The `build2` Toolchain Introduction

Copyright © 2014-2024 the build2 authors.
Permission is granted to copy, distribute and/or modify this document under the terms of the MIT License.

Revision `0.17`, May 2024
This revision of the document describes the `build2` toolchain `0.17.x` series.

Table of Contents

Preface 1
1 Getting Started Guide 1
1.1 Hello, World 1
1.2 Package Repositories 11
1.3 Adding and Removing Dependencies 14
1.4 Upgrading and Downgrading Dependencies 18
1.5 Build-Time Dependencies and Linked Configurations 20
1.6 Versioning and Release Management 25
1.7 Developing Multiple Packages and Projects 31
1.8 Package Consumption 36
1.9 Using System-Installed Dependencies 40
1.10 Using Unpackaged Dependencies 43
2 Canonical Project Structure 45
2.1 Source Subdirectory 47
2.2 Source Naming 50
2.3 Source Contents 52
2.4 Tests 53
2.5 Build Output 54

Preface

This document is an overall introduction to the `build2` toolchain that shows how the main components, namely the build system, the package dependency manager, and the project dependency manager are used together to handle the entire C/C++ project development lifecycle: creation, development, testing, and delivery. For additional information, including documentation for individual toolchain components, man pages, HOWTOs, etc., refer to the `build2` project Documentation page.

1 Getting Started Guide

The aim of this guide is to get you started developing C/C++ projects with the `build2` toolchain. All the examples in this section include the relevant command output so if you just want to get a sense of what `build2` is about, then you don't have to install the toolchain and run the commands in order to follow along. Or, alternatively, you can take a short detour to the Installation Instructions and then try the examples for yourself.

One of the primary goals of the `build2` toolchain is to provide a uniform interface across all the platforms and compilers. While most of the examples in this document assume a UNIX-like operation system, they will look pretty similar if you are on Windows. You just have to use appropriate paths, compilers, and options.

The question we will try to answer in this section can be summarized as:

```
$ git clone .../hello.git && now-what?
```

That is, we clone an existing C/C++ project or would like to create a new one and then start hacking on it. We want to spend as little time and energy as possible on the initial and ongoing infrastructure maintenance: setting up build configurations, managing dependencies, continuous integration and testing, release management, etc. Or, as one C++ user aptly put it, "*All I want to do is program.*"

1.1 Hello, World

Let's see what programming with `build2` feels like by starting with a customary *"Hello, World!"* program (here we assume our current working directory is `/tmp`):

```
$ bdep new -l c++ -t exe hello
created new executable project hello in /tmp/hello/
```

The **`bdep-new(1)`** command creates a `build2` project. In this case it is an executable implemented in C++.

To create a library, pass `-t lib`. By default `new` also initializes a `git` repository and generates suitable `.gitignore` files (pass `-s none` if you don't want that). And for details on naming your projects, see Package Name.

Note to Windows users: the `build2-baseutils` package includes core `git` utilities that are sufficient for the `bdep` functionality.

Let's take a look inside our new project:

```
$ tree hello
hello/
|-- .git/
|-- .bdep/
|-- build/
|-- hello/
|   |-- hello.cxx
|   |-- buildfile
|   `-- testscript
|-- buildfile
|-- manifest
|-- README.md
`-- repositories.manifest
```

See Canonical Project Structure for a detailed discussion and rationale behind this layout. While it is recommended, especially for new projects, `build2` is flexible enough to support various arrangements used in today's C and C++ projects. Furthermore, the **`bdep-new(1)`** command provides a number of customization options and chances are good you will be able to create your preferred layout automatically. See SOURCE LAYOUT for more information and examples.

Similar to version control tools, we normally run all `build2` tools from the project's source directory or one of its subdirectories, so:

```
$ cd hello
```

While the project layout is discussed in more detail in later sections, let's examine a couple of interesting files to get a sense of what's going on. We start with the source file which should look familiar:

```
$ cat hello/hello.cxx

#include <iostream>

int main (int argc, char* argv[])
{
  using namespace std;

  if (argc < 2)
  {
    cerr << "error: missing name" << endl;
    return 1;
  }

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

If you prefer the `.?pp` extensions over `.?xx` for your C++ source files, pass `-l c++,cpp` to the `new` command. See **`bdep-new(1)`** for details on this and other customization options.

Let's take a look at the accompanying `buildfile`:

```
$ cat hello/buildfile

libs =
#import libs += libhello%lib{hello}

exe{hello}: {hxx ixx txx cxx}{**} $libs testscript
```

As the name suggests, this file describes how to build things. While its content might look a bit cryptic, let's try to infer a couple of points without going into too much detail (for details see the build system Introduction).

That `exe{hello}` on the left of `:` is a *target* (executable named `hello`) and what we have on the right are *prerequisites* (C++ source files, libraries, etc). This `buildfile` uses wildcard patterns (that `**`) to automatically locate all the C++ source files. This means we don't have to edit our `buildfile` every time we add, remove, or rename a source file in our project. There also appears to be some (commented out) infrastructure for importing and linking libraries (that `libs` variable). We will see how to use it in a moment.

In simple projects that follow the canonical structure we can often completely ignore the presence of the build definition files thus approaching the *build system-less* workflow found in languages like Rust and Go.

Finally, the `buildfile` also lists `testscript` as a prerequisite of `hello`. This file tests our target. Let's take a look inside:

```
$ cat hello/testscript

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

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

Again, we are not going into detail here (see Testscript Introduction for a proper introduction), but to give you an idea, here we have two tests: the first (with id `basics`) verifies that our program prints the expected greeting while the second makes sure it handles the missing name error condition. Tests written in Testscript are concise, portable, and executed in parallel.

Next up is `manifest`:

```
$ cat manifest
: 1
name: hello
version: 0.1.0-a.0.z
language: c++
summary: hello C++ executable
license: other: proprietary
description-file: README.md
url: https://example.org/hello
email: you@example.org
#depends: libhello ^1.0.0
```

The `manifest` file is what makes a build system project a *package*. It contains all the meta-data that a user of a package might need to know: its name, version, license, dependencies, etc., all in one place.

Refer to Manifest Format for the general format of `build2` manifest files and to Package Manifest for details on the package manifest values.

As you can see, `manifest` created by **`bdep-new(1)`** contains some dummy values which you would want to adjust before publishing your package. Specifically, you would want to review `summary`, `license`, `url`, and `email` as well as the `README.md` file referenced by `description-file`. Let's, however, resist the urge to adjust that strange looking `0.1.0-a.0.z` until we discuss package versioning.

Next to `manifest` you might have noticed the `repositories.manifest` file – we will discuss its function later, when we talk about dependencies and where they come from.

Project in hand, let's build it. Unlike other programming languages, C++ development usually involves juggling a handful of build configurations: several compilers and/or targets (`build2` is big on cross-compiling), debug/release, different sanitizers and/or static analysis tools, and so on. As a result, `build2` is optimized for multi-configuration usage. However, as we will see shortly, one build configuration can be designated as the default with additional conveniences.

The **`bdep-init(1)`** command is used to initialize a project in a build configuration. As a shortcut, it can also create a new build configuration in the process, which is just what we need here.

To create build configurations separately from initialization and to manage them after that, use the **`bdep-config(1)`** subcommands.

Let's start with GCC (remember we are in the project's root directory):

```
$ bdep init -C ../hello-gcc @gcc cc config.cxx=g++
initializing in project /tmp/hello/
created configuration @gcc /tmp/hello-gcc/ default,auto-synchronized
synchronizing:
  new hello/0.1.0-a.0.19700101000000
```

The `--config-create|-C` option instructs `init` to create a new configuration in the specified directory (`../hello-gcc` in our case). To make referring to configurations easier, we can give it a name, which is what we do with `@gcc`. The next argument (`cc`, stands for *C-common*) is the build system module we would like to configure. It implements compilation and linking rules for the C and C++ languages. Finally, `config.cxx=g++` is (one of) this module's configuration variables that specifies the C++ compiler we would like to use (the corresponding C compiler will be determined automatically). Let's for now also ignore that `synchronizing:...` bit along with strange-looking `19700101000000` in the version – it will become clear what's going on here in a moment.

Note to Windows users: a command line argument with leading `@` has a special meaning in PowerShell. To work around this, you can use the alternative `-@gcc` syntax or the `-n gcc` option.

Now the same for Clang:

```
$ bdep init -C ../hello-clang @clang cc config.cxx=clang++
initializing in project /tmp/hello/
created configuration @clang /tmp/hello-clang/ auto-synchronized
synchronizing:
  new hello/0.1.0-a.0.19700101000000
```

If we check the parent directory, we should now see two build configurations next to our project:

```
$ ls ..
hello/
hello-gcc/
hello-clang/
```

If, as in the above examples, our configuration directories are next to the project and their names are in the *`prj-name-cfg-name`* form, then we can use the shortcut version of the `init` command:

```
$ bdep init -C @clang cc config.cxx=clang++
```

Things will also look pretty similar if you are on Windows instead of a UNIX-like operating system. For example, to initialize our project on Windows with Visual Studio, start a command prompt and then run:

```
> bdep init -C ..\hello-debug @debug cc ^
  "config.cxx=cl /MDd"                  ^
  "config.cc.coptions=/Od /Zi"          ^
  config.cc.loptions=/DEBUG:FULL

> bdep init -C ..\hello-release @release cc ^
  config.cxx=cl                             ^
  config.cc.coptions=/O2
```

For Visual Studio, `build2` by default will use the latest available version and build for the `x86_64` target (`x64` in the Microsoft's terminology). You can, however, override these defaults by either running from a suitable Visual Studio development command prompt or by

specifying an absolute path to `cl.exe` that you wish to use. For example:

```
> bdep init -C ..\hello-debug-32 @debug-32 cc                         ^
  "config.cxx=...\VC\Tools\MSVC\14.23.28105\bin\Hostx64\x86\cl.exe"
  ...
```

In case of the command prompt, you may also want to make your configuration *hermetic* (Hermetic Build Configurations):

```
> bdep init -C ... cc ... config.config.hermetic=true
```

Hermetically configuring our project in a suitable Visual Studio command prompt makes us free to build it from any other prompt or shell, IDE, etc.

Besides the `coptions` (compile options) and `loptions` (link options), other commonly used `cc` module configuration variables are `poptions` (preprocess options) and `libs` (extra libraries to link). Here is the complete list with their rough `make` equivalents:

```
*.poptions   preprocess        CPPFLAGS
*.coptions   compile           CFLAGS/CXXFLAGS
*.loptions   link              LDFLAGS
*.aoptions   archive           ARFLAGS
*.libs       extra libraries   LIBS/LDLIBS
```

We can also use their `config.c.*` (C compilation) and `config.cxx.*` (C++ compilation) variants if we only want them applied during the respective language compilation/linking. For example:

```
$ bdep init ... cc                        \
  config.cxx=g++                          \
  config.cc.poptions=-D_FORTIFY_SOURCE=2 \
  config.cxx.poptions=-D_GLIBCXX_ASSERTIONS
```

Finally, we can specify the "compiler mode" options as part of the compiler executable in `config.c` and `config.cxx`. Such options cannot be modified by buildfiles and they will appear last on the command lines. For example:

```
$ bdep init ... cc      \
  config.c="clang -m32" \
  config.cxx="clang++ -m32 -stdlib=libc++"
```

The compiler mode options are also the correct place to specify *system-like* header (`-I`) and library (`-L`, `/LIBPATH`) search paths. Where by system-like we mean common installation directories like `/usr/include` or `/usr/local/lib` which may contain older versions of the libraries we are trying to build and/or use. By specifying these paths as part of the mode options (as opposed to `config.*.poptions` and `config.*.loptions`) we make sure they will be considered last, similar to the compiler's build-in search paths. For example:

```
$ bdep init ... cc config.cxx="g++ -L/opt/install"
```

One difference you might have noticed when creating the `gcc` and `clang` configurations above is that the first one was designated as the default. The default configuration is used by `bdep` commands if no configuration is specified explicitly (see **`bdep-projects-configs(1)`** for details). It is also the configuration that is used if we run the build system in the project's source directory. So, normally, you would make your every day development configuration the default. Let's try that:

```
$ bdep status
hello configured 0.1.0-a.0.19700101000000

$ b
c++ hello/cxx{hello} -> ../hello-gcc/hello/hello/obje{hello}
ld ../hello-gcc/hello/hello/exe{hello}
ln ../hello-gcc/hello/hello/exe{hello} -> hello/

$ b test
test ../hello-gcc/hello/hello/exe{hello} + hello/testscript{testscript}

$ hello/hello World
Hello, World!
```

To see the actual compilation command lines, run `b -v` and for even more details, run `b -V`. See **`b(1)`** for more information on these and other build system options.

In contrast, the Clang configuration has to be requested explicitly:

```
$ bdep status @clang
hello configured 0.1.0-a.0.19700101000000

$ b ../hello-clang/hello/
c++ hello/cxx{hello} -> ../hello-clang/hello/hello/obje{hello}
ld ../hello-clang/hello/hello/exe{hello}

$ b test: ../hello-clang/hello/
test ../hello-clang/hello/hello/exe{hello} +
     hello/testscript{testscript}

$ ../hello-clang/hello/hello/hello World
Hello, World!
```

As you can see, using the build system directly on configurations other than the default requires explicitly specifying their paths. It would have been more convenient if we could refer to them by names. The **`bdep-update(1)`** and **`bdep-test(1)`** commands allow us to do exactly that:

```
$ bdep test @clang
c++ hello/cxx{hello} -> ../hello-clang/hello/hello/obje{hello}
ld ../hello-clang/hello/hello/exe{hello}
test ../hello-clang/hello/hello/exe{hello} +
     hello/testscript{testscript}
```

And we can also perform the desired build system operation on several (or `--all|-a`) configurations at once:

```
$ bdep test @gcc @clang
in configuration @gcc:
test ../hello-gcc/hello/hello/exe{hello} + hello/testscript{testscript}

in configuration @clang:
test ../hello-clang/hello/hello/exe{hello} +
     hello/testscript{testscript}
```

As we will see later, the **`bdep-test(1)`** command also allows us to test immediate (`--immediate|-i`) or all (`--recursive|-r`) dependencies of our project. We call it *deep testing*.

While we are here, let's also check how hard it would be to cross-compile:

```
$ bdep init -C @mingw cc config.cxx=x86_64-w64-mingw32-g++
initializing in project /tmp/hello/
created configuration @mingw /tmp/hello-mingw/ auto-synchronized
synchronizing:
  new hello/0.1.0-a.0.19700101000000

$ bdep update @mingw
c++ hello/cxx{hello} -> ../hello-mingw/hello/hello/obje{hello}
ld ../hello-mingw/hello/hello/exe{hello}
```

As you can see, cross-compiling in `build2` is nothing special. In our case, on a properly setup GNU/Linux machine (that automatically uses `wine` as an `.exe` interpreter) we can even run tests (in `build2` this is called *cross-testing*):

```
$ bdep test @mingw
test ../hello-mingw/hello/hello/exe{hello} +
     hello/testscript{testscript}

$ ../hello-mingw/hello/hello/hello.exe Windows
Hello, Windows!
```

Let's review what it takes to initialize a project's infrastructure and perform the first build. For an existing project:

```
$ git clone .../hello.git
$ cd hello
$ bdep init -C ../hello-gcc @gcc cc config.cxx=g++
$ b
```

For a new project:

```
$ bdep new -l c++ -t exe hello
$ cd hello
$ bdep init -C ../hello-gcc @gcc cc config.cxx=g++
$ b
```

If you prefer, the `new` and `init` steps can be combined into a single command:

```
$ bdep new -l c++ -t exe hello -C hello-gcc @gcc cc config.cxx=g++
```

And if you need to deinitialize a project in one or more build configurations, there is the **bdep-deinit(1)** command for that:

```
$ bdep deinit @gcc @clang
deinitializing in project /tmp/hello/
in configuration @gcc:
synchronizing:
  drop hello

in configuration @clang:
synchronizing:
  drop hello
```

By default `bdep` initializes a project for development by automatically passing `config.<project>.develop=true` unless a custom value is specified. For example:

```
$ bdep init ... @gcc cc config.cxx=g++ config.hello.develop=false
```

To change the development mode of an already initialized project, use **bdep-sync(1)**:

```
$ bdep sync @gcc config.hello.develop=false
```

See Project Configuration for background on the development mode.

As mentioned earlier, by default **bdep-new(1)** initializes a `git` repository for us. Now that we have successfully built and tested our project, it might be a good idea to make a first commit and publish it to a remote repository where others can find it. Using GitHub as an example:

```
$ git add .
$ git commit -m "Initial implementation"
$ git remote add origin git@github.com:john-doe/hello.git
$ git push -u
```

We could have also done it the other way around: first created a project on one of the hosting services (GitHub, GitLab, etc) cloned it, and then ran `new` on that. One advantage of this approach is the `new` command's ability to automatically extract the license and description from the existing `LICENSE` and `README.md` files and use that to generate the `manifest` file. This way we only need to specify things once and everything is nice and consistent. Here is an example of this streamlined project creation workflow (notice also the omitted project name in the `new` command):

```
# Create a project with LICENSE and README.md on one of the Git
# hosting services (GitHub, GitLab, etc) and then:

$ git clone .../hello.git
$ cd hello

$ bdep new -l c++ -t exe
```

While we have managed to test a couple of platforms (Linux and Windows) and compiler versions (Clang and GCC) locally, there are quite a few combinations that we haven't tried (Mac OS with Apple Clang and Windows with MSVC, to name the major ones). We could test them manually, some with the help of virtualization while for others (such as Mac OS) we

may need physical hardware. Add a few versions for each compiler and we are looking at a dozen build configurations. Needless to say, testing on all of them manually is a lot of work. Now that we have our project available from a public remote repository, we can instead use the remote testing functionality offered by the **bdep-ci(1)** command. For example:

```
$ bdep ci
submitting:
  to:      https://ci.cppget.org
  in:      https://github.com/john-doe/hello.git#master@93e1dbc94baa
  package: hello
  version: 0.1.0-a.0.20180907091517.93e1dbc94baa
continue? [y/n] y
############################################################ 100.0%
CI request is queued:
  https://ci.cppget.org/@d6ee90f4-21a9-47a0-ab5a-7cd2f521d3d8
```

Let's see what's going on here. By default `ci` submits a test request to ci.cppget.org, a public CI service run by the `build2` project (see available Build Configurations and Use Policies). In our case it will be testing the current working tree state (branch and commit) of our package which should be available from our remote repository (on GitHub in this example) since that's where the CI service expects to get it from. In response we get a URL where we can see the build and test results, logs, etc.

This *push* CI model works particularly well with the "feature branch" development workflow. Specifically, you would develop a new feature in a separate branch, publishing and remote-testing it as necessary. When the feature is ready, you would merge any changes from `master`, test the result one more time, and then merge (fast-forward) the feature into master.

Now is a good time to get an overview of the `build2` toolchain. After all, we have already used two of its tools (`bdep` and `b`) without a clear understanding of what they actually are.

Unlike most other programming languages that encapsulate the build system, package dependency manager, and project dependency manager into a single tool (such as Rust's `cargo` or Go's `go`), `build2` is a hierarchy of several tools that you will be using directly and which together with your version control system (VCS) will constitute the core of your project management toolset.

While `build2` can work without a VCS, this will result in reduced functionality.

At the bottom of the hierarchy is the `build2` build system, which we invoke using the **b(1)** driver. Next comes the package dependency manager, **bpkg(1)**. It is primarily used for *package consumption* and depends on the build system. The top of the hierarchy is the project dependency manager, **bdep(1)**. It is used for *project development* and relies on `bpkg` for building project packages and their dependencies.

The main reason for this separation is modularity and the resulting flexibility: there are situations where we only need the build system (for example, when building a package for a system package manager where all the dependencies should be satisfied from the system repository), or only the build system and package manager (for example, when a build bot is building a package for testing).

Note also that strictly speaking `build2` is not C/C++-specific; its build model is general enough to handle any DAG-based operations and its package/project dependency management can be used for any compiled language.

As we will see in a moment, `build2` also integrates with your VCS in order to automate project versioning. Note that currently only `git(1)` is supported.

Now that we understand the tooling, let's also revisit the notion of *build configuration* (those `hello-gcc` and `hello-clang` directories). While we often talk of build configurations in the abstract, as a set of common options used to build our code, in `build2` this term also has a very concrete meaning – a directory where our projects and their dependencies are built with such a set of common options.

The concept of a build configuration appears prominently throughout the toolchain: a `bdep` build configuration is actually a `bpkg` build configuration which, in the build system terms, is a special kind of an *amalgamation* – a project that contains *subprojects*. In our case, the subprojects in these amalgamations will be the projects we have initialized with `init` and, as we will see in a moment, packages that they depend on. For example, here is what our `hello-gcc` contains:

```
$ tree hello-gcc
hello-gcc/
|-- .bpkg/
|-- build/
|   `-- config.build
`-- hello/
    |-- build/
    |   `-- config.build
    `-- hello/
        |-- hello
        `-- hello.o
```

Underneath **`bdep-init(1)`** with the `--config-create|-C` option calls **`bpkg-cfg-create(1)`** which, in turn, performs the build system `create` meta-operation (see **`b(1)`** for details).

The important point here is that the `bdep` build configuration is not a black box that you should never look inside of. On the contrary, it is a well-defined concept of the package manager and the build system and as long as you understand what you are doing, you should feel free to interact with it directly.

Let's now move on to the reason why there is *dep* in the `bdep` name: dependency management.

1.2 Package Repositories

Say we have realized that writing *"Hello, World!"* programs is a fairly common task and that someone must have written a library to help with that. So let's see if we can find something suitable to use in our project.

Where should we look? That's a good question. But before we can try to answer it, we need to understand where `build2` can source dependencies. In `build2` packages usually come from *package repositories*. Two commonly used repository types are *version control* and *archive*-based (see **`bpkg-repository-types(1)`** for details).

As the name suggests, a version control-based repository uses a VCS as its distribution mechanism. Currently, only `git` is supported. Such a repository normally contains multiple versions of a single package or, perhaps, of a few related packages.

An archive-based repository contains multiple, potentially unrelated packages/versions as archives along with some metadate (package list, prerequisite/complement repositories, signatures, etc) that are all accessible via HTTP(S).

Version control and archive-based repositories have different trade-offs. Version control-based repositories are great for package developers since with services like GitHub they are trivial to setup. In fact, your project's (already existing) VCS repository will normally be the `build2` package repository – you might need to add a few files, but that's about it.

However, version control-based repositories are not without drawbacks: It will be hard for your users to discover your packages (try searching for "hello library" on GitHub – most of the results are not even in C++ let alone packaged for `build2`). There is also the issue of continuous availability: users can delete their repositories, services may change their policies or go out of business, and so on. Version control-based repositories also lack repository authentication and package signing. Finally, obtaining the available package list for such repositories can be slow.

A central, archive-based repository would address all these drawbacks: It would be a single place to search for packages. Published packages will never disappear and can be easily mirrored. Packages are signed and the repository is authenticated (see **`bpkg-repository-signing(1)`** for details). And, last, but not least, archive-based repositories are fast.

cppget.org is the `build2` community's central package repository. While centralized, it is also easy to mirror since its contents are accessible via plain HTTPS (you can browse pkg.cppget.org to get an idea). As an added benefit, packages on cppget.org are continuously built and tested on all the major platform/compiler combinations with the results available as part of the package description.

The main drawback of archive-based repositories is the setup cost. Getting a basic repository going is relatively easy – all you need is an HTTP(S) server. Adding a repository web interface like that on cppget.org will require running `brep`. And adding CI will require running a bunch of build bots (`bbot`). Note also that in `build2` archive-based repositories can be federated with different sections of the repository being hosted/managed potentially independently.

To summarize, version control-based repositories are great for package developers while a central, archive-based repository is convenient for package consumers. A reasonable strategy then is for package developers to publish their releases to a central repository. Package consumers can then decide which repository to use based on their needs. For example, one

could use cppget.org as a (fast, reliable, and secure) source of stable versions but also add, say, `git` repositories for select packages (perhaps with the `#HEAD` fragment filter to improve download speed) for testing development snapshots. In this model the two repository types complement each other.

Publishing of packages to archive-based repositories is discussed in Versioning and Release Management.

Let's see how all this works in practice. Go over to cppget.org and type "hello library" in the search box. At the top of the search result you should see the `libhello` package and if you follow the link you will see the package description page along with a list of available versions. Pick a version that you like and you will see the package version description page with quite a bit of information, including the list of platform/compiler combinations that this version has been successfully (or unsuccessfully) tested with. If you like what you see, copy the `repository` value – this is the repository location where this package version can be sourced from.

The cppget.org repository is split into several sections: `stable`, `testing`, `beta`, `alpha` and `legacy`, with each section having its own repository location (see the repository's about page for details on each section's policies). Note also that `testing` is complemented by `stable`, `beta` by `testing`, and so on, so you only need to choose the lowest stability level and you will automatically "see" packages from the more stable sections.

The cppget.org `stable` sections will always contain the `libhello` library version `1.0.X` that was generated using the following **`bdep-new(1)`** command line:

```
$ bdep new -l c++ -t lib libhello
```

It can be used as a predictable test dependency when setting up new projects.

Let's say we've visited the `libhello` project's home page (for example by following a link from the package details page) and noticed that it is being developed in a `git` repository. How can we see what's available there? If the releases are tagged, then we can infer the available released versions from the tags. But that doesn't tell us anything about what's happening on the `HEAD` or in the branches. For that we can use the package manager's **`bpkg-rep-info(1)`** command:

```
$ bpkg rep-info https://git.build2.org/hello/libhello.git
libhello/1.0.0
libhello/1.1.0
```

As you can see, besides `1.0.0` that we have seen in `cppget.org/stable`, there is also `1.1.0` (which is perhaps being tested in `cppget.org/testing`). We can also check what might be available on the `HEAD` (see **`bpkg-repository-types(1)`** for details on the `git` repository URL format):

```
$ bpkg rep-info https://git.build2.org/hello/libhello.git#HEAD
libhello/1.1.1-a.0.20180504111511.2e82f7378519
```

We can also use the `rep-info` command on archive-based repositories, however, if available, the web interface is usually more convenient and provides more information.

To summarize, we found two repositories for the `libhello` package: the archive-based cppget.org that contains the released versions as well as its development `git` repository where we can get the bleeding edge stuff. Let's now see how we can add `libhello` to our project.

1.3 Adding and Removing Dependencies

So we found `libhello` that we would like to use in our `hello` project. First, we edit the `repositories.manifest` file found in the root directory of our project and add one of the `libhello` repositories as a prerequisite. Let's start with cppget.org:

```
role: prerequisite
location: https://pkg.cppget.org/1/stable
```

Refer to Repository Manifest for details on the repository manifest values.

Next, we edit the `manifest` file (again, found in the root of our project) and specify the dependency on `libhello` with optional version constraint. For example:

```
depends: libhello ^1.0.0
```

Let's briefly discuss version constraints (for details see the `depends` value documentation). A version constraint can be expressed with a comparison operator (==, >, <, >=, <=), a range shortcut operator (~ and ^), or a range. Here are a few examples:

```
depends: libhello == 1.2.3
depends: libhello >= 1.2.3

depends: libhello ~1.2.3
depends: libhello ^1.2.3

depends: libhello [1.2.3 1.2.9)
```

You may already be familiar with the tilde (~) and caret (^) constraints from dependency managers for other languages. To recap, tilde allows upgrades to any further patch versions while caret also allows upgrades to further minor versions. They are equivalent to the following ranges:

```
~X.Y.Z  [X.Y.Z  X.Y+1.0)

^X.Y.Z  [X.Y.Z  X+1.0.0)  if X >  0
^0.Y.Z  [0.Y.Z  0.Y+1.0)  if X == 0
```

Zero major version component is customarily used during early development where the minor version effectively becomes major. As a result, the caret constraint has a special treatment of this case.

Unless you have good reasons not to (for example, a dependency does not use semantic versioning), we suggest that you use the ^ constraint which provides a good balance between compatibility and upgradability with ~ being a more conservative option.

Besides the version constraint, the dependency declaration supports a number of more advanced features, including conditional dependencies, dependency alternatives, and dependency configuration. For details, see the `depends` value documentation.

Ok, we've specified where our package comes from (`repositories.manifest`) and which versions we find acceptable (`manifest`). The next step is to edit `hello/buildfile` and import the `libhello` library into our build:

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

Finally, we modify our source code to use the library:

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

int main (int argc, char* argv[])
{
  ...
  hello::say_hello (cout, argv[1]);
}
```

You are probably wondering why we have to specify this repeating information in so many places. Let's start with the source code: we can't specify the version constraint or location there because it will have to be repeated in every source file that uses the dependency.

Moving up, `buildfile` is also not a good place to specify this information for the same reason (a library can be imported in multiple buildfiles) plus the build system doesn't really know anything about version constraints or repositories which is the purview of the dependency management tools.

Finally, we have to separate the version constraint and the location because the same package can be present in multiple repositories with different policies. For example, when a package from a version control-based repository is published in an archive-based repository, its `repositories.manifest` file is ignored and all its dependencies should be available from the archive-based repository itself (or its fixed set of prerequisite repositories). In other words, `manifest` belongs to a package while `repositories.manifest` – to a repository.

Also note that this is unlikely to become burdensome since adding new dependencies is not something that happens often. There are also ideas to automate this with a `bdep-add(1)` command in the future.

To summarize, these are the files we had to modify to add a dependency to our project:

```
repositories.manifest   # add https://pkg.cppget.org/1/stable
manifest                # add 'depends: libhello ^1.0.0'
buildfile               # import libhello library
hello.cxx               # include libhello header (or import module)
```

While the repository URL and package name are easy to find on the cppget.org's package description page, the C/C++ library ecosystem unfortunately does not follow any predictable library or header naming scheme. If the library documentation does not provide any clues, then another place to check are the library tests and examples that can often be found in the package source directory (or source repository). In particular, every library in the `stable` section of the cppget.org repository should provide at least a basic test.

With a new dependency added, let's check the status of our project:

```
$ bdep status
fetching pkg:cppget.org/stable (prerequisite of dir:/tmp/hello)
warning: authenticity of the certificate for pkg:cppget.org/stable
         cannot be established
certificate is for cppget.org, "Code Synthesis" <admin@cppget.org>
certificate SHA256 fingerprint:
70:64:FE:E4:E0:F3:60:F1:B4:<...>:E5:C2:68:63:4C:A6:47:39:43
trust this certificate? [y/n] y

hello configured 0.1.0-a.0.19700101000000
      available  0.1.0-a.0.19700101000000#1
```

The **`bdep-status(1)`** command has detected that the dependency information has changed and tells us that a new *iteration* of our project (that `#1`) is now available for *synchronization* with the build configuration.

We've also been prompted to authenticate the prerequisite repository. This will have to happen once for every build configuration we initialize our project in and can quickly become tedious. To overcome this, we can mention the certificate fingerprint that we wish to automatically trust in the `repositories.manifest` file (replace it with the actual fingerprint from the repository's about page):

```
role: prerequisite
location: https://pkg.cppget.org/1/stable
trust: 70:64:FE:E4:E0:F3:60:F1:B4:<...>:E5:C2:68:63:4C:A6:47:39:43
```

To synchronize a project with one or more build configurations we use the **`bdep-sync(1)`** command:

```
$ bdep sync
synchronizing:
  new libhello/1.0.0 (required by hello)
  upgrade hello/0.1.0-a.0.19700101000000#1
```

Or we could just build the project without an explicit `sync` – if necessary, it will be automatically synchronized:

```
$ b
synchronizing:
  new libhello/1.0.0 (required by hello)
  upgrade hello/0.1.0-a.0.19700101000000#1
c++ ../hello-gcc/libhello-1.0.0/libhello/cxx{hello} ->
    ../hello-gcc/libhello-1.0.0/libhello/objs{hello}
ld ../hello-gcc/libhello-1.0.0/libhello/libs{hello}
c++ hello/cxx{hello} -> ../hello-gcc/hello/hello/obje{hello}
ld ../hello-gcc/hello/hello/exe{hello}
ln ../hello-gcc/hello/hello/exe{hello} -> hello/
```

The synchronization as performed by the `sync` command is two-way: dependency packages are first added, removed, upgraded, or downgraded in build configurations according to the project's version constraints and user input. Then the actual versions of the dependencies present in the build configurations are recorded in the project's `lockfile` so that if desired, the build can be reproduced exactly. The `lockfile` functionality is not yet implemented. For a new dependency the latest available version that satisfies the version constraint is used.

Synchronization is also the last step in the **`bdep-init(1)`** command's logic.

Let's now examine the status in all (`--all|-a`) the build configurations and include the immediate dependencies (`--immediate|-i`):

```
$ bdep status -ai
in configuration @gcc:
hello configured 0.1.0-a.0.19700101000000#1
  libhello ^1.0.0 configured 1.0.0

in configuration @clang:
hello configured 0.1.0-a.0.19700101000000
      available 0.1.0-a.0.19700101000000#1
```

Since we didn't specify a configuration explicitly, only the default (`gcc`) was synchronized. Normally, you would try a new dependency in one configuration, make sure everything looks good, then synchronize the rest with `--all|-a` (or, again, just build what you need directly). Here are a few examples (see **`bdep-projects-configs(1)`** for details):

```
$ bdep sync -a
$ bdep sync @gcc @clang
$ bdep sync -c ../hello-mingw
```

After adding a new (or upgrading/downgrading existing) dependency, it's a good idea to *deep-test* our project: run not only our own tests but also of its immediate (`--immediate|-i`) or even all (`--recursive|-r`) dependencies. For example:

```
$ bdep test -ai
in configuration @gcc:
test ../hello-gcc/libhello-1.0.0/tests/basics/exe{driver}
test ../hello-gcc/hello/hello/exe{hello} + hello/testscript{testscript}

in configuration @clang:
test ../hello-clang/libhello-1.0.0/tests/basics/exe{driver}
test ../hello-clang/hello/hello/exe{hello} +
     hello/testscript{testscript}
```

To get rid of a dependency, we simply remove it from the `manifest` file and synchronize the project. For example, assuming `libhello` is no longer mentioned as a dependency in our `manifests`:

```
$ bdep status
hello configured 0.1.0-a.0.19700101000000#1
      available  0.1.0-a.0.19700101000000#2

$ bdep sync
synchronizing:
  drop libhello/1.0.0 (unused)
  upgrade hello/0.1.0-a.0.19700101000000#2
```

If instead of building a dependency from source you would prefer to use a version that is installed by your system package manager, see Using System-Installed Dependencies. And for information on using dependencies that are not `build2` packages refer to Using Unpackaged Dependencies.

1.4 Upgrading and Downgrading Dependencies

Let's say we would like to try that `1.1.0` version we have seen in the `libhello` `git` repository. First, we need to add the repository to the `repositories.manifest` file:

```
role: prerequisite
location: https://git.build2.org/hello/libhello.git
```

Note that we don't need the `trust` value since `git` repositories are not authenticated.

To refresh the list of available dependency versions we use the **`bdep-fetch(1)`** command (or the `--fetch|-f` option to `status`):

```
$ bdep fetch
$ bdep status libhello
libhello configured 1.0.0 available [1.1.0]
```

To upgrade (or downgrade) dependencies we again use the **`bdep-sync(1)`** command. We can upgrade one or more specific dependencies by listing them as arguments to `sync`:

```
$ bdep sync libhello
synchronizing:
  new libformat/1.0.0 (required by libhello)
  new libprint/1.0.0 (required by libhello)
  upgrade libhello/1.1.0
  upgrade hello/0.1.0-a.0.19700101000000#3
```

Without an explicit version or the `--patch|-p` option, `sync` will upgrade the specified dependencies to the latest available versions. For example, if we don't like version `1.1.0`, we can downgrade it back to `1.0.0` by specifying the version explicitly (we pass `--old-available|-o` to `status` to see the old versions):

```
$ bdep status -o libhello
libhello configured 1.1.0 available (1.1.0) [1.0.0]

$ bdep sync libhello/1.0.0
synchronizing:
  drop libprint/1.0.0 (unused)
  drop libformat/1.0.0 (unused)
  downgrade libhello/1.0.0
  reconfigure hello/0.1.0-a.0.19700101000000#3
```

The available versions are listed in the descending order with `[]` indicating that the version is only available as a dependency and `()` marking the current version.

Instead of specific dependencies we can also upgrade (`--upgrade|-u`) or patch (`--patch|-p`) immediate (`--immediate|-i`) or all (`--recursive|-r`) dependencies of our project.

As a more realistic example, version `1.1.0` of `libhello` depends on two other libraries: `libformat` and `libprint`. Here is our project's dependency tree while we were still using that version:

```
$ bdep status -r
hello configured 0.1.0-a.0.19700101000000#3
  libhello ^1.0.0 configured 1.1.0
    libformat ^1.0.0 configured 1.0.0
    libprint ^1.0.0 configured 1.0.0
```

A typical conservative dependency management workflow would look like this:

```
$ bdep status -fi  # refresh and examine immediate dependencies
hello configured 0.1.0-a.0.19700101000000#3
  libhello configured 1.1.0 available [2.0.0] [1.2.0] [1.1.2] [1.1.1]

$ bdep sync -pi    # upgrade immediate to latest patch version
synchronizing:
  upgrade libhello/1.1.2
  reconfigure hello/0.1.0-a.0.19700101000000#3
continue? [Y/n] y
```

Notice that in case of such mass upgrades you are prompted for confirmation before anything is actually changed (unless you pass `--yes|-y`).

In contrast, the following would be a fairly aggressive workflow where we upgrade everything to the latest available version (version constraints permitting; here we assume `^1.0.0` was used for all the dependencies):

```
$ bdep status -fr  # refresh and examine all dependencies
hello configured 0.1.0-a.0.19700101000000#3
  libhello configured 1.1.0 available [2.0.0] [1.2.0] [1.1.1]
    libprint configured 1.0.0 available [2.0.0] [1.1.0] [1.0.1]
    libformat configured 1.0.0 available [2.0.0] [1.1.0] [1.0.1]

$ bdep sync -ur    # upgrade all to latest available version
synchronizing:
  upgrade libprint/1.1.0
```

```
  upgrade libformat/1.1.0
  upgrade libhello/1.2.0
  reconfigure hello/0.1.0-a.0.19700101000000#3
continue? [Y/n] y
```

We can also have something in between: patch all (`sync -pr`), upgrade immediate (`sync -ui`), or even upgrade immediate and patch the rest (`sync -ui` followed by `sync -pr`).

1.5 Build-Time Dependencies and Linked Configurations

The `libhello` dependency we've been playing with in the previous two sections is a *runtime dependency*, that is, our `hello` executable needs it at run-time. This is typical of libraries and most of our dependencies will be of this kind. However, sometimes we may only wish to use a dependency during the build, typically a tool, such as a source code generator. This kind of dependency is called a *build-time dependency*.

Build-time dependencies are an advanced topic and if you don't have an immediate need for this functionality, you may skip this section without any loss of continuity.

Why do we need to distinguish between the two kinds of dependencies? The primary reason is cross-compilation: if we build a tool in the same (cross-compiling) build configuration as our project, then we will not be able to execute it during the build (since it's built for a different target than what we are running). But even if you are not planning to cross-compile, there are other good reasons: if you have multiple build configurations for your project, you may want to share a single build of your tool between them (why waste time building the same thing multiple times). And even if you only have a single build of your project, you may want to build the tool with different options (for example, optimized instead of debug).

You can probably see where this is going: in order to properly support build-time dependencies, we need to distinguish them from runtime and we need an ability to build them in a separate build configuration.

Let's see how all this works using the `xxd` tool as an example. If you are not familiar, `xxd` is a hexdump utility which can be used to embed external binary data into C/C++ code in a portable manner. Specifically, it can read a binary file and produce a C array definition of its contents. For example:

```
$ xxd -i names.txt

unsigned char names_txt[] = {
  0x57, 0x6f, 0x72, 0x6c, 0x64, 0x0a, 0x55, 0x6e, 0x69, 0x76, 0x65,
  0x72, 0x73, 0x65, 0x0a, 0x50, 0x65, 0x6f, 0x70, 0x6c, 0x65, 0x0a,
  0x4d, 0x61, 0x72, 0x74, 0x69, 0x61, 0x6e, 0x73, 0x0a
};
unsigned int names_txt_len = 31;
```

While the above output is a bit old school (using `unsigned int` instead of `size_t`) and the array/length names are derived from the input file name (including directories), `xxd` can also produce just the array values allowing us to wrap it into an array of our choice. See the

`xxd` package description for examples of `build2` recipes that do that.

So here is an idea: instead of failing if the user did not specify the name to greet, let's improve our `hello` program to greet a random generic name from a pre-defined list. To make this list easier to maintain, let's keep it in a separate file called `names.txt` and use `xxd` to embed it into our `hello` executable. We can use the one name per line format, for example:

```
$ cat names.txt
World
Universe
People
Martians
```

The first step in our plan is to add a build-time dependency on `xxd` to our project's `manifest`, similar to how we did for `libhello`:

```
...
depends: libhello ^1.0.0
depends: * xxd >= 8.2.0
```

The `*` mark in front of the `xxd` name indicates that it's a build-time dependency.

Next we import `xxd` in our `buildfile`:

```
...

import libs += libhello%lib{hello}

import! [metadata] xxd = xxd%exe{xxd}

...
```

There are two main differences compared to the way we import the `libhello` library: we request metadata (`[metadata]`) and we do immediate importation (`import!`). Let's briefly discuss what this means (for details, refer to Target Importation in the build system manual). Metadata for an executable contains information that helps the build system do a better job when an executable is used as part of the build. For example, it includes the uniform program name to be used for low-verbosity diagnostics as well as the version, checksum, and environment that are used to detect changes. And immediate importation instructs the build system to skip rule-specific importation (for example, search for libraries in compiler-specific search paths) and import the target here and now, failing if that's not possible. It is usually appropriate for importing executables. Note also that the metadata can only be requested in immediate importation.

While requesting the metadata means that you will have a simpler `buildfile` and a more reliable build, it also likely means that you won't be able to use the system-installed version of the executable since it needs to be patched to provide the metadata.

Now that we have the `xxd` tool, let's use it from an ad hoc recipe to convert `names.txt` to `names.cxx`. Here is the complete `buildfile` for our `hello` executable:

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

import! [metadata] xxd = xxd%exe{xxd}

exe{hello}: {hxx ixx txx cxx}{** -names} cxx{names} $libs testscript

cxx{names}: file{names.txt} $xxd
{{
  i = $path($<[0])
  env --cwd $directory($i) -- $xxd -i $leaf($i) >$path($>)
}}
```

The last bit that we need to do is to modify `hello.cxx` to use the list of fallback names (the actual implementation is left as an exercise for the reader):

```
#include <iostream>

extern unsigned char names_txt[];
extern unsigned int names_txt_len;

int main (int argc, char* argv[])
{
  using namespace std;

  if (argc < 2)
  {
    // TODO: pick a random name from names_txt using newline as
    //       a name separator.
  }

  ...
}
```

Let's recap what we've achieved so far: we've added a build-time dependency on `xxd`, we've imported it in our `buildfile` and used it in an ad hoc recipe to generate `names.cxx`, and we've modified `hello.cxx` to use the generated list of names. The only step left is to actually try to build it. But before doing that, let's also print the list of build configurations we currently have associated with our project (see the `list` subcommand in **`bdep-config(1)`**):

```
$ bdep config list
@gcc /tmp/hello-gcc/ 1 target default,forwarded,auto-synchronized
@clang /tmp/hello-clang/ 2 target auto-synchronized

$ b
creating configuration of host type in /tmp/hello-host/ and
associating it with project(s):
  /tmp/hello/
as if by executing command(s):
  bdep config create @host --type host --no-default /tmp/hello-host \
  cc config.config.load=~host
while searching for configuration for build-time dependency xxd of
package hello/0.1.0-a.0.19700101000000#4
while synchronizing configuration /tmp/hello-gcc/
continue? [Y/n] y

synchronizing /tmp/hello-gcc/:
  new xxd/8.2.3075 [/tmp/hello-host/] (required by hello)
```

```
  upgrade hello/0.1.0-a.0.19700101000000#4

c ../hello-host/xxd-8.2.3075+1/c{xxd} ->
  ../hello-host/xxd-8.2.3075+1/obje{xxd}
ld ../hello-host/xxd-8.2.3075/exe{xxd}
xxd hello/file{names.txt} -> ../hello-gcc/hello/hello/cxx{names}
c++ ../hello-gcc/hello/hello/cxx{names} ->
    ../hello-gcc/hello/hello/obje{names}
c++ hello/cxx{hello} -> ../hello-gcc/hello/hello/obje{hello}
ld ../hello-gcc/hello/hello/exe{hello}
```

While the diagnostics is hopefully fairly self-explanatory, let's go over the key points. The first part goes exactly as in the previous section: because we've added a new dependency, the build configuration needs to be synchronized with the project state. However, this is a build-time dependency and build-time dependencies are built in configurations of type `host`. So `bdep` first looks for such a configuration among the configurations already associated with the project. In our case there isn't one (from the listing above we can see that all our configurations are of type `target`). In this case, `bdep` offers to create one automatically. We accept this offer by answering `y` at the prompt and the rest should again look familiar: the new dependency is configured and built (but now in the host configuration) and our project is updated (which involves running the new dependency). If we now again print the list of build configurations associated with our project, we will see the new configuration among them:

```
$ bdep config list
@gcc /tmp/hello-gcc/ 1 target default,forwarded,auto-synchronized
@clang /tmp/hello-clang/ 2 target auto-synchronized
@host /tmp/hello-host/ 3 host forwarded,auto-synchronized
```

Let's also try to update our project in the `clang` configuration:

```
$ bdep update @clang
synchronizing:
  upgrade hello/0.1.0-a.0.19700101000000#4

xxd hello/file{names.txt} -> ../hello-clang/hello/hello/cxx{names}
c++ ../hello-clang/hello/hello/cxx{names} ->
    ../hello-clang/hello/hello/obje{names}
c++ hello/cxx{hello} -> ../hello-clang/hello/hello/obje{hello}
ld ../hello-clang/hello/hello/exe{hello}
```

This time we are neither prompted to create another configuration nor is a new instance of `xxd` built – as we would have expected, the existing host configuration with the already built `xxd` is reused.

From the above output we can see that `bdep` creates the host configuration using the default host compiler and build options (`~host`) which means the result will most likely be optimized. But if we don't like something about the host configuration that `bdep` offers us to create, we can answer `n` at the prompt, create one ourselves (by perhaps copying and tweaking the command line `bdep` was going to use), and then restart the build.

Besides the `target` and `host` types, the third pre-defined configuration type is `build2`, which is used for build system modules. If you would like to try a build-time dependency on a build system module, there is a dummy `libbuild2-hello` module that you can use.

Simply add the following line to your `manifest`:

```
depends: * libbuild2-hello
```

And the following line somewhere in your `buildfile`:

```
using hello
```

Then build the project and see what happens.

The `target` type signifies a configuration for the end-result of our build. If no type is specified during the configuration creation with the `--type` option (or `--config-type` if using `bdep-new`), then `target` is assumed.

The `host` type signifies a configuration corresponding to the host machine, that is, the machine on which the build is performed. It is expected that an executable built in the host configuration can be executed. Oftentimes, target and host are the same. In this case, if you would prefer not to have separate configurations, then you can make your target configuration *self-hosted* by using the `host` type rather than `target`. For example:

```
$ bdep init -C ../hello-gcc @gcc --type host cc config.cxx=g++
```

The `build2` type is a special kind of host configuration that is used to build build system modules. It cannot be self-hosted.

Building build-time dependencies in separate configurations is just one application of the more general configuration linking mechanism which allows us to build a package in one configuration while its dependencies – in one or more linked configurations. This, for example, can be used to create a "base" configuration with common dependencies that are shared between multiple configurations (sometimes also referred to as build configuration overlaying).

Let's see how this works on our `hello` project. Imagine `libhello` that we depend on is very big and takes a while to compile. We also aren't really interested in building it in both `gcc` and `clang` configurations (it's our project that we are interested in building with different compilers). Since these two compilers are ABI-compatible (at least on Linux), we could build `libhello` with just one of them and reuse the result with the other. Let's see how we can achieve this with linked configurations (refer to **`bdep-config(1)`** for details on subcommands involved):

```
$ bdep config create ../hello-base @base --no-default cc config.cxx=g++
$ bdep config create ../hello-gcc @gcc --default cc config.cxx=g++
$ bdep config create ../hello-clang @clang cc config.cxx=clang++

$ bdep config link @gcc @base
$ bdep config link @clang @base

$ bdep init @gcc { @base }+ ?libhello
$ bdep init @clang
```

Most of the commands are hopefully self-explanatory except for the `{ @base }+ ?libhello` part. Here `?` is a package flag that instructs `bdep` to treat `libhello` as a dependency. And `{ @base }+` tells it to build this dependency in the `base` configuration (we don't have to do the same for `clang` since the dependency is already built). See **`bdep-sync(1)`** for details on this syntax.

1.6 Versioning and Release Management

Let's now discuss versioning and release management and, yes, that strange-looking `0.1.0-a.0.19700101000000` we keep seeing. While a build system project doesn't need a version and a `bpkg` package can use custom versioning schemes (see Package Version), a project managed by `bdep` must use *standard versioning*. A dependency, which is a `bpkg` package, need not use standard versioning.

Standard versioning (*stdver*) is a semantic versioning (*semver*) scheme with a more precisely defined pre-release component and without any build metadata.

If you believe that *semver* is just *`major.minor.patch`*, then in your worldview *stdver* would be the same as *semver*. In reality, *semver* also allows loosely defined pre-release and build metadata components. For example, `1.2.3-beta.1+build.23456` is a valid *semver*.

A standard version has the following form:

`major.minor.patch`[**`-`***`prerel`*]

The *`major`*, *`minor`*, and *`patch`* components have the same meaning as in *semver*. The *`prerel`* component is used to provide *continuous versioning* of our project between releases. Specifically, during development of a new version we may want to publish several pre-releases, for example, alpha or beta. In between those we may also want to publish a number of snapshots, for example, for CI. With continuous versioning all these releases, pre-releases, and snapshots are assigned unique, properly ordered versions.

Continuous versioning is a cornerstone of the `build2` project dependency management. In case of snapshots, an appropriate version is assigned automatically in cooperation with your VCS.

The *`prerel`* component for a pre-release has the following form:

(**`a`**|**`b`**)**`.`***`num`*

Here **`a`** stands for alpha, **`b`** stands for beta, and *`num`* is the alpha/beta number. For example:

```
1.1.0        # final              release  for 1.1.0
1.2.0-a.1    # first  alpha   pre-release  for 1.2.0
1.2.0-a.2    # second alpha   pre-release  for 1.2.0
1.2.0-b.1    # first  beta    pre-release  for 1.2.0
1.2.0        # final              release  for 1.2.0
```

The *prerel* component for a snapshot has the following form:

(**a**|**b**).*num*.*snapsn*[.*snapid*]

Where *snapsn* is the snapshot sequence number and *snapid* is the snapshot id. In case of `git`, *snapsn* is the commit timestamp in the `YYYYMMDDhhmmss` form and UTC timezone while *snapid* is a 12-character abbreviated commit id. For example:

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

Notice also that a snapshot version is ordered *after* the corresponding pre-release version. That is, `1.2.3-a.1 < 1.2.3-a.1.1`. As a result, it is customary to start the development of a new version with `X.Y.Z-a.0.z`, that is, a snapshot after the (non-existent) zero'th alpha release. We will explain the meaning of **z** in this version momentarily. The following chronologically-ordered versions illustrate a typical release flow of a project that uses `git` as its VCS:

```
0.1.0-a.0.19700101000000               # snapshot (no commits yet)
0.1.0-a.0.20180319215815.26efe301f4a7  # snapshot (first commit)
...                                    # more commits/snapshots
0.1.0-a.1                              # pre-release (first alpha)
0.1.0-a.1.20180319221826.a6f0f41205b8  # snapshot
...                                    # more commits/snapshots
0.1.0-a.2                              # pre-release (second alpha)
0.1.0-a.2.20180319231937.b701052316c9  # snapshot
...                                    # more commits/snapshots
0.1.0-b.1                              # pre-release (first beta)
0.1.0-b.1.20180319242038.c812163417da  # snapshot
...                                    # more commits/snapshots
0.1.0                                  # release
0.2.0-a.0.20180319252139.d923274528eb  # snapshot (first in 0.2.0)
...
```

For a more detailed discussion of standard versioning and its support in `build2` refer to `version` Module.

Let's now see how this works in practice by publishing a couple of versions for our `hello` project. By now it should be clear what that `0.1.0-a.0.19700101000000` means – it is the first snapshot version of our project. Since there are no commits yet, it has the UNIX epoch as its commit timestamp. Let's see what changes after we've made our first commit:

```
$ git add .
$ git commit -m "Initial implementation"

$ bdep status
hello configured 0.1.0-a.0.19700101000000
      available  0.1.0-a.0.20180507062614.ee006880fc7e
```

Just like with changes to dependency information, `status` has detected that a new (snapshot) version of our project is available for synchronization.

Another way to view the project's version (which works even if we are not using `bdep`) is with the build system's `info` meta-operation:

```
$ b info
project: hello
version: 0.1.0-a.0.20180507062614.ee006880fc7e
summary: hello C++ executable
...
```

Let's synchronize with the default build configuration:

```
$ bdep sync
synchronizing:
  upgrade hello/0.1.0-a.0.20180507062614.ee006880fc7e

$ bdep status
hello configured 0.1.0-a.0.20180507062614.ee006880fc7e
```

Notice that we didn't have to manually change the version anywhere. All we had to do was commit our changes and a new snapshot version was automatically derived by `build2` from the new `git` commit. Without this automation continuous versioning would hardly be practical.

If we now make another commit, we will see a similar picture:

```
$ bdep status
hello configured 0.1.0-a.0.20180507062614.ee006880fc7e
      available  0.1.0-a.0.20180507062615.8fb9de05b38f
```

Note that you don't need to manually run `sync` after every commit. As discussed earlier, you can simply run the build system to update your project and things will get automatically synchronized if necessary.

Ok, time for our first release. Let's start with `0.1.0-a.1`. Unlike snapshots, for pre-releases as well as final releases we have to change the version in the `manifest` file:

```
version: 0.1.0-a.1
```

The `manifest` file is the singular place where we specify the package version. The build system's `version` module makes it available in various forms in buildfiles and even source code.

To ensure continuous versioning, this change to version must be the last commit for this (pre-)release which itself must be immediately followed by a second change to the version starting the development of the next (pre-)release. We also recommend that you tag the release commit with a tag name in the **`v`***`X.Y.Z`* form.

Having regular release tag names with the **`v`** prefix allows one to distinguish them from other tags, for example, with wildcard patterns.

Here is the release workflow for our example:

```
$ git commit -a -m "Release version 0.1.0-a.1"
$ git tag -a v0.1.0-a.1 -m "Tag version 0.1.0-a.1"
$ git push --follow-tags

# Version 0.1.0-a.1 is now public.

$ edit manifest  # change 'version: 0.1.0-a.1.z'
$ git commit -a -m "Change version to 0.1.0-a.1.z"
$ git push

# Master is now open for business.
```

Notice also that when specifying a snapshot version in `manifest` we use the special **`z`** snapshot value (for example, `0.1.0-a.1.z`) which is recognized and automatically replaced by `build2` with, in case of `git`, the current commit timestamp and id (refer to `version` Module for details).

While not particularly complicated, performing the release steps manually is both tedious and error-prone. Instead, this process can be automated with the **`bdep-release(1)`** command. Specifically, in its default mode, this command will update the version in the `manifest` file, commit and tag this change, open the next development cycle (again, by changing `manifest` and committing), and, finally, if `--push` is specified, push everything to the remote. So, instead of the above manual steps, we could have simply run:

```
$ bdep release --alpha --push
releasing:
  package: hello
  current: 0.1.0-a.0.z
  release: 0.1.0-a.1
  open:    0.1.0-a.1.z
  commit:  yes
  tag:     v0.1.0-a.1
  push:    origin/master
continue? [y/n] y
[master 82a7e65] Release version 0.1.0-a.1
[master e6cf3c0] Change version to 0.1.0-a.1.z
pushing branch master, tag v0.1.0-a.1
To github.com:john-doe/hello.git
   26ec5c9..e6cf3c0  master -> master
 * [new tag]         v0.1.0-a.1 -> v0.1.0-a.1
```

The `release` command has a number of alternative modes, such as for releasing a package revision, as well as a number of options that control which version will be released and which version will be opened. See **`bdep-release(1)`** for details.

Publishing the final release to the version control repository is exactly the same. This time, however, let's see how we can also publish it to an archive-based repository. The first step is again to make the release, which we will do with the help of the `release` command. Except now we will delay opening the next development cycle by passing `--no-open` (there is also no `--alpha` since this is the final release):

```
$ bdep release --no-open --push
releasing:
  package: hello
  current: 0.1.0-a.1.z
  release: 0.1.0
  commit:  yes
  tag:     v0.1.0
  push:    origin/master
continue? [y/n] y
[master 00ed45a] Release version 0.1.0
pushing branch master, tag v0.1.0
To github.com:john-doe/hello.git
   5d5094c..00ed45a  master -> master
 * [new tag]         v0.1.0 -> v0.1.0
```

To publish our project to an archive-based repository we use the **`bdep-publish(1)`** command. For example:

```
$ bdep publish
publishing:
  to:      https://cppget.org
  as:      John Doe <john@example.org>
  package: hello
  version: 0.1.0
  project: hello
  section: alpha
  control: https://github.com/john-doe/hello.git
continue? [y/n] y
pushing branch build2-control
submitting hello-0.1.0.tar.gz
############################################################ 100.0%
package submission is queued: https://queue.cppget.org/hello/0.1.0
reference: 0c596fca2017
```

Let's see what's going on here. By default `publish` submits to the cppget.org repository. On `cppget.org` package names are assigned on a first come first serve basis. But instead of using logins or emails to authenticate package ownership, `cppget.org` uses your version control repository as a proxy. In a nutshell, when we submit a package for the first time, its control repository is associated with its name and all subsequent submissions have to use the same control repository (the authentication part). When submitting a package, `publish` also adds a file to the `build2-control` branch of the control repository with the package archive checksum. On the other side, `cppget.org` checks for the presence of this file to make sure that whomever is making this submission has write access to the control repository (the authorization part). See **`bdep-publish(1)`** for details.

The rest should be pretty straightforward: `publish` prepares and uploads a distribution of our package which goes into the `alpha` section of the repository (because it has `0` major version). In response we get a URL which we can use to check the status of our submission on queue.cppget.org. And after some basic testing and verification, our package should appear on `cppget.org` (the exact steps are described in Submission Policies). Note also that package submissions to `cppget.org` are public and permanent and cannot be removed under any circumstances.

Finally, we also shouldn't forget to increment the version for the next development cycle. For that we can use the `--open` mode of the `release` command. For example:

```
$ bdep release --open --push
opening:
  package: hello
  current: 0.1.0
  open:    0.2.0-a.0.z
  commit:  yes
  push:    origin/master
continue? [y/n] y
[master ace2f6e] Change version to 0.2.0-a.0.z
pushing branch master
To github.com:john-doe/hello.git
   00ed45a..ace2f6e  master -> master
```

One sticky point of continuous versioning is choosing the next version. For example, above should we continue with `0.1.1-a.0`, `0.2.0-a.0`, or `1.0.0-a.0`? The important rule to keep in mind is that we can jump forward to any further version at any time and without breaking continuous versioning. But we can never jump backwards.

For example, we can start with `0.2.0-a.0` but if we later realize that this will actually be a new major release, we can easily change it to `1.0.0-a.0`. As a result, the general guideline is to start conservatively by either incrementing the patch or the minor version component. And the recommended strategy is to increment the minor component and, if required, release patch versions from a separate branch (created by branching off from the release commit). This is the default behavior of the `release` command.

Note also that you don't have to make any pre-releases if you don't need them. While during development you would still keep the version as `X.Y.Z-a.0`, at release you simply change it directly to the final `X.Y.Z`.

When publishing the final release you may also want to clean up now obsolete pre-release tags. For example:

```
$ git tag -l 'v0.1.0-*' | xargs git push --delete origin
$ git tag -l 'v0.1.0-*' | xargs git tag --delete
```

While at first removing such tags may seem like a bad idea, pre-releases are by nature temporary and their use only makes sense until the final release is published.

Also note that having a `git` repository with a large number of published but unused version tags may result in a significant download overhead.

Let's also briefly discuss in which situations we should increment each of the version components. While *semver* gives basic guidelines, there are several ways to apply them in the context of C/C++ where there is a distinction between binary and source compatibility. We recommend that you reserve *patch* releases for specific bug fixes and security issues that you can guarantee with a high level of certainty to be binary-compatible. Otherwise, if the changes are source-compatible, increment *minor*. And if they are breaking (that is, the user code likely will need adjustments), increment *major*. During early development, when breaking changes

are frequent, it is customary to use the `0.Y.Z` versions where `Y` effectively becomes the *major* component. Again, refer to the `version` Module for a more detailed discussion of this topic.

1.7 Developing Multiple Packages and Projects

How does a library like `libhello` get developed? It's possible someone woke up one day and realized that they were going to build a useful library that everyone was going to use. But somehow this doesn't feel like how it really works. In the real world things start organically: someone had a project like `hello` and then needed the same functionality in another project. Or someone else needed it and asked the author to factor it out into a library. For this approach to work, however, moving such common functionality into a library and then continue its parallel development must be a simple, frictionless process. Let's see how this works in `build2`.

First, we need to decide whether to make `libhello` another package in our `hello` project (that is, in the same `git` repository) or a separate project (with a separate repository). Both arrangements are equally well supported.

A multi-package project works best if all the packages have the same version and are released together. While the packages themselves can have different versions (since each has its own `manifest`), in this scenario following the release tagging recommendations discussed earlier will be problematic.

Let's start with a separate project since it is simpler. As the first step we use **`bdep-new(1)`** to create a new library project next to our `hello`:

```
$ bdep new -l c++ -t lib libhello
created new library project libhello in /tmp/libhello/

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

$ tree libhello
libhello/
|-- build/
|   `-- ...
|-- libhello/
|   |-- hello.hxx
|   |-- hello.cxx
|   `-- buildfile
|-- buildfile
|-- manifest
|-- README.md
`-- repositories.manifest
```

Similar to the executable project, this layout is not mandatory and **`bdep-new(1)`** can create a number of alternative library structures. For example, if you prefer the `include/src` split, try:

```
$ bdep new -l c++ -t lib,split libhello
```

See SOURCE LAYOUT for more examples.

Let's edit the generated `manifest` file and add the `project` value (customarily after `version`) to indicate that our library belongs to the same overall project as our executable:

```
$ cat libhello/manifest
: 1
name: libhello
version: 0.1.0-a.0.z
project: hello
summary: hello C++ library
...
```

The `project` value is used to group related packages together in order to help with their organization and discovery. For example, if later we create `libhello2` or `libhello-extra`, then it would make sense for them to also belong to the `hello` project. See the `project` value documentation for details.

Our two projects will be sharing the same set of build configurations, so next we initialize `libhello` in `hello-gcc` and `hello-clang` (notice the use of `--config-add|-A` instead of `--config-create|-C`):

```
$ cd libhello

$ bdep init -A ../hello-gcc @gcc
initializing in project /tmp/libhello/
added configuration @gcc /tmp/hello-gcc/ default,auto-synchronized
synchronizing:
  new libhello/0.1.0-a.0.19700101000000

$ bdep init -A ../hello-clang @clang
initializing in project /tmp/libhello/
added configuration @clang /tmp/hello-clang/ auto-synchronized
synchronizing:
  new libhello/0.1.0-a.0.19700101000000
```

If two or more projects share the same build configuration, then all of them are always synchronized at once, regardless of the originating project. It also makes sense to have the same default configuration and use identical configuration names in all the projects.

The last step is to move the desired functionality from `hello` to `libhello` and at the same time add a dependency on `libhello`, just as we did earlier (add a `depends` entry to `manifest`, then import the library in `buildfile`, and so on). One interesting question is what to put as a prerequisite repository in `repositories.manifest`. Our own setup will work even if we don't put anything there – the dependency will be automatically resolved to our local version of `libhello` since we have initialized it in all our build configurations. However, in case our `hello` repository is used by someone else, it's a good idea to add the remote `git` repository for `libhello` as a prerequisite.

By now you have probably realized that our project directory is just another type of package repository. See **`bpkg-repository-types(1)`** for more information.

And that's it, now we can build and test our new arrangement:

```
$ cd ../hello   # back to hello project root
$ bdep test -i
c++ ../libhello/libhello/cxx{hello} ->
    ../hello-gcc/libhello/libhello/objs{hello}
c++ ../libhello/tests/basics/cxx{driver} ->
    ../hello-gcc/libhello/tests/basics/obje{driver}
c++ hello/cxx{hello} -> ../hello-gcc/hello/hello/obje{hello}
ld ../hello-gcc/libhello/libhello/libs{hello}
ld ../hello-gcc/libhello/tests/basics/exe{driver}
ld ../hello-gcc/hello/hello/exe{hello}
test ../hello-gcc/libhello/tests/basics/exe{driver}
test ../hello-gcc/hello/hello/exe{hello} + hello/testscript{testscript}
```

This is also the approach we would use if we wanted to fix a bug in someone else's library. That is, we would clone their library repository and initialize it in the build configurations of our project which will "upgrade" the dependency to use the local version. Then we make the fix, submit it upstream, and continue using the local version until our fix is merged/published, at which point we deinitialize their library repository and our project will be automatically switched back to using the new upstream version of the library. Here is the summary of the steps in this workflow:

```
$ cd hello/                       # Our project.
$ bdep init -C @gcc ...           # Configures libhello as a dependency.

$ git clone .../libhello.git      # Need to fix a bug in libhello.
$ cd libhello
$ bdep init -A ../hello-gcc @gcc # Upgrades libhello to local version.

# Fix the bug in libhello, test, and submit upstream.
# Continue using local libhello until the bugfix is published.

$ cd libhello                     # Bugfix has been published.
$ bdep deinit @gcc                # Switches libhello back to dependency.

$ rm -r libhello                  # If no longer needed.
```

Let's now examine the second option: making `libhello` a package inside `hello`. Here is the original structure of our `hello` project:

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

As the first step, we move the `hello` program into its own subdirectory:

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

Next we again use **`bdep-new(1)`** to create a new library but this time as a package inside an already existing project:

```
$ cd hello
$ bdep new --package -l c++ -t lib libhello
created new library package libhello in /tmp/hello/libhello/
```

Let's see what our project looks like now:

```
hello/
|-- .git/
|-- hello/
|   |-- ...
|   `-- manifest
|-- libhello/
|   |-- ...
|   `-- manifest
|-- buildfile
|-- packages.manifest
`-- repositories.manifest
```

Notice that, as discussed earlier, `repositories.manifest` belongs to the project (repository) while `manifest` – to the package.

Besides the `libhello` directory the `new` command also created the `buildfile` and `packages.manifest` files in the root directory of our project. First let's take a look inside `buildfile`:

```
import pkgs = */

./: $pkgs
```

This is what we call a *glue* `buildfile`. Its purpose is to "pull" together several packages so that we are able to invoke the build system driver from the project root. See Target Importation for details.

Now let's examine `packages.manifest`:

```
$ cat packages.manifest
: 1
location: libhello/
```

Up until now our `hello` was a simple, single-package project that didn't need this file – `manifest` in its root directory was sufficient (see **bpkg-repository-types(1)** for details on the project repository structure). But now it contains several packages and we need to specify where they are located within the project. So let's go ahead and add the location of the `hello` package:

```
$ cat packages.manifest
: 1
location: libhello/
:
location: hello/
```

Packages in a project can reside next to each other or in subdirectories but they cannot nest. When published to an archive-based repository, each such package will be placed into its own archive.

Next we initialize the new package in all our build configurations:

```
$ cd libhello
$ bdep init -a
initializing in project /tmp/hello/
in configuration @gcc:
synchronizing:
  upgrade hello/0.1.0-a.0.19700101000000#1
  new libhello/0.1.0-a.0.19700101000000

in configuration @clang:
synchronizing:
  upgrade hello/0.1.0-a.0.19700101000000#1
  new libhello/0.1.0-a.0.19700101000000
```

Notice that the `hello` package has been "upgraded" to reflect its new location.

Finally, as before, we move the desired functionality from `hello` to `libhello` and at the same time add a dependency on `libhello`. Note, however, that in this case we don't need to add anything to `repositories.manifest` since both packages are in the same project (repository). And that's it, now we can build and test our new arrangement:

```
$ cd ..      # back to hello project root
$ bdep test
c++ libhello/libhello/cxx{hello} ->
    ../hello-gcc/libhello/libhello/objs{hello}
c++ libhello/tests/basics/cxx{driver} ->
    ../hello-gcc/libhello/tests/basics/obje{driver}
c++ hello/hello/cxx{hello} -> ../hello-gcc/hello/hello/obje{hello}
ld ../hello-gcc/libhello/libhello/libs{hello}
ld ../hello-gcc/libhello/tests/basics/exe{driver}
ld ../hello-gcc/hello/hello/exe{hello}
test ../hello-gcc/libhello/tests/basics/exe{driver}
test ../hello-gcc/hello/hello/exe{hello} +
     hello/hello/testscript{testscript}
```

A multi-package project could have several files, such as `README.md` and `LICENSE`, which, while shared by all the packages, must nevertheless reside within each package's directory. The recommended way to avoid the duplication is to use symlinks. For example:

```
hello/
|-- .git/
|-- hello/
|   |-- ...
|   |-- LICENSE  -> ../LICENSE
|   `-- manifest
|-- libhello/
|   |-- ...
|   |-- LICENSE  -> ../LICENSE
|   `-- manifest
|-- LICENSE
|-- buildfile
|-- packages.manifest
`-- repositories.manifest
```

See Using Symlinks in `build2` Projects for details.

1.8 Package Consumption

Ok, now that we have published a few releases of `hello`, how would the users of our project get them? While they could clone the repository and use `bdep` just like we did, this is more of a development than a consumption workflow. For consumption it is much easier to use the package dependency manager, **`bpkg(1)`**, directly.

Note that this approach also works for libraries in case you wish to use them in a project with a build system other than `build2`. See Using Unpackaged Dependencies for background on cross-build system library consumption.

First, we create a suitable build configuration with the **`bpkg-cfg-create(1)`** command. We can use the same place for building all our tools so let's call the directory `tools`. Seeing that we are only interested in using (rather than developing) such tools, let's build them optimized and also configure a suitable installation location:

```
$ bpkg create -d tools cc        \
  config.cxx=g++                 \
  config.cc.coptions=-O3         \
  config.install.root=/usr/local \
  config.install.sudo=sudo
created new configuration in /tmp/tools/
```

The same step on Windows using Visual Studio would look like this:

```
$ bpkg create -d tools cc ^
  config.cxx=cl           ^
  config.cc.coptions=/O2  ^
  config.install.root= C:\install
```

To fetch and build packages (as well as all their dependencies) we use the **`bpkg-pkg-build(1)`** command. We can use either an archive-based repository like cppget.org or build directly from `git`:

```
$ cd tools

$ bpkg build hello@https://git.build2.org/hello/hello.git
fetching from https://git.build2.org/hello/hello.git
  new libformat/1.0.0 (required by libhello)
  new libprint/1.0.0 (required by libhello)
  new libhello/1.1.0 (required by hello)
  new hello/1.0.0
continue? [Y/n] y
configured libformat/1.0.0
configured libprint/1.0.0
configured libhello/1.1.0
configured hello/1.0.0
c++ libprint-1.0.0/libprint/cxx{print} ->
    libprint-1.0.0/libprint/objs{print}
c++ hello-1.0.0/hello/cxx{hello} -> hello-1.0.0/hello/obje{hello}
c++ libhello-1.1.0/libhello/cxx{hello} ->
    libhello-1.1.0/libhello/objs{hello}
c++ libformat-1.0.0/libformat/cxx{format} ->
    libformat-1.0.0/libformat/objs{format}
ld libprint-1.0.0/libprint/libs{print}
ld libformat-1.0.0/libformat/libs{format}
ld libhello-1.1.0/libhello/libs{hello}
ld hello-1.0.0/hello/exe{hello}
updated hello/1.0.0
```

Passing a repository URL to the `build` command is a shortcut to the following sequence of commands:

```
$ bpkg add https://git.build2.org/hello/hello.git  # add repository
$ bpkg fetch                                       # fetch package list
$ bpkg build hello                                 # build package by name
```

If building a package involves building a build-time dependency and no configuration of type `host` (or `build2`, if the dependency is a build system module) is linked with the target configuration, then a private configuration of a suitable type is automatically created and linked. See Build-Time Dependencies and Linked Configurations for background on build-time dependencies and **`bpkg-cfg-create(1)`** for more information on `bpkg` configuration linking.

Once built, we can install the package to the location that we have specified with `config.install.root` using the **`bpkg-pkg-install(1)`** command:

```
$ bpkg install hello
...
install libformat-1.0.0/libformat/libs{format} -> /usr/local/lib/
install libprint-1.0.0/libprint/libs{print} -> /usr/local/lib/
install libhello-1.1.0/libhello/libs{hello} -> /usr/local/lib/
install hello-1.0.0/hello/exe{hello} -> /usr/local/bin/

$ hello World
Hello, World!
```

If on your system the installed executables don't run from `/usr/local` because of the unresolved shared libraries (or if you are installing somewhere else, such as `/opt`), then the easiest way to fix this is with *rpath*. Simply add the following configuration variable when

creating the build configuration (or as an argument to the `install` command):

```
config.bin.rpath=/usr/local/lib
```

Note to Windows users: this is not an issue on this platform since executables and shared (DLL) libraries are installed into the same subdirectory (`bin`) of the installation directory.

The installation contents and layout under `config.install.root` would be along these lines:

```
/usr/local/
|-- bin/
|   `-- hello
|-- include/
|   |-- libformat/
|   |   |-- export.hxx
|   |   |-- format.hxx
|   |   `-- version.hxx
|   |-- libhello/
|   |   |-- export.hxx
|   |   |-- hello.hxx
|   |   `-- version.hxx
|   `-- libprint/
|       |-- export.hxx
|       |-- print.hxx
|       `-- version.hxx
|-- lib/
|   |-- libformat-1.0.so
|   |-- libformat.so -> libformat-1.0.so
|   |-- libhello-1.1.so
|   |-- libhello.so -> libhello-1.1.so
|   |-- libprint-1.0.so
|   |-- libprint.so -> libprint-1.0.so
|   `-- pkgconfig/
|       |-- libformat.pc
|       |-- libhello.pc
|       `-- libprint.pc
`-- share/
    `-- doc/
        |-- libformat/
        |   |-- manifest
        |   `-- README.md
        |-- libhello/
        |   |-- manifest
        |   `-- README.md
        `-- libprint/
            |-- manifest
            `-- README.md
```

The installation locations of various types of files (executables, libraries, headers, documentation, etc) can be customized using a number of the `config.install.*` variables with the most commonly used ones and their defaults (relative to `config.install.root`) listed below (see the `install` build system module documentation for the complete list).

```
config.install.bin     = root/bin/
config.install.lib     = root/lib/
config.install.doc     = root/share/doc/
config.install.man     = root/share/man/
config.install.include = root/include/
```

If we need to uninstall a previously installed package, there is the **`bpkg-pkg-uninstall(1)`** command:

```
$ bpkg uninstall hello
uninstall hello-1.0.0/hello/exe{hello} <- /usr/local/bin/
uninstall libhello-1.1.0/libhello/libs{hello} <- /usr/local/lib/
uninstall libprint-1.0.0/libprint/libs{print} <- /usr/local/lib/
uninstall libformat-1.0.0/libformat/libs{format} <- /usr/local/lib/
...
```

Rather than installing the package locally we could instead generate a *binary distribution package* for it using the **`bpkg-pkg-bindist(1)`** command. Such a binary package can then be installed on a different machine. Currently, the `bindist` command supports producing Debian (and alike, such as Ubuntu) and Fedora (and alike, such as RHEL) packages as well as installation archives for all operating systems. For example, to generate a Debian package for our `hello` (running on Debian or alike):

```
$ bpkg bindist --recursive=auto --private -o /tmp/hello-deb/ hello
...
generated debian package for hello/1.0.0:
  /tmp/hello-deb/hello_1.0.0-0~debian12_amd64.deb
  /tmp/hello-deb/hello-dbgsym_1.0.0-0~debian12_amd64.deb
  /tmp/hello-deb/hello_1.0.0-0~debian12_amd64.buildinfo
  /tmp/hello-deb/hello_1.0.0-0~debian12_amd64.changes

$ sudo apt-get install /tmp/hello-deb/hello_1.0.0-0~debian12_amd64.deb
```

And to generate a Fedora package (running on Fedora or alike):

```
$ bpkg bindist --recursive=auto --private hello
...
generated fedora package for hello/1.0.0:
  ~/rpmbuild/RPMS/x86_64/hello-1.0.0-1.fc38.x86_64.rpm
  ~/rpmbuild/RPMS/x86_64/hello-debuginfo-1.0.0-1.fc38.x86_64.rpm

$ sudo dnf install ~/rpmbuild/RPMS/x86_64/hello-1.0.0-1.fc38.x86_64.rpm
```

And to generate an installation archive (running on Windows in this example):

```
$ bpkg bindist --recursive=auto   ^
  --private                       ^
  --distribution=archive          ^
  -o C:\tmp\hello-zip\            ^
  config.install.relocatable=true ^
  hello
...
generated archive package for hello/1.0.0:
  C:\tmp\hello-zip\hello-1.0.0-x86_64-windows10.zip
```

To upgrade or downgrade packages we again use the `build` command. Here is a typical upgrade workflow:

```
$ bpkg fetch              # refresh available package list
$ bpkg status             # see if new versions are available

$ bpkg uninstall hello    # uninstall old version
$ bpkg build     hello    # upgrade to the latest version
$ bpkg install   hello    # install new version
```

Similar to `bdep`, to downgrade we have to specify the desired version explicitly. There are also the `--upgrade|-u` and `--patch|-p` as well as `--immediate|-i` and `--recursive|-r` options that allow us to upgrade or patch packages that we have built and/or their immediate or all dependencies (see **`bpkg-pkg-build(1)`** for details). For example, to make sure everything is patched, run:

```
$ bpkg fetch
$ bpkg build -pr
```

If a package is no longer needed, we can remove it from the configuration with **`bpkg-pkg-drop(1)`**:

```
$ bpkg drop hello
following dependencies were automatically built but
will no longer be used:
  libhello
  libformat
  libprint
drop unused packages? [Y/n] y
  drop hello
  drop libhello
  drop libformat
  drop libprint
continue? [Y/n] y
purged hello
purged libhello
purged libformat
purged libprint
```

1.9 Using System-Installed Dependencies

Our operating system might already have a package manager (which we will refer to as *system package manager*) and for various reasons we may want to use the system-installed version of a dependency rather than building one from source.

Using system-installed versions works best for mature rather than rapidly-developed packages since for the latter you often need to track the latest version (which may not yet be available from the system repository) and/or test with multiple versions (which is not something that many system package managers support).

We can also have some build configurations using a system-installed version of a dependency while in others building it from source, for example, for testing.

We can instruct `build2` to configure a dependency package as available from the system rather than building it from source. Specifically, we can install a suitable version manually (for example, using the system package manager) and then communicate this fact as well as the version installed to `build2` so that it can use this information when resolving version constraints. Furthermore, for Debian (and alike, such as Ubuntu) and Fedora (and alike, such as RHEL) `build2` can automatically query the system package manager for the installed version and, if requested, automatically install a suitable version from the system repository if none is already installed.

Let's see how all this works in an example. Say, we want to use `libsqlite3` in our `hello` project.

The first step is to add it as a dependency, just like we did for `libhello`. That is, add another `depends` entry to `manifest`, then import it in `buildfile`, and so on.

Now, if we just run `sync` or try to build our project, `build2` will download and build the new dependency from source, just like it did for `libhello`. Instead, we can issue an explicit `sync` command that configures the `libsqlite3` package as coming from the system:

```
$ bdep sync ?sys:libsqlite3
```

Here `?` is a package flag that instructs `build2` to treat it as a dependency and `sys` is a package scheme that tells `build2` it comes from the system. See **`bpkg-pkg-build(1)`** for details.

Now what exactly happens in this case depends on which operating system we are running as well as whether `libsqlite3` is already installed. Let's examine each combination in turn.

If we are running on an operating system for which there is `build2` support for the system package manager interactions (currently Debian, Fedora, or alike) and `libsqlite3` is already installed, then `build2` will get its version from the system package manager and use that when resolving version constraints. For example, running the above command on Debian with `libsqlite3-dev` version `3.42.0` already installed:

```
$ bdep sync ?sys:libsqlite3
synchronizing:
  configure sys:libsqlite3/3.42.0 (required by hello)
  upgrade hello/0.1.0-a.0.19700101000000#3
```

If, on the other hand, we are running on an operating system for which there is `build2` support for the system package manager interactions but `libsqlite3` is not installed, then `build2` will fail:

```
$ bdep sync ?sys:libsqlite3
error: no installed system package for libsqlite3
  info: specify --sys-install to try to install it
  info: specify libsqlite3/* if package is not installed with system
        package manager
  info: specify --sys-no-query to disable system package manager
        interactions
```

As you can see, `build2` will not attempt to automatically install system packages unless explicitly requested with the `--sys-install` option. Let's try to add that (again, running on Debian):

```
$ bdep sync --sys-install ?sys:libsqlite3
updating debian package index...
synchronizing:
  sys-install libsqlite3-0/3.42.0-1 (required by sys:libsqlite3)
  configure sys:libsqlite3/3.42.0 (required by hello)
  upgrade hello/0.1.0-a.0.19700101000000#3
installing debian packages...
The following NEW packages will be installed:
  libsqlite3-dev
The following packages will be upgraded:
  libsqlite3-0 sqlite3
Do you want to continue? [Y/n] y
...
Setting up libsqlite3-0:amd64 (3.42.0-1) ...
Setting up libsqlite3-dev:amd64 (3.42.0-1) ...
Setting up sqlite3 (3.42.0-1) ...
```

You can suppress the system package manager confirmation prompt with the `--sys-yes` option. By default `build2` uses `sudo` for system package manager interactions that normally require administrative privileges (fetch package metadata, install packages, etc). This can be customized with the `--sys-sudo` option.

Finally, if we are running on an operating system for which there is no `build2` support for the system package manager interactions, then, as mentioned earlier, it is the user's responsibility to make sure a suitable package is installed and, optionally, communicate its version. In this case, unless we specify the installed version explicitly, a system-installed package is assumed to satisfy any dependency constraint (indicated with the `*` wildcard instead of the version):

```
$ bdep sync ?sys:libsqlite3
synchronizing:
  configure sys:libsqlite3/* (required by hello)
  upgrade hello/0.1.0-a.0.19700101000000#3
```

You can reduce the supported system package manager case to this case by disabling the system package manager interactions with the `--sys-no-query` option.

The system-installed dependency doesn't really have to come from the system package manager. It can also be manually installed and, as discussed in Using Unpackaged Dependencies, not necessarily into the system-default location like `/usr/local`.

In the above examples our dependency (`libsqlite3`) still has to be packaged and available from one of the project's prerequisite repositories. But it can be a *stub* – a package that does not contain any source code and that can only be "obtained" from the system.

The purpose of a stub is to provide the `build2` package to system package name and version mapping, in case it cannot be deduced automatically. See Package Version and `*-{name, version, to-downstream-version}` package manifest values for details.

If we would like to use a completely unpackaged dependency, then, for the supported system package manager case, we will need to pass the `--sys-no-stub` option:

```
$ bdep sync --sys-install --sys-no-stub ?sys:libsqlite3
```

And for the unsupported system package manager case we will have to specify the system version explicitly either as the actual version or as the `*` wildcard, for example:

```
$ bdep sync ?sys:libsqlite3/* ?sys:libcurl/7.47.0
```

The reason at least a stub is required by default is due to the automatic mapping between `build2` and system packages often being unreliable.

1.10 Using Unpackaged Dependencies

Generally, we will have a much better time if all our dependencies come as `build2` packages. Unfortunately, this won't always be the case in the real world and some libraries that you may need will use other build systems.

There is also the opposite problem: you may want to consume a library that uses `build2` in a project that uses a different build system. For that refer to Package Consumption.

The standard way to consume such unpackaged libraries is to install them (not necessarily into a system-default location like `/usr/local`) so that we have a single directory with their headers and a single directory with their libraries. We can then configure our builds to use these directories when searching for imported libraries.

Needless to say, none of the `build2` dependency management mechanisms such as version constraints or upgrade/downgrade will work on such unpackaged libraries. You will have to manage all these yourself manually.

Let's see how this all works in an example. Say, we want to use `libextra` that uses a different build system in our `hello` project. The first step is to manually build and install this library for each build configuration that we have. For example, we can install all such unpackaged libraries into `unpkg-gcc` and `unpkg-clang`, next to our `hello-gcc` and `hello-clang` build configurations:

```
$ ls
hello/
hello-gcc/
unpkg-gcc/
hello-clang/
unpkg-clang/
```

If you would like to try this out but don't have a suitable `libextra`, you can create and install one with these commands:

```
$ bdep new -l c++ -t lib libextra -C libextra-gcc cc config.cxx=g++
$ b install: libextra-gcc/ config.install.root=/tmp/unpkg-gcc
```

If we look inside one of these `unpkg-*` directories, we should see something like this:

```
$ tree unpkg-gcc
unpkg-gcc/
|-- include/
|   `-- libextra/
|       `-- extra.hxx
`-- lib/
    |-- libextra.a
    |-- libextra.so
    `-- pkgconfig/
        `-- libextra.pc
```

Notice that `libextra.pc` – it's a **`pkg-config(1)`** file that contains any extra compile and link options that may be necessary to consume this library. This is the *de facto* standard for build systems to communicate library build information to each other and is today supported by most commonly used implementations. Speaking of `build2`, it both recognizes `.pc` files when consuming third-party libraries and automatically produces them when installing its own.

While this may all seem foreign to Windows users, there is nothing platform-specific about this approach, including support for `pkg-config`, which, at least in case of `build2`, works equally well on Windows.

Next, we create a build configuration and configure it to use one of these `unpkg-*` directories (replace **`...`** with the absolute path):

```
$ bdep init -C ../hello-gcc @gcc cc config.cxx=g++ \
  config.cc.poptions=-I.../unpkg-gcc/include       \
  config.cc.loptions=-L.../unpkg-gcc/lib
```

If using Visual Studio, replace `-I` with `/I` and `-L` with `/LIBPATH:`.

Alternatively, if you want to reconfigure one of the existing build configurations, then simply edit the `build/config.build` file (that is, `hello-gcc/build/config.build` in our case) and adjust the `poptions` and `loptions` values. Or you can use the build system directly to reconfigure the build configuration (see **`b(1)`** for details):

```
b configure: ../hello-gcc/                    \
  config.cc.poptions+=-I.../unpkg-gcc/include \
  config.cc.loptions+=-L.../unpkg-gcc/lib
```

If all the unpackaged libraries included `.pc` files, then the `-L` alone would have been sufficient. However, it doesn't hurt to also add `-I`, for good measure.

Once this is done, adjust your `buildfile` to import the library:

```
import libs += libextra%lib{extra}
```

And your source code to use it:

```
#include <libextra/extra.hxx>
```

Notice that we don't add the corresponding `depends` value to the project's `manifest` since this library is not a package. However, it is a good idea to instead add a `requires` entry as a documentation to users of our project.

2 Canonical Project Structure

The goal of establishing a canonical `build2` project structure is to create an ecosystem of packages that can coexist, are easy to comprehend by both humans and tools, scale to complex, real-world requirements, and, last but not least, are pleasant to work with.

Here by *canonical* we mean a structure that on balance achieves these objectives in the simplest possible way. However, not everyone agrees with where that balance should be struck. As a result, this structure is only recommended and `build2` is flexible enough to support various arrangements used in modern C and C++ projects. Furthermore, the **`bdep-new(1)`** command provides a number of customization options and chances are you will be able to create your preferred layout automatically. See SOURCE LAYOUT for more information and examples.

This canonical structure is primarily meant for a package – a single library or program (or, sometimes, a collection of related libraries or programs) with a specific and well-defined function. While it may be less suitable for more elaborate, multi-library/program *end-products* that are not meant to be packaged, most of the recommendations discussed below would still apply. Oftentimes, you would start with a canonical project and expand from there. Note also that while the discussion below focuses on C++, most of it applies equally to C projects.

We often find ourselves factoring common functionality out of such end-products and into separate packages, for example, in order to be reused in another end-product. In this light, it can be helpful to organize a new end-product project as a composition of individual packages or source subdirectories that follow the canonical structure. The **`bdep-new(1)`** `--package` and `--source` modes can be used to automate this process.

By default, projects created by the **`bdep-new(1)`** command have the canonical structure. The overall layouts for executable (`-t exe`) and library (`-t lib`) projects are presented below.

```
<name>/
|-- build/
|-- <name>/
|   |-- <name>.cxx
|   |-- <name>.test.cxx
|   |-- testscript
|   `-- buildfile
|-- buildfile
|-- manifest
`-- README.md

lib<name>/
|-- build/
|-- lib<name>/
|   |-- <name>.hxx
|   |-- <name>.cxx
|   |-- <name>.test.cxx
|   |-- export.hxx
|   |-- version.hxx.in
|   `-- buildfile
|-- tests/
|-- buildfile
|-- manifest
`-- README.md
```

The canonical structure for both project types is discussed in detail in the following sections with a short summary of the key points presented below.

- *Header and source files (or module interface and implementation files) are next to each other (no `include/` and `src/` split).*
- *Headers are included with <> and contain the project name as a subdirectory prefix, for example, `<libhello/hello.hxx>`.*
- *Header and source file extensions are either `.hpp/.cpp` or `.hxx/.cxx` (`.mpp` or `.mxx` for module interfaces).*
- *No special characters other than _ and − in file names with `.` only used for extensions.*

Let's start with naming our projects: A project name should only contain ASCII alphabetic characters (`[a-zA-Z]`), digits (`[0-9]`), underscores (_), plus/minus (+−), and dots (`.`) as well as be at least two characters long (see Package Name for additional restrictions and recommendations).

If a project consists of a library and an executable, then they should be split into separate packages (see Developing Multiple Packages and Projects for some common arrangements). In this case, by convention, the library name should start with the `lib` prefix, for example, `libhello` and `hello`. It is also recommended (but not required) to follow this convention in new projects, even if there are no plans to have a related executable.

Using the `lib` prefix consistently offers several benefits:

1. It is clear from the name to both humans and tools what kind of project it is.
2. All libraries are consistently named (as opposed to some with the `lib` prefix and some without).
3. All library names are future-proofed to co-exist with executables. If one starts with a

library without the `lib` prefix but later decides to add an executable, renaming the library would unlikely be an option. And there is no need to spend mental energy on thinking whether it's possible that an executable will be added later.

The project's root directory should contain the root `buildfile` and package `manifest` file. Other recommended top-level subdirectory names are `examples/` (for libraries it is normally a subproject like `tests/`, as discussed below), `doc/`, and `etc/` (sample configurations, scripts, third-party contributions, etc). See also build system Project Structure for details on the build-related files (`buildfile`) and subdirectories (`build/`) as well as the available alternative naming scheme.

2.1 Source Subdirectory

The project's source code is placed into a subdirectory of the root directory named the same as the project, for example, `hello/hello/` or `libhello/libhello/`. It is called the project's *source subdirectory*.

There are several reasons for this layout: It implements the canonical inclusion scheme (discussed below) where each header is prefixed with its project name as a subdirectory. It also has a predictable name where users (and tools) 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 (like `README`, `LICENSE`) and directories (like `doc/`, `tests/`, `examples/`).

Another popular approach is to place public headers into the `include/` subdirectory and source files as well as private headers into `src/`. The cited advantage of this layout is the predictable location (`include/`) that contains only the project's public headers (that is, its API). This can make the project easier to navigate and understand while harder to misuse, for example, by including a private header.

However, this split layout is not without drawbacks:

- Navigating between corresponding headers and sources is cumbersome. This affects editing, grep'ing, as well as code browsing (for example, on GitHub).
- Implementing the canonical inclusion scheme would require an extra level of subdirectories (for example, `include/libhello/` and `src/libhello/`), which only amplifies the previous issue.
- Supporting generated source code can be challenging: Source code generators rarely provide support for writing headers and sources into different directories. Even if we can move things around post-generation, build systems may not support this arrangement (for example, `build2` does not currently support target groups with members in different directories).

Also, the stated advantage of this layout – separation of public headers from private – is not as clear cut as it may seem at first. The common assumption of the split layout is that only headers from `include/` are installed and, conversely, to use the headers in-place, all one has to do is add `-I` pointing to `include/`. On the other hand, it is common for public headers to

include private headers to, for example, call an implementation detail function in inline or template code (note that the same applies to private modules imported in public module interfaces). Which means such private (or probably now more accurately called *implementation detail*) headers have to be placed in the `include/` directory as well, perhaps into a subdirectory (such as `details/`) or with a file name suffix (such as `-impl`) to signal to the user that they are still "private". Needless to say, in an actively developed project, keeping track of which private headers can still stay in `src/` and which have to be moved to `include/` (and vice versa) is a tedious, error-prone task. As a result, practically, the split layout quickly degrades into the "all headers in `include/`" arrangement which negates its main advantage.

It is also not clear how the split layout will translate to modularized projects. With modules, both the interface and implementation (including non-inline/template function definitions) can reside in the same file with a substantial number of C++ developers finding this arrangement appealing. If a project consists of only such single-file modules, then `include/` and `src/` have effectively become the same thing (note that there couldn't be any "private" modules in `src/` since there would be nobody to import them). In a sense, we already have this situation with header-only libraries except that, in the case of modules, calling the directory `include/` would be an anachronism.

To summarize, the split directory arrangement offers little benefit over the combined directory layout, has a number of real drawbacks, and does not fit modularized projects well. In practice, private headers are placed into `include/`, often either in a subdirectory or with a special file name suffix, a mechanism that is readily available in the combined directory layout.

All headers within a project should be included using the <> style inclusion and contain the project name as a subdirectory prefix. And all headers means *all headers* – public, private, or implementation detail, in executables or in libraries.

As an example, let's say we've added `utility.hxx` to our `hello` project. This is how it should be included in `hello.cxx`:

```
// #include "utility.hxx"           // Wrong.
// #include <utility.hxx>           // Wrong.
// #include "../hello/utility.hxx"  // Wrong.

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

Similarly, if we want to include `hello.hxx` from `libhello`, then the inclusion should look like this:

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

The problem with the `""` style inclusion is if the header is not found relative to the including file, most compilers will continue looking for it in the include search paths, the same as for <>. As a result, if the header is not present in the right place (for example, because it was mistakenly not listed as to be installed), chances are that a completely unrelated header with the same name will be found and included. Needless to say, debugging situations like these is unpleasant.

Prefixing all inclusions with the project name as subdirectory also makes sure that headers with common names (for example, `utility.hxx`) can coexist (for example, when installed into a system-wide directory, such as `/usr/include`). The subdirectory prefix also plays an important role in supporting auto-generated headers.

Note also that this header inclusion scheme is consistent with the module importation, for example:

```
import hello.utility;
```

Finally, note that while adding the subdirectory prefix to the `""` style inclusion (for example, `"libhello/hello.hxx"`) will make finding an unrelated header unlikely, there is still a possibility. And it is not clear why take the chance when there are no benefits. So let's imagine the `""` style inclusion does not exist and we will all have a much better time.

If you have to disregard every rule and recommendation in this section but one, for example, because you are working on an existing library, then at minimum insist on this: **public header inclusions must use the library name as a subdirectory prefix**.

The project's source subdirectory can have subdirectories of its own, for example, to organize the code into components. Naturally, header inclusions will need to contain such subdirectories, for example `<libhello/core/hello.hxx>`. When the project's headers are installed (for example, into `/usr/include`), this subdirectory hierarchy is automatically recreated.

If you would like to separate public API headers/modules from implementation details, the convention is to place them into the `details/` subdirectory. For example:

```
libhello/
·-- libhello/
    |-- details/
    |   ·-- utility.hxx
    ·-- ...
```

If a project has truly private headers (for example, proprietary code) that must be clearly separated from public and implementation detail headers, then they can be placed into the `private/` subdirectory, next to `details/`. In a sense, this arrangement mimics the C++ public/protected/private member access.

It is recommended that you still install the implementation detail headers and modules for the reasons discussed above. If, however, you would like to disable their installation, you can add the following line to your source subdirectory `buildfile`:

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

If you are creating a *family of libraries* with a common name prefix, then it may make sense to use a nested source subdirectory layout with a common top-level directory. As an example, let's say we have the `libstud-path` and `libstud-url` libraries that belong to the same `libstud` family. Their source subdirectory layouts could look like this:

```
libstud-path/
·-- libstud/
    ·-- path/
        |-- path.hxx
        |-- path-io.hxx
        |-- ...
        ·-- buildfile

libstud-url/
·-- libstud/
    ·-- url/
        |-- url.hxx
        |-- url-io.hxx
        |-- ...
        ·-- buildfile
```

With the header inclusion paths adjusted accordingly:

```
#include <libstud/path/path.hxx>
#include <libstud/url/url.hxx>
```

The **`bdep-new(1)`** command provides the `subdir` project type sub-option that allows us to customize the source subdirectory within a project. For example:

```
$ bdep new -l c++ -t lib,subdir=libstud/path libstud-path
```

2.2 Source Naming

When naming source files, only use ASCII alphabetic characters, digits, as well as _ (underscore) and − (minus). Use `.` (dot) only for extensions, that is, trailing parts of the name that *classify* your files. Examples of good names:

```
SmallVector.hxx
small-vector.hxx
small_vector.hxx
small-vector.test.cxx
```

Examples of bad names:

```
small+vector.hxx
small.vector.hxx
```

If you are using _ or − as word separators in filesystem names, pick one and use it consistently throughout the project.

The C source file extensions are always `.h`/`.c`. The two alternative C++ source file extension schemes are `.?pp` and `.?xx`:

```
file        .?pp  .?xx

header      .hpp  .hxx
module      .mpp  .mxx
inline      .ipp  .ixx
template    .tpp  .txx
source      .cpp  .cxx
```

The `.mxx/.mpp` extension is for the module interface translation units with module implementation units (if any) using the `.cxx/.cpp` extension. If both are present, then it makes sense to use the same base name, similar to headers. For example:

```
hello-core.mxx
hello-core.cxx
```

The use of inline and template files is a matter of taste. If used, they are included at the end of the header/module files and contain definitions of inline and non-inline template functions, respectively. The `.?xx/.?pp` files with the same name (or, sometimes, name prefix) are assumed to be related and are collectively called a *module*. This term is meant to correspond directly to a C++ module.

By default the **`bdep-new(1)`** command uses the naming `.?xx` scheme. To use `.?pp` instead, pass `-t c++,cpp`.

There are several reasons not to "reuse" the `.h` C header extension for C++ files:

- There can be a need for both C and C++ headers for the same module.
- It allows tools to accurately determine the language from the file name.
- It is easier to search for C++ source code using wildcard patterns (`*.?pp`).

The last two reasons are also why headers without extensions are probably not worth the trouble.

Source files corresponding to C++ modules need to embed a sufficient amount of "module name tail" in their names to unambiguously resolve all the modules used in a project. When deriving file names from C++ module names, `.` (dot) should be replaced with either _ (underscore), - (minus), a case change, or a directory separator, according to your project's file naming scheme. For example, if our `libhello` had two modules, `hello.core` and `hello.extra`, then their interface units could be named as follows:

```
hello-core.mxx
hello-extra.mxx

hello_core.mxx
hello_extra.mxx

HelloCore.mxx
HelloExtra.mxx

hello/core.mxx
hello/extra.mxx

core.mxx
extra.mxx
```

As discussed in the next section, public module names should start with the project name and for such modules it is customary to omit this first component from file names (the last variant in the above example). See also Building Modules for a more detailed discussion of the module name to file name mapping.

2.3 Source Contents

Let's now move inside our source files. All macros defined by a project, such as include guards, version and symbol export macros, etc., must all start with the project name (including the `lib` prefix for libraries), for example `LIBHELLO_VERSION`. Similarly, the library's namespace and module names (both public and implementation detail) should all start with the library name but without the `lib` prefix. For example:

```
// libhello/hello.mxx

export module hello.core;

namespace hello
{
  ...
}
```

An executable project may use a namespace (in which case it is natural to name it after the project) and its (private) modules shouldn't be qualified with the project name (in order not to clash with similarly named modules from the corresponding library, if any). A library may also have private modules in which case they shouldn't be qualified either.

Hopefully by now the recommendation for the `lib` prefix should be easy to understand: oftentimes executables and libraries come in pairs, for example `hello` and `libhello`, with the reusable functionality being factored out from the executable into the library. It is natural to want to use the same name *stem* (`hello` in our case) for both.

The above naming scheme (with the `lib` prefix present in some names but not others) is carefully chosen to allow such library/executable pairs to coexist and be used together without too much friction. For example, both the library and executable can have a header called `utility.hxx` with the executable being able to include both and even get the "merged" functionality without extra effort (since they use the same namespace):

```
// hello/hello.cxx

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

namespace hello
{
  // Contains names from both utilities.
}
```

A canonical library project contains two special headers: `export.hxx` (or `export.hpp`) that defines the library's symbol exporting macro as well as `version.hxx` (or `version.hpp`) that defines the library's version macros (see `version` Module for details).

2.4 Tests

A project may have *unit* and/or *functional/integration* tests. Unit tests exercise each module's (potentially private) functionality in isolation. In contrast, functional/integration tests exercise the project via its public API, just like the real users of the project would.

A source file that implements a module's unit tests should be placed next to that module's files and be called with the module's name plus the `.test` second-level extension. It is expected to implement an executable (that is, define `main()`). If a module uses Testscript for unit testing, then the corresponding file should be called with the module's name plus the `.test.testscript` extension. For example:

```
libhello/
·-- libhello/
    |-- hello.hxx
    |-- hello.cxx
    |-- hello.test.cxx
    ·-- hello.test.testscript
```

All source files (that is, headers, modules, etc) with the `.test` second-level extension are assumed to belong to unit tests and are automatically excluded from the library/executable sources.

A library's functional/integration tests should go into the `tests/` subdirectory. Each such test should reside in a separate subdirectory, potentially organized into nested subdirectories (for instance, to correspond to the source subdirectory components). For example, if we were creating an XML parsing and serialization library, then our `tests/` could have the following layout:

```
tests/
|-- basics/
|   |-- driver.cxx
|   ·-- buildfile
|-- parser/
|   |-- pull/
|   |   |-- driver.cxx
|   |   ·-- buildfile
|   ·-- push/
|       |-- driver.cxx
|       ·-- buildfile
·-- serializer/
    ·-- ...
```

In the canonical library project created by `bdep-new` the `tests/` subdirectory is an unnamed subproject (in the build system terms). This allows us to build and run tests against an installed version of the library (see Testing for more information on the contents of this directory).

The `build2` CI implementation will automatically perform the installation test if a project contains the `tests/` subproject. See `bbot` Worker Logic for details.

By default executable projects do not have the `tests/` subprojects instead placing integration tests next to the source code (the `testscript` file; see The build2 Testscript Language for details). However, if desired, executable projects can have the `tests/` subproject, the same as libraries.

By default projects created by `bdep-new` include support for functional/integration testing but exclude support for unit testing. These defaults, however, can be overridden with `no-tests` and `unit-tests` options, respectively. For example:

```
$ bdep new -l c++ -t lib,unit-tests libhello
```

The rationale behind these defaults is that if a functionality can be tested through the public API, then we should generally prefer integration to unit testing. And in simple projects the entire functionality is often exposed through the public API. At the same time, support for unit testing adds extra complexity to the build infrastructure. Note also that it is fairly straightforward to add support for unit testing at a later stage. The relevant build logic is localized in the source subdirectory `buildfile` so you can simply generate a new project with unit tests enabled and copy over the relevant parts.

2.5 Build Output

There are no `bin/` or `obj/` subdirectories: build output (object files, libraries, executables, etc) go into a parallel directory structure (in case of an out of source build) or next to the sources (in case of an in source build). See Output Directories and Scopes for details on in and out of source builds.

Projects managed with **`bdep(1)`** are always built out of source. However, by default, the source directory is configured as *forwarded* to one of the out of source builds. This has two effects: we can run the build system driver **`b(1)`** directly in the source directory and certain "interesting" targets (such as executables, documentation, test results, etc) will be automatically *backlinked* to the source directory (see Configuration for details on forwarded configurations). The following listing illustrates this setup for our `hello` project (executables are marked with `*`):

```
                hello-gcc/
hello/    ~~>   `-- hello/
|-- build/   ~~>   |-- build/
`-- hello/   ~~>   `-- hello/
    |-- hello.cxx       |-- hello.o
    `-- hello     -->   `-- *hello
```

The result is an *as-if* in source build with all the benefits (such as having both source and relevant output in the same directory) but without any of the drawbacks (such as the inability to have multiple builds or source directory cluttered with object files).

The often cited motivation for placing executables into `bin/` is that in many build systems it is the only way to make things runnable in a reasonably cross-platform manner. The major drawback of this arrangement is the need for unique executable names which is especially constraining when writing tests where it is convenient to call the executable just `driver` or

`test`.

In `build2` there is no such restriction and all executables can run *in-place*. This is achieved with `rpath` which is emulated with DLL assemblies on Windows.