The `build2` Toolchain Introduction

Revision `0.2`, January 2016
This revision of the document describes the `build2` toolchain `0.2.x` series.

TL;DR

```
$ bpkg create -d hello cxx
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+1 (required by hello)
build hello 1.0.0
continue? [Y/n] y
libhello-1.0.0+1.tar.gz      100% of 1489  B  983 kBps 00m01s
fetched libhello 1.0.0+1
unpacked libhello 1.0.0+1
hello-1.0.0.tar.gz           100% of 1030  B 6882 kBps 00m01s
fetched hello 1.0.0
unpacked hello 1.0.0
configured libhello 1.0.0+1
configured hello 1.0.0
c++ hello-1.0.0/cxx{hello}
c++ libhello-1.0.0+1/hello/cxx{hello}
ld libhello-1.0.0+1/hello/libso{hello}
ld hello-1.0.0/exe{hello}
updated hello 1.0.0
```

Warning

The `build2` toolchain `0.x.y` series are alpha releases. Interfaces *will* change in backwards-incompatible ways, guaranteed. Currently, it is more of a technology preview rather than anything final. 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 be extracting header dependencies on every run (no caching yet) they do play well with auto-generated source code. In other words, we go depth rather than breadth-first. As a result, there are plenty of limitations and missing pieces, especially in the build system. The most notable ones are:

- Limited documentation.
- No C compiler rules.
- No support for Windows/VC++.
- No support for parallel builds.
- No support for custom build system rules/modules.

Introduction

The `build2` toolchain is a set of tools designed for building and packaging 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 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 then serve as its own example. However, before the toolchain can unpack and 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, the following section jumps right into bootstrapping and installation (and, yes, you get to run that coveted `bpkg update 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 and the `hello` program and we will learn how to obtain and build them as well as keep up with their updates. And so, without further ado, let's begin.

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, 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/
```

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 a 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` option. 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 during the configuration creation 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 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).

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
build2.org/hello/stable https://build2.org/pkg/1/hello/stable
hello 1.0.0
libhello 1.0.0+1
```

Or we could 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 `brep 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)
```

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 packages, we can finally get down to business:

```
$ bpkg build hello
build libhello 1.0.0+1 (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 that's what we want to do (you can add `--yes|-y` to skip the confirmation). Let's answer *yes* and see what happens:

```
...
continue? [Y/n] y
libhello-1.0.0+1.tar.gz      100% of 1489  B 1364 kBps 00m01s
fetched libhello 1.0.0+1
unpacked libhello 1.0.0+1
hello-1.0.0.tar.gz           100% of 1030  B   20 MBps 00m01s
fetched hello 1.0.0
unpacked hello 1.0.0
configured libhello 1.0.0+1
configured hello 1.0.0
c++ hello-1.0.0/cxx{hello}
c++ libhello-1.0.0+1/hello/cxx{hello}
ld libhello-1.0.0+1/hello/libso{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 of 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 not interested.

```
$ bpkg build -v -y hello
fetching libhello-1.0.0+1.tar.gz from build2.org/hello/stable
curl ... https://build2.org/pkg/1/hello/stable/libhello-1.0.0+1.tar.gz
  % Total    % Received Average Speed   Time    Time     Time  Current
                        Dload  Upload   Total   Spent    Left  Speed
100  1489  100  1489      1121      0  0:00:01  0:00:01 --:--:--  1122
fetched libhello 1.0.0+1
tar -xf libhello-1.0.0+1.tar.gz
unpacked libhello 1.0.0+1
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  1030  100  1030      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+1/)
config::save libhello-1.0.0+1/build/config.build
configured libhello 1.0.0+1
b -v configure(./hello-1.0.0/)
config::save hello-1.0.0/build/config.build
configured hello 1.0.0
hold package hello
b -v update(./hello-1.0.0/)
g++-5 -Ilibhello-1.0.0+1 -O3 -std=c++11 -o hello-1.0.0/hello.o -c hello-1.0.0/hello.cxx
g++-5 -Ilibhello-1.0.0+1 -O3 -std=c++11 -fPIC -o libhello-1.0.0+1/hello/hello-so.o -c libhello-1.0.0+1/hello/hello.cxx
g++-5 -O3 -std=c++11 -shared -o libhello-1.0.0+1/hello/libhello.so
g++-5 -O3 -std=c++11 -o hello-1.0.0/hello hello-1.0.0/hello.o libhello-1.0.0+1/hello/libhello.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` means, check the command's documentation):

```
$ bpkg status libhello
configured 1.0.0+1

$ bpkg status hello
configured 1.0.0 hold_package

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

$ bpkg status hello
available 1.0.0

$ 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 state of the *"Hello, World!"* art, 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 configuration 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)

$ bpkg status libhello
configured 1.0.0+1; available 1.1.0
```

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
```

Nothing happens. That's because `bpkg` will only upgrade (or downgrade) to a new version if we explicitly ask it to. As it is now, 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 (required by 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:

```
...
continue? [Y/n] y
disfigured hello 1.0.0
disfigured libhello 1.0.0+1
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/libso{format}
c++ libprint-1.0.0/print/cxx{print}
ld libprint-1.0.0/print/libso{print}
```

```
ld libhello-1.1.0/hello/libso{hello}
c++ libhello-1.1.0/tests/test/cxx{driver}
ld libhello-1.1.0/tests/test/exe{driver}
updated libhello 1.1.0
```

If you paid really close attention, you might have noticed something surprising: the `hello` package wasn't *updated*. Yes, it was reconfigured, but we didn't see any compile or link commands for this project. In fact, `hello` is now pretty *out-of-date*.

While it may sound surprising, `bpkg` doesn't try to keep your packages *up-to-date*. Configured – yes, but not up-to-date. Trying to guarantee up-to-date-ness of packages is in the end futile. For example, if you upgrade your compiler or system headers, `bpkg` has no way of realizing that some packages are now out-of-date. Only the build system, which has the complete information about all the dependencies, can make such a realization (and correct it).

But it is easy to make sure a package is up-to-date at any given time with the `bpkg update` command (there is also `bpkg clean`), for example:

```
$ 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 desired version for upgrades, in case you are wondering).

```
$ bpkg build libhello/1.0.0 hello
downgrade libhello 1.0.0+1
reconfigure/build hello 1.0.0
continue? [Y/n] y
disfigured hello 1.0.0
disfigured libhello 1.1.0
libhello-1.0.0+1.tar.gz      100% of 1489  B  983 kBps 00m01s
fetched libhello 1.0.0+1
unpacked libhello 1.0.0+1
configured libhello 1.0.0+1
configured hello 1.0.0
c++ libhello-1.0.0+1/hello/cxx{hello}
ld libhello-1.0.0+1/hello/liba{hello}
c++ libhello-1.0.0+1/hello/cxx{hello}
ld libhello-1.0.0+1/hello/libso{hello}
c++ libhello-1.0.0+1/tests/test/cxx{driver}
ld libhello-1.0.0+1/tests/test/exe{driver}
updated libhello 1.0.0+1
c++ hello-1.0.0/cxx{hello}
ld hello-1.0.0/exe{hello}
updated hello 1.0.0
```

Notice how this time we updated `hello` as part of the `libhello` downgrade – yes, you can do that. Perhaps there should be an option to automatically update all the dependents?

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+1/tests/test/exe{driver}
test hello-1.0.0/exe{hello}
tested libhello 1.0.0+1
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+1/
bpkg.sqlite3
buildfile
hello-1.0.0.tar.gz
libhello-1.0.0+1.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.cxx
hello.o
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. On the contrary, it is a normal building block of the build system and if you understand what you are doing, feel free to muck around. Now, confess, did you run `sqlite3 bpkg.sqlite3 .dump`?

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.root.sudo=sudo hello
install /opt/hello/
install /opt/hello/include/hello/
install libhello-1.0.0+1/hello/hxx{hello}
install /opt/hello/lib/
install libhello-1.0.0+1/hello/libso{hello}
install /opt/hello/bin/
install hello-1.0.0/exe{hello}
install /opt/hello/share/doc/hello/
install hello-1.0.0/doc{version}
installed hello 1.0.0

$ tree -F /opt/hello/
/opt/hello/
âãâ bin/
â   âãâ hello*
âãâ include/
â   âãâ hello/
â       âãâ hello
âãâ lib/
â   âãâ libhello.so*
âãâ share/
    âãâ doc/
        âãâ hello/
            âãâ version

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

The `config.install.root.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!

What if we wanted to use `libhello` in our own project? While the installed version is always an option, it 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 various versions of `libhello` (and maybe even with that heinous `1.1.0`). 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 <hello/hello>

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

What build system will 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

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

$ 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/
âÂ Â  âãâ bootstrap.build
âÂ 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
test g++
error: unable to import target libhello%lib{hello}
  info: consider explicitly specifying its project out_root via the config.import.libhello command line variable
info: while applying rule cxx.compile to update obja{hello}
info: while applying rule cxx.link to update exe{hello}
info: while applying rule alias to update dir{./}
```

No magic but we got a hint: looks like we need to tell `build2` where `libhello` 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* for `libhello` is.

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

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

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

$ ./hello
Hello, World!
```

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

```
$ touch hello.cpp

$ b
test g++
error: unable to import target libhello%lib{hello}
  info: consider explicitly specifying its project out_root via the config.import.libhello command line variable
info: while applying rule cxx.compile to update obja{hello}
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 that `config.import.libhello` and who wants that? Also, the `test g++` line is getting annoying. To get rid of both 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.cxx.coptions=-O3 \
config.import.libhello=/tmp/hello-gcc5-release \
'configure(hello2/@hello2-gcc5-release/)'
test g++-5
mkdir hello2-gcc5-release/build/
mkdir hello2-gcc5-release/build/bootstrap/
save hello2-gcc5-release/build/bootstrap/src-root.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 obja{hello}

$ b -v
g++-5 -I/tmp/hello-gcc5-release/libhello-1.0.0+1 -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+1/hello/libhello.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/
mkdir /tmp/hello-gcc5-release/hello2/build/
mkdir /tmp/hello-gcc5-release/hello2/build/bootstrap/
save /tmp/hello-gcc5-release/hello2/build/bootstrap/src-root.build
save /tmp/hello-gcc5-release/hello2/build/config.build

$ b /tmp/hello-gcc5-release/hello2/
c++ hello2/cxx{hello}
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.cxx` values this time? Because they are inherited from the set specified for `/tmp/hello-gcc5-release` when we created this 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 as we now are.

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 `bpkg` *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/obja{hello}
rm /tmp/hello-gcc5-release/hello2/build/config.build
rm /tmp/hello-gcc5-release/hello2/build/bootstrap/src-root.build
rmdir /tmp/hello-gcc5-release/hello2/build/bootstrap/
rmdir /tmp/hello-gcc5-release/hello2/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 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}
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/ libhello
upgrade libhello 1.1.0
reconfigure hello2 (required by libhello)
continue? [Y/n] y
disfigured hello2 1.0.0
disfigured libhello 1.0.0+1
unpacked libhello 1.1.0
configured libhello 1.1.0
configured hello2 1.0.0
mkdir fsdir{/tmp/hello-gcc5-release/libhello-1.1.0/hello/}
c++ libhello/hello/cxx{hello}
ld /tmp/hello-gcc5-release/libhello-1.1.0/hello/liba{hello}
c++ libhello/hello/cxx{hello}
ld /tmp/hello-gcc5-release/libhello-1.1.0/hello/libso{hello}
updated libhello 1.1.0
```

As promised, `hello2` got reconfigured. We can now update it and give it a try:

```
$ bpkg update -d /tmp/hello-gcc5-release/ hello2
c++ hello2/cxx{hello}
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 cxx config.cxx=clang++-3.6 config.cxx.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
fetching build2.org/hello/stable (complements build2.org/hello/testing)
5 package(s) in 2 repository(s)

$ bpkg build libhello/1.0.0 .../hello2/
build libhello 1.0.0+1
build hello2 1.0.0
continue? [Y/n] y
libhello-1.0.0+1.tar.gz      100% of 1489  B  983 kBps 00m01s
fetched libhello 1.0.0+1
unpacked libhello 1.0.0+1
unpacked hello2 1.0.0
configured libhello 1.0.0+1
configured hello2 1.0.0
c++ libhello-1.0.0+1/hello/cxx{hello}
ld libhello-1.0.0+1/hello/liba{hello}
c++ libhello-1.0.0+1/hello/cxx{hello}
ld libhello-1.0.0+1/hello/libso{hello}
c++ libhello-1.0.0+1/tests/test/cxx{driver}
ld libhello-1.0.0+1/tests/test/exe{driver}
updated libhello 1.0.0+1
c++ ~/work/build2/hello/hello2/cxx{hello}
ld hello2-1.0.0/exe{hello}
updated hello2 1.0.0
```

Are you still here? Ok, one last example. This one is for *STL* (for those missing the connection, Stephan T. Lavavej, said, and I am paraphrasing, that he will never build a shared library and will never use a build system/package manager more complex than a single makefile; got to respect the man's convictions).

The Warning section above said there is no Windows support yet. But nobody said anything about cross-compilers:

```
$ mkdir hello-mingw32
$ cd hello-mingw32

$ bpkg create cxx                    \
config.cxx=x86_64-w64-mingw32-g++    \
config.bin.ar=x86_64-w64-mingw32-ar \
config.bin.lib=static config.cxx.loptions=-static
created new configuration in /tmp/hello-mingw32/

$ 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 -y hello
bpkg build -y hello
libhello-1.0.0+1.tar.gz      100% of 1489  B  983 kBps 00m01s
fetched libhello 1.0.0+1
unpacked libhello 1.0.0+1
hello-1.0.0.tar.gz           100% of 1030  B 6882 kBps 00m01s
fetched hello 1.0.0
unpacked hello 1.0.0
configured libhello 1.0.0+1
configured hello 1.0.0
c++ hello-1.0.0/cxx{hello}
c++ libhello-1.0.0+1/hello/cxx{hello}
ld libhello-1.0.0+1/hello/libso{hello}
ld hello-1.0.0/exe{hello}
updated hello 1.0.0

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

Installation

The `build2` toolchain requires a C++11 compiler with limited C++14 support. GCC 4.8 or later and Clang 3.4 or later are known to work. If you only need the `build2` build system without the `bpkg` package manager, then the C++ compiler is all you will need. If, however, you would also like to build `bpkg`, then you will first need to obtain SQLite as well as the `libodb` and `libodb-sqlite` libraries.

In this guide we install everything that we build into `/usr/local`. If you would like to use a different installation location, you will need to make adjustments to the commands below.

Note on `/usr/local`: most distributions these days "cripple" this location by either not searching `/usr/local/include` for headers during compilation (so we add the `-I` option) or not searching `/usr/local/lib` for libraries either during linking (so we add the `-L` option) or at runtime (which we fix with the help of `-rpath`). If you know that your installation doesn't suffer from (some of) these issues, then you can adjust the commands below accordingly. Note that even if `/usr/local/lib` is searched in at runtime, you may still have to run **`ldconfig(1)`** (as `root`) after the installation to refresh the library cache.

Note to Mac OS users: you will need version 10.5 (Leopard) or later. We will also be using the system C++ toolchain that comes with the Xcode Command Line Tools. To verify it is installed, run:

```
$ g++ --version
```

To install Command Line Tools, run:

```
$ xcode-select --install
```

1. Installing SQLite

   Skip this step if you are only interested in the `build2` build system.

   To install SQLite, use your distribution's package manager and make sure you install both the libraries (most likely already installed) and the development files.

   For Debian/Ubuntu:

   ```
   $ sudo apt-get install libsqlite3-dev
   ```

   For RedHat/Fedora:

   ```
   $ sudo yum install sqlite-devel
   ```

   For FreeBSD:

   ```
   # pkg install sqlite3
   ```

   For Mac OS:

   You should already have a system-default version installed. To verify:

   ```
   $ ls /usr/include/sqlite3.h /usr/lib/libsqlite3.dylib
   ```

   To see which version you have, run:

   ```
   $ grep '#define SQLITE_VERSION' /usr/include/sqlite3.h
   ```

   Any recent version (i.e., greater than 3.5.0) should work. If for some reason you don't seem to have SQLite, download the source code then build and install it into `/usr/local`.

2. Installing `libodb` and `libodb-sqlite`

   Again, skip this step if you are only interested in the `build2` build system.

   [Currently we use pre-release versions of these libraries so they have to be built from source.]

   Download source packages for the two libraries from the same location as `build2-toolchain`. Then unpack, build, and install:

```
$ cd lib*-X.Y.Z

$ ./configure --prefix=/usr/local \
CPPFLAGS=-I/usr/local/include     \
LDFLAGS=-L/usr/local/lib

$ make
$ sudo make install
```

See the `INSTALL` file for each library for more information.

3. Bootstrapping `build2`

   Download `build2-toolchain` then unpack and bootstrap the `build2` build system:

```
$ cd build2-toolchain-X.Y.Z
$ cd build2/
$ ./bootstrap
$ ./build2/b-boot config.bin.rpath=/usr/local/lib update
```

   For more information on this step (for example, how to specify the C++ compiler, options, etc.), refer to the `INSTALL` file in the `build2/` subdirectory of `build2-toolchain`.

4. Configuring, Building, and Installing the Rest of the Toolchain

```
$ cd .. # back to build2-toolchain-X.Y.Z

$ ./build2/build2/b                       \
config.cxx.poptions=-I/usr/local/include \
config.cxx.loptions=-L/usr/local/lib     \
config.bin.rpath=/usr/local/lib          \
config.install.root=/usr/local           \
config.install.root.sudo=sudo            \
configure

$ ./build2/build2/b update
$ ./build2/build2/b install
```

   To test the installation, run:

```
$ which b
/usr/local/bin/b
$ b --version

$ which bpkg
/usr/local/bin/bpkg
$ bpkg --version
```

5. Setting up updates with the package manager

   If you only need to build this specific version of the toolchain, then you are done and can skip this step. However, if you are planning to upgrade to future versions, then going every time through the bootstrap steps will be tedious. Instead, we can use the `bpkg` package manager to manage upgrades automatically. Note also that without periodic upgrades your version of the toolchain may become too old to be able to upgrade itself. In this case you will have to fall back onto the bootstrap process.

First, choose a directory where you would like `bpkg` to build everything, for example, `build2-toolchain`. Then:

```
$ cd # back to home directory
$ mkdir build2-toolchain
$ cd build2-toolchain

$ bpkg create                                \
cxx                                          \
config.cxx.poptions=-I/usr/local/include     \
config.cxx.loptions=-L/usr/local/lib         \
config.bin.rpath=/usr/local/lib              \
config.install.root=/usr/local               \
config.install.root.sudo=sudo

$ bpkg add https://pkg.cppget.org/1/alpha
$ bpkg fetch
$ bpkg build build2 bpkg
$ bpkg install build2 bpkg
```

Later, to upgrade to a new version of the toolchain, simply do:

```
$ bpkg fetch
$ bpkg status build2 bpkg   # See if any upgrades are available.
$ bpkg build build2 bpkg
$ bpkg install build2 bpkg
```