` module also automates versioning of snapshots between releases.

The `manifest` file is what makes our build system project a *package*. It contains all the metadata that a user of a package might need to know: name, version, dependencies, etc., all in one place. However, even if you don't plan to package your project, it is a good idea to create a basic `manifest` if only to take advantage of the version management offered by the `version` module. So let's go ahead and add it next to our root `buildfile`:

```
$ tree hello/
hello/
|-- build/
|   `-- ...
|-- ...
|-- buildfile
`-- manifest

$ cat hello/manifest
: 1
name: hello
version: 0.1.0
summary: hello C++ executable
```

The `config` module provides support for persistent configurations. While project configuration is a large topic that we will discuss in detail later, in a nutshell `build2` support for configuration is an integral part of the build system with the same mechanisms available to the build system core, modules, and your projects. However, without `config`, the configuration information is *transient*. That is, whatever configuration information was automatically discovered or that you have supplied on the command line is discarded after each build system invocation. With the `config` module, however, we can *configure* a project to make the configuration *persistent*. We will see an example of this shortly.

Next up are the `test`, `install`, and `dist` modules. As their names suggest, they provide support for testing, installation and preparation of distributions. Specifically, the `test` module defines the `test` operation, the `install` module defines the `install` and `uninstall` operations, and the `dist` module defines the `dist` (meta-)operation. Again, we will try them out in a moment.

Moving on, the `root.build` file is optional though most projects will have it. This is the place where we normally establish project-wide settings as well as load build system modules that provide support for the languages/tools that we use. Here is what it could look like for our `hello` example:

```
cxx.std = latest

using cxx

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

As you can see, we've moved the loading of the `cxx` modules and setting of the default file extensions from the root `buildfile` in our simple project to `root.build` when using the standard layout. We've also set the `cxx.std` variable to tell the `cxx` module to select the latest C++ standard available in any particular C++ compiler this project might be built with.

Selecting the C++ standard for our project is a messy issue. If we don't specify the standard explicitly with `cxx.std`, then the default standard in each compiler will be used, which, currently, can range from C++98 to C++14. So unless you carefully write your code to work with any standard, this is probably not a good idea.

Fixing the standard (for example, to `c++11`, `c++14`, etc) should work theoretically. In practice, however, compilers add support for new standards incrementally and many versions, while perfectly usable, are not feature-complete. As a result, a better practical strategy is to specify the set of minimum supported compiler versions rather than the C++ standard.

There is also the issue of using libraries that require newer standard in older code. For example, headers from a library that relies on C++14 features will not compile when included in a project that is built as C++11. And, even if the headers compile (that is, C++14 features are only used in the implementation), strictly speaking, there is no guarantee that codebases compiled with different C++ standards are ABI compatible (in fact, some changes to the C++ language leave the implementations no choice but to break the ABI).

As result, our recommendation is to set the standard to `latest` and specify the minimum supported compilers and versions in your project's documentation (see package manifest `requires` value for one possible place). Practically, this should allow you to include and link any library, regardless of the C++ standard that it uses.

Let's now take a look at the root `buildfile`:

```
./: {*/ -build/}
```

In plain English, this `buildfile` declares that building this directory (and, since it's the root of our project, building this entire project) means building all its subdirectories excluding `build/`. Let's now try to understand how this is actually achieved.

We already know this is a dependency declaration, `./` is the target, and what's after `:` are its prerequisites, which seem to be generated with some kind of a name pattern (the wildcard character in `*/` should be the giveaway). What's unusual about this declaration, however, is the lack of any target types plus that strange-looking `./`.

Let's start with the missing target types. In fact, the above `buildfile` can be rewritten as:

```
dir{.}: dir{* -build}
```

So the trailing slash (always forward, even on Windows) is a special shorthand notation for `dir{}`. As we will see shortly, it fits naturally with other uses of directories in `buildfiles` (for example, in scopes).

The `dir{}` target type is an *alias* (and, in fact, is derived from more general `alias{}`). Building it means building all its prerequisites.

If you are familiar with `make`, then you can probably see the similarity with the ubiquitous `all` pseudo-target. In `build2` we instead use directory names as more natural aliases for the "build everything in this directory" semantics.

Note also that `dir{}` is purely an alias and doesn't have anything to do with the filesystem. In particular, it does not create any directories. If you do want explicit directory creation (which should be rarely needed), use the `fsdir{}` target type instead.

The `./` target is a special *default target*. If we run the build system without specifying the target explicitly, then this target is built by default. Every `buildfile` has the `./` target. If we don't declare it explicitly, then its declaration is implied with the first target in the `buildfile` as its prerequisite. Recall our `buildfile` from the simple `hello` project:

```
exe{hello}: cxx{hello}
```

It is equivalent to:

```
./: exe{hello}
exe{hello}: cxx{hello}
```

If, however, we had several targets in the same directory that we wanted built by default, then we would need to explicitly list them as prerequisites of the default target. For example:

```
./: exe{hello}
exe{hello}: cxx{hello}

./: exe{goodby}
exe{goodby}: cxx{goodby}
```

While straightforward, this is somewhat inelegant in its repetitiveness. To tidy things up we can use *dependency declaration chains* that allow us to chain together several target-prerequisite declarations in a single line. For example:

```
./: exe{hello}: cxx{hello}

./: exe{goodby}: cxx{goodby}
```

With dependency chains a prerequisite of the preceding target becomes a target itself for the following prerequisites.

Let's get back to our root `buildfile`:

```
./: {*/ -build/}
```

The last unexplained bit is the `{*/ -build/}` name pattern. All it does is exclude `build/` from the subdirectories to build. See Name Patterns for details.

Let's take a look at a slightly more realistic root `buildfile`:

```
./: {*/ -build/} doc{README.md LICENSE} manifest
```

Here we have the customary `README.md` and `LICENSE` files as well as the package `manifest`. Listing them as prerequisites achieves two things: they will be installed if/when our project is installed and, as mentioned earlier, they will be included into the project distribution.

The `README.md` and `LICENSE` files use the `doc{}` target type. We could have used the generic `file{}` but using the more precise `doc{}` makes sure that they are installed into the appropriate documentation directory. The `manifest` file doesn't need an explicit target type since it has a fixed name (`manifest{manifest}` is valid but redundant).

Standard project infrastructure in place, where should we put our source code? While we could have everything in the root directory of our project, just like we did with the simple layout, it is recommended to instead place the source code into a subdirectory named the same as the project. For example:

```
hello/
|-- build/
|   `-- ...
|-- hello/
|   |-- hello.cxx
|   `-- buildfile
|-- buildfile
|-- manifest
`-- README.md
```

There are several reasons for this layout: It implements the canonical inclusion scheme where each header is prefixed with its project name. It also has a predictable name where users can expect to find our project's source code. Finally, this layout prevents clutter in the project's root directory which usually contains various other files. See Canonical Project Structure for more information.

Note also that while we can name our header and source files however we like (but, again, see Canonical Project Structure for some sensible guidelines), C++ module interface files need to embed a sufficient amount of the module name suffix in their names to unambiguously resolve all the modules within a project. See Building Modules for details.

The source subdirectory `buildfile` is identical to the simple project's minus the parts moved to `root.build`:

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

Let's now build our project and see where the build system output ends up in this new layout:

```
$ cd hello/  # Change to project root.
$ b
c++ hello/cxx{hello}
ld hello/exe{hello}

$ tree ./
./
|-- build/
|   `-- ...
|-- hello/
|   |-- hello.cxx
|   |-- hello
|   |-- hello.d
|   |-- hello.o
|   |-- hello.o.d
|   `-- buildfile
|-- buildfile
`-- manifest

$ hello/hello
Hello, World!
```

If we don't specify a target to build (as we did above), then `build2` will build the current directory or, more precisely, the default target in the `buildfile` in the current directory. We can also build a directory other than the current, for example:

```
$ b hello/
```

Note that the trailing slash is required. In fact, `hello/` in the above command line is a target and is equivalent to `dir{hello}`, just like in the `buildfiles`.

Or we can build a specific target:

```
$ b hello/exe{hello}
```

Naturally, nothing prevents us from building multiple targets or even projects in the same build system invocation. For example, if we had the `libhello` project next to our `hello/`, then we could build both at once:

```
$ ls -1
hello/
libhello/

$ b hello/ libhello/
```

Speaking of libraries, let's see what the standard project structure looks like for one, using `libhello` created by **`bdep-new(1)`** as an example:

```
$ bdep new --no-init -t lib libhello

$ tree libhello/
libhello/
|-- build/
|   |-- bootstrap.build
|   |-- root.build
|   `-- export.build
|-- libhello/
|   |-- hello.hxx
|   |-- hello.cxx
|   |-- export.hxx
|   |-- version.hxx.in
|   `-- buildfile
|-- tests/
|   `--  ...
|-- buildfile
|-- manifest
`-- README.md
```

The overall layout (`build/`, `libhello/` source directory) as well as the contents of the root files (`bootstrap.build`, `root.build`, root `buildfile`) are exactly the same. There is, however, a new file, `export.build`, in `build/`, a new subdirectory, `tests/`, and the contents of the project's source subdirectory, `libhello/`, look quite a bit different. We will examine all of these differences in the coming sections, as we learn more about the build system.

The standard project structure is not type (executable, library, etc) or even language specific. In fact, the same project can contain multiple executables and/or libraries (for example, both `hello` and `libhello`). However, if you plan to package your projects, it is a good idea to keep them as separate build system projects (they can still reside in the same version control repository, though).

Speaking of projects, this term is unfortunately overloaded to mean two different things at different levels of software organization. At the bottom we have *build system projects* which, if packaged, become *packages*. And at the top, related packages are often grouped into what is also commonly referred to as *projects*. At this point both usages are probably too well established to look for alternatives.

And this completes the conversion of our simple `hello` project to the standard structure. Earlier, when examining `bootstrap.build`, we mentioned that modules loaded in this file usually provide additional operations. So we still need to discuss what exactly the term *build system operation* means and see how to use operations that are provided by the modules we have loaded. But before we do that, let's see how we can build our projects *out of source* tree and learn about another cornerstone `build2` concept: *scopes*.

1.3 Output Directories and Scopes

Two common requirements placed on modern build systems are the ability to build projects out of the source directory tree (referred to as just *out of source* vs *in source*) as well as isolation of `buildfiles` from each other when it comes to target and variable names. In `build2` these mechanisms are closely-related, integral parts of the build system.

This tight integration has advantages, like being always available and working well with other build system mechanisms, as well as disadvantages, like the inability to implement a completely different out of source arrangement and/or isolation model. In the end, if you find yourself "fighting" this aspect of `build2`, it will likely be easier to use a different build system than subvert it.

Let's start with an example of an out of source build for our `hello` project. To recap, this is what we have:

```
$ ls -1
hello/

$ tree hello/
hello/
|-- build/
|   `-- ...
|-- hello/
|   `-- ...
|-- buildfile
`-- manifest
```

To start, let's build it in the `hello-out/` directory next to the project:

```
$ b hello/@hello-out/
mkdir fsdir{hello-out/}
mkdir hello-out/fsdir{hello/}
c++ hello/hello/cxx{hello}@hello-out/hello/
ld hello-out/hello/exe{hello}
```

```
$ ls -1
hello/
hello-out/

$ tree hello-out/
hello-out/
`-- hello/
    |-- hello
    |-- hello.d
    |-- hello.o
    `-- hello.o.d
```

This definitely requires some explaining. Let's start from the bottom, with the `hello-out/` layout. It is *parallel* to the source directory. This mirrored side-by-side listing (of the relevant parts) should illustrate this clearly:

```
hello/             ~~>  hello-out/
`-- hello/         ~~>  `-- hello/
    `-- hello.cxx  ~~>      `-- hello.o
```

In fact, if we copy the contents of `hello-out/` over to `hello/`, we will end up with exactly the same result as in the in source build. And this is not accidental: an in source build is just a special case of an out of source build where the *out* directory is the same as *src*.

In `build2` this parallel structure of the out and src directories is a cornerstone design decision and is non-negotiable, so to speak. In particular, out cannot be inside src. And while we can stash the build system output (object files, executables, etc) into (potentially different) subdirectories, this is not recommended. As will be shown later, `build2` offers better mechanisms to achieve the same benefits (like reduced clutter, ability to run executables) but without the drawbacks (like name clashes).

Let's now examine how we invoked the build system to achieve this out of source build. Specifically, if we were building in source, our command line would have been:

```
$ b hello/
```

but for the out of source build, we have:

```
$ b hello/@hello-out/
```

In fact, that strange-looking construct, `hello/@hello-out/` is just a more elaborate target specification that explicitly spells out the target's src and out directories. Let's add an explicit target type to make it clearer:

```
$ b hello/@hello-out/dir{.}
```

What we have on the right of `@` is the target in the out directory and on the left – its src directory. In plain English, this command line says "build me the default target from `hello/` in the `hello-out/` directory".

As an example, if instead we wanted to build only the `hello` executable out of source, then the invocation would have looked like this:

```
$ b hello/hello/@hello-out/hello/exe{hello}
```

We could have also specified out for an in source build, but that's redundant:

```
$ b hello/@hello/
```

There is another example of this elaborate target specification in the build diagnostics:

```
c++ hello/hello/cxx{hello}@hello-out/hello/
```

Notice, however, that now the target (`cxx{hello}`) is on the left of `@`, that is, in the src directory. It does, however, make sense if you think about it: our `hello.cxx` is a *source file*, it is not built and it resides in the project's source directory. This is in contrast, for example, to the `exe{hello}` target which is the output of the build system and goes to the out directory. So in `build2` targets can be either in src or in out (there can also be *out of any project* targets, for example, installed files).

The elaborate target specification can also be used in `buildfiles`. We haven't encountered any so far because targets mentioned without explicit src/out default to out and, naturally, most of the targets we mention in `buildfiles` are things we want built. One situation where you may encounter an src target mentioned explicitly is when specifying its installability (discussed in the next section). For example, if our project includes the customary `INSTALL` file, it probably doesn't make sense to install it. However, since it is a source file, we have to use the elaborate target specification when disabling its installation:

```
doc{INSTALL}@./: install = false
```

Note also that only targets and not prerequisites have this notion of src/out directories. In a sense, prerequisites are relative to the target they are prerequisites of and are resolved to targets in a manner that is specific to their target types. For `file{}`-based prerequisites the corresponding target in out is first looked up and if found used. Otherwise, an existing file in src is searched for and if found the corresponding target (now in src) is used. In particular, this semantics gives preference to generated code over static.

More precisely, a prerequisite is relative to the scope (discussed below) in which the dependency is declared and not to the target that it is a prerequisite of. However, in most practical cases, this means the same thing.

And this pretty much covers out of source builds. Let's summarize the key points we have established so far: Every build has two parallel directory trees, src and out, with the in source build being just a special case where they are the same. Targets in a project can be either in the src or out directory though most of the time targets we mention in our `buildfiles` will be in out, which is the default. Prerequsites are relative to targets they are prerequisites of and `file{}`-based prerequisites are first searched for as declared targets in out and then as existing files in src.

Note also that we can have as many out of source builds as we want and we can place them anywhere we want (but not inside src), say, on a RAM-backed disk/filesystem. As an example, let's build our `hello` project with two different compilers:

```
$ b hello/@hello-gcc/    config.cxx=g++
$ b hello/@hello-clang/  config.cxx=clang++
```

In the next section we will see how to permanently configure our out of source builds so that we don't have to keep repeating these long command lines.

While technically you can have both in source and out of source builds at the same time, this is not recommended. While it may work for basic projects, as soon as you start using generated source code (which is fairly common in `build2`), it becomes difficult to predict where the compiler will pick generated headers. There is support for remapping mis-picked headers but this may not always work with older C/C++ compilers. Plus, as we will see in the next section, `build2` supports *forwarded configurations* which provide most of the benefits of an in source build but without the drawbacks.

Let's now turn to `buildfile` isolation. It is a common, well-established practice to organize complex software projects in directory hierarchies. One of the benefits of this organization is isolation: we can use the same, short file names in different subdirectories. In `build2` the project's directory tree is used as a basis for its *scope* hierarchy. In a sense, scopes are like C++ namespaces that automatically track the project's filesystem structure and use directories as their names. The following listing illustrates the parallel directory and scope hierarchies for our `hello` project. The `build/` subdirectory is special and does not have a corresponding scope.

```
hello/                  hello/
|                       {
`-- hello/                hello/
    |                     {
    `-- ...                 ...
                          }
                        }
```

Every `buildfile` is loaded in its corresponding scope, variables set in a `buildfile` are set in this scope and relative targets mentioned in a `buildfile` are relative to this scope's directory. Let's "load" the `buildfile` contents from our `hello` project to the above listing:

```
hello/                  hello/
|                       {
|-- buildfile             ./: {*/ -build/}
|
`-- hello/                hello/
    |                     {
    `-- buildfile           exe{hello}: {hxx cxx}{**}
                          }
                        }
```

In fact, to be absolutely precise, we should also add the contents of `bootstrap.build` and `root.build` to the project's root scope (module loading is omitted for brevity):

```
hello/                  hello/
|                       {
|-- build/
|   |-- bootstrap.build   project = hello
|   |
|   `-- root.build        cxx.std = latest
|                         hxx{*}: extension = hxx
|                         cxx{*}: extension = cxx
|
|-- buildfile             ./: {*/ -build/}
|
`-- hello/                hello/
    |                     {
    `-- buildfile           exe{hello}: {hxx cxx}{**}
                          }
                        }
```

The above scope structure is very similar to what you will see (besides a lot of other things) if you build with `--dump match`. With this option the build system driver dumps the build state after matching rules to targets (see Diagnostics and Debugging for more information). Here is an abbreviated output of bulding our `hello` with `--dump` (assuming an in source build in `/tmp/hello`):

```
$ b --dump match

/
{
  [target_triplet] build.host = x86_64-linux-gnu
  [string] build.host.class = linux
  [string] build.host.cpu = x86_64
  [string] build.host.system = linux-gnu

  /tmp/hello/
  {

    [dir_path] src_root = /tmp/hello/
    [dir_path] out_root = /tmp/hello/

    [dir_path] src_base = /tmp/hello/
    [dir_path] out_base = /tmp/hello/
```

```
    [project_name] project = hello
    [string] project.summary = hello executable
    [string] project.url = https://example.org/hello

    [string] version = 1.2.3
    [uint64] version.major = 1
    [uint64] version.minor = 2
    [uint64] version.patch = 3

    [string] cxx.std = latest

    [string] cxx.id = gcc
    [string] cxx.version = 8.1.0
    [uint64] cxx.version.major = 8
    [uint64] cxx.version.minor = 1
    [uint64] cxx.version.patch = 0

    [target_triplet] cxx.target = x86_64-w64-mingw32
    [string] cxx.target.class = windows
    [string] cxx.target.cpu = x86_64
    [string] cxx.target.system = mingw32

    hxx{*}: [string] extension = hxx
    cxx{*}: [string] extension = cxx

    hello/
    {
      [dir_path] src_base = /tmp/hello/hello/
      [dir_path] out_base = /tmp/hello/hello/

      dir{./}: exe{hello}
      exe{hello.}: cxx{hello.cxx}
    }

    dir{./}: dir{hello/} manifest{manifest}
  }
}
```

This is probably quite a bit more information than what you've expected to see so let's explain a couple of things. Firstly, it appears there is another scope outer to our project's root. In fact, `build2` extends scoping outside of projects with the root of the filesystem (denoted by the special `/`) being the *global scope*. This extension becomes useful when we try to build multiple unrelated projects or import one project into another. In this model all projects are part of a single scope hierarchy with the global scope at its root.

The global scope is read-only and contains a number of pre-defined *build-wide* variables such as the build system version, host platform (shown in the above listing), etc.

Next, inside the global scope, we see our project's root scope (`/tmp/hello/`). Besides the variables that we have set ourselves (like `project`), it also contains a number of variables set by the build system core (for example, `out_base`, `src_root`, etc) as well by build system modules (for example, `project.*` and `version.*` variables set by the `version` module and `cxx.*`

variables set by the `cxx` module).

The scope for our project's source directory (`hello/`) should look familiar. We again have a few special variables (`out_base`, `src_base`). Notice also that the name patterns in prerequisites have been expanded to the actual files.

As you can probably guess from their names, the `src_*` and `out_*` variables track the association between scopes and src/out directories. They are maintained automatically by the build system core with the `src/out_base` pair set on each scope within the project and an additional `src/out_root` pair set on the project's root scope so that we can get the project's root directories from anywhere in the project. Note that directory paths in these variables are always absolute and normalized.

In the above example the corresponding src/out variable pairs have the same values because we were building in source. As an example, this is what the association will look like for an out of source build:

```
hello/  ~~>      hello-out/                   <~~  hello-out/
|                {                                 |
|                  src_root = .../hello/           |
|                  out_root = .../hello-out/       |
|                                                  |
|                  src_base = .../hello/           |
|                  out_base = .../hello-out/       |
|                                                  |
`-- hello/  ~~>    hello/                     <~~  `-- hello/
                   {
                     src_base = .../hello/hello/
                     out_base = .../hello-out/hello/
                   }
                 }
```

Now that we have some scopes and variables to play with, it's a good time to introduce variable expansion. To get the value stored in a variable we use `$` followed by the variable's name. The variable is first looked up in the current scope (that is, the scope in which the expansion was encountered) and, if not found, in the outer scopes all the way to the global scope.

To be precise, this is for the default *variable visibility*. Variables, however, can have more limited visibilities, such as *project*, *scope*, *target*, or *prerequisite*.

To illustrate the lookup semantics, let's add the following line to each `buildfile` in our `hello` project:

```
$ cd hello/  # Change to project root.

$ cat buildfile
...
info "src_base: $src_base"

$ cat hello/buildfile
...
info "src_base: $src_base"
```

And then build it:

```
$ b
buildfile:3:1: info: src_base: /tmp/hello/
hello/buildfile:8:1: info: src_base: /tmp/hello/hello/
```

In this case `src_base` is defined in each of the two scopes and we get their respective values. If, however, we change the above line to print `src_root` instead of `src_base`, we will get the same value from the root scope:

```
buildfile:3:1: info: src_root: /tmp/hello/
hello/buildfile:8:1: info: src_root: /tmp/hello/
```

In this section we've only scratched the surface when it comes to variables. In particular, variables and variable values in `build2` are optionally typed (those `[string]`, `[uint64]` we've seen in the build state dump). And in certain contexts the lookup semantics actually starts from the target, not from the scope (target-specific variables; there are also prerequisite-specific). These and other variable-related topics will be covered in subsequent sections.

One typical place to find `src/out_root` expansions is in the include search path options. For example, the source directory `buildfile` generated by **`bdep-new(1)`** for an executable project actually looks like this (`poptions` stands for *preprocessor options*):

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

cxx.poptions =+ "-I$out_root" "-I$src_root"
```

The strange-looking =+ line is a *prepend* variable assignment. It adds the value on the right hand side to the beginning of the existing value. So, in the above example, the two header search paths will be added before any of the existing preprocessor options (and thus will be considered first).

There are also the *append* assignment, +=, which adds the value on the right hand side to the end of the existing value, as well as, of course, the normal or *replace* assignment, =, which replaces the existing value with the right hand side. One way to remember where the existing and new values end up in the =+ and += results is to imagine the new value taking the position of = and the existing value – of +.

The above `buildfile` allows us to include our headers using the project's name as a prefix, inline with the Canonical Project Structure guidelines. For example, if we added the `utility.hxx` header to our `hello` project, we would include it like this:

```
#include <iostream>

#include <hello/utility.hxx>

int main ()
{
...
}
```

Besides `poptions`, there are also `coptions` (compile options), `loptions` (link options) and `libs` (extra libraries to link). If you are familiar with `make`, these are roughly equivalent to `CPPFLAGS`, `CFLAGS`/`CXXFLAGS`, `LDFLAGS`, and `LIBS`.

More specifically, there are three sets of these variables: `cc.*` (stands for *C-common*) which applies to all C-like languages as well as `c.*` and `cxx.*` which only apply during the C and C++ compilation, respectively. We can use these variables in our `buildfiles` to adjust the compiler/linker behavior. For example:

```
if ($cc.class == 'gcc')
{
  cc.coptions  += -fno-strict-aliasing  # C and C++
  cxx.coptions += -fno-exceptions       # only C++
}

if ($c.target.class != 'windows')
  c.libs += -lpthread  # only C
```

Additionally, as we will see in Configuration, there are also the `config.cc.*`, `config.c.*`, and `config.cxx.*` sets which are used by the users of our projects to provide external configuration. The initial values of the `cc.*`, `c.*`, and `cxx.*` variables are taken from the corresponding `config.*.*` values.

And, as we will learn in Library Exportation, there are also the `cc.export.*`, `c.export.*`, and `cxx.export.*` sets that are used to specify options that should be exported to the users of our library.

If we adjust the `cc.*`, `c.*`, and `cxx.*` variables at the scope level, as in the above fragment, then the changes will apply when building every target in this scope (as well as in the nested scopes, if any). Usually this is what we want but sometimes we may need to pass additional options only when compiling certain source files or linking certain libraries or executables. For that we use the target-specific variable assignment. For example:

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

obj{utility}: cxx.poptions += -DNDEBUG
exe{hello}: cxx.loptions += -static
```

Note that we set these variables on targets which they affect. In particular, those with a background in other build systems may, for example, erroneously expect that setting `poptions` on a library target will affect compilation of its prerequisites. For example, the following does not work:

```
exe{hello}: cxx.poptions += -DNDEBUG
```

The recommended way to achieve this behavior in `build2` is to organize your targets into subdirectories, in which case we can just set the variables on the scope. And if this is impossible or undesirable, then we can use target type/pattern-specific variables (if there is a common pattern) or simply list the affected targets explicitly. For example:

```
obj{*.test}: cxx.poptions += -DDEFINE_MAIN
obj{main utility}: cxx.poptions += -DNDEBUG
```

The first line covers compilation of source files that have the `.test` second-level extension (see Implementing Unit Testing for background) while the second simply lists the targets explicitly.

It is also possible to specify different options when producing different types of object files (`obje{}` – executable, `obja{}` – static library, or `objs{}` – shared library) or when linking different libraries (`liba{}` – static library or `libs{}` – shared library). See Library Exportation and Versioning for an example.

As mentioned above, each `buildfile` in a project is loaded into its corresponding scope. As a result, we rarely need to open scopes explicitly. In the few cases that we do, we use the following syntax:

```
<directory>/
{
  ...
}
```

If the scope directory is relative, then it is assumed to be relative to the current scope. As an exercise for understanding, let's reimplement our `hello` project as a single `buildfile`. That is, we move the contents of the source directory `buildfile` into the root `buildfile`:

```
$ tree hello/
hello/
|-- build/
|   `-- ...
|-- hello/
|   `-- hello.cxx
`-- buildfile
```

```
$ cat hello/buildfile

./: hello/

hello/
{
  ./: exe{hello}: {hxx cxx}{**}
}
```

While this single `buildfile` setup is not recommended for new projects, it can be useful for non-intrusive conversion of existing projects to `build2`. One approach is to place the unmodified original project into a subdirectory (potentially automating this with a mechanism such as `git(1)` submodules) then adding the `build/` subdirectory and the root `buildfile` which explicitly opens scopes to define the build over the upstream project's subdirectory structure.

Seeing this merged `buildfile` may make you wonder what exactly caused the loading of the source directory `buildfile` in our normal setup. In other words, when we build our `hello` from the project root, who and why loads `hello/buildfile`?

Actually, in the earlier days of `build2`, we had to explicitly load `buildfiles` that define targets we depend on with the `include` directive. In fact, we still can (and have to if we are depending on targets other than directories). For example:

```
./: hello/

include hello/buildfile
```

We can also omit `buildfile` for brevity and have just:

```
include hello/
```

This explicit inclusion, however, quickly becomes tiresome as the number of directories grows. It also makes using wildcard patterns for subdirectory prerequisites a lot less appealing.

To overcome this the `dir{}` target type implements an interesting prerequisite to target resolution semantics: if there is no existing target with this name, a `buildfile` that (presumably) defines this target is automatically loaded from the corresponding directory. In fact, this mechanism goes a step further and, if the `buildfile` does not exist, then it assumes one with the following contents was implied:

```
./: */
```

That is, it simply builds all the subdirectories. This is especially handy when organizing related tests into directory hierarchies.

As mentioned above, this automatic inclusion is only triggered if the target we depend on is `dir{}` and we still have to explicitly include the necessary `buildfiles` for other targets. One common example is a project consisting of a library and an executable that links it, each residing in a separate directory next to each other (as noted earlier, this is not recommended for projects that you plan to package). For example:

```
hello/
|-- build/
|   `-- ...
|-- hello/
|   |-- main.cxx
|   `-- buildfile
|-- libhello/
|   |-- hello.hxx
|   |-- hello.cxx
|   `-- buildfile
`-- buildfile
```

In this case the executable `buildfile` would look along these lines:

```
include ../libhello/ # Include lib{hello}.

exe{hello}: {hxx cxx}{**} lib{hello}
```

Note also that `buildfile` inclusion should only be used for accessing targets within the same project. For cross-project references we use Target Importation.

1.4 Operations

Modern build systems have to perform operations other than just building: cleaning the build output, running tests, installing/uninstalling the build results, preparing source distributions, and so on. And, if the build system has integrated configuration support, configuring the project would naturally belong to this list as well.

If you are familiar with `make`, you should recognize the parallel with the common `clean` `test`, `install`, and `dist`, "operation" pseudo-targets.

In `build2` we have the concept of a *build system operation* performed on a target. The two pre-defined operations are `update` and `clean` with other operations provided by build system modules.

Operations to be performed and targets to perform them on are specified on the command line. As discussed earlier, `update` is the default operation and `./` in the current directory is the default target if no operation and/or target is specified explicitly. And, similar to targets, we can specify multiple operations (not necessarily on the same target) in a single build system invocation. The list of operations to perform and targets to perform them on is called a *build specification* or *buildspec* for short (see **`b(1)`** for details). Here are a few examples:

```
$ cd hello        # Change to project root.

$ b               # Update current directory.
$ b ./            # Same as above.
$ b update        # Same as above.
$ b update: ./    # Same as above.

$ b clean update  # Rebuild.

$ b clean:  hello/             # Clean specific target.
$ b update: hello/exe{hello}   # Update specific target

$ b update: libhello/ tests/   # Update two targets.
```

Let's revisit `build/bootstrap.build` from our `hello` project:

```
project = hello

using version
using config
using test
using install
using dist
```

Other than `version`, all the modules we load define new operations. Let's examine each of them starting with `config`.

1.4.1 Configuration

As mentioned briefly earlier, the `config` module provides support for persisting configurations by having us *configure* our projects. At first it may feel natural to call `configure` another operation. There is, however, a conceptual problem: we don't really configure a target. And, perhaps after some meditation, it should become clear that what we are really doing is configuring operations on targets. For example, configuring updating a C++ project might involve detecting and saving information about the C++ compiler while configuring installing it may require specifying the installation directory.

In other words, `configure` is an operation on operation on targets – a meta-operation. And so in `build2` we have the concept of a *build system meta-operation*. If not specified explicitly (as part of the buildspec), the default is `perform`, which is to simply perform the operation.

Back to `config`, this module provides two meta-operations: `configure` which saves the configuration of a project into the `build/config.build` file as well as `disfigure` which removes it.

While the common meaning of the word *disfigure* is somewhat different to what we make it mean in this context, we still prefer it over the commonly suggested alternative (*deconfigure*) for the symmetry of their Latin *con-* ("together") and *dis-* ("apart") prefixes.

Let's say for the in source build of our `hello` project we want to use `Clang` and enable debug information. Without persistence we would have to repeat this configuration on every build system invocation:

```
$ cd hello/  # Change to project root.

$ b config.cxx=clang++ config.cxx.coptions=-g
```

Instead, we can configure our project with this information once and from then on invoke the build system without any arguments:

```
$ b configure config.cxx=clang++ config.cxx.coptions=-g

$ tree ./
./
|-- build/
|   |-- ...
|   `-- config.build
`-- ...

$ b
$ b clean
$ b
...
```

Let's take a look at `config.build`:

```
$ cat build/config.build

config.cxx = clang++
config.cxx.poptions = [null]
config.cxx.coptions = -g
config.cxx.loptions = [null]
config.cxx.libs = [null]
...
```

As you can see, it's just a buildfile with a bunch of variable assignments. In particular, this means you can tweak your build configuration by modifying this file with your favorite editor. Or, alternatively, you can adjust the configuration by reconfiguring the project:

```
$ b configure config.cxx=g++

$ cat build/config.build

config.cxx = g++
config.cxx.poptions = [null]
config.cxx.coptions = -g
config.cxx.loptions = [null]
config.cxx.libs = [null]
...
```

Any variable value specified on the command line overrides those specified in the `build-files`. As a result, `config.cxx` was updated while the value of `config.cxx.coptions` was preserved.

Command line variable overrides are also handy to adjust the configuration for a single build system invocation. For example, let's say we want to quickly check that our project builds with optimization but without permanently changing the configuration:

```
$ b config.cxx.coptions=-O3  # Rebuild with -O3.
$ b                          # Rebuild with -g.
```

We can also configure out of source builds of our projects. In this case, besides `config.build`, `configure` also saves the location of the source directory so that we don't have to repeat that either. Remember, this is how we used to build our `hello` out of source:

```
$ b hello/@hello-gcc/   config.cxx=g++
$ b hello/@hello-clang/ config.cxx=clang++
```

And now we can do:

```
$ b configure: hello/@hello-gcc/   config.cxx=g++
$ b configure: hello/@hello-clang/ config.cxx=clang++

$ hello-clang/
hello-clang/
'-- build/
    |-- bootstrap/
    |   '-- src-root.build
    '-- config.build

$ b hello-gcc/
$ b hello-clang/
$ b hello-gcc/ hello-clang/
```

One major benefit of an in source build is the ability to run executables as well as examine build and test output (test results, generated source code, documentation, etc) without leaving the source directory. Unfortunately, we cannot have multiple in source builds and as was discussed earlier, mixing in and out of source builds is not recommended.

To overcome this limitation `build2` has a notion of *forwarded configurations*. As the name suggests, we can configure a project's source directory to forward to one of its out of source builds. Once done, whenever we run the build system from the source directory, it will automatically build in the corresponded forwarded output directory. Additionally, it will *backlink* (using symlinks or another suitable mechanism) certain "interesting" targets (`exe{}`, `doc{}`) to the source directory for easy access. As an example, let's configure our `hello/` source directory to forward to the `hello-gcc/` build:

```
$ b configure: hello/@hello-gcc/,forward

$ cd hello/  # Change to project root.
$ b
c++ hello/cxx{hello}@../hello-gcc/hello/
ld ../hello-gcc/hello/exe{hello}
ln ../hello-gcc/hello/exe{hello} -> hello/
```

Notice the last line in the above listing: it indicates that `exe{hello}` from the out directory was backlinked in our project's source subdirectory:

```
$ tree ./
./
|-- build/
|   |-- bootstrap/
|   |   `-- out-root.build
|   `-- ...
|-- hello/
|   |-- ...
|   `-- hello -> ../../hello-gcc/hello/hello*
`-- ...

$ ./hello/hello
Hello World!
```

By default only `exe{}` and `doc{}` targets are backlinked. This, however, can be customized with the `backlink` target-specific variable.

1.4.2 Testing

The next module we load in `bootstrap.build` is `test` which defines the `test` operation. As the name suggests, this module provides support for running tests.

There are two types of tests that we can run with the `test` module: simple and scripted.

A simple test is just an executable target with the `test` target-specific variable set to `true`. For example:

```
exe{hello}: test = true
```

A simple test is executed once and in its most basic form (typical for unit testing) doesn't take any inputs nor produce any output, indicating success via the zero exit status. If we test our `hello` project with the above addition to the `buildfile`, then we will see the following output:

```
$ b test
test hello/exe{hello}
Hello, World!
```

While the test passes (since it exited with zero status), we probably don't want to see that `Hello, World!` every time we run it (this can, however, be quite useful when running examples). More importantly, we don't really test its functionality and if tomorrow our `hello` starts swearing rather than greeting, the test will still pass.

Besides checking its exit status we can also supply some basic information to a simple test (more common for integration testing). Specifically, we can pass command line options (`test.options`) and arguments (`test.arguments`) as well as input (`test.stdin`, used to supply test's `stdin`) and output (`test.stdout`, used to compare to test's `stdout`).

Let's see how we can use this to fix our `hello` test by making sure our program prints the expected greeting. First, we need to add a file that will contain the expected output, let's call it `test.out`:

```
$ ls -1 hello/
hello.cxx
test.out
buildfile

$ cat hello/test.out
Hello, World!
```

Next, we arrange for it to be compared to our test's `stdout`. Here is the new `hello/buildfile`:

```
exe{hello}: {hxx cxx}{**}
exe{hello}: file{test.out}: test.stdout = true
```

The last line looks new. What we have here is a *prerequisite-specific variable* assignment. By setting `test.stdout` for the `file{test.out}` prerequisite of target `exe{hello}` we mark it as expected `stdout` output of *this* target (theoretically, we could have marked it as `test.input` for another target). Notice also that we no longer need the `test` target-specific variable; it's unnecessary if one of the other `test.*` variables is specified.

Now, if we run our test, we won't see any output:

```
$ b test
test hello/exe{hello}
```

And if we try to change the greeting in `hello.cxx` but not in `test.out`, our test will fail printing the `diff(1)` comparison of the expected and actual output:

```
$ b test
c++ hello/cxx{hello}
ld hello/exe{hello}
test hello/exe{hello}
--- test.out
+++ -
@@ -1 +1 @@
-Hello, World!
+Hi, World!
error: test hello/exe{hello} failed
```

Notice another interesting thing: we have modified `hello.cxx` to change the greeting and our test executable was automatically rebuilt before testing. This happened because the `test` operation performs `update` as its *pre-operation* on all the targets to be tested.

Let's make our `hello` program more flexible by accepting the name to greet on the command line:

```
#include <iostream>

int main (int argc, char* argv[])
{
  if (argc < 2)
  {
    std::cerr << "error: missing name" << std::endl;
    return 1;
  }

  std::cout << "Hello, " << argv[1] << '!' << std::endl;
}
```

We can exercise its successful execution path with a simple test fairly easily:

```
exe{hello}: test.arguments = 'World'
exe{hello}: file{test.out}: test.stdout = true
```

What if we also wanted to test its error handling? Since simple tests are single-run, this won't be easy. Even if we could overcome this, having expected output for each test in a separate file will quickly become untidy. And this is where script-based tests come in. Testscript is `build2`'s portable language for running tests. It vaguely resembles Bash and is optimized for concise test implementation and fast, parallel execution.

Just to give you an idea (see Testscript Introduction for a proper introduction), here is what testing our `hello` program with Testscript would look like:

```
$ ls -1 hello/
hello.cxx
testscript
buildfile

$ cat hello/buildfile

exe{hello}: {hxx cxx}{**} testscript
```

And this is the contents of `hello/testscript`:

```
: basics
:
$* 'World' >'Hello, World!'

: missing-name
:
$* 2>>EOE != 0
error: missing name
EOE
```

A couple of key points: The `test.out` file is gone with all the test inputs and expected outputs incorporated into `testscript`. To test an executable with Testscript, all we have to do is list the corresponding `testscript` file as its prerequisite (and which, being a fixed name, doesn't need an explicit target type, similar to `manifest`).

To see Testscript in action, let's say we've made our program more forgiving by falling back to a default name if one wasn't specified:

```
#include <iostream>

int main (int argc, char* argv[])
{
  const char* n (argc > 1 ? argv[1] : "World");
  std::cout << "Hello, " << n << '!' << std::endl;
}
```

If we forget to adjust the `missing-name` test, then this is what we could expect to see when running the tests:

```
b test
c++ hello/cxx{hello}
ld hello/exe{hello}
test hello/testscript{testscript} hello/exe{hello}
hello/testscript:7:1: error: hello/hello exit code 0 == 0
  info: stdout: hello/test-hello/missing-name/stdout
```

Testscript-based integration testing is the default setup for executable (`-t exe`) projects created by **`bdep-new(1)`**. Here is the recap of the overall layout:

```
hello/
|-- build/
|   `-- ...
|-- hello/
|   |-- hello.cxx
|   |-- testscript
|   `-- buildfile
|-- buildfile
`-- manifest
```

For libraries (`-t lib`), however, the integration testing setup is a bit different. Here are the relevant parts of the layout:

```
libhello/
|-- build/
|   `-- ...
|-- libhello/
|   |-- hello.hxx
|   |-- hello.cxx
|   |-- export.hxx
|   |-- version.hxx.in
|   `-- buildfile
|-- tests/
|   |-- build/
|   |   |-- bootstrap.build
|   |   `-- root.build
|   |-- basics/
|   |   |-- driver.cxx
|   |   `-- buildfile
|   `-- buildfile
|-- buildfile
`-- manifest
```

Specifically, there is no `testscript` in `libhello/`, the project's source directory. Instead, we have the `tests/` subdirectory which itself looks like a project: it contains the `build/` subdirectory with all the familiar files, etc. In fact, `tests` is a *subproject* of our `libhello` project.

While we will be examining `tests` in greater detail later, in a nutshell, the reason it is a subproject is to be able to test an installed version of our library. By default, when `tests` is built as part of its parent project (called *amalgamation*), the locally built `libhello` library will be automatically imported. However, we can also configure a build of `tests` out of its amalgamation, in which case we can import an installed version of `libhello`. We will learn how to do all that as well as the underlying concepts (*subproject*/*amalgamation*, *import*, etc) in the coming sections.

Inside `tests/` we have the `basics/` subdirectory which contains a simple test for our library's API. By default it doesn't use Testscript but if you want to, you can. You can also rename `basics/` to something more meaningful and add more tests next to it. For example, if we were creating an XML parsing and serialization library, then our `tests/` could have the following layout:

```
tests/
|-- build/
|   `-- ...
|-- parser/
|   `-- ...
|-- serializer/
|   `-- ...
`-- buildfile
```

Nothing prevents us from having the `tests/` subdirectory for executable projects. And it can be just a subdirectory or a subproject, the same as for libraries. Making it a subproject makes sense if your program has complex installation, for example, if its execution requires configuration and/or data files that need to be found, etc. For simple programs, however, testing the executable before installing it is usually sufficient.

For a general discussion of functional/integration and unit testing refer to the Tests section in the toolchain introduction. For details on the unit test support implementation see Implementing Unit Testing.

1.4.3 Installation

The `install` module defines the `install` and `uninstall` operations. As the name suggests, this module provides support for project installation.

Installation in `build2` is modeled after UNIX-like operation systems though the installation directory layout is highly customizable. While `build2` projects can import `build2` libraries directly, installation is often a way to "export" them in a form usable by other build systems.

The root installation directory is specified with the `config.install.root` configuration variable. Let's install our `hello` program into `/tmp/install`:

```
$ cd hello/  # Change to project root.

$ b install config.install.root=/tmp/install/
```

And see what we've got (executables are marked with `*`):

```
$ tree /tmp/install/

/tmp/install/
|-- bin/
|   `-- *hello
`-- share/
    `-- doc/
        `-- hello/
            `-- manifest
```

Similar to the `test` operation, `install` performs `update` as a pre-operation for targets that it installs.

We can also configure our project with the desired `config.install.*` values so that we don't have to repeat them on every install/uninstall. For example:

```
$ b configure config.install.root=/tmp/install/
$ b install
$ b uninstall
```

Now let's try the same for `libhello` (symbolic link targets are shown with `->` and actual static/shared library names may differ on your operating system):

```
$ rm -r /tmp/install

$ cd libhello/  # Change to project root.

$ b install config.install.root=/tmp/install/

$ tree /tmp/install/

/tmp/install/
|-- include/
|   `-- libhello/
|       |-- hello.hxx
|       |-- export.hxx
|       `-- version.hxx
|-- lib/
|   |-- pkgconfig/
|   |   |-- libhello.shared.pc
|   |   `-- libhello.static.pc
|   |-- libhello.a
|   |-- libhello.so -> libhello-0.1.so
|   `-- libhello-0.1.so
`-- share/
    `-- doc/
        `-- libhello/
            `-- manifest
```

As you can see, the library headers go into the customary `include/` subdirectory while static and shared libraries (and their `pkg-config(1)` files) – into `lib/`. Using this installation we should be able to import this library from other build systems or even use it in a manual build:

```
$ g++ -I/tmp/install/include -L/tmp/install/lib greet.cxx -lhello
```

If we want to install into a system-wide location like `/usr` or `/usr/local`, then we most likely will need to specify the `sudo(1)` program:

```
$ b config.install.root=/usr/local/ config.install.sudo=sudo
```

In build2 only actual install/uninstall commands are executed with sudo(1). And while on the topic of sensible implementations, uninstall can be generally trusted to work reliably.

The default installability of a target as well as where it is installed is determined by its target type. For example, exe{} is by default installed into bin/, doc{} – into share/doc/<project>/, and file{} is not installed.

We can, however, override these defaults with the install target-specific variable. Its value should be either special false indicating that the target should not be installed or the directory to install the target to. As an example, here is what the root buildfile from our libhello project looks like:

```
./: {*/ -build/} manifest

tests/: install = false
```

The first line we have already seen and the purpose of the second line should now be clear: it makes sure we don't try to install anything in the tests/ subdirectory.

If the value of the install variable is not false, then it is normally a relative path with the first path component being one of these names:

```
name        default                          override
----        -------                          --------
root                                         config.install.root

data_root   root/                            config.install.data_root
exec_root   root/                            config.install.exec_root

bin         exec_root/bin/                   config.install.bin
sbin        exec_root/sbin/                  config.install.sbin
lib         exec_root/lib/                   config.install.lib
libexec     exec_root/libexec/<project>/     config.install.libexec
pkgconfig   lib/pkgconfig/                   config.install.pkgconfig

data        data_root/share/<project>/       config.install.data
include     data_root/include/               config.install.include

doc         data_root/share/doc/<project>/   config.install.doc
man         data_root/share/man/             config.install.man
man<N>      man/man<N>/                      config.install.man<N>
```

Let's see what's going on here: The default install directory tree is derived from the config.install.root value but the location of each node in this tree can be overridden by the user that installs our project using the corresponding config.install.* variables. In our buildfiles, in turn, we use the node names instead of actual directories. As an example, here is a buildfile fragment from the source directory of our libhello project:

```
hxx{*}:
{
  install         = include/libhello/
  install.subdirs = true
}
```

Here we set the installation location for headers to be the `libhello/` subdirectory of the `include` installation location. Assuming `config.install.root` is `/usr/`, the `install` module will perform the following steps to resolve this relative path to the actual, absolute installation directory:

```
include/libhello/
data_root/include/libhello/
root/include/libhello/
/usr/include/libhello/
```

In the above `buildfile` fragment we also see the use of the `install.subdirs` variable. Setting it to `true` instructs the `install` module to recreate subdirectories starting from this point in the project's directory hierarchy. For example, if our `libhello/` source directory had the `details/` subdirectory with the `utility.hxx` header, then this header would have been installed as `.../include/libhello/details/utility.hxx`.

1.4.4 Distribution

The last module that we load in our `bootstrap.build` is `dist` which provides support for the preparation of distributions and defines the `dist` meta-operation. Similar to `configure`, `dist` is a meta-operation rather than an operation because, conceptually, we are preparing a distribution for performing operations (like `update`, `test`) on targets rather than targets themselves.

The preparation of a correct distribution requires that all the necessary project files (sources, documentation, etc) be listed as prerequisites in the project's `buildfiles`.

You may wonder why not just use the export support offered by many version control systems? The main reason is that in most real-world projects version control repositories contain a lot more than what needs to be distributed. In fact, it is not uncommon to host multiple build system projects/packages in a single repository. As a result, with this approach we seem to inevitably end up maintaining an exclusion list, which feels backwards: why specify all the things we don't want in a new list instead of making sure the already existing list of things that we do want is complete? Also, once we have the complete list, it can be put to good use by other tools, such as editors, IDEs, etc.

The preparation of a distribution also requires an out of source build. This allows the `dist` module to distinguish between source and output targets. By default, targets found in src are included into the distribution while those in out are excluded. However, we can customize this with the `dist` target-specific variable.

As an example, let's prepare a distribution of our `hello` project using the out of source build configured in `hello-out/`. We use `config.dist.root` to specify the directory to write the distribution to:

```
$ b dist: hello-out/ config.dist.root=/tmp/dist

$ ls -1 /tmp/dist
hello-0.1.0/

$ tree /tmp/dist/hello-0.1.0/
/tmp/dist/hello-0.1.0/
|-- build/
|   |-- bootstrap.build
|   `-- root.build
|-- hello/
|   |-- hello.cxx
|   |-- testscript
|   `-- buildfile
|-- buildfile
`-- manifest
```

As we can see, the distribution directory includes the project version (comes from the `version` variable which, in our case, is extracted from `manifest` by the `version` module). Inside the distribution directory we have our project's source files (but, for example, without any `.gitignore` files that we may have had in `hello/`).

We can also ask the `dist` module to package the distribution directory into one or more archives and generate their checksum files for us. For example:

```
$ b dist: hello-out/ \
  config.dist.root=/tmp/dist \
  config.dist.archives="tar.gz zip" \
  config.dist.checksums=sha256

$ ls -1 /tmp/dist
hello-0.1.0/
hello-0.1.0.tar.gz
hello-0.1.0.tar.gz.sha256
hello-0.1.0.zip
hello-0.1.0.zip.sha256
```

We can also configure our project with the desired `config.dist.*` values so we don't have to repeat them every time. For example:

```
$ b configure: hello-out/ config.dist.root=/tmp/dist ...
$ b dist
```

Let's now take a look at an example of customizing what gets distributed. Most of the time you will be using this mechanism to include certain targets from out. Here is a fragment from the `libhello` source directory `buildfile`:

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

Our library provides the `version.hxx` header that the users can include to obtain its version. This header is generated by the `version` module from the `version.hxx.in` template. In essence, the `version` module takes the version value from our `manifest`, splits it into various components (major, minor, patch, etc) and then preprocesses the `in{}` file substituting these values (see `version` Module for details). The end result is an automatically maintained version header.

One problem with auto-generated headers is that if one does not yet exist, then the compiler may still find it somewhere else. For example, we may have an older version of a library installed somewhere where the compiler searches for headers by default (for example, `/usr/local/include/`). To overcome this problem it is a good idea to ship pre-generated headers in our distributions. But since they are output targets, we have to explicitly request this with `dist=true`.

1.5 Target Importation

Recall that if we need to depend on a target defined in another `buildfile` within our project, then we simply include said `buildfile` and reference the target. For example, if our `hello` included both an executable and a library in separate subdirectories next to each other:

```
hello/
|-- build/
|   `-- ...
|-- hello/
|   |-- ...
|   `-- buildfile
`-- libhello/
    |-- ...
    `-- buildfile
```

Then our executable `buildfile` could look like this:

```
include ../libhello/ # Include lib{hello}.

exe{hello}: {hxx cxx}{**} lib{hello}
```

What if instead `libhello` were a separate project? The inclusion approach would no longer work for two reasons: we don't know the path to `libhello` (after all, it's an independent project and can reside anywhere) and we can't assume the path to the `lib{hello}` target within `libhello` (the project directory layout can change).

To depend on a target from a separate project we use *importation* instead of inclusion. This mechanism is also used to depend on targets that are not part of any project, for example, installed libraries.

The importing project's side is pretty simple. This is what the above `buildfile` will look like if `libhello` were a separate project:

```
import libs = libhello%lib{hello}

exe{hello}: {hxx cxx}{**} $libs
```

The `import` directive is a kind of variable assignment that resolves a *project-qualified* relative target (`libhello%lib{hello}`) to an unqualified absolute target and stores it in the variable (`libs` in our case). We can then expand the variable (`$libs`), normally in the dependency declaration, to get the imported target.

If we needed to import several libraries, then we simply repeat the `import` directive, usually accumulating the result in the same variable, for example:

```
import libs  = libformat%lib{format}
import libs += libprint%lib{print}
import libs += libhello%lib{hello}

exe{hello}: {hxx cxx}{**} $libs
```

Let's now try to build our `hello` project that uses imported `libhello`:

```
$ b hello/
error: unable to import target libhello%lib{hello}
  info: use config.import.libhello command line variable to specify
        its project out_root
```

While that didn't work out well, it does make sense: the build system cannot know the location of `libhello` or which of its builds we want to use. Though it does helpfully suggest that we use `config.import.libhello` to specify its out directory (`out_root`). Let's point it to `libhello` source directory to use its in source build (`out_root == src_root`):

```
$ b hello/ config.import.libhello=libhello/
c++ libhello/libhello/cxx{hello}
ld libhello/libhello/libs{hello}
c++ hello/hello/cxx{hello}
ld hello/hello/exe{hello}
```

And it works. Naturally, the importation mechanism works the same for out of source builds and we can persist the `config.import.*` variables in the project's configuration. As an example, let's configure Clang builds of the two projects out of source:

```
$ b configure: libhello/@libhello-clang/ config.cxx=clang++
$ b configure: hello/@hello-clang/ config.cxx=clang++ \
  config.import.libhello=libhello-clang/

$ b hello-clang/
c++ libhello/libhello/cxx{hello}@libhello-clang/libhello/
ld libhello-clang/libhello/libs{hello}
c++ hello/hello/cxx{hello}@hello-clang/hello/
ld hello-clang/hello/exe{hello}
```

If the corresponding `config.import.*` variable is not specified, `import` searches for a project in a couple of other places. First, it looks in the list of subprojects starting from the importing project itself and then continuing with its outer amalgamations and their subprojects (see Subprojects and Amalgamations for details on this subject).

We've actually seen an example of this search step in action: the `tests` subproject in `libhello`. The test imports `libhello` which is automatically found as an amalgamation containing this subproject.

If the project being imported cannot be located using any of these methods, then `import` falls back to the rule-specific search. That is, a rule that matches the target may provide support for importing certain target types based on rule-specific knowledge. Support for importing installed libraries by the C++ link rule is a good example of this. Internally, the `cxx` module extracts the compiler's library search paths (that is, paths that would be used to resolve `-lfoo`) and then the link rule uses them to search for installed libraries. This allows us to use the same `import` directive regardless of whether we import a library from a separate build, from a subproject, or from an installation directory.

Importation of an installed library will work even if it is not a `build2` project. Besides finding the library itself, the link rule will also try to locate its `pkg-config(1)` file and, if present, extract additional compile/link flags from it. The link rule also automatically produces `pkg-config(1)` files for libraries that it installs.

Let's now examine the exporting side of the importation mechanism. While a project doesn't need to do anything special to be found by `import`, it does need to handle locating the exported target (or targets; there could be several) within the project as well as loading their `build-files`. And this is the job of an *export stub*, the `build/export.build` file that you might have noticed in the `libhello` project:

```
libhello
|-- build/
|   `-- export.build
`-- ...
```

Let's take a look inside:

```
$out_root/
{
  include libhello/
}

export $out_root/libhello/$import.target
```

An export stub is a special kind of `buildfile` that bridges from the importing project into exporting. It is loaded in a special temporary scope out of any project, in a "no man's land" so to speak. The only variables set on the temporary scope are `src_root` and `out_root` of the project being imported as well as `import.target` containing the name of the target being imported (without project qualification; that is, `lib{hello}` in our example).

Typically, an export stub will open the scope of the exporting project, load the `buildfile` that defines the target being exported and finally "return" the absolute target name to the importing project using the `export` directive. And this is exactly what the export stub in our `libhello` does.

We now have all the pieces of the importation puzzle in place and you can probably see how they all fit together. To summarize, when the build system sees the `import` directive, it looks for a project with the specified name. If found, it creates a temporary scope, sets the `src/out_root` variables to point to the project and `import.target` – to the target name specified in the `import` directive. And then it load the project's export stub in this scope. Inside the export stub we switch to the project's root scope, load its `buildfile` and then use the `export` directive to return the exported target. Once the export stub is processed, the build system obtains the exported target and assigns it to the variable specified in the `import` directive.

Our export stub is quite "loose" in that it allows importing any target defined in the project's source subdirectory `buildfile`. While we found it to be a good balance between strictness and flexibility, if you would like to "tighten" your export stubs, you can. For example:

```
if ($import.target == lib{hello})
  export $out_root/libhello/$import.target
```

If no `export` directive is executed in an export stub then the build system assumes that the target is not exported by the project and issues appropriate diagnostics.

1.6 Library Exportation and Versioning

By now we have examined and explained every line of every `buildfile` in our `hello` executable project. There are, however, still a few lines to be covered in the source subdirectory `buildfile` in `libhello`. Here it is in its entirety:

```
int_libs = # Interface dependencies.
imp_libs = # Implementation dependencies.

lib{hello}: {hxx ixx txx cxx}{** -version} hxx{version} \
  $imp_libs $int_libs

# Include the generated version header into the distribution (so that
# we don't pick up an installed one) and don't remove it when cleaning
# in src (so that clean results in a state identical to distributed).
#
hxx{version}: in{version} $src_root/manifest
{
  dist  = true
  clean = ($src_root != $out_root)
}

# Build options.
#
cxx.poptions =+ "-I$out_root" "-I$src_root"

obja{*}: cxx.poptions += -DLIBHELLO_STATIC_BUILD
objs{*}: cxx.poptions += -DLIBHELLO_SHARED_BUILD

# Export options.
#
lib{hello}:
{
  cxx.export.poptions = "-I$out_root" "-I$src_root"
  cxx.export.libs = $int_libs
}

liba{hello}: cxx.export.poptions += -DLIBHELLO_STATIC
libs{hello}: cxx.export.poptions += -DLIBHELLO_SHARED

# For pre-releases use the complete version to make sure they cannot
# be used in place of another pre-release or the final version. See
# the version module for details on the version.* variable values.
#
if $version.pre_release
  lib{hello}: bin.lib.version = @"-$version.project_id"
else
  lib{hello}: bin.lib.version = @"-$version.major.$version.minor"

# Install into the libhello/ subdirectory of, say, /usr/include/
# recreating subdirectories.
#
{hxx ixx txx}{*}:
{
  install         = include/libhello/
  install.subdirs = true
}
```

Let's start with all those `cxx.export.*` variables. It turns out that merely exporting a library target is not enough for the importers of the library to be able to use it. They also need to know where to find its headers, which other libraries to link, etc. This information is carried in a set of target-specific `cxx.export.*` variables that parallel the `cxx.*` set and that together with the

library's prerequisites constitute the *library meta-information protocol*. Every time a source file that depends on a library is compiled or a binary is linked, this information is automatically extracted by the compile and link rules from the library dependency chain, recursively. And when the library is installed, this information is carried over to its `pkg-config(1)` file.

Similar to the `c.*` and `cc.*` sets discussed earlier, there are also `c.export.*` and `cc.export.*` sets.

Here are the parts relevant to the library meta-information protocol in the above `buildfile`:

```
int_libs = # Interface dependencies.
imp_libs = # Implementation dependencies.

lib{hello}: ... $imp_libs $int_libs

lib{hello}:
{
  cxx.export.poptions = "-I$out_root" "-I$src_root"
  cxx.export.libs = $int_libs
}

liba{hello}: cxx.export.poptions += -DLIBHELLO_STATIC
libs{hello}: cxx.export.poptions += -DLIBHELLO_SHARED
```

As a first step we classify all our library dependencies into *interface dependencies* and *implementation dependencies*. A library is an interface dependency if it is referenced from our interface, for example, by including (importing) one of its headers (modules) from one of our (public) headers (modules) or if one of its functions is called from our inline or template functions. Otherwise, it is an implementation dependency.

To illustrate the distinction between interface and implementation dependencies, let's say we've reimplemented our `libhello` to use `libformat` to format the greeting and `libprint` to print it. Here is our new header (`hello.hxx`):

```
#include <libformat/format.hxx>

namespace hello
{
  void
  say_hello_formatted (std::ostream&, const std::string& hello);

  inline void
  say_hello (std::ostream& o, const std::string& name)
  {
    say_hello_formatted (o, format::format_hello ("Hello", name));
  }
}
```

And this is the new source file (`hello.cxx`):

```
#include <libprint/print.hxx>

namespace hello
{
  void
  say_hello_formatted (ostream& o, const string& h)
  {
    print::print_hello (o, h);
  }
}
```

In this case, `libformat` is our interface dependency since we both include its header in our interface and call it from one of our inline functions. In contrast, `libprint` is only included and used in the source file and so we can safely treat it as an implementation dependency. The corresponding `import` directives in our `buildfile` will therefore look like this:

```
import int_libs = libformat%lib{format}
import imp_libs = libprint%lib{print}
```

The preprocessor options (`poptions`) of an interface dependency must be made available to our library's users. The library itself should also be explicitly linked whenever our library is linked. All this is achieved by listing the interface dependencies in the `cxx.export.libs` variable:

```
lib{hello}:
{
  cxx.export.libs = $int_libs
}
```

More precisely, the interface dependency should be explicitly linked if a user of our library may end up with a direct call to the dependency in one of their object files. Not linking such a library is called *underlinking* while linking a library unnecessarily (which can happen because we've included its header but are not actually calling any of its non-inline/template functions) is called *overlinking*. Underlinking is an error on some platforms while overlinking may slow down the process startup and/or waste its memory.

Note also that this only applies to shared libraries. In case of static libraries, both interface and implementation dependencies are always linked, recursively.

The remaining lines in the library meta-information fragment are:

```
lib{hello}:
{
  cxx.export.poptions = "-I$out_root" "-I$src_root"
}

liba{hello}: cxx.export.poptions += -DLIBHELLO_STATIC
libs{hello}: cxx.export.poptions += -DLIBHELLO_SHARED
```

The first line makes sure the users of our library can locate its headers by exporting the relevant `-I` options. The last two lines define the library type macros that are relied upon by the `export.hxx` header to properly setup symbol exporting.

The `liba{}` and `libs{}` target types correspond to the static and shared libraries, respectively. And `lib{}` is actually a target group that can contain one, the other, or both as its members.

Specifically, when we build a `lib{}` target, which members will be built is determined by the `config.bin.lib` variable with the `static`, `shared`, and `both` (default) possible values. So to only build a shared library we can run:

```
$ b config.bin.lib=shared
```

When it comes to linking `lib{}` prerequisites, which member is picked is controlled by the `config.bin.{exe,liba,libs}.lib` variables for the executable, static library, and shared library targets, respectively. Each contains a list of `shared` and `static` values that determine the linking preferences. For example, to build both shared and static libraries but to link executable to static libraries we can run:

```
$ b config.bin.lib=both config.bin.exe.lib=static
```

See `bin` Module for more information.

Note also that we don't need to change anything in the above `buildfile` if our library is header-only. In `build2` this is handled dynamically and automatically based on the absence of source file prerequisites. In fact, the same library can be header-only on some platforms or in some configuration and "source-full" in others.

In `build2` a header-only library (or a module interface-only library) is not a different kind of library compared to static/shared libraries but is rather a binary-less, or *binless* for short, static or shared library. So, theoretically, it is possible to have a library that has a binless static and a binary-full (*binfull*) shared variants. Note also that binless libraries can depend on binfull libraries and are fully supported where the `pkg-config(1)` functionality is concerned.

If you are creating a new library with **`bdep-new(1)`** and are certain that it will always be binless and in all configurations, then you can produce a simplified `buildfile` by specifying the `binless` option, for example:

```
$ bdep new -t lib -l c++,binless libheader-only
```

Let's now turn to the second subject of this section and the last unexplained bit in our `buildfile`: shared library versioning. Here is the relevant fragment:

```
if $version.pre_release
  lib{hello}: bin.lib.version = @"-$version.project_id"
else
  lib{hello}: bin.lib.version = @"-$version.major.$version.minor"
```

Shared library versioning is a murky, platform-specific area. Instead of trying to come up with a unified versioning scheme that few are likely to comprehend (similar to `autoconf`), `build2` provides a platform-independent versioning scheme as well as the ability to specify platform-specific versions in a native format.

The library version is specified with the `bin.lib.version` target-specific variable. Its value should be a sequence of `@`-pairs with the left hand side (key) being the platform name and the right hand side (value) being the version. An empty key signifies the platform-independent version (see `bin` Module for the exact semantics). For example:

```
lib{hello}: bin.lib.version = @-1.2 linux@3
```

While the interface for platform-specific versions is defined, their support is not yet implemented by the C/C++ link and install rules.

A platform-independent version is embedded as a suffix into the library name (and into its `soname` on relevant platforms) while platform-specific versions are handled according to the platform. Continuing with the above example, these would be the resulting shared library names on select platforms:

```
libhello.so.3       # Linux
libhello-1.2.dll    # Windows
libhello-1.2.dylib  # Mac OS
```

With this background we can now explain what's going in our `buildfile`:

```
if $version.pre_release
  lib{hello}: bin.lib.version = @"-$version.project_id"
else
  lib{hello}: bin.lib.version = @"-$version.major.$version.minor"
```

Here we only use platform-independent library versioning. For releases we embed both major and minor version components assuming that patch releases are binary compatible. For pre-releases, however, we use the complete version to make sure it cannot be used in place of another pre-release or the final version.

The `version.project_id` variable contains the project's (as opposed to package's), shortest "version id". See the `version` Module for details.

1.7 Subprojects and Amalgamations

In `build2` projects can contain other projects, recursively. In this arrangement the outer project is called an *amalgamation* and the inner – *subprojects*. In contrast to importation where we merely reference a project somewhere else, amalgamation is physical containment. It can be *strong* where the src directory of a subproject is within the amalgamating project or *weak* where only the out directory is contained.

There are several distinct use cases for amalgamations. We've already discussed the `tests/` subproject in `libhello`. To recap, traditionally, it is made a subproject rather than a subdirectory to support building it as a standalone project in order to test library installations.

As discussed in Target Importation, subprojects and amalgamations (as well as their subprojects, recursively) are automatically considered when resolving imports. As a result, amalgamation can be used to *bundle* dependencies to produce an external dependency-free distribution. For example, if our `hello` project imports `libhello`, then we could copy the `libhello` project into `hello`, for example:

```
$ tree hello/
hello/
|-- build/
|   `-- ...
|-- hello/
|   |-- hello.cxx
|   `-- ...
|-- libhello/
|   |-- build/
|   |   `-- ...
|   |-- libhello/
|   |   |-- hello.hxx
|   |   |-- hello.cxx
|   |   `-- ...
|   |-- tests/
|   |   `-- ...
|   `-- buildfile
`-- buildfile

$ b hello/
c++ hello/libhello/libhello/cxx{hello}
ld hello/libhello/libhello/libs{hello}
c++ hello/hello/cxx{hello}
ld hello/hello/exe{hello}
```

Note, however, that while project bundling can be useful in certain cases, it does not scale as a general dependency management solution. For that, independent packaging and proper dependency management are the appropriate mechanisms.

By default `build2` looks for subprojects only in the root directory of a project. That is, every root subdirectory is examined to see if it itself is a project root. If you need to place a subproject somewhere else in your project's directory hierarchy, then you will need to specify its location (and of all other subprojects) explicitly with the `subprojects` variable in `bootstrap.build`. For example, if above we placed `libhello` into the `extras/` subdirectory of `hello`, then our `bootstrap.build` would need to start like this:

```
project = hello
subprojects = extras/libhello/
...
```

Note also that while importation of specific targets from subprojects is always performed, whether they are loaded and built as part of the overall project build is controlled using the standard subdirectories inclusion and dependency mechanisms. Continuing with the above example, if we adjust the root `buildfile` in `hello` to exclude the `extras/` subdirectory from the build:

```
./: {*/ -build/ -extras/}
```

Then while we can still import `libhello` from any `buildfile` in our project, the entire `libhello` (for example, its tests) will never be built as part of the `hello` build.

Similar to subprojects we can also explicitly specify the project's amalgamation with the `amalgamation` variable (again, in `bootstrap.build`). This is rarely necessary except if you want to prevent the project from being amalgamated, in which case you should set it to the empty value.

If either of these variables is not explicitly set, then they will contain the automatically discovered values.

Besides affecting importation, another central property of amalgamation is configuration inheritance. As an example, let's configure the above bundled `hello` project in its src directory:

```
$ b configure: hello/ config.cxx=clang++ config.cxx.coptions=-g

$ b tree
hello/
|-- build/
|   |-- config.build
|   `-- ...
|-- libhello/
|   |-- build/
|   |   |-- config.build
|   |   `-- ...
|   `-- ...
`-- ...
```

As you can see, we now have the `config.build` files in both project's `build/` subdirectories. If we examine the amalgamation's `config.build`, we will see the familiar picture:

```
$ cat hello/build/config.build

config.cxx = clang++
config.cxx.poptions = [null]
config.cxx.coptions = -g
config.cxx.loptions = [null]
config.cxx.libs = [null]

...
```

The subproject's `config.build`, however, is pretty much empty:

```
$ cat hello/libhello/build/config.build

# Base configuration inherited from ../
```

As the comment suggests, the base configuration is inherited from the outer project. We can, however, override some values if we need to. For example (note that we are re-configuring the `libhello` subproject):

```
$ b configure: hello/libhello/ config.cxx.coptions=-O2

$ cat hello/libhello/build/config.build

# Base configuration inherited from ../

config.cxx.coptions = -O2
```

This configuration inheritance combined with import resolution is behind the most common use of amalgamations in `build2` – shared build configurations. Let's say we are developing multiple projects, for example, `hello` and `libhello` that it imports:

```
$ ls -1
hello/
libhello/
```

And we want to build them with several compilers, let's say GCC and Clang. As we have already seen in Configuration, we can configure several out of source builds for each compiler, for example:

```
$ b configure: libhello/@libhello-gcc/   config.cxx=g++
$ b configure: libhello/@libhello-clang/ config.cxx=clang++

$ b configure: hello/@hello-gcc/   \
               config.cxx=g++      \
               config.import.libhello=libhello-gcc/
$ b configure: hello/@hello-clang/ \
               config.cxx=clang++  \
               config.import.libhello=libhello-clang/
```

```
$ ls -l
hello/
hello-gcc/
hello-clang/
libhello/
libhello-gcc/
libhello-clang/
```

Needless to say, this is a lot of repetitive typing. Another problem is future changes to the configurations. If, for example, we need to adjust compile options in the GCC configuration, then we will have to (remember to) do it in both places.

You can probably sense where this is going: why not create a shared build configuration (that is, an amalgamation) for GCC and Clang where we build both of our projects (as its subprojects)? This is how we can do that:

```
$ b create: build-gcc/,cc   config.cxx=g++
$ b create: build-clang/,cc config.cxx=clang++

$ b configure: libhello/@build-gcc/libhello/
$ b configure: hello/@build-gcc/hello/

$ b configure: libhello/@build-clang/libhello/
$ b configure: hello/@build-clang/hello/

$ ls -l
hello/
libhello/
build-gcc/
build-clang/
```

Let's explain what's going on here. First, we create two build configurations using the `create` meta-operation. These are real `build2` projects just tailored for housing other projects as subprojects. In `create`, after the directory name, we specify the list of modules to load in the project's `root.build`. In our case we specify `cc` which is a common module for C-based languages (see **`b(1)`** for details on `create` and its parameters).

When creating build configurations it is a good idea to get into the habit of using the `cc` module instead of `c` or `cxx` since with more complex dependency chains we may not know whether every project we build only uses C or C++. In fact, it is not uncommon for a C++ project to have C implementation details and even the other way around (yes, really, there are C libraries with C++ implementations).

Once the configurations are ready we simply configure our `libhello` and `hello` as subprojects in each of them. Note that now we neither need to specify `config.cxx`, because it will be inherited from the amalgamation, nor `config.import.*`, because the import will be automatically resolved to a subproject.

Now, to build a specific project in a particular configuration we simply build the corresponding subdirectory. We can also build the entire build configuration if we want to. For example:

```
$ b build-gcc/hello/

$ b build-clang/
```

In case you've already looked into **`bpkg(1)`** and/or **`bdep(1)`**, their build configurations are actually these same amalgamations (created underneath with the `create` meta-operation) and their packages are just subprojects. And with this understanding you are free to interact with them directly using the build system interface.

1.8 Buildfile Language

By now we should have a good overall sense of what writing `buildfiles` feels like. In this section we will examine the language in slightly more detail and with more precision.

Buildfile is primarily a declarative language with support for variables, pure functions, repetition (`for`-loop), and conditional inclusion/exclusion (`if-else`).

Buildfile is a line-oriented language. That is, every construct ends at the end of the line unless escaped with line continuation (trailing `\`). For example:

```
exe{hello}: {hxx cxx}{**} \
  $libs
```

Some lines may start a *block* if followed by `{` on the next line. Such a block ends with a closing `}` on a separate line. Some types of blocks can nest. For example:

```
if ($cxx.target.class == 'windows')
{
  if ($cxx.target.system == 'ming32')
  {
    ...
  }
}
```

A comment starts with `#` and everything from this character and until the end of the line is ignored. A multi-line comment starts with `#\` on a separate line and ends with the same character sequence, again on a separate line. For example:

```
# Single line comment.

info 'Hello, World!' # Trailing comment.

#\
Multi-
line
comment.
#\
```

The three primary Buildfile constructs are dependency declaration, directive, and variable assignment. We've already used all three but let's see another example:

```
include ../libhello/                  # Directive.

exe{hello}: {hxx cxx}{**} lib{hello}  # Dependency declaration.

cxx.poptions += -DNDEBUG              # Variable assignment (append).
```

There is also the scope opening (we've seen one in `export.build`) as well as target-specific and prerequisite-specific variable assignment blocks. The latter two are used to assign several entity-specific variables at once. For example:

```
details/                          # scope
{
  hxx{*}: install = false
}

hxx{version}:                     # target-specific
{
  dist  = true
  clean = ($src_root != $out_root)
}

exe{test}: file{test.roundtrip}:  # prerequisite-specific
{
  test.stdin  = true
  test.stdout = true
}
```

Variable assignment blocks can be combined with dependency declarations, for example:

```
h{config}: in{config}
{
  in.symbol = '@'
  in.substitution = lax

  SYSTEM_NAME = $c.target.system
  SYSTEM_PROCESSOR = $c.target.cpu
}
```

In case of a dependency chain, the block applies to the set of prerequisites (note: *not targets*) before last `:`. For example:

```
./: exe{test}: libue{test}: cxx{test}
{
  bin.whole = false  # Applies to the libue{test} prerequisite.
}
```

All prerequisite-specific variables must be assigned at once as part of the dependency declaration since repeating the same dependency again duplicates the prerequisite rather than references the already existing one.

There is also the target type/pattern-specific variable assignment block, for example:

```
exe{*.test}:
{
  test = true
  install = false
}
```

See Variables for more information.

Each `buildfile` is processed linearly with directives executed and variables expanded as they are encountered. However, certain variables, for example, `cxx.poptions` are also expanded by rules during execution in which case they will "see" the final value set in the `buildfile`.

Unlike GNU `make(1)`, which has deferred (`=`) and immediate (`:=`) variable assignments, all assignments in `build2` are immediate. For example:

```
x = x
y = $x
x = X
info $y # Prints 'x', not 'X'.
```

1.8.1 Expansion and Quoting

While we've discussed variable expansion and lookup earlier, to recap, to get the variable's value we use `$` followed by its name. The variable name is first looked up in the current scope (that is, the scope in which the expansion was encountered) and, if not found, in the outer scopes, recursively.

There are two other kinds of expansions: function calls and evaluation contexts, or *eval contexts* for short. Let's start with the latter since function calls are built on top of eval contexts.

An eval context is essentially a fragment of a line with additional interpretations of certain characters to support value comparison, logical operators, and a few other constructs. Eval contexts begin with `(`, end with `)`, and can nest. Here are a few examples:

```
info ($src_root != $out_root)                 # Prints true or false.
info ($src_root == $out_root ? 'in' : 'out')  # Prints in or out.

macos = ($cxx.target.class == 'macos')  # Assigns true or false.
linux = ($cxx.target.class == 'linux')  # Assigns true or false.

if ($macos || $linux)  # Also eval context.
  ...
```

Below is the eval context grammar that shows supported operators and their precedence.

```
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 a single value only (no commas).

A function call starts with `$` followed by its name and an eval context listing its arguments. Note that there is no space between the name and `(`. For example:

```
x =
y = Y

info $empty($x)  # true
info $empty($y)  # false

if $regex.match($y, '[A-Z]')
  ...

p = $src_base/foo.txt

info $path.leaf($src_base)              # foo.txt
info $path.directory($src_base)         # $src_base
info $path.base($path.leaf($src_base))  # foo
```

Note that functions in `build2` are *pure* in a sense that they do not alter the build state in any way.

Functions in `build2` are currently defined either by the build system core or build system modules and are implemented in C++. In the future it will be possible to define custom functions in `buildfiles` (also in C++).

Variable and function names follow the C identifier rules. We can also group variables into namespaces and functions into families by combining multiple identifiers with `.`. These rules are used to determine the end of the variable name in expansions. If, however, a name is recognized as being longer than desired, then we can use the eval context to explicitly specify its boundaries. For example:

```
base = foo
name = $(base).txt
```

What is the structure of a variable value? Consider this assignment:

```
x = foo bar
```

The value of `x` could be a string, a list of two strings, or something else entirely. In `build2` the fundamental, untyped value is a *list of names*. A value can be typed to something else later but it always starts as a list of names. So in the above example we have a list of two names, `foo` and `bar`, the same as in this example (notice the extra spaces):

```
x = foo    bar
```

The motivation behind going with a list of names instead of a string or a list of strings is that at its core we are dealing with targets and their prerequisites and it would be natural to make the representation of their names (that is, the way we refer to them) the default. Consider the following two examples; it would be natural for them to mean the same thing:

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

prereqs = {hxx cxx}{**}
exe{hello}: $prereqs
```

Note also that the name semantics was carefully tuned to be *reversible* to its syntactic representation for common non-name values, such as paths, command line options, etc., that are usually found in `buildfiles`.

Names are split into a list at whitespace boundaries with certain other characters treated as syntax rather than as part of the value. Here are a few example:

```
x = $y          # expansion
x = (a == b)    # eval context
x = {foo bar}   # name generation
x = [null]      # attributes
x = name@value  # pairs
x = # comments
```

The complete set of syntax characters is `$(){}[]@#` plus space and tab. Additionally, `*?` will be treated as wildcards in a name pattern. If instead we need these characters to appear literally as part of the value, then we either have to *escape* or *quote* them.

To escape a special character, we prefix it with a backslash (`\`; to specify a literal backslash double it). For example:

```
x = \$
y = C:\\Program\ Files
```

Similar to UNIX shells, `build2` supports single (`''`) and double (`""`) quoting with roughly the same semantics. Specifically, expansions (variable, function call, and eval context) and escaping are performed inside double-quoted strings but not in single-quoted. Note also that quoted strings can span multiple lines with newlines treated literally (unless escaped in double-quoted strings). For example:

```
x = "(a != b)"  # true
y = '(a != b)'  # (a != b)

x = "C:\\Program Files"
y = 'C:\Program Files'

t = 'line one
line two
line three'
```

Since quote characters are now also part of the syntax, if you need to specify them literally in the value, then they will either have to be escaped or quoted. For example:

```
cxx.poptions += -DOUTPUT='"debug"'
cxx.poptions += -DTARGET=\"$cxx.target\"
```

An expansion can be one of two kinds: *spliced* or *concatenated*. In a spliced expansion the variable, function, or eval context is separated from other text with whitespaces. In this case, as the name suggests, the resulting list of names is spliced into the value. For example:

```
x = 'foo fox'
y = bar $x baz  # Three names: 'bar' 'foo fox' 'baz'.
```

This is an important difference compared to the semantics of UNIX shells where the result of expansion is re-parsed. In particular, this is the reason why you won't see quoted expansions in `buildfiles` as often as in (well-written) shell scripts.

In a concatenated expansion the variable, function, or eval context are combined with unseparated text before and/or after the expansion. For example:

```
x = 'foo fox'
y = bar$(x)foz  # Single name: 'barfoo foxbaz'
```

A concatenated expansion is typed unless it is quoted. In a typed concatenated expansion the parts are combined in a type-aware manner while in an untyped – literally, as string. To illustrate the difference, consider this `buildfile` fragment:

```
info $src_root/foo.txt
info "$src_root/foo.txt"
```

If we run it on a UNIX-like operating system, we will see two identical lines, along these lines:

```
/tmp/test/foo.txt
/tmp/test/foo.txt
```

However, if we run it on Windows (which uses backslashes as directory separators), we will see the output along these lines:

```
C:\test\foo.txt
C:\test/foo.txt
```

The typed concatenation resulted in a native directory separator because `dir_path` (the `src_root` type) did the right thing.

Not every typed concatenation is well defined and in certain situations we may need to force untyped concatenation with quoting. Options specifying header search paths (`-I`) are a typical case, for example:

```
cxx.poptions =+ "-I$out_root" "-I$src_root"
```

If we were to remove the quotes, we would see the following error:

```
buildfile:6:20: error: no typed concatenation of <untyped> to dir_path
  info: use quoting to force untyped concatenation
```

1.8.2 Conditions (`if-else`)

The `if` directive can be used to conditionally exclude `buildfile` fragments from being processed. The conditional fragment can be a single (separate) line or a block with the initial `if` optionally followed by a number of `elif` directives and a final `else`, which together form the `if-else` chain. An `if-else` block can contain nested `if-else` chains. For example:

```
if ($cxx.target.class == 'linux')
  info 'linux'
elif ($cxx.target.class == 'windows')
{
  if ($cxx.target.system == 'mingw32')
    info 'windows-mingw'
  elif ($cxx.target.system == 'win32-msvc')
    info 'windows-msvc'
  else
    info 'windows-other'
}
else
  info 'other'
```

The `if` and `elif` directive names must be followed by something that expands to a single, literal `true` or `false`. This can be a variable expansion, a function call, an eval context, or a literal value. For example:

```
if $version.pre_release
  ...

if $regex.match($x, '[A-Z]')
  ...

if ($cxx.target.class == 'linux')
  ...

if false
{
  # disabled fragment
}

x = X
if $x  # Error, must expand to true or false.
  ...
```

There are also `if!` and `elif!` directives which negate the condition that follows (note that there is no space before `!`). For example:

```
if! $version.pre_release
  ...
elif! $regex.match($x, '[A-Z]')
  ...
```

Note also that there is no notion of variable locality in `if-else` blocks and any value set inside is visible outside. For example:

```
if true
{
  x = X
}

info $x  # Prints 'X'.
```

The `if-else` chains should not be used for conditional dependency declarations since this would violate the expectation that all of the project's source files are listed as prerequisites, irrespective of the configuration. Instead, use the special `include` prerequisite-specific variable to conditionally include prerequisites into the build. For example:

```
# Incorrect.
#
if ($cxx.target.class == 'linux')
  exe{hello}: cxx{hello-linux}
elif ($cxx.target.class == 'windows')
  exe{hello}: cxx{hello-win32}

# Correct.
#
exe{hello}: cxx{hello-linux}: include = ($cxx.target.class == 'linux')
exe{hello}: cxx{hello-win32}: include = ($cxx.target.class == 'windows')
```

1.8.3 Repetitions (`for`)

The `for` directive can be used to repeat the same `buildfile` fragment multiple times, once for each element of a list. The fragment to repeat can be a single (separate) line or a block, which together form the `for` loop. A `for` block can contain nested `for` loops. For example:

```
for n: foo bar baz
{
  exe{$n}: cxx{$n}
}
```

The `for` directive name must be followed by the variable name (called *loop variable*) that on each iteration will be assigned the corresponding element, `:`, and something that expands to a potentially empty list of values. This can be a variable expansion, a function call, an eval context, or a literal list as in the above fragment. Here is a somewhat more realistic example that splits a space-separated environment variable value into names and then generates a dependency declaration for each of them:

```
for n: $regex.split($getenv(NAMES), ' +', '')
{
  exe{$n}: cxx{$n}
}
```

Note also that there is no notion of variable locality in `for` blocks and any value set inside is visible outside. At the end of the iteration the loop variable contains the value of the last element, if any. For example:

```
for x: x X
{
  y = Y
}

info $x  # Prints 'X'.
info $y  # Prints 'Y'.
```

1.9 Implementing Unit Testing

As an example of how many of these features fit together to implement more advanced functionality, let's examine a `buildfile` that provides support for unit testing. This support is added by the **`bdep-new(1)`** command if we specify the `unit-tests` option when creating executable (`-t exe,unit-tests`) or library (`-t lib,unit-tests`) projects. Here is the source subdirectory `buildfile` of an executable created with this option:

```
./: exe{hello}: libue{hello}: {hxx cxx}{** -**.test...}

# Unit tests.
#
exe{*.test}
{
  test = true
  install = false
}

for t: cxx{**.test...}
{
  d = $directory($t)
  n = $name($t)...

  ./: $d/exe{$n}: $t $d/hxx{+$n} $d/testscript{+$n}
  $d/exe{$n}: libue{hello}: bin.whole = false
}

cxx.poptions =+ "-I$out_root" "-I$src_root"
```

The basic idea behind this unit testing arrangement is to keep unit tests next to the source code files that they test and automatically recognize and build them into test executables without having to manually list each in the `buildfile`. Specifically, if we have `hello.hxx` and `hello.cxx`, then to add a unit test for this module all we have to do is drop the `hello.test.cxx` source file next to them and it will be automatically picked up, built into an executable, and run during the `test` operation.

As an example, let's say we've renamed `hello.cxx` to `main.cxx` and factored the printing code into the `hello.hxx/hello.cxx` module that we would like to unit-test. Here is the new layout:

```
hello/
|-- build
|   `-- ...
|-- hello
|   |-- hello.cxx
|   |-- hello.hxx
|   |-- hello.test.cxx
|   |-- main.cxx
|   `-- buildfile
`-- ...
```

Let's examine how this support is implemented in our `buildifle`, line by line. Because now we link `hello.cxx` object code into multiple executables (unit tests and the `hello` program itself), we have to place it into a *utility library*. This is what the first line does (it has to explicitly list `exe{hello}` as a prerequisite of the default targets since we now have multiple targets that should be built by default):

```
./: exe{hello}: libue{hello}: {hxx cxx}{** -**.test...}
```

A utility library (**`u`** in `lib`**`u`**`e`) is a static library that is built for a specific type of a *primary target* (**`e`** in `libu`**`e`** for executable). If we were building a utility library for a library then we would have used the `libul{}` target type instead. In fact, this would be the only difference in the above unit testing implementation if it were for a library project instead of an executable:

```
./: lib{hello}: libul{hello}: {hxx cxx}{** -**.test...}

...

# Unit tests.
#
...

for t: cxx{**.test...}
{
  ...

  $d/exe{$n}: libul{hello}: bin.whole = false
}
```

Going back to the first three lines of the executable `buildfile`, notice that we had to exclude source files in the `*.test.cxx` form from the utility library. This makes sense since we don't want unit testing code (each with its own `main()`) to end up in the utility library.

The exclusion pattern, `-**.test...`, looks a bit cryptic. What we have here is a second-level extension (`.test`) which we use to classify our source files as belonging to unit tests. Because it is a second-level extension, we have to indicate this fact to the pattern matching machinery with the trailing triple dot (meaning "there are more extensions coming"). If we didn't do that, `.test` would have been treated as a first-level extension explicitly specified for our source files.

If you need to specify a name that does not have an extension, then end it with a single dot. For example, for a header `utility` you would write `hxx{utility.}`. If you need to specify a name with an actual trailing dot, then escape it with a double dot, for example, `hxx{utility..}`.

The next couple of lines set target type/pattern-specific variables to treat all unit test executables as tests that should not be installed:

```
exe{*.test}:
{
  test = true
  install = false
}
```

You may be wondering why we had to escape the second-level `.test` extension in the name pattern above but not here. The answer is that these are different kinds of patterns in different contexts. In particular, patterns in the target type/pattern-specific variables are only matched against target names without regard for extensions. See Name Patterns for details.

Then we have the `for`-loop that declares an executable target for each unit test source file. The list of these files is generated with a name pattern that is the inverse of what we've used for the utility library:

```
for t: cxx{**.test...}
{
  d = $directory($t)
  n = $name($t)...

  ./: $d/exe{$n}: $t $d/hxx{+$n} $d/testscript{+$n}
  $d/exe{$n}: libue{hello}: bin.whole = false
}
```

In the loop body we first split the test source file into the directory (remember, we can have sources, including tests, in subdirectories) and name (which contains the `.test` second-level extension and which we immediately escape with `...`). And then we use these components to declare a dependency for the corresponding unit test executable. There is nothing here that we haven't already seen except for using variable expansions instead of literal names.

By default utility libraries are linked in the "whole archive" mode where every object file from the static library ends up in the resulting executable or library. This behavior is what we want when linking the primary target but can normally be relaxed for unit tests to speed up linking. This is what the last line in the loop does using the `bin.whole` prerequisite-specific variable.

You can easily customize this and other aspects on a test-by-test basis by excluding the specific test(s) from the loop and then providing a custom implementation. For example:

```
for t: cxx{**.test... -special.test...}
{
  ...
}

./: exe{special.test...}: cxx{special.test...} libue{hello}
```

Note also that if you plan to link any of your unit tests in the whole archive mode, then you will also need to exclude the source file containing the primary executable's `main()` from the utility library. For example:

```
./: exe{hello}: cxx{main} libue{hello}
libue{hello}: {hxx cxx}{** -main -**.test...}
```

1.10 Diagnostics and Debugging

Sooner or later we will run into a situation where our `buildfiles` don't do what we expect them to. In this section we examine a number of techniques and mechanisms that can help us understand the cause of a misbehaving build.

To perform a build the build system goes through several phases. During the *load* phase the `buildfiles` are loaded and processed. The result of this phase is the in-memory *build state* that contains the scopes, targets, variables, etc., defined by the `buildfiles`. Next, is the *match* phase during which rules are matched to the targets that need to be built, recursively. Finally, during the *execute* phase the matched rules are executed to perform the build.

The load phase is always serial and stops at the first error. In contrast, by default, both match and execute are parallel and continue in the presence of errors (similar to the "keep going" `make` mode). While beneficial in normal circumstances, during debugging this can lead to both interleaved output that is hard to correlate as well as extra noise from cascading errors. As a result, for debugging, it is usually helpful to run serially and stop at the first error, which can be achieved with the `--serial-stop|-s` option.

The match phase can be temporarily switched to either (serial) load or (parallel) execute. The former is used, for example, to load additional `buildfiles` during the `dir{}` prerequisite to target resolution, as described in Output Directories and Scopes. While the latter is used to update generated source code (such as headers) that is required to complete the match.

Debugging issues in each phase requires different techniques. Let's start with the load phase. As mentioned in Build Language, `buildfiles` are processed linearly with directives executed and variables expanded as they are encountered. As we have already seen, to print a variable value we can use the `info` directive. For example:

```
x = X
info $x
```

This will print something along these lines:

```
buildfile:2:1: info: X
```

Or, if we want to clearly see where the value begins and ends (useful when investigating whitespace-related issues):

```
x = " X "
info "'$x'"
```

Which prints:

```
buildfile:2:1: info: ' X '
```

Besides the `info` directive, there are also `text`, which doesn't print the `info:` prefix, `warn`, which prints a warning, as well as `fail` which prints an error and causes the build system to exit with an error. Here is an example of using each:

```
text 'note: we are about to get an error'
warn 'the error is imminent'
fail 'this is the end'
info 'we will never get here'
```

This will produce the following output:

```
buildfile:1:1: note: we are about to get an error
buildfile:2:1: warning: the error is imminent
buildfile:3:1: error: this is the end
```

If you find yourself writing code like this:

```
if ($cxx.target.class == 'windows')
  fail 'Windows is not supported'
```

Then the `assert` directive is a more concise way to express the same:

```
assert ($cxx.target.class != 'windows') 'Windows is not supported'
```

The assert condition must be something that evaluates to `true` or `false`, similar to the `if` directive (see Conditions (`if-else`) for details). The description after the condition is optional and, similar to `if`, there is also the `assert!` variant, which fails if the condition is `true`.

All the diagnostics directives write to `stderr`. If instead we need to write something to `stdout`, for example, to send some information back to our caller, then we can use the `print` directive. For example, this will print the C++ compiler id and its target:

```
print "$cxx.id $cxx.target"
```

To query the value of a target-specific variable we use the qualified name syntax (the `eval-qual` production) of eval context, for example:

```
obj{main}: cxx.poptions += -DMAIN
info $(obj{main}: cxx.poptions)
```

There is no direct way to query the value of a prerequisite-specific variable since a prerequisite has no identity. Instead, we can use the `dump` directive discussed next to print the entire dependency declaration, including prerequisite-specific variables for each prerequisite.

While printing variables values is the most common mechanism for diagnosing `buildfile` issues, sometimes it is also helpful to examine targets and scopes. For that we use the `dump` directive.

Without any arguments, `dump` prints (to `stderr`) the contents of the scope it was encountered in and at that point of processing the `buildfile`. Its output includes variables, targets and their prerequsites, as well as nested scopes, recursively. As an example, let's print the source directory scope of our `hello` executable project. Here is its `buildfile` with the `dump` directive at the end:

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

cxx.poptions =+ "-I$out_root" "-I$src_root"

dump
```

This will produce the output along these lines:

```
buildfile:5:1: dump:
  /tmp/hello/hello/
  {
    [strings] cxx.poptions = -I/tmp/hello -I/tmp/hello
    [dir_path] out_base = /tmp/hello/hello/
    [dir_path] src_base = /tmp/hello/hello/

    build{buildfile.}:

    exe{hello.?}: cxx{hello.?}
  }
```

The question marks (`?`) in the dependency declaration mean that the file extensions haven't been assigned yet, which happens during the match phase.

Instead of printing the entire scope, we can also print individual targets by specifying one or more target names in `dump`. To make things more interesting, let's convert our `hello` project to use a utility library, similar to the unit testing setup (Implementing Unit Testing). We will also link to the `pthread` library to see an example of a target-specific variable being dumped:

```
exe{hello}: libue{hello}: bin.whole = false
exe{hello}: cxx.libs += -lpthread
libue{hello}: {hxx cxx}{**}

dump exe{hello}
```

The output will look along these lines:

```
buildfile:5:1: dump:
  /tmp/hello/hello/exe{hello.?}:
  {
    [strings] cxx.libs = -lpthread
  }
  /tmp/hello/hello/exe{hello.?}: /tmp/hello/hello/:libue{hello.?}:
  {
    [bool] bin.whole = false
  }
```

The output of `dump` might look familiar: in Output Directories and Scopes we've used the `--dump` option to print the entire build state, which looks pretty similar. In fact, the `dump` directive uses the same mechanism but allows us to print individual scopes and targets.

There is, however, an important difference to keep in mind: `dump` prints the state of a target or scope at the point in the `buildfile` load phase where it was executed. In contrast, the `--dump` option can be used to print the state after the load phase (`--dump load`) and/or after the match phase (`--dump match`). In particular, the after match printout reflects the changes to the build state made by the matching rules, which may include entering of additional dependencies, setting of additional variables, resolution of prerequsites to targets, assignment of file extensions, etc. As a result, while the `dump` directive should be sufficient in most cases, sometimes you may need to use the `--dump` option to examine the build state just before rule execution.

Let's now move from state to behavior. As we already know, to see the underlying commands executed by the build system we use the `-v` options (which is equivalent to `--verbose 2`). Note, however, that these are *logical* rather than actual commands. You can still run them and they should produce the desired result, but in reality the build system may have achieved the same result in a different way. To see the actual commands we use the `-V` option instead (equivalent to `--verbose 3`). Let's see the difference in an example. Here is what building our `hello` executable with `-v` might look like:

```
$ b -s -v
g++ -o hello.o -c hello.cxx
g++ -o hello hello.o
```

And here is the same build with `-V`:

```
$ b -s -V
g++ -MD -E -fdirectives-only -MF hello.o.t -o hello.o.ii hello.cxx
g++ -E -fpreprocessed -fdirectives-only hello.o.ii
g++ -o hello.o -c -fdirectives-only hello.o.ii
g++ -o hello hello.o
```

From the second listing we can see that in reality `build2` first partially preprocessed `hello.cxx` while extracting its header dependency information. It then preprocessed it fully, which is used to extract module dependency information, calculate the checksum for ignorable change detection, etc. When it comes to producing `hello.o`, the build system compiled the partially preprocessed output rather than the original `hello.cxx`. The end result, however, is

the same as in the first listing.

Verbosity level `3` (`-V`) also triggers printing of the build system module configuration information. Here is what we would see for the `cxx` module:

```
cxx hello@/tmp/hello/
  cxx         g++@/usr/bin/g++
  id          gcc
  version     7.2.0 (Ubuntu 7.2.0-1ubuntu1~16.04)
  major       7
  minor       2
  patch       0
  build       (Ubuntu 7.2.0-1ubuntu1~16.04)
  signature   gcc version 7.2.0 (Ubuntu 7.2.0-1ubuntu1~16.04)
  checksum    09b3b59d337eb9a760dd028fa0df585b307e6a49c2bfa00b3[...]
  target      x86_64-linux-gnu
  runtime     libgcc
  stdlib      libstdc++
  c stdlib    glibc
...
```

Verbosity levels higher than `3` enable build system tracing. In particular, level `4` is useful for understanding why a rule doesn't match a target or if it does, why it determined the target to be out of date. For example, assuming we have an up-to-date build of our `hello`, let's change a compile option:

```
$ b -s --verbose 4
info: /tmp/hello/dir{hello/} is up to date

$ b -s --verbose 4 config.cxx.poptions+=-DNDEBUG
trace: cxx::compile_rule::apply: options mismatch forcing update
of /tmp/hello/hello/obje{hello.o}
...
```

Higher verbosity levels result in more and more tracing statements being printed. These include `buildfile` loading and parsing, prerequisite to target resolution, as well as build system module and rule-specific logic.

Another useful diagnostics option is `--mtime-check`. When specified, the build system performs a number of file modification time sanity checks that can be helpful in diagnosing spurious rebuilds.

If neither state dumps nor behavior analysis are sufficient to understand the problem, there is always an option of running the build system under a C++ debugger in order to better understand what's going on. This can be particularly productive for debugging complex rules.

Finally, to help with diagnosing the build system performance issues, there is the `--stat` option. It causes `build2` to print various execution statistics which can be useful for pin-pointing the bottlenecks. There are also a number of options for tuning the build system's perfor-

mance, such as, the number of jobs to perform in parallel, the stack size, queue depths, etc. See the **`b(1)`** man pages for details.

2 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, but excluding directories that contain the `.buildignore` file. 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 as well as to allow patterns without wildcards, the first pattern can also start with the plus sign. For example:

```
exe{hello}: cxx{f* -foo}            # Exclude foo if exists.
exe{hello}: cxx{f* +bar}            # Include bar if exists.
exe{hello}: cxx{f* -fo?}            # Exclude foo and fox if exist.
exe{hello}: cxx{f* +b* -foo -bar}   # Exclude foo and bar if exist.
exe{hello}: cxx{+f* +b* -foo -bar}  # Same as above.
exe{hello}: cxx{+foo}               # Pattern without wildcards.
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 first 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.

3 Variables

Note: this section is a work in progress.

Note that while expansions in the target and prerequisite-specific assignments happen in the corresponding target and prerequisite contexts, respectively, for type/pattern-specific assignments they happen in the scope context. Plus, a type/pattern-specific prepend/append is applied at the time of expansion for the actual target. For example:

```
x = s

file{foo}:              # target
{
  x += t    # s t
  y = $x y  # s t y
}

file{foo}: file{bar}    # prerequisite
{
  x += p    # x t p
  y = $x y  # x t p y
}

file{b*}:               # type/pattern
{
  x += w   # <append w>
  y = $x w # <assign s w>
}
```

```
x = S

info $(file{bar}: x) # S w
info $(file{bar}: y) # s w
```

4 `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/}`).

5 `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
+2-1.2.3-a.1+2
```

The *major*, *minor*, and *patch* should be numeric values between `0` and `99999` 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 time-zone 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:

```
AAAAABBBBBCCCCCDDDE

AAAAA - major
BBBBB - minor
CCCCC - 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 *AAAAABBBBBCCCCC* 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:

```
             AAAAABBBBBCCCCCDDDE
0.1.0        0000000001000000000
0.1.2        0000000001000020000
1.2.3        0000100002000030000
2.2.0-a.1    0000200001999990010
3.0.0-b.2    0000299999999995020
2.2.0-a.1.z  0000200001999990011
```

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     # 0000100002000025041
[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 distributions.

The version header rule is based on the `in` module rule and can be used to preprocesses a template file with version information. While it is usually used to generate C/C++ version headers (thus the name), it can really generate any kind of files.

The rule matches a `file`-based target that has the 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 accomplish 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}
{
  dist = true
}
```

The header rule is a line-based preprocessor that substitutes fragments enclosed between (and including) a pair of dollar signs (`$`) with `$$` being the escape sequence (see the `in` module for details). 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     1000020000030000ULL
#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     1000020000030000ULL
#define LIBHELLO_VERSION_STR "1.2.3"

#include <libprint/version.hxx>

#ifdef LIBPRINT_VERSION
#  if !(LIBPRINT_VERSION >= 2000030000040000ULL)
#    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)
```

6 `bin` Module

Coming soon.

7 `cxx` 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.

7.1 C++ Modules Support

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

7.1.1 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.

7.1.2 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).

7.1.3 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.

7.1.4 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.

7.1.5 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.

7.1.6 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 modularized 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.

8 `in` Module

The `in` build system module provides support for `.in` (input) file preprocessing. Specifically, the `.in` file can contain a number of *substitutions* – build system variable names enclosed with the substitution symbol (`$` by default) – which are replaced with the corresponding variable values to produce the output file. For example:

```
# build/root.build

using in
```

```
// config.hxx.in

#define TARGET "$cxx.target$"

# buildfile

hxx{config}: in{config}
```

The `in` module defines the `in{}` target type and implements the `in` build system rule.

While we can specify the `.in` extension explicitly, it is not necessary because the `in{}` target type implements *target-dependent search* by taking into account the target it is a prerequisite of. In other words, the following dependency declarations produce the same result:

```
hxx{config}:     in{config}
hxx{config.hxx}: in{config}
hxx{config.hxx}: in{config.hxx.in}
```

By default the `in` rule uses `$` as the substitution symbol. This can be changed using the `in.symbol` variable. For example:

```
// data.cxx.in

const char data[] = "@data@";

# buildfile

cxx{data}: in{data}
{
  in.symbol = '@'
  data = 'Hello, World!'
}
```

Note that the substitution symbol must be a single character.

The default substitution mode is strict. In this mode every substitution symbol is expected to start a substitution with unresolved (to a variable value) names treated as errors. The double substitution symbol (for example, `$$`) serves as an escape sequence.

The substitution mode can be relaxed using the `in.substitution` variable. Its valid values are `strict` (default) and `lax`. In the lax mode a pair of substitution symbols is only treated as a substitution if what's between them looks like a build system variable name (that is, it doesn't contain spaces, etc). Everything else, including unterminated substitution symbols, is copied as is. Note also that in this mode the double substitution symbol is not treated as an escape sequence.

The lax mode is mostly useful when trying to reuse existing `.in` files from other build systems, such as `autoconf`. Note, however, that the lax mode is still stricter than the `autoconf`'s semantics which also leaves unresolved substitutions as is. For example:

```
# buildfile

h{config}: in{config} # config.h.in
{
  in.symbol = '@'
  in.substitution = lax

  CMAKE_SYSTEM_NAME = $c.target.system
  CMAKE_SYSTEM_PROCESSOR = $c.target.cpu
}
```

The `in` rule tracks changes to the input file as well as the substituted variable values and automatically regenerates the output file if any were detected. Substituted variable values are looked up starting from the target-specific variables. Typed variable values are converted to string using the corresponding `builtin.string()` function overload before substitution.

A number of other build system modules, for example, `version` and `bash`, are based on the `in` module and provide extended functionality. The `in` preprocessing rule matches any `file{}`-based target that has the corresponding `in{}` prerequisite provided none of the extended rules match.

9 `bash` Module

The `bash` build system module provides modularization support for `bash` scripts. It is based on the `in` build system module and extends its preprocessing rule with support for *import substitutions* in the `@import <module>@` form. During preprocessing, such imports are replaced with suitable `source` builtin calls. For example:

```
# build/root.build

using bash

# hello/say-hello.bash

function say_hello ()
{
  echo "Hello, $1!"
}

#!/usr/bin/env bash

# hello/hello.in

@import hello/say-hello@

say_hello 'World'
```

```
# hello/buildfile

exe{hello}: in{hello} bash{say-hello}
```

By default the `bash` preprocessing rule uses the lax substitution mode and `@` as the substitution symbol but this can be overridden using the standard `in` module mechanisms.

In the above example, `say-hello.bash` is a *module*. By convention, `bash` modules have the `.bash` extension and we use the `bash{}` target type (defined by the `bash` build system module) to refer to them in buildfiles.

The `say-hello.bash` module is *imported* by the `hello` script with the `@import hello/say-hello@` substitution. The *import path* (`hello/say-hello` in our case) is a relative path to the module file within the project. Its first component (`hello` in our case) must be the project base name and the `.bash` module extension can be omitted. The constraint placed on the first component of the import path is required to implement importation of installed modules, as discussed below.

During preprocessing, the import substitution will be replaced with a `source` builtin call and the import path resolved to one of the `bash{}` prerequisites from the script's dependency declaration. The actual module path used in `source` depends on whether the script is preprocessed for installation. If it's not (development build), then the absolute path to the module file is used. Otherwise, a path relative to the sourcing script's directory is derived. This allows installed scripts and their modules to be moved around.

The derivation of the sourcing script's directory works even if the script is executed via a symbolic link from another directory. Implementing this, however, requires `readlink(1)` with support for the `-f` option. One notable platform that does not provide such `readlink(1)` by default is Mac OS. The script, however, can provide a suitable implementation as a function. See the `bash` module tests for a sample implementation of such a function.

By default, `bash` modules are installed into a subdirectory of the `bin/` installation directory named as the project base name. For instance, in the above example, the script will be installed as `bin/hello` and the module as `bin/hello/say-hello.bash` with the script sourcing the module relative to the `bin/` directory. Note that currently it is assumed the script and all its modules are installed into the same `bin/` directory.

Naturally, modules can import other modules and modules can be packaged into *module libraries* and imported using the standard build system import mechanism. For example, we could factor the `say-hello.bash` module into a separate `libhello` project:

```
# build/export.build

$out_root/
{
  include libhello/
}

export $src_root/libhello/$import.target

# libhello/say-hello.bash

function hello_say_hello ()
{
  echo "Hello, $1!"
}
```

And then import it in a module of our `hello` project:

```
# hello/hello-world.bash.in

@import libhello/say-hello@

function hello_world ()
{
  hello_say_hello 'World'
}

#!/usr/bin/env bash

# hello/hello.in

@import hello/hello-world@

hello_world

# hello/buildfile

import mods = libhello%bash{say-hello}

exe{hello}:        in{hello}       bash{hello-world}
bash{hello-world}: in{hello-world} $mods
```

The `bash` preprocessing rule also supports importation of installed modules by searching in the `PATH` environment variable.

By convention, `bash` module libraries should use the `lib` name prefix, for example, `libhello`. If there is also a native library (that is, one written in C/C++) that provides the same functionality (or the `bash` library is a language binding for said library), then it is customary to add the `.bash` extension to the `bash` library name, for example, `libhello.bash`. Note that in this case the project base name is `libhello`.

Modules can be *private* or *public*. Private modules are implementation details of a specific project and are not expected to be imported from other projects. The `hello/hello-world.bash.in` module above is an example of a private module. Public modules are meant to be used by other projects and are normally packaged into libraries, like the `libhello/say-hello.bash` module above.

Public modules must take care to avoid name clashes. Since `bash` does not have a notion of namespaces, the recommended way is to prefix all module functions (and global variables, if any) with the library name (without the `lib` prefix), like in the `libhello/say-hello.bash` module above.

While using such decorated function names can be unwieldy, it is relatively easy to create wrappers with shorter names and use those instead. For example:

```
@import libhello/say-hello@

function say_hello () { hello_say_hello "$@"; }
```

A module should normally also prevent itself from being sourced multiple times. The recommended way to achieve this is to begin the module with a *source guard*. For example:

```
# libhello/say-hello.bash

if [ "$hello_say_hello" ]; then
  return 0
else
  hello_say_hello=true
fi

function hello_say_hello ()
{
  echo "Hello, $1!"
}
```

The `bash` preprocessing rule matches `exe{}` targets that have the corresponding `in{}` and one or more `bash{}` prerequisites as well as `bash{}` targets that have the corresponding `in{}` prerequisite (if you need to preprocess a script that does not depend on any modules, you can use the `in` module's rule).