


Documentation

NAVIGATION

 Edit (<https://github.com/bazelbuild/bazel/tree/master/site/docs/cpp-use-cases.md>)

Introduction to Bazel: Common C++ Build Use Cases

Here you will find some of the most common use cases for building C++ projects with Bazel. If you have not done so already, get started with building C++ projects with Bazel by completing the tutorial [Introduction to Bazel: Build a C++ Project \(tutorial/cpp.html\)](#).

Contents

- Including multiple files in a target
- Using transitive includes
- Adding include paths
- Including external libraries
- Writing and running C++ tests
- Adding dependencies on precompiled libraries

Including multiple files in a target

You can include multiple files in a single target with `glob` ([../be/functions.html#glob](#)). For example:

```
cc_library(  
    name = "build-all-the-files",  
    srcs = glob(["*.cc"])  
    hdrs = glob(["*.h"]),  
)
```

With this target, Bazel will build all the `.cc` and `.h` files it finds in the same directory as the `BUILD` file that contains this target (excluding subdirectories).

Using transitive includes

If a file includes a header, then the file's rule should depend on that header's library. Conversely, only direct dependencies need to be specified as dependencies. For example, suppose `sandwich.h` includes `bread.h` and `bread.h` includes `flour.h`. `sandwich.h` doesn't include `flour.h` (who wants flour in their sandwich?), so the `BUILD` file would look like this:

```
cc_library(  
    name = "sandwich",  
    srcs = ["sandwich.cc"],  
    hdrs = ["sandwich.h"],  
    deps = [":bread"],  
)  
  
cc_library(  
    name = "bread",  
    srcs = ["bread.cc"],  
    hdrs = ["bread.h"],  
    deps = [":flour"],  
)  
  
cc_library(  
    name = "flour",  
    srcs = ["flour.cc"],  
    hdrs = ["flour.h"],  
)
```

Here, the `sandwich` library depends on the `bread` library, which depends on the `flour` library.

Adding include paths

Sometimes you cannot (or do not want to) root include paths at the workspace root. Existing libraries might already have an include directory that doesn't match its path in your workspace. For example, suppose you have the following directory structure:

```

└─ my-project
    └─ legacy
        └─ some_lib
            └─ BUILD
            └─ include
                └─ some_lib.h
            └─ some_lib.cc
└─ WORKSPACE

```

Bazel will expect `some_lib.h` to be included as `legacy/some_lib/include/some_lib.h`, but suppose `some_lib.cc` includes `"include/some_lib.h"`. To make that include path valid, `legacy/some_lib/BUILD` will need to specify that the `some_lib/` directory is an include directory:

```

cc_library(
    name = "some_lib",
    srcs = ["some_lib.cc"],
    hdrs = ["include/some_lib.h"],
    copts = ["-Ilegacy/some_lib/include"],
)

```

This is especially useful for external dependencies, as their header files must otherwise be included with a `/` prefix.

Including external libraries

Suppose you are using Google Test (<https://github.com/google/googletest>). You can use one of the `new_` repository functions in the `WORKSPACE` file to download Google Test and make it available in your repository:

```

new_http_archive(
    name = "gtest",
    url = "https://github.com/google/googletest/archive/release-1.7.0.zip",
    sha256 = "b58cb7547a28b2c718d1e38aee18a3659c9e3ff52440297e965f5edffe34b6d0",
    build_file = "gtest.BUILD",
)

```

NOTE: If the destination already contains a `BUILD` file, you can use one of the `non-new_` functions.

Then create `gtest.BUILD`, a `BUILD` file used to compile Google Test. Google Test has several "special" requirements that make its `cc_library` rule more complicated:

- `googletest-release-1.7.0/src/gtest-all.cc` #include s all of the other files in `googletest-release-1.7.0/src/`, so we need to exclude it from the compile or we'll get link errors for duplicate symbols.
- It uses header files that are relative to the `googletest-release-1.7.0/include/` directory (`"gtest/gtest.h"`), so we must add that directory to the include paths.
- It needs to link in `pthread`, so we add that as a `linkopt`.

The final rule therefore looks like this:

```
cc_library(
  name = "main",
  srcs = glob(
    ["googletest-release-1.7.0/src/*.cc"],
    exclude = ["googletest-release-1.7.0/src/gtest-all.cc"]
  ),
  hdrs = glob([
    "googletest-release-1.7.0/include/**/*.h",
    "googletest-release-1.7.0/src/*.h"
  ]),
  copts = [
    "-Iexternal/gtest/googletest-release-1.7.0/include"
  ],
  linkopts = ["-pthread"],
  visibility = ["//visibility:public"],
)
```

This is somewhat messy: everything is prefixed with `googletest-release-1.7.0` as a byproduct of the archive's structure. You can make `new_http_archive` strip this prefix by adding the `strip_prefix` attribute:

```

new_http_archive(
    name = "gtest",
    url = "https://github.com/google/googletest/archive/release-1.7.0.zip",
    sha256 = "b58cb7547a28b2c718d1e38aee18a3659c9e3ff52440297e965f5edffe34b6d0",
    build_file = "gtest.BUILD",
    strip_prefix = "googletest-release-1.7.0",
)

```

Then `gtest.BUILD` would look like this:

```

cc_library(
    name = "main",
    srcs = glob(
        ["src/*.cc"],
        exclude = ["src/gtest-all.cc"]
    ),
    hdrs = glob([
        "include/**/*.h",
        "src/*.h"
    ]),
    copts = ["-Iexternal/gtest/include"],
    linkopts = ["-pthread"],
    visibility = ["//visibility:public"],
)

```

Now `cc_` rules can depend on `@gtest//:main`.

Writing and running C++ tests

For example, we could create a test `./test/hello-test.cc` such as:

```
#include "gtest/gtest.h"
#include "lib/hello-greet.h"

TEST(HelloTest, GetGreet) {
    EXPECT_EQ(get_greet("Bazel"), "Hello Bazel");
}
```

Then create `./test/BUILD` file for your tests:

```
cc_test(
    name = "hello-test",
    srcs = ["hello-test.cc"],
    copts = ["-Iexternal/gtest/include"],
    deps = [
        "@gtest//:main",
        "//lib:hello-greet",
    ],
)
```

Note that in order to make `hello-greet` visible to `hello-test`, we have to add `"//test:__pkg__"`, to the `visibility` attribute in `./lib/BUILD`.

Now you can use `bazel test` to run the test.

```
bazel test test:hello-test
```

This produces the following output:

```
INFO: Found 1 test target...
Target //test:hello-test up-to-date:
  bazel-bin/test/hello-test
INFO: Elapsed time: 4.497s, Critical Path: 2.53s
//test:hello-test PASSED in 0.3s
```

```
Executed 1 out of 1 tests: 1 test passes.
```

Adding dependencies on precompiled libraries

If you want to use a library of which you only have a compiled version (for example, headers and a `.so` file) wrap it in a `cc_library` rule:

```
cc_library(  
    name = "mylib",  
    srcs = ["mylib.so"],  
    hdrs = ["mylib.h"],  
)
```

This way, other C++ targets in your workspace can depend on this rule.

About

Who's using Bazel (<https://github.com/bazelbuild/bazel/wiki/Bazel-Users>)

Roadmap (<https://www.bazel.build/roadmap.html>)

Contribute (<https://www.bazel.build/contributing.html>)

Governance Plan (<https://www.bazel.build/governance.html>)

Support

Stack Overflow (<http://stackoverflow.com/questions/tagged/bazel>)

Issue Tracker (<https://github.com/bazelbuild/bazel/issues>)

Documentation (<https://docs.bazel.build>)

FAQ (<https://www.bazel.build/faq.html>)

Support Policy (<https://www.bazel.build/support.html>)

Stay Connected

Twitter (<https://twitter.com/bazelbuild>)

Blog (<https://blog.bazel.build>)

GitHub (<https://github.com/bazelbuild/bazel>)

Discussion group (<https://groups.google.com/forum/#!forum/bazel-discuss>)

© 2018 Google