The `build2` Toolchain Introduction

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

Table of Contents

1 TL;DR

```
$ bpkg create -d hello cc
created new configuration in hello/

$ cd hello/
$ bpkg add https://build2.org/pkg/1/hello/stable
added repository build2.org/hello/stable

$ bpkg fetch
fetching build2.org/hello/stable
2 package(s) in 1 repository(s)

$ bpkg build hello
  build libhello/1.0.0 (required by hello)
  build hello/1.0.0
continue? [Y/n] y

libhello-1.0.0.tar.gz        100% of 2428  B  983 kBps 00m01s
fetched libhello/1.0.0
unpacked libhello/1.0.0

hello-1.0.0.tar.gz           100% of 1057  B 6882 kBps 00m01s
fetched hello/1.0.0
unpacked hello/1.0.0

configured libhello/1.0.0
configured hello/1.0.0

c++ hello-1.0.0/cxx{hello}
c++ libhello-1.0.0/hello/cxx{hello}
ld libhello-1.0.0/hello/libs{hello}
ld hello-1.0.0/exe{hello}

updated hello/1.0.0
```

2 Warning

The `build2` toolchain `0.X.Y` series are alpha releases. Interfaces *will* most likely change in backwards-incompatible ways. But if you want to start playing with it, welcome and join the mailing list!

Our approach to developing `build2` is to first get the hard parts right before focusing on completeness. So while we might still have no support for custom build rules, we do handle auto-generated source code (and, in particular, headers) properly. In other words, we go depth rather than breadth-first. As a result, there are some limitations and missing pieces, especially in the build system. The most notable ones are:

- Limited documentation.
- No support for custom build system rules/modules.

3 Introduction

The `build2` toolchain is a set of tools designed for building and packaging C and C++ code (though, if it can handle C++, it can handle anything, right?). The toolchain currently includes the *build system* (`build2`), the *package manager* (`bpkg`), and the *repository web interface* (`brep`). More tools, such as the *build robot* (`bbot`), are in the works. Then there is cppget.org (running `brep`) which we hope will become *the C++ package repository*.

The goal of this document is to give you a basic idea of what the `build2` toolchain can do so that you can decide if you are interested and want to learn more. Further documentation is referenced at the end of this introduction.

The `build2` toolchain is self-hosted and self-packaged (and, yes, it is on cppget.org). It could have served as its own example, however, before the toolchain can build itself, we have to bootstrap it (that chicken and egg problem again). And this step wouldn't serve our goal of quickly learning what `build2` is about. So, instead, we will start with a customary *"Hello, World!"* example which you won't yet be able to try yourself (but don't worry, complete terminal output will be shown). If at the end you find `build2` appealing, you can jump straight to The `build2` Toolchain Installation and Upgrade (and, yes, there you get to run that coveted `bpkg build bpkg`). Once the `build2` installation is complete, you can come back to the *"Hello, World!"* example and try all of the steps for yourself.

This introduction explores the *consumer* side of *"Hello, World!"*. That is, we assume that someone was kind enough to create and package the `libhello` library as well as the `hello` program and we will learn how to obtain and build them as well as keep up with their updates. At the end we will also see how to write our own, `hello2`, program that depends on `libhello`. And so, without further ado, let's begin.

Actually, one more thing: if you have a recent enough compiler and would like to try the new C++ Modules support, then you can instead use the modularized variants of these packages: simply replace `hello` with `mhello` and `libhello` with `libmhello` in the commands below.

The first step in using `bpkg` is to create a *configuration*. A configuration is a directory where packages that require similar compile settings will be built. You can create as many configurations as you want: for different C++ compilers, targets (`build2` is big on cross-compiling), debug/release, 32/64-bit, or even for different days of the week, if you are so inclined. Say we are in the mood for a GCC 5 release build today:

```
$ mkdir hello-gcc5-release
$ cd hello-gcc5-release
$ bpkg create cxx config.cxx=g++-5 config.cxx.coptions=-O3
created new configuration in /tmp/hello-gcc5-release/
```

Or perhaps you are on Windows and prefer Visual Studio (running from the Visual Studio Tools Command Prompt):

```
> mkdir hello-vc14-release
> cd hello-vc14-release
> bpkg create cxx config.cxx=cl config.cxx.coptions=/O2
created new configuration in C:\projects\hello-vc14-release\
```

One of the primary goals of the `build2` toolchain is to provide a uniform build interface across all the platforms and compilers. While the following examples use the `hello-gcc5-release` configuration and assume a UNIX-like operation system, everything will work if you use `hello-vc14-release` (or `hello-mingw-release`) on Windows. Just use appropriate paths, compilers, and options.

Let's discuss that last command line: `bpkg create` is the command for creating a new configuration. As a side note, if you ever want to get help for any `bpkg` command, run `bpkg help <command>`. To see the list of commands, run just `bpkg help` (or see **`bpkg(1)`**). While we are at it, if you ever want to see what `bpkg` is running underneath, there is the `-v` (essential commands) and `-V` (all commands) options. And if you really want to get under the hood, use `--verbose <level>`.

After the command we have `cxx` which is the name of the `build2` build system module. As you might have guessed, `cxx` provides support for the C++ compilation. By specifying this module when creating the configuration we configure it (yes, with those `config.cxx.*` variables that follow) for the entire configuration. That is, every package that we will build in this configuration and that uses the `cxx` module will by default inherit these settings.

The rest of the command line are the configuration variables for the `cxx` module with `coptions` standing for *compile options* (there are also `poptions` for *preprocess options*, `loptions` for *link options*, and `libs` for extra libraries to link).

There is also the `c` module for the C compilation. So if we were planning to build both C and C++ projects, then we could have run:

```
$ bpkg create c cxx ...
```

The problem, of course, is that you may not know what mix of languages those projects (or their dependencies) might use. For example, the use of C might be an implementation detail of a C++ library. To solve this, `build2` provides another module called `cc` which stands for *C-common*. So, in this context, instead of using the `c` and `cxx` modules directly, it's a good idea to get into the habit of using `cc`:

```
$ bpkg create cc config.cxx=g++-5 config.cc.coptions=-O3
```

Notice two things about this command line: we don't need to specify the C compiler with `config.c` – `build2` is smart enough to figure it out from `config.cxx` (or vice versa). We also used `config.cc.coptions` instead of `config.cxx.coptions` so that the options apply to all the C-common languages (we can still use `config.{c,cxx}.*` for the language-specific options).

Ok, configuration in hand, where can we get some packages? `bpkg` packages come from *repositories*. A repository can be a local filesystem directory or a remote URL. Our example packages come from their own remote *"Hello, World!"* repository: `https://build2.org/pkg/1/hello/stable/` (go ahead, browse it, I will wait).

Instead of scouring repository manifests by hand (I know you couldn't resist), we can ask `bpkg` to interrogate a repository location for us:

```
$ bpkg rep-info https://build2.org/pkg/1/hello/stable
warning: authenticity of the certificate for repository build2.org/hello/stable cannot be established
certificate is for build2.org, "Code Synthesis" <admin@build2.org>
certificate SHA256 fingerprint:
FF:DF:7D:38:67:4E:C3:82:[...]:30:56:B9:77:B9:F2:01:94
trust this certificate? [y/n]
```

The `bpkg` repositories are normally signed to prevent tampering with packages. If the repository certificate is seen (in this configuration) for the first time, `bpkg` will ask you to authenticate it. A good way to authenticate a certificate is to compare the displayed fingerprint to the one you have received earlier, for example, in an email announcement. The repository's about page also lists the fingerprint (see the about page for our repository). For more details on repository signing see the **`bpkg-repository-signing(1)`** help topic.

If we answer *yes*, we will see the basic repository information (its *canonical name*, location, certificate subject and fingerprint) followed by the list of available packages:

```
build2.org/hello/stable https://build2.org/pkg/1/hello/stable
CN=build2.org/O=Code Synthesis/admin@build2.org
FF:DF:7D:38:67:4E:C3:82:[...]:30:56:B9:77:B9:F2:01:94

hello/1.0.0
libhello/1.0.0
```

We can also use the repository's web interface (implemented by `brep`). Our repository has one, check it out: `https://build2.org/pkg/hello/`.

Ok, back to the command line. If we want to use a repository as a source of packages in our configuration, we have to first add it:

```
$ bpkg add https://build2.org/pkg/1/hello/stable
added repository build2.org/hello/stable
```

If we want to add several repositories, we just execute the `bpkg add` command for each of them. Once this is done, we fetch the list of available packages for all the added repositories:

```
$ bpkg fetch
fetching build2.org/hello/stable
2 package(s) in 1 repository(s)
```

Note that you would normally re-run the `bpkg fetch` command after you've added another repository or to refresh the list of available packages.

Now that `bpkg` knows where to get the packages, we can finally get down to business:

```
$ bpkg build hello
  build libhello/1.0.0 (required by hello)
  build hello/1.0.0
continue? [Y/n]
```

Let's see what's going on here. We ran `bpkg build` to build the `hello` program which happens to depend on the `libhello` library. So `bpkg` presents us with a *plan of action*, that is, the steps it will have to perform in order to build us `hello` and then asks us to confirm if that's what we want to do (you can add `--yes|-y` to skip the confirmation). In the real-world usage the plan will be more complex, with upgrades/downgrades, reconfigurations, etc.

Let's answer *yes* and see what happens:

```
libhello-1.0.0.tar.gz        100% of 2428  B 1364 kBps 00m01s
fetched libhello/1.0.0
unpacked libhello/1.0.0
hello-1.0.0.tar.gz           100% of 1057  B   20 MBps 00m01s
fetched hello/1.0.0
unpacked hello/1.0.0
configured libhello/1.0.0
configured hello/1.0.0
c++ hello-1.0.0/cxx{hello}
c++ libhello-1.0.0/hello/cxx{hello}
ld libhello-1.0.0/hello/libs{hello}
ld hello-1.0.0/exe{hello}
updated hello/1.0.0
```

While the output is mostly self-explanatory, in short, `bpkg` downloaded, unpacked, and configured both packages and then proceeded to building the `hello` executable which happens to require building the `libhello` library. Note that the download progress may look differently on your machine depending on which *fetch tool* (`wget`, `curl`, or `fetch`) is used. If you ever considered giving that `-v` option a try, now would be a good time. But let's first drop (`bpkg drop`) the `hello` package so that we get the same build from scratch:

```
$ bpkg drop hello
following prerequisite packages were automatically built and will no longer be necessary:
  libhello
drop prerequisite packages? [Y/n] y
  drop hello
  drop libhello
continue? [Y/n] y
disfigured hello
disfigured libhello
purged hello
purged libhello
```

Ok, ready for some `-v` details? Feel free to skip the following listing if you are not interested.

```
$ bpkg build -v -y hello
fetching libhello-1.0.0.tar.gz from build2.org/hello/stable
curl ... https://build2.org/pkg/1/hello/stable/libhello-1.0.0.tar.gz
  % Total    % Received Average Speed   Time    Time     Time  Current
                        Dload  Upload   Total   Spent    Left  Speed
100  2428  100  2428      1121      0  0:00:01  0:00:01 --:--:--  1122
fetched libhello/1.0.0
```

```
tar -xf libhello-1.0.0.tar.gz
unpacked libhello/1.0.0
fetching hello-1.0.0.tar.gz from build2.org/hello/stable
curl ... https://build2.org/pkg/1/hello/stable/hello-1.0.0.tar.gz
  % Total    % Received Average Speed   Time    Time     Time  Current
                        Dload  Upload   Total   Spent    Left  Speed
100  1057  100  1057      773      0  0:00:01  0:00:01 --:--:--   772
fetched hello/1.0.0
tar -xf hello-1.0.0.tar.gz
unpacked hello/1.0.0
b -v configure(./libhello-1.0.0/)
cat >libhello-1.0.0/build/config.build
configured libhello/1.0.0
b -v configure(./hello-1.0.0/)
cat >hello-1.0.0/build/config.build
configured hello/1.0.0
hold package hello
b -v update(./hello-1.0.0/)
g++-5 -I libhello-1.0.0 -O3 -std=c++11 -o hello-1.0.0/hello.o -c hello-1.0.0/hello.cxx
g++-5 -I libhello-1.0.0 -O3 -std=c++11 -fPIC -o libhello-1.0.0/hello/hello.so.o -c libhello-1.0.0/hello/hello.cxx
g++-5 -O3 -std=c++11 -shared -o libhello-1.0.0/hello/libhello-1.0.so libhello-1.0.0/hello/hello.so.o
g++-5 -O3 -std=c++11 -o hello-1.0.0/hello hello-1.0.0/hello.o libhello-1.0.0/hello/libhello-1.0.so
updated hello/1.0.0
```

Another handy command is `bpkg status`. It can be used to examine the state of a package in the configuration. Here are a few examples (if you absolutely must know what `hold_package` and `sys:?` mean, check **`bpkg-pkg-status(1)`**):

```
$ bpkg status libhello
configured 1.0.0; available sys:?

$ bpkg status hello
configured 1.0.0 hold_package; available sys:?

$ bpkg drop -y hello
disfigured hello
disfigured libhello
purged hello
purged libhello

$ bpkg status hello
available 1.0.0 sys:?

$ bpkg status libfoobar
unknown
```

Let's say we got wind of a new development: the `libhello` author released a new version of the library. It is such an advance in the art of *"Hello, World!"*, it's only currently available from `testing`. Of course, we must check it out.

Now, what exactly is `testing`? You must have noticed that the repository location that we've been using so far ended with `/stable`. Quite often it is useful to split our repository into sub-repositories or *sections*. For example, to reflect the maturity of packages (say, `stable` and `testing`, as in our case) or to divide them into sub-categories (`misc` and `math`) or even some combination (`math/testing`). Note, however, that to `bpkg` these sub-repositories or *sections* are just normal repositories and there is nothing special about them.

We are impatient to try the new version so we will skip interrogating the repository with `rep-info` and just add it to our configuration. After all, we can always check with `status` if any upgrades are available for packages we are interested in. Here we assume the configura-

tion has `hello` built (run `bpkg build -y hello` to get to that state).

```
$ bpkg add https://build2.org/pkg/1/hello/testing
added repository build2.org/hello/testing

$ bpkg fetch
fetching build2.org/hello/stable
fetching build2.org/hello/testing
5 package(s) in 2 repository(s)
```

Notice that this time we don't see any authentication-related messages or prompts since `bpkg` remembered (in this configuration) that we trust the certificate (`testing` naturally uses the same one as `stable`).

Let's see what's new:

```
$ bpkg status libhello
configured 1.0.0; available 1.1.0 sys:?
```

Ok, `libhello/1.1.0` is now available. How do we upgrade? We can try to build `hello` again:

```
$ bpkg build -y hello
info: dir{hello-1.0.0/} is up to date
updated hello/1.0.0
```

Why did nothing happenned? Because `bpkg` will only upgrade (or downgrade) to a new version if we explicitly ask it to. As things stand, all dependencies for `hello` are satisfied and `bpkg` is happy to twiddle its thumbs. Let's tell `bpkg` to build us `libhello` instead:

```
$ bpkg build libhello
  build libformat/1.0.0 (required by libhello)
  build libprint/1.0.0 (required by libhello)
  upgrade libhello/1.1.0
  reconfigure hello (dependent of libhello)
continue? [Y/n]
```

Ok, now we are getting somewhere. It looks like the new version of `libhello` went really enterprise-grade (or is it called web-scale these days?). There are now two new dependencies (`libformat` and `libprint`) that we will have to build in order to upgrade. Maybe we should answer *no* here?

Notice also that `reconfigure hello` line. If you think about this, it makes sense: we are getting a new version of `libhello` and `hello` depends on it so it might need a chance to make some adjustments to its configuration.

Let's answer *yes* if only to see what happens:

```
update dependent packages? [Y/n]
```

Another question. This one has to do with that `reconfigure hello` line we just talked about. If you were wondering why we were only offered to reconfigure and not actually update the dependent package, you should know that `bpkg` is a very lazy package manager, it

only does what it must do, not what might be nice to do. It must reconfigure but it doesn't really have to update. And this could be a good thing if, for example, you have a hundred dependents in your configuration but right now you only want to build just those specific packages. However, quite often, you do want to keep all the packages in your configuration up to date and `bpkg` graciously offers to take care of this task. Ok, let's answer *yes* again:

```
...
update dependent packages? [Y/n] y
disfigured hello/1.0.0
disfigured libhello/1.0.0
libformat-1.0.0.tar.gz       100% of 1064  B   11 MBps 00m01s
fetched libformat/1.0.0
unpacked libformat/1.0.0
libprint-1.0.0.tar.gz        100% of 1040  B    9 MBps 00m01s
fetched libprint/1.0.0
unpacked libprint/1.0.0
libhello-1.1.0.tar.gz        100% of 1564  B 4672 kBps 00m01s
fetched libhello/1.1.0
unpacked libhello/1.1.0
configured libformat/1.0.0
configured libprint/1.0.0
configured libhello/1.1.0
configured hello/1.0.0
c++ libhello-1.1.0/hello/cxx{hello}
c++ libformat-1.0.0/format/cxx{format}
ld libformat-1.0.0/format/liba{format}
c++ libprint-1.0.0/print/cxx{print}
ld libprint-1.0.0/print/liba{print}
ld libhello-1.1.0/hello/liba{hello}
c++ libhello-1.1.0/hello/cxx{hello}
c++ libformat-1.0.0/format/cxx{format}
ld libformat-1.0.0/format/libs{format}
c++ libprint-1.0.0/print/cxx{print}
ld libprint-1.0.0/print/libs{print}
ld libhello-1.1.0/hello/libs{hello}
c++ libhello-1.1.0/tests/test/cxx{driver}
ld libhello-1.1.0/tests/test/exe{driver}
c++ hello-1.0.0/cxx{hello}
ld hello-1.0.0/exe{hello}
updated libhello/1.1.0
updated hello/1.0.0
```

A lot of output but nothing really new. If you were to answer *no* to the "update dependent packages?" question above, it is easy to make sure a package is up-to-date at a later time with the `bpkg update` command (there is also `bpkg clean`), for example:

```
$ bpkg clean hello
rm hello-1.0.0/exe{hello}
rm hello-1.0.0/obje{hello}
cleaned hello/1.0.0

$ bpkg update hello
c++ hello-1.0.0/cxx{hello.cxx}
ld hello-1.0.0/exe{hello}
updated hello/1.0.0
```

Let's say we really don't like the direction `libhello` is going and would rather stick to version `1.0.0`. Just like upgrades, downgrades are explicit plus, in this case, we need to specify the version (you can also specify the desired version for upgrades).

```
$ bpkg build libhello/1.0.0
  downgrade libhello/1.0.0
  reconfigure hello (dependent of libhello)
continue? [Y/n] y
update dependent packages? [Y/n] y
disfigured hello/1.0.0
disfigured libhello/1.1.0
libhello-1.0.0.tar.gz        100% of 2428  B  983 kBps 00m01s
fetched libhello/1.0.0
unpacked libhello/1.0.0
configured libhello/1.0.0
configured hello/1.0.0
following prerequisite packages were automatically built and will no longer be necessary:
  libprint
  libformat
drop prerequisite packages? [Y/n] y
disfigured libprint
disfigured libformat
purged libprint
purged libformat
c++ libhello-1.0.0/hello/cxx{hello}
ld libhello-1.0.0/hello/liba{hello}
c++ libhello-1.0.0/hello/cxx{hello}
ld libhello-1.0.0/hello/libs{hello}
c++ libhello-1.0.0/tests/test/cxx{driver}
ld libhello-1.0.0/tests/test/exe{driver}
c++ hello-1.0.0/cxx{hello}
ld hello-1.0.0/exe{hello}
updated libhello/1.0.0
updated hello/1.0.0
```

Notice how `bpkg` helpfully offered to get rid of `libprint` and `libformat` which we won't be needing anymore. Note also that while we can use `--yes|y` as an answer to all the numerous prompts, there are also more granular options. For example, this is how we can instruct `bpkg` to drop prerequisites (`--drop-prerequisite|-D`) but leave dependents just reconfigured (`--leave-dependent|-L`):

```
$ bpkg build -D -L libhello/1.0.0
```

Ok, so all this might look nice and all, but we haven't actually seen anything of what we've presumably built; it can all be a charade, for all we know. Can we see some libraries and run the `hello` program?

There are several ways we can do this. If the package provides tests (as all good packages should), we can run them with the `bpkg test` command:

```
$ bpkg test libhello hello
test libhello-1.0.0/tests/test/exe{driver}
test hello-1.0.0/exe{hello}
tested libhello/1.0.0
tested hello/1.0.0
```

But that doesn't quite count for seeing libraries and running programs. Well, if you insist, let's see what's inside `hello-gcc5-release/`. The `bpkg` configuration (this `hello-gcc5-release/` directory) is, in the `build2` build system terms, an *amalgamation* – a project that contains *subprojects*. Not surprisingly, the subprojects in this amalgamation are the packages that we've built:

```
$ ls -1F
build/
hello-1.0.0/
libhello-1.0.0/
buildfile
hello-1.0.0.tar.gz
libhello-1.0.0.tar.gz
```

And if we look inside `hello-1.0.0/` we will see what looks like the `hello` program:

```
$ ls -1F hello-1.0.0/
build/
buildfile
hello*
hello.d
hello.cxx
hello.o
hello.o.d
manifest
test.out
version

$ hello-1.0.0/hello
usage: hello <name>...

$ hello-1.0.0/hello World
Hello, World!
```

The important point here is this: the `bpkg` configuration is not some black box that you should never look inside of. On the contrary, it is a normal and predictable concept of the build system and as long as you understand what you are doing, feel free to muck around.

Another way to get hold of a package's goodies is to install it with `bpkg install`. Let's try that:

```
$ bpkg install                    \
  config.install.root=/opt/hello \
  config.install.sudo=sudo       \
  hello

install /opt/hello/
install /opt/hello/include/
install /opt/hello/include/hello/
install libhello-1.0.0/hello/hxx{hello}
install libhello-1.0.0/hello/hxx{export}
install /opt/hello/lib/
install libhello-1.0.0/hello/libs{hello}
install /opt/hello/bin/
install hello-1.0.0/exe{hello}
install /opt/hello/share/
```

```
install /opt/hello/share/doc/
install /opt/hello/share/doc/hello/
install hello-1.0.0/doc{version}
installed hello/1.0.0
```

The `config.install.sudo` value is the optional *sudo*-like program that should be used to run the `install` program. For those feeling queasy running `sudo make install`, here is your answer. If you are wondering whether you could have specified those `config.install.*` values during the configuration creation, the answer is yes, indeed!

Let's see what we've got:

```
$ tree -F /opt/hello/
/opt/hello/
âãã bin/
â   âãã hello*
âãã include/
â   âãã libhello/
â       âãã export
â       âãã hello
âãã lib/
â   âãã libhello-1.0.so*
â   âãã libhello.so -> libhello-1.0.so*
âãã share/
    âãã doc/
        âãã hello/
            âãã version
```

We can also try to run the installed program:

```
$ /opt/hello/bin/hello World
/opt/hello/bin/hello: error while loading shared libraries: libhello-1.0.so: cannot open shared object file: No such file or directory
```

Not what we hoped to see. Note to the Windows users: this will actually work since `hello-1.0.dll` will be installed into `bin\`, next to the executable; for once things are working better on Windows.

The problem is with our installation location: the runtime linker won't look for `libhello-1.0.so` in `/opt/hello/lib` unless we somehow tell it to (for example, using `LD_LIBRARY_PATH` or equivalent). There are several ways we can resolve this. We could give up on shared libraries and link our prerequisite libraries statically (`config.bin.exe.lib=static`). Or we could use the *rpath* mechanism:

```
$ bpkg install                     \
  config.install.root=/opt/hello   \
  config.install.sudo=sudo         \
  config.bin.rpath=/opt/hello/lib \
  hello

ld hello-1.0.0/exe{hello}
install /opt/hello/
install /opt/hello/include/
install /opt/hello/include/hello/
install libhello-1.0.0/hello/hxx{hello}
install libhello-1.0.0/hello/hxx{export}
install /opt/hello/lib/
install libhello-1.0.0/hello/libs{hello}
```

```
install /opt/hello/bin/
install hello-1.0.0/exe{hello}
install /opt/hello/share/
install /opt/hello/share/doc/
install /opt/hello/share/doc/hello/
install hello-1.0.0/doc{version}
installed hello/1.0.0

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

Notice that `ld` line above – this is where our executable is re-linked with the `-rpath` option.

We can also uninstall what we have installed with `bpkg uninstall`:

```
$ bpkg uninstall                  \
  config.install.root=/opt/hello  \
  config.install.sudo=sudo        \
  hello

uninstall hello-1.0.0/doc{version}
uninstall /opt/hello/share/doc/hello/
uninstall /opt/hello/share/doc/
uninstall /opt/hello/share/
uninstall hello-1.0.0/exe{hello}
uninstall /opt/hello/bin/
uninstall libhello-1.0.0/hello/libs{hello}
uninstall /opt/hello/lib/
uninstall libhello-1.0.0/hello/hxx{export}
uninstall libhello-1.0.0/hello/hxx{hello}
uninstall /opt/hello/include/hello/
uninstall /opt/hello/include/
uninstall /opt/hello/
uninstalled hello/1.0.0

$ ls /opt/hello
ls: cannot access /opt/hello: No such file or directory
```

What if we wanted to use `libhello` in our own project? While installing it is always an option, this may not be convenient when we develop our code. We may have multiple builds per project, for example, with GCC and Clang to catch all the warnings. We may also want to make sure our application works well with several versions of `libhello` (and maybe even with that heinous `1.1.X`). While we can install different configurations into different directories, it's hard to deny things are getting a bit hairy: multiple configurations, multiple installations... I guess we will have to get our hands into that cookie jar, I mean, configuration, again.

In fact, let's just start writing our own version of the `hello` program and see how it goes:

```
$ mkdir hello2
$ cd hello2

$ cat >hello.cpp

#include <libhello/hello>

int main ()
{
  hello::say ("World");
}
```

What build system shall we use? I can't believe you are even asking this question...

```
$ mkdir build

$ cat >build/bootstrap.build

project = hello2              # project name
using config                  # config module (those config.*)

$ cat >build/root.build

cxx.std = 11                  # C++ standard
using cxx                     # C++ module
cxx{*}: extension = cpp       # C++ source file extension

$ cat >buildfile

import libs = libhello%lib{hello}
exe{hello}: cxx{hello} $libs
```

While some of this might not be crystal clear (like why do we have `bootstrap.build` *and* `root.build`), I am sure you at least have a fuzzy idea of what's going on. And that's enough for what we are after here. Completely explaining what's going on here and, more importantly, *why* it's going this way is for another time and place (the `build2` build system manual).

To recap, these are the contents of our project so far:

```
$ tree -F
.
âãâ build/
â   âãâ bootstrap.build
â   âãâ root.build
âãâ buildfile
âãâ hello.cpp
```

Let's try to build it and see what happens – maybe it will magically work (**`b(1)`** is the `build2` build system driver).

```
$ b config.cxx=g++-5
error: unable to import target libhello%lib{hello}
  info: use config.import.libhello command line variable to specifying its project out_root
  info: while applying rule cxx.link to update exe{hello}
  info: while applying rule alias to update dir{./}
```

No magic, unfortunately (or fortunately). But we got a hint: looks like we need to tell `build2` where `libhello` is using `config.import.libhello`. Without fretting too much about what exactly `out_root` means, let's point `build2` to our `bpkg` configuration and see what happens. After all, that's where, more or less, our *out*-put for `libhello` is.

```
$ b config.cxx=g++-5 \
    config.import.libhello=/tmp/hello-gcc5-release
c++ cxx{hello}
ld exe{hello}
```

Almost magic. Let's see what we've got:

```
$ tree -F
.
âãâ build/
â   âãâ bootstrap.build
â   âãâ root.build
âãâ buildfile
âãâ hello*
âãâ hello.d
âãâ hello.cpp
âãâ hello.o
âãâ hello.o.d

$ ./hello
Hello, World!
```

Let's change something in our source code and try to update:

```
$ touch hello.cpp

$ b
error: unable to import target libhello%lib{hello}
  info: use config.import.libhello command line variable to specifying its project out_root
  info: while applying rule cxx.link to update exe{hello}
  info: while applying rule alias to update dir{./}
```

Looks like we have to keep repeating those `config.*` values and who wants that? To get rid of this annoyance we have to make our configuration *permanent*. Also, seeing that we plan to have several of them (GCC/Clang, different version of `libhello`), it makes sense to create them *out of source tree*. Let's get to it:

```
$ cd ..
$ mkdir hello2-gcc5-release
$ ls -1F
hello2/
hello2-gcc5-release/

$ b config.cxx=g++-5                              \
  config.cc.coptions=-O3                          \
  config.import.libhello=/tmp/hello-gcc5-release \
  'configure(hello2/@hello2-gcc5-release/)'

mkdir -p hello2-gcc5-release/build/
save hello2-gcc5-release/build/config.build
```

Translated, `configure(hello2/@hello2-gcc5-release/)` means *"configure the `hello2/` source directory in the `hello2-gcc5-release/` output directory"*. In `build2` this *source directory* is called `src_root` and *output directory* – `out_root`. Hm, we've already heard `out_root` mentioned somewhere before...

Once the configuration is saved, we can develop our project without any annoyance:

```
$ b hello2-gcc5-release/
c++ hello2/cxx{hello}
ld hello2-gcc5-release/exe{hello}

$ cd hello2-gcc5-release/

$ b
info: dir{./} is up to date

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

$ b -v
g++-5 -I/tmp/hello-gcc5-release/libhello-1.0.0 -O3 -std=c++11 -o hello.o -c ../hello2/hello.cpp
g++-5 -O3 -std=c++11 -o hello hello.o /tmp/hello-gcc5-release/libhello-1.0.0/hello/libhello-1.0.so
```

Some of you might have noticed that `hello2-gcc5-release/` and `/tmp/hello-gcc5-release/` look awfully similar and are now wondering if we could instead build `hello2` *inside* `/tmp/hello-gcc5-release/`? I am glad you've asked. In fact, we can just do:

```
$ cd ..
$ ls -1F
hello2/
hello2-gcc5-release/

$ b 'configure(hello2/@/tmp/hello-gcc5-release/hello2/)'
mkdir -p /tmp/hello-gcc5-release/hello2/build/
save /tmp/hello-gcc5-release/hello2/build/config.build

$ b /tmp/hello-gcc5-release/hello2/
c++ hello2/cxx{hello}@/tmp/hello-gcc5-release/hello2/
ld /tmp/hello-gcc5-release/hello2/exe{hello}
```

Now that might seem like magic, but it's actually pretty logical. Why don't we need to specify any of the `config.c*` values this time? Because they are inherited from those specified for `/tmp/hello-gcc5-release` when we created the configuration with `bpkg create`. What about `config.import.libhello`, don't we need at least that? Not really – `libhello` will be found automatically since it is part of the same amalgamation.

Of course, `bpkg` has no idea `hello2` is now part of its configuration:

```
$ bpkg status -d /tmp/hello-gcc5-release/ hello2
unknown
```

This is what I meant when I said you can muck around in `bpkg`'s back yard as long as you understand the implications.

But is there a way to make `bpkg` aware of our little project? You seem to really have all the right questions today. Actually, there is a very good reason why we would want that: if we upgrade `libhello` we would want `bpkg` to automatically reconfigure our project. As it is now, we will have to remember and do it ourselves.

The only way to make `bpkg` aware of `hello2` is to turn it from merely a `build2` *project* into a `build2` *package*. While the topic of packaging is also for another time and place (the `build2` package manager manual), we can get away with something as simple as this:

```
$ cat >hello2/manifest
: 1
name: hello2
version: 1.0.0
summary: Improved "Hello World" program
license: proprietary
url: http://example.org/hello2
email: hello2@example.org
depends: libhello >= 1.0.0
```

For our purposes, the only really important value in this manifest is `depends` since it tells `bpkg` which package(s) we need. Let's give it a try. But first we will clean up our previous attempt at building `hello2` inside `/tmp/hello-gcc5-release/`:

```
$ b '{clean disfigure}(/tmp/hello-gcc5-release/hello2/)'
rm /tmp/hello-gcc5-release/hello2/exe{hello}
rm /tmp/hello-gcc5-release/hello2/obje{hello}
rm /tmp/hello-gcc5-release/hello2/build/config.build
rmdir /tmp/hello-gcc5-release/hello2/
```

Next, we use the `bpkg build` command but instead of giving it a package name like we did before, we will point it to our `hello2` package directory (`bpkg` can fetch packages or it can build local package archives or package directories):

```
$ bpkg build -d /tmp/hello-gcc5-release/ ./hello2/
  build hello2/1.0.0
continue? [Y/n] y
unpacked hello2/1.0.0
configured hello2/1.0.0
c++ hello2/cxx{hello}@/tmp/hello-gcc5-release/hello2-1.0.0/
ld /tmp/hello-gcc5-release/hello2-1.0.0/exe{hello}
updated hello2/1.0.0
```

Let's upgrade `libhello` and see what happens:

```
$ bpkg build -d /tmp/hello-gcc5-release/ -L libhello
  build libformat/1.0.0 (required by libhello)
  build libprint/1.0.0 (required by libhello)
  upgrade libhello/1.1.0
  reconfigure hello2 (dependent of libhello)
continue? [Y/n] y
disfigured hello2/1.0.0
disfigured libhello/1.0.0
[ ... fetching & unpacking ... ]
configured libformat/1.0.0
configured libprint/1.0.0
```

```
configured libhello/1.1.0
configured hello2/1.0.0
[ ... updating libprint, libformat, and libhello ... ]
updated libhello/1.1.0
```

As promised, `hello2` got reconfigured (it didn't get updated because of the `-L` option). We can now update it and give it a try:

```
$ bpkg update -d /tmp/hello-gcc5-release/ hello2
c++ hello2/cxx{hello}@/tmp/hello-gcc5-release/hello2-1.0.0/
ld /tmp/hello-gcc5-release/hello2-1.0.0/exe{hello}
updated hello2/1.0.0

$ /tmp/hello-gcc5-release/hello2-1.0.0/hello
Hello, World!
```

To finish off, let's see how hard it will be to get a Clang build going:

```
$ cd /tmp
$ mkdir hello-clang36-release
$ cd hello-clang36-release

$ bpkg create cc config.cxx=clang++-3.6 config.cc.coptions=-O3
created new configuration in /tmp/hello-clang36-release/

$ bpkg add https://build2.org/pkg/1/hello/testing
added repository build2.org/hello/testing

$ bpkg fetch
fetching build2.org/hello/testing
[... certificate authentication ...]
fetching build2.org/hello/stable (complements build2.org/hello/testing)
5 package(s) in 2 repository(s)

$ bpkg build libhello/1.0.0 path/to/hello2/
  build libhello/1.0.0
  build hello2/1.0.0
continue? [Y/n] y
libhello-1.0.0.tar.gz        100% of 2428  B  983 kBps 00m01s
fetched libhello/1.0.0
unpacked libhello/1.0.0
unpacked hello2/1.0.0
configured libhello/1.0.0
configured hello2/1.0.0
c++ libhello-1.0.0/hello/cxx{hello}
ld libhello-1.0.0/hello/liba{hello}
c++ libhello-1.0.0/hello/cxx{hello}
ld libhello-1.0.0/hello/libs{hello}
c++ libhello-1.0.0/tests/test/cxx{driver}
ld libhello-1.0.0/tests/test/exe{driver}
c++ /path/to/hello2/cxx{hello}@hello2-1.0.0/
ld hello2-1.0.0/exe{hello}
updated libhello/1.0.0
updated hello2/1.0.0
```

Are you still there? Ok, one last example. Let's see how hard it is to cross-compile.

```
$ mkdir hello-mingw64
$ cd hello-mingw64

$ bpkg create cc config.cxx=x86_64-w64-mingw32-g++
created new configuration in /tmp/hello-mingw64/

$ bpkg add https://build2.org/pkg/1/hello/stable
added repository build2.org/hello/stable

$ bpkg fetch
fetching build2.org/hello/stable
[... certificate authentication ...]
2 package(s) in 1 repository(s)

$ bpkg build -y hello
libhello-1.0.0.tar.gz        100% of 2428  B  983 kBps 00m01s
fetched libhello/1.0.0
unpacked libhello/1.0.0
hello-1.0.0.tar.gz           100% of 1057  B 6882 kBps 00m01s
fetched hello/1.0.0
unpacked hello/1.0.0
configured libhello/1.0.0
configured hello/1.0.0
c++ hello-1.0.0/cxx{hello}
c++ libhello-1.0.0/hello/cxx{hello}
ld libhello-1.0.0/hello/libs{hello}
ld hello-1.0.0/exe{hello}
updated hello/1.0.0

$ wine hello-1.0.0/hello.exe Windows
Hello, Windows!
```

In fact, on a properly setup GNU/Linux machine (that automatically uses `wine` as an `.exe` interpreter) we can even run tests:

```
$ bpkg test libhello hello
c++ libhello-1.0.0/tests/test/cxx{driver}
ld libhello-1.0.0/tests/test/exe{driver}
test libhello-1.0.0/tests/test/exe{driver}
test hello-1.0.0/exe{hello}
tested libhello/1.0.0
tested hello/1.0.0
```