

Build Scripts

Some packages need to compile third-party non-Rust code, for example C libraries. Other packages need to link to C libraries which can either be located on the system or possibly need to be built from source. Others still need facilities for functionality such as code generation before building (think parser generators).

Cargo does not aim to replace other tools that are well-optimized for these tasks, but it does integrate with them with the `build` configuration option.

```
[package]
# ...
build = "build.rs"
```



The Rust file designated by the `build` command (relative to the package root) will be compiled and invoked before anything else is compiled in the package, allowing your Rust code to depend on the built or generated artifacts. By default Cargo looks up for `"build.rs"` file in a package root (even if you do not specify a value for `build`). Use `build = "custom_build_name.rs"` to specify a custom build name or `build = false` to disable automatic detection of the build script.

Some example use cases of the `build` command are:

- Building a bundled C library.
- Finding a C library on the host system.
- Generating a Rust module from a specification.
- Performing any platform-specific configuration needed for the crate.

Each of these use cases will be detailed in full below to give examples of how the `build` command works.

Inputs to the Build Script

When the build script is run, there are a number of inputs to the build script, all passed in the form of [environment variables](#).

In addition to environment variables, the build script's current directory is the source directory of the build script's package.

Outputs of the Build Script

All the lines printed to stdout by a build script are written to a file like `target/debug/build/<pkg>/output` (the precise location may depend on your configuration). If you would like to see such output directly in your terminal, invoke cargo as 'very verbose' with the `-vv` flag. Note that if neither the build script nor project source files are modified, subsequent calls to cargo with `-vv` will **not** print output to the terminal because a new build is not executed. Run `cargo clean` before each cargo invocation if you want to ensure that output is always displayed on your terminal. Any line that starts with `cargo:` is interpreted directly by Cargo. This line must be of the form `cargo:key=value`, like the examples below:

```
# specially recognized by Cargo
cargo:rustc-link-lib=static=foo
cargo:rustc-link-search=native=/path/to/foo
cargo:rustc-cfg=foo
cargo:rustc-env=F00=bar
# arbitrary user-defined metadata
cargo:root=/path/to/foo
cargo:libdir=/path/to/foo/lib
cargo:include=/path/to/foo/include
```



On the other hand, lines printed to stderr are written to a file like `target/debug/build/<pkg>/stderr` but are not interpreted by cargo.

There are a few special keys that Cargo recognizes, some affecting how the crate is built:

- `rustc-link-lib=[KIND=]NAME` indicates that the specified value is a library name and should be passed to the compiler as a `-l` flag. The optional `KIND` can be one of `static`, `dllib` (the default), or `framework`, see `rustc --help` for more details.
- `rustc-link-search=[KIND=]PATH` indicates the specified value is a library search path and should be passed to the compiler as a `-L` flag. The optional `KIND` can be one of `dependency`, `crate`, `native`, `framework` or `all` (the default), see `rustc --help` for more details.
- `rustc-flags=FLAGS` is a set of flags passed to the compiler, only `-l` and `-L` flags are supported.
- `rustc-cfg=FEATURE` indicates that the specified feature will be passed as a `--cfg` flag to the compiler. This is often useful for performing compile-time detection of various features.

- `rustc-env=VAR=VALUE` indicates that the specified environment variable will be added to the environment which the compiler is run within. The value can be then retrieved by the `env!` macro in the compiled crate. This is useful for embedding additional metadata in crate's code, such as the hash of Git HEAD or the unique identifier of a continuous integration server.
- `rerun-if-changed=PATH` is a path to a file or directory which indicates that the build script should be re-run if it changes (detected by a more-recent last-modified timestamp on the file). Normally build scripts are re-run if any file inside the crate root changes, but this can be used to scope changes to just a small set of files. (If this path points to a directory the entire directory will not be traversed for changes -- only changes to the timestamp of the directory itself (which corresponds to some types of changes within the directory, depending on platform) will trigger a rebuild. To request a re-run on any changes within an entire directory, print a line for the directory and another line for everything inside it, recursively.) Note that if the build script itself (or one of its dependencies) changes, then it's rebuilt and rerun unconditionally, so `cargo:rerun-if-changed=build.rs` is almost always redundant (unless you want to ignore changes in all other files except for `build.rs`).
- `rerun-if-env-changed=VAR` is the name of an environment variable which indicates that if the environment variable's value changes the build script should be rerun. This basically behaves the same as `rerun-if-changed` except that it works with environment variables instead. Note that the environment variables here are intended for global environment variables like `CC` and such, it's not necessary to use this for env vars like `TARGET` that Cargo sets. Also note that if `rerun-if-env-changed` is printed out then Cargo will *only* rerun the build script if those environment variables change or if files printed out by `rerun-if-changed` change.
- `warning=MESSAGE` is a message that will be printed to the main console after a build script has finished running. Warnings are only shown for path dependencies (that is, those you're working on locally), so for example warnings printed out in crates.io crates are not emitted by default.

Any other element is a user-defined metadata that will be passed to dependents. More information about this can be found in the `links` section.

Build Dependencies

Build scripts are also allowed to have dependencies on other Cargo-based crates.

Dependencies are declared through the `build-dependencies` section of the manifest.

```
[build-dependencies]
foo = { git = "https://github.com/your-packages/foo" }
```



The build script **does not** have access to the dependencies listed in the `dependencies` or `dev-dependencies` section (they're not built yet!). All build dependencies will also not be available to the package itself unless explicitly stated as so.

The `links` Manifest Key

In addition to the manifest key `build`, Cargo also supports a `links` manifest key to declare the name of a native library that is being linked to:

```
[package]
# ...
links = "foo"
build = "build.rs"
```



This manifest states that the package links to the `libfoo` native library, and it also has a build script for locating and/or building the library. Cargo requires that a `build` command is specified if a `links` entry is also specified.

The purpose of this manifest key is to give Cargo an understanding about the set of native dependencies that a package has, as well as providing a principled system of passing metadata between package build scripts.

Primarily, Cargo requires that there is at most one package per `links` value. In other words, it's forbidden to have two packages link to the same native library. Note, however, that there are [conventions in place](#) to alleviate this.

As mentioned above in the output format, each build script can generate an arbitrary set of metadata in the form of key-value pairs. This metadata is passed to the build scripts of **dependent** packages. For example, if `libbar` depends on `libfoo`, then if `libfoo` generates `key=value` as part of its metadata, then the build script of `libbar` will have the environment variables `DEP_FOO_KEY=value`.

Note that metadata is only passed to immediate dependents, not transitive dependents. The motivation for this metadata passing is outlined in the linking to system libraries case study below.

Overriding Build Scripts

If a manifest contains a `links` key, then Cargo supports overriding the build script specified with a custom library. The purpose of this functionality is to prevent running the build script in question altogether and instead supply the metadata ahead of time.

To override a build script, place the following configuration in any acceptable Cargo [configuration location](#).

```
[target.x86_64-unknown-linux-gnu.foo]
rustc-link-search = ["/path/to/foo"]
rustc-link-lib = ["foo"]
root = "/path/to/foo"
key = "value"
```



This section states that for the target `x86_64-unknown-linux-gnu` the library named `foo` has the metadata specified. This metadata is the same as the metadata generated as if the build script had run, providing a number of key/value pairs where the `rustc-flags`, `rustc-link-search`, and `rustc-link-lib` keys are slightly special.

With this configuration, if a package declares that it links to `foo` then the build script will **not** be compiled or run, and the metadata specified will instead be used.

Case study: Code generation

Some Cargo packages need to have code generated just before they are compiled for various reasons. Here we'll walk through a simple example which generates a library call as part of the build script.

First, let's take a look at the directory structure of this package:

```
.
├─ Cargo.toml
├─ build.rs
└─ src
    └─ main.rs
```



1 directory, 3 files

Here we can see that we have a `build.rs` build script and our binary in `main.rs`. Next, let's take a look at the manifest:

```
# Cargo.toml

[package]
name = "hello-from-generated-code"
version = "0.1.0"
authors = ["you@example.com"]
build = "build.rs"
```



Here we can see we've got a build script specified which we'll use to generate some code. Let's see what's inside the build script:

```
// build.rs

use std::env;
use std::fs::File;
use std::io::Write;
use std::path::Path;

fn main() {
    let out_dir = env::var("OUT_DIR").unwrap();
    let dest_path = Path::new(&out_dir).join("hello.rs");
    let mut f = File::create(&dest_path).unwrap();

    f.write_all(b"
        pub fn message() -> &'static str {
            \"Hello, World!\"
        }
    ").unwrap();
}
```



There's a couple of points of note here:

- The script uses the `OUT_DIR` environment variable to discover where the output files should be located. It can use the process' current working directory to find where the input files should be located, but in this case we don't have any input files.
- This script is relatively simple as it just writes out a small generated file. One could imagine that other more fanciful operations could take place such as generating a Rust module from a C header file or another language definition, for example.

Next, let's peek at the library itself:

```
// src/main.rs

include!(concat!(env!("OUT_DIR"), "/hello.rs"));

fn main() {
    println!("{}", message());
}
```



This is where the real magic happens. The library is using the rustc-defined `include!` macro in combination with the `concat!` and `env!` macros to include the generated file (`hello.rs`) into the crate's compilation.

Using the structure shown here, crates can include any number of generated files from the build script itself.

Case study: Building some native code

Sometimes it's necessary to build some native C or C++ code as part of a package. This is another excellent use case of leveraging the build script to build a native library before the Rust crate itself. As an example, we'll create a Rust library which calls into C to print "Hello, World!".

Like above, let's first take a look at the project layout:

```
.
├─ Cargo.toml
├─ build.rs
└─ src
   └─ hello.c
      └─ main.rs
```



1 directory, 4 files

Pretty similar to before! Next, the manifest:

```
# Cargo.toml

[package]
name = "hello-world-from-c"
version = "0.1.0"
authors = ["you@example.com"]
build = "build.rs"
```



For now we're not going to use any build dependencies, so let's take a look at the

build script now:

```
// build.rs

use std::process::Command;
use std::env;
use std::path::Path;

fn main() {
    let out_dir = env::var("OUT_DIR").unwrap();

    // note that there are a number of downsides to this approach, the
    // comments
    // below detail how to improve the portability of these commands.
    Command::new("gcc").args(&["src/hello.c", "-c", "-fPIC", "-o"])
        .arg(&format!("{}", out_dir))
        .status().unwrap();
    Command::new("ar").args(&["crus", "libhello.a", "hello.o"])
        .current_dir(&Path::new(out_dir))
        .status().unwrap();


    println!("cargo:rustc-link-search=native={}", out_dir);
    println!("cargo:rustc-link-lib=static=hello");
}
```

This build script starts out by compiling our C file into an object file (by invoking `gcc`) and then converting this object file into a static library (by invoking `ar`). The final step is feedback to Cargo itself to say that our output was in `out_dir` and the compiler should link the crate to `libhello.a` statically via the `-l static=hello` flag.

Note that there are a number of drawbacks to this hardcoded approach:

- The `gcc` command itself is not portable across platforms. For example it's unlikely that Windows platforms have `gcc`, and not even all Unix platforms may have `gcc`. The `ar` command is also in a similar situation.
- These commands do not take cross-compilation into account. If we're cross compiling for a platform such as Android it's unlikely that `gcc` will produce an ARM executable.

Not to fear, though, this is where a `build-dependencies` entry would help! The Cargo ecosystem has a number of packages to make this sort of task much easier, portable, and standardized. For example, the build script could be written as:


```
// build.rs   
  
// Bring in a dependency on an externally maintained `cc` package which  
// manages  
// invoking the C compiler.  
extern crate cc;  
  
fn main() {  
    cc::Build::new()  
        .file("src/hello.c")  
        .compile("hello");  
}
```

Add a build time dependency on the `cc` crate with the following addition to your `Cargo.toml`:


```
[build-dependencies]   
gcc = "1.0"
```

The `cc` crate abstracts a range of build script requirements for C code:

- It invokes the appropriate compiler (MSVC for windows, `gcc` for MinGW, `cc` for Unix platforms, etc.).
- It takes the `TARGET` variable into account by passing appropriate flags to the compiler being used.
- Other environment variables, such as `OPT_LEVEL`, `DEBUG`, etc., are all handled automatically.
- The stdout output and `OUT_DIR` locations are also handled by the `cc` library.

Here we can start to see some of the major benefits of farming as much functionality as possible out to common build dependencies rather than duplicating logic across all build scripts!

Back to the case study though, let's take a quick look at the contents of the `src` directory:

```
// src/hello.c   
  
#include <stdio.h>  
  
void hello() {  
    printf("Hello, World!\n");  
}
```

```
// src/main.rs

// Note the lack of the `#[link]` attribute. We're delegating the
responsibility
// of selecting what to link to over to the build script rather than
hardcoding
// it in the source file.
extern { fn hello(); }

fn main() {
    unsafe { hello(); }
}
```



And there we go! This should complete our example of building some C code from a Cargo package using the build script itself. This also shows why using a build dependency can be crucial in many situations and even much more concise!

We've also seen a brief example of how a build script can use a crate as a dependency purely for the build process and not for the crate itself at runtime.

Case study: Linking to system libraries

The final case study here will be investigating how a Cargo library links to a system library and how the build script is leveraged to support this use case.

Quite frequently a Rust crate wants to link to a native library often provided on the system to bind its functionality or just use it as part of an implementation detail. This is quite a nuanced problem when it comes to performing this in a platform-agnostic fashion, and the purpose of a build script is again to farm out as much of this as possible to make this as easy as possible for consumers.

As an example to follow, let's take a look at one of [Cargo's own dependencies](#), [libgit2](#). The C library has a number of constraints:

- It has an optional dependency on OpenSSL on Unix to implement the https transport.
- It has an optional dependency on libssh2 on all platforms to implement the ssh transport.
- It is often not installed on all systems by default.
- It can be built from source using `cmake`.

To visualize what's going on here, let's take a look at the manifest for the relevant Cargo package that links to the native C library.



```
[package]
name = "libgit2-sys"
version = "0.1.0"
authors = ["..."]
links = "git2"
build = "build.rs"

[dependencies]
libssh2-sys = { git = "https://github.com/alexcrichton/ssh2-rs" }

[target.'cfg(unix)'.dependencies]
openssl-sys = { git = "https://github.com/alexcrichton/openssl-sys" }

# ...
```

As the above manifests show, we've got a `build` script specified, but it's worth noting that this example has a `links` entry which indicates that the crate (`libgit2-sys`) links to the `git2` native library.

Here we also see that we chose to have the Rust crate have an unconditional dependency on `libssh2` via the `libssh2-sys` crate, as well as a platform-specific dependency on `openssl-sys` for `*nix` (other variants elided for now). It may seem a little counterintuitive to express *C dependencies* in the *Cargo manifest*, but this is actually using one of Cargo's conventions in this space.

*-sys Packages

To alleviate linking to system libraries, Cargo has a *convention* of package naming and functionality. Any package named `foo-sys` will provide two major pieces of functionality:

- The library crate will link to the native library `libfoo`. This will often probe the current system for `libfoo` before resorting to building from source.
- The library crate will provide **declarations** for functions in `libfoo`, but it does **not** provide bindings or higher-level abstractions.

The set of `*-sys` packages provides a common set of dependencies for linking to native libraries. There are a number of benefits earned from having this convention of native-library-related packages:

- Common dependencies on `foo-sys` alleviates the above rule about one package per value of `links`.
- A common dependency allows centralizing logic on discovering `libfoo` itself

(or building it from source).

- These dependencies are easily overridable.

Building libgit2

Now that we've got libgit2's dependencies sorted out, we need to actually write the build script. We're not going to look at specific snippets of code here and instead only take a look at the high-level details of the build script of `libgit2-sys`. This is not recommending all packages follow this strategy, but rather just outlining one specific strategy.

The first step of the build script should do is to query whether libgit2 is already installed on the host system. To do this we'll leverage the preexisting tool `pkg-config` (when its available). We'll also use a `build-dependencies` section to refactor out all the `pkg-config` related code (or someone's already done that!).

If `pkg-config` failed to find libgit2, or if `pkg-config` just wasn't installed, the next step is to build libgit2 from bundled source code (distributed as part of `libgit2-sys` itself). There are a few nuances when doing so that we need to take into account, however:

- The build system of libgit2, `cmake`, needs to be able to find libgit2's optional dependency of libssh2. We're sure we've already built it (it's a Cargo dependency), we just need to communicate this information. To do this we leverage the metadata format to communicate information between build scripts. In this example the libssh2 package printed out `cargo:root=...` to tell us where libssh2 is installed at, and we can then pass this along to `cmake` with the `CMAKE_PREFIX_PATH` environment variable.
- We'll need to handle some `CFLAGS` values when compiling C code (and tell `cmake` about this). Some flags we may want to pass are `-m64` for 64-bit code, `-m32` for 32-bit code, or `-fPIC` for 64-bit code as well.
- Finally, we'll invoke `cmake` to place all output into the `OUT_DIR` environment variable, and then we'll print the necessary metadata to instruct `rustc` how to link to libgit2.

Most of the functionality of this build script is easily refactorable into common dependencies, so our build script isn't quite as intimidating as this descriptions! In reality it's expected that build scripts are quite succinct by farming logic such as above to build dependencies.