Linking Rust Crates, Part 1

⊕ ☐ This post is the first part in a planned series on linking Rust crates. My desire is to get informal feedback on these posts, and then turn it into more structured content suitable for the Rust reference, or maybe the Rustonomicon (or a book on its own.) Working on the Rust compiler, one topic that I come across from time to time is "what is *supposed* to happen when we use these particular features of my tools?" More specifically, Rust has various metaphorical knobs that allow fine-grained control of the object code generated by the compiler, several of which are related to the process of linking that code to other object code.

From Linkage chapter of the Rust Reference, we can see there are seven kinds of crates: `bin`, `lib`, `dylib`, `staticlib`, `cdylib`, `rlib`, and `proc-macro`.

⊕ ☐ I had originally intended to cover all seven, but the time got late and the post got long and I decided that `proc-macro` can be dealt with another day.

What this post is going to do is walk through the first six of the crate types listed above and demonstrate: how to build an example of such a crate, how to link to it, and how to run with that linked crate.

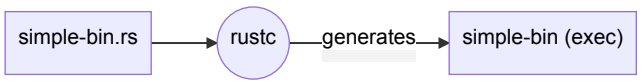
In later posts, I will explore the various attributes and command-line flags that can influence the linking step. But right now, I want to establish the foundation for that later discussion.

⊕ ☐ Some phenomena require mixing multiple crate types in order to observe corner cases that are worth addressing. I am leaving that for a future post as well.

These initial examples are as simple as possible. We will want to actually demonstrate each case running. Since most crate types are not executable, that means we will need multiple crates in almost all of our examples.

Also, I will not be using Cargo in any of my examples. I will try to note points where things I am doing are deviating from what you would normally do if you were using Cargo (such as my use of `extern crate` in these examples), and hopefully a later post will explore the status quo of how Cargo handles various linkage scenarios. But, I am taking baby steps here; lets focus on `rustc` alone for now.

At a very high level, we will be looking at object file structures that look something like this:



That is: Given some input like `simple-bin.rs`, you can feed into `rustc` and get an executable as its output.

The actual code for a case like the above is trivial:

```
1 // simple-bin.rs
2 #![crate_type="bin"] // ("bin" is the default; other examples vary here.)
3 fn main() { println!("Running main from {}", file!()); }
```

With that code in place, you can compile and run the program. (In practice developers would usally do this via `cargo run`.)

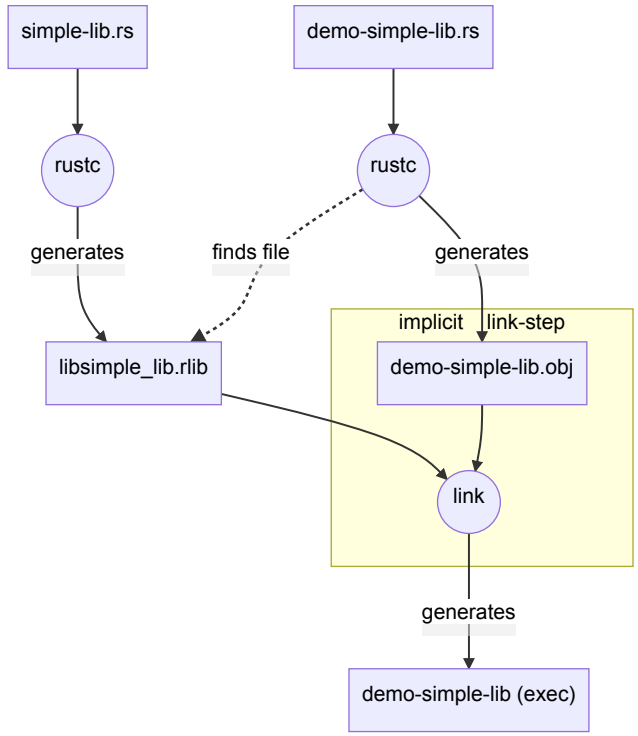
```
1 % rustc --out-dir out/bins/ps/dsb simple-bin.rs
2 % ls out/bins/ps/dsb
3 simple-bin
4 % out/bins/ps/dsb/simple-bin
5 Running main from simple-bin.rs
6 %
```

⊕ ☐ I am using a bespoke directory naming convention unique to this blog post. For example, in `--out-dir` in this invocation: All output goes under `out`. Executable binaries go under `out/bin`. If static linkage was preferred during the build, then it goes under `out/bin/ps` (for "prefer static"); if dynamic linkage was preferred, then under `out/bin/pd` (for "prefer dynamic"); the post explains this "preference" further down, with the discussion of `-Cprefer-dynamic`. Finally, I added a directory named via a unique key for each test (here, `dsb`, for

“demo-simple-bin”), so that the `ls` invocations are tidy. In these examples, I will be overriding the default output directory so that each example will have its own direcrory. This forces the examples to specify precisely where it is getting its libraries from; it is a good way to double-check one’s understanding of what is actually happening under-the-hood.

Simple linkage of a `lib` crate

Of course, since our subject of interest is linking, we will want to look at examples that involve library crates. Here is perhaps the simplest intance of that:



What is the advantage of separating your code like this, into separate libraries that are subsequently linked? One reason to do this is to identify which code is under active development, and focus on its development separately from the overall product. So, if `simple-lib.rs` were under development, we could focus just on that, and not have our tools spend time processing `demo-simple-lib.rs`. Or vice-versa: If `demo-simple-lib.rs` were under development, then we could focus on that, and our tools would process, at most, the `libsimple_lib.rlib` library produced by running `rustc` on `simple-lib.rs`.

Here are the code and commands that correspond to the picture above for compiling `demo-simple-lib`.

```
1 // simple-lib.rs
2
3 // generate a library (what kind? The "default" for this platform)
4 #![crate_type="lib"]
5
6 // Here's the function we'll provide to our clients.
7 pub fn main() { println!("Running main from {}", file!()); }
```

```
1 // demo-simple-lib.rs
2
3 // link to library built from simple-lib.rs
4 extern crate simple_lib;
5
6 // call function it exports
7 fn main() { simple_lib::main(); }
```

We have omitted `#![crate_type="bin"]` from `demo-simple-lib.rs` because that is the default crate type.

(In practice, most modern Rust code does not use `extern crate`; instead, people let Cargo handle injecting compiler options that achieve a similar effect. For now, we will use `extern crate` in our initial examples, so that the Rust code itself indicates its dependence on a separate library.)

With the above two files in place, we can compile each of them and run the resulting binary.

```
1 % rustc --out-dir out/ps/sl simple-lib.rs
2 % ls out/ps/sl
3 libsimple_lib.rlib
4 % rustc --out-dir out/bins/ps/dsl demo-simple-lib.rs -Lout/ps/sl
5 % ls out/bins/ps/dsl
6 demo-simple-lib
```

```
7 % out/bins/ps/dsl/demo-simple-lib
8 Running main from simple-lib.rs
9 %
```

⊕ ☐ If you're squinting at the diagram and wondering how you might confirm that the relations it describes properly reflect what these commands are doing, I recommend you check out the "Who is using that library?" appendix at the end of this post. It walks through some of the issues that arise when linking crates together.

The first `rustc` invocation, compiling `simple-lib.rs`, is much like our `simple-bin.rs` example. The second `rustc` invocation, compiling `demo-simple-lib.rs`, has something new: it is passing the option `-Lout/sl`, which tells the compiler "if you need to resolve any external crates, you should add `out/sl` to the list of paths you will search for them."

The `demo-simple-lib` example demonstrates Rust's `lib` crate type, which is specified as a compiler-defined choice from one of the flavors of libraries that Rust supports.

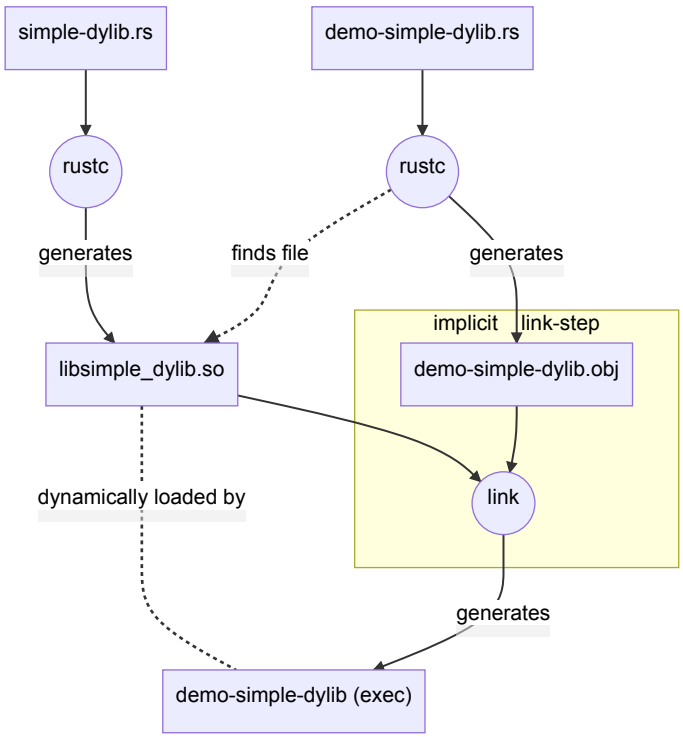
At the time of this blog post, Rust's `lib` crate type maps to `rlib`, at least on my machine. We will talk more about `rlib` further down; but for now, you can just keep in mind the points made above for `simple-lib`: the compiler itself will read the metadata stored in `rlib` crates, *and* the linker also extracts definitions from `rlib` crates as well.

We will now step through other supported crate types, and provide a similar demonstration of how they operate.

⊕ ☐ Re-reading the overall post now, I am thinking that I should have put `dylib` at the end, due to the number of rabbit holes it opens up. Maybe I will edit the post in the future and do that rearrangement, so that people can see the "easy cases" first.

Dynamic Libraries: `dylib`

The next crate type listed in the Rust reference is `dylib`.



This is a similar picture to the one presented above for `lib`. In fact, if you look at the two pictures side-by-side, most of the differences can be attributed to uniform renaming ... except for one: There is now an extra arc in the diagram, connecting the `demo-simple-dylib` executable directly to the `libsimple_dylib.so` library file.

This is because a `dylib` crate is a dynamic library; it is meant to be loaded *dynamically*, when the program is executed. There are a couple different reasons one might want this: Perhaps the program is meant to support a "plug-in" architecture, where one can get load up new behaviors by swapping in a different dynamic library. Another common reason is to reduce executable binary sizes: if many programs depend on the same external crate, then it might be more efficient to have them all share the same `dylib`.

Note: The `dylib` format is not guaranteed to remain stable between different versions of the Rust compiler. Therefore, if you use the `dylib` format, you need to ensure that all crates that are sharing the same `dylib` and the `dylib` itself are all built with the same version of the Rust compiler.

Here are the code and commands that correspond to the picture above for compiling `demo-simple-dylib".

```
1 // simple-dylib.rs
2 #![crate_type="dylib"]
3 pub fn main() { println!("Running main from {}", file!()); }
```

```
1 // demo-simple-dylib
2 extern crate simple_dylib;
3 fn main() { simple_dylib::main(); }
```

With the above two files in place, we can compile each of them and run the resulting binary.

```
1 % rustc --out-dir out/pd/sd -C prefer-dynamic simple-dylib.rs
2 % ls out/pd/sd
3 libsimple_dylib.so
4 % rustc --out-dir out/bins/ps/dsd demo-simple-dylib.rs -Lout/pd/sd
5 % ls out/bins/ps/dsd
6 demo-simple-dylib
7 % LD_LIBRARY_PATH=out/pd/sd:$(rustc --print=sysroot)/lib out/bins/ps/dsd/demo-simple-dylib
```

You might have noticed that our `demo-simple-dylib` code is very similar to the `demo-simple-lib` code; the only effective change to the source is the difference in `crate_type` for `simple-dylib.rs`. On the other hand, the commands to compile these inputs and run the executable have had some pretty severe changes applied to them. Let us explore why those changes were necessary.

Adapting command lines to meet needs of crate type

If we tried to reuse the `demo-simple-lib` sequence of steps to compile these files and run the presumably generated executable, we hit some roadblocks.

Why was -Cprefer-dynamic added

First, if we tried to compile `simple-dylib.rs` using the same command that was used for `simple-lib.rs`, it *seems* to work at first:

```
1 % mkdir -p out/ps/sd
2 % rustc --out-dir out/ps/sd simple-dylib.rs
3 % ls out/ps/sd
4 libsimple_dylib.so
5 %
```

The problem arises when we try to *use* that generated dylib that was compiled; that step causes the following error output:

```
1  ```sh
2  % rustc --out-dir out/bins/ps/dsd demo-simple-dylib.rs -Lout/ps/sd
3  error: cannot satisfy dependencies so `std` only shows up once
4    |
5    = help: having upstream crates all available in one format will likely make this go away
6
7  error: cannot satisfy dependencies so `core` only shows up once
8    |
9    = help: having upstream crates all available in one format will likely make this go away
10 [...]
```

and so on, for all of the upstram crates `std`, `core`, `compiler_builtins`, `rustc_std_workspace_core`, `alloc`, `libc`, `unwind`, `cfg_if`, `hashbrown`, `rustc_std_workspace_alloc`, `std_detect`, `rustc_demangle`, `addr2line`, `gimli`, `object`, `memchr`, `miniz_oxide`, `adler`, and `panic_unwind`.

Understanding why this happens requires we take a step back.

The compiler needs to decide, in the absence of explicit indication from the programmer, how each dependency should be incorporated into the executable binary being generated. Namely, should a given dependency be “statically linked” into the output object (which effectively means that anything the object needs will be copied into the output from the referenced library), or should the given dependency be “dynamically linked” (which means the executable carries a dependence on the dynamic library, that will need to be resolved at runtime). This is a non-trivlal decision, because Rust allows individual crates to opt-into supporting multiple distinct crate types: a single crate can say “I can be used as an `rlib` or a `dylib`; I will let my downstream client decide which object file they want to pull in for their needs.”

But if no one tells the compiler what choice to make, the Rust compiler applies a simple-minded tactic for guessing what options would be best, and currently, that tactic is heavily influenced by the presence or absence of `-C prefer-dynamic`.

When `simple-dylib.rs` was compiled *without* `-C prefer-dynamic`, the compiler interpreted the absence of that flag as a signal that the compiler should attempt to link all dependencies of `simple-dylib.rs` statically.

Furthermore, the compiler manages to succeed at this static linkage of those dependencies, but in doing so, it has it impossible to link the resulting statically-linked crate into `demo-simple-dylib.rs`.

⊕ ☐ A reasonable person might note here: “Why is the linkage of `std` from `demo-simple-dylib` treated as its own distinct thing? In other words, why doesn’t the compiler just let `simple-dylib`, which has already statically-linked in those crates, provide them to `demo-simple-dylib`? And you wouldn’t be alone in thinking this: Alex Crichton, who is responsible for the basic logic in use here, made the same suggestion in rust-lang/rust#34909. I plan to explore this question more in a future blog post. The `demo-simple-dylib` crate *also* wants to link to all the same upstream crates, and the compiler rejects this, saying it is not legal for both the `simple-dylib` and the `demo-simple-dylib` crates to have duplicate copies of those dependencies.

⊕ ☐ The semantics described here dates from RFC 404, which was introduced in 2014 before Rust had even hit 1.0 status. That RFC itself refers to the comments in the code as the documentation; those comments have moved as the compiler has gone through various refactorings, but the most recent version can be found here in rustc_metadata::dependency_format

In the absence of `-Cprefer-dynamic` and `--extern` flags (and also absent any constraints forcing everything to be statically linked), the default logic of the compiler when trying to decide which upstream crate types to use is as follows:

1. First try to statically link all of the upstream dependencies via their `.rlib` libraries. If that succeeds, we are done.
2. If static linking failed, then either linking in general is impossible, or at least one dependency will have to be a dynamic library. Since at least one dependency has to be dynamic, the compiler, as a simple-minded tactic, tries to satisfy as many upstream dependencies as possible via their `dylib` object files.

However, if one *does* provide `-Cprefer-dynamic`, then that tells the compiler to not attempt the static link step, and instead to prefer to use `dylib` whenever possible for its upstream dependencies. And *that* is the fix we need here: we need `simple-lib.dylib` to prefer the `dylib` version of `std`.

```
1 % rustc --out-dir out/pd/sd -C prefer-dynamic simple-dylib.rs
```

Once that is in place, everything else follows suit.

First, consider how `demo-simple-dylib.rs` is built.

```
1 % rustc --out-dir out/bins/ps/dsd demo-simple-dylib.rs -Lout/pd/sd
```

⊕ ☐ Note that this story here is simple in part because `simple-dylib.rs` was only compiled to one crate type. If we had generated both `dylib` and `rlib` crates for it, *then* the presence/absence of `-Cprefer-dynamic` would become significant for building `demo-simple-dylib`.

Once we have established that the `simple-dylib` crate is only available as a `dylib`, then it does not matter whether we pass `-Cprefer-dynamic` or not when building `demo-simple-dylib`: if we leave it off, then all that happens is the compiler will first explore trying to statically link all of the dependencies. Once it determines it cannot (due to `simple-dylib`), it will go back to trying to use dynamic libraries for as much as possible, and thus it will resolve both `simple-dylib` *and* `std` to dynamic libraries.

Next, we consider the way that `demo-simple-dylib` needs to be invoked:

```
1 % LD_LIBRARY_PATH=out/pd/sd:$(rustc --print=sysroot)/lib out/bins/ps/dsd/demo-simple-dylib
```

The main point of interest here is that we had to add some entries to the `LD_LIBRARY_PATH`: when the program runs, it needs to satisfy its upstream dylib dependencies, which we just finished establishing are `simple-dylib` (in `out/pd/sd`) and `std` (which we map to a directory by asking `rustc` itself where those support files all live by running `rustc --print=sysroot`).

Rust libraries: `rlib`

Phew, that was exhausting.

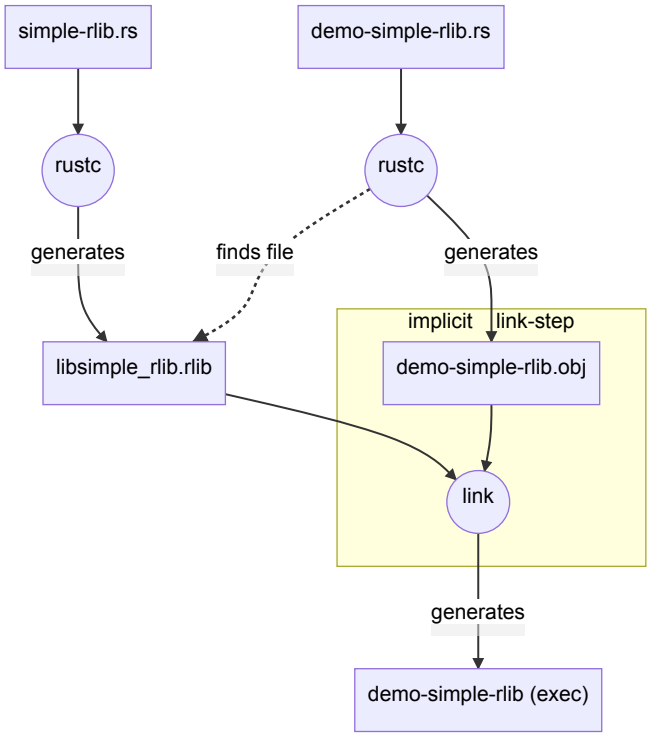
Lets try to go through an easier case next: the Rust library type, `rlib`.

This is not necessarily a simple case, but it is an obvious one to deal with, because we've been discussing it this whole time; we just did not say that we were.

Specifically: the `demo-simple-lib` example covered Rust's `lib` crate type, which is specified as a compiler-defined choice from one of the flavors of libraries that Rust supports. At the time of this blog post, Rust's `lib` crate type maps to `rlib`, at least on my machine.

So, the usage patterns and issues we described up above for `lib` all apply to `rlib`.

For completeness, here is a diagram showing how `rlib` is used; it will look very familiar.



Likewise, here is the source code for the files listed in the diagram.

```
1 // simple-rlib.rs
2 #![crate_type="rlib"]
3 pub fn main() { println!("Running main from {}", file!()); }
```

```
1 // demo-simple-rlib.rs
2 extern crate simple_rlib;
3 fn main() { simple_rlib::main(); }
```

Finally, the command invocations that implement the diagram above.

```
1 % rustc --out-dir out/ps/sr simple-rlib.rs
2 % ls out/ps/sr
3 libsimple_rlib.rlib
4 % rustc --out-dir out/bins/ps/dsr demo-simple-rlib.rs -Lout/ps/sr
5 % ls out/bins/ps/dsr
6 demo-simple-rlib
7 % out/bins/ps/dsr/demo-simple-rlib
8 Running main from simple-rlib.rs
9 %
```

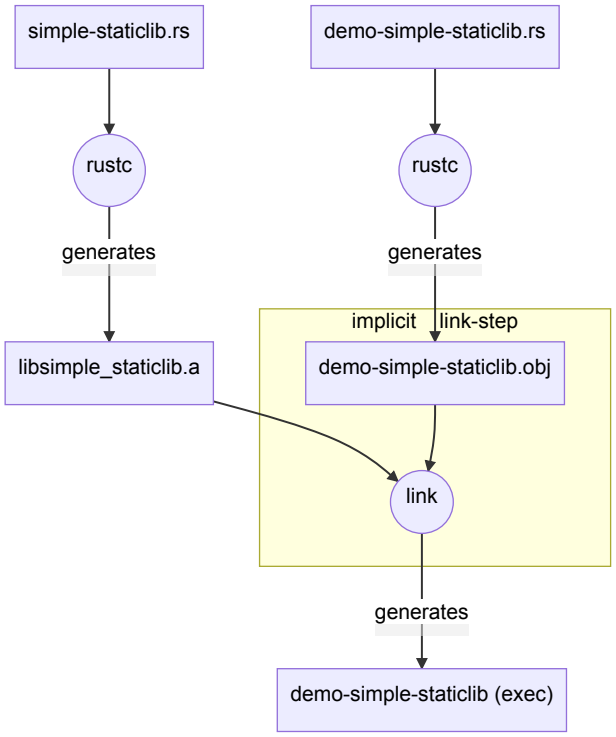
But, there are no surprises here.

Static libraries for non Rust code: `staticlib`

If you want to call Rust from another language, you can do that by compiling your Rust code into a static library, and then linking to that static library from your foreign program.

Now, for purposes of demonstration in this example, I am using Rust to implement `demo-simple-staticlib`; but, crucially, I **didn't have to**. It could have been written in C, or I could have used Java and JNI to interface with it. (Or `simple-staticlib` could have been a Python extension, *et cetera.*)

Here's what our linkage picture looks like for `staticlib`:



If we compare this against our picture for `rlib`, the main difference now is that there is no longer an arc from `rustc` to `libsimple_staticlib.a`. This is actually a pretty big difference!

1. The generated archive file, `libsimple_staticlib.a`, is *not* a Rust crate. It does not have the metadata the Rust compiler would need to intepret it as a crate. Instead, it is just another library archive, like the others typically used in a C project.
2. When compiling `demo-simple-staticlib.rs`, we will have to pass flags to the compiler that tell it to link to the static library. The compiler will no longer magically figure this out for us.
3. Since `libsimple_staticlib.a` is not a Rust crate, we have to provide explicit declarations in `demo-simple-staticlib.rs` for the functions it provides.

The differences above can arguably be summed up in: It is like you are programming in C, in terms of having to deal with keeping function signatures consistent and juggling linker flags. If you have experience with that, none of this should seem surprising.

That said, here is the source code for the files in the diagram.

```
1 // simple-staticlib.rs
2 #![crate_type="staticlib"]
3 #[no_mangle]
4 pub extern "C" fn staticlib_main() { println!("Running staticlib_main from {}", file!()); }
```

```
1 fn main() {
2     extern "C" { fn staticlib_main(); }
3     unsafe { staticlib_main(); }
4 }
```

And here are the command invocations that complete the diagram above.

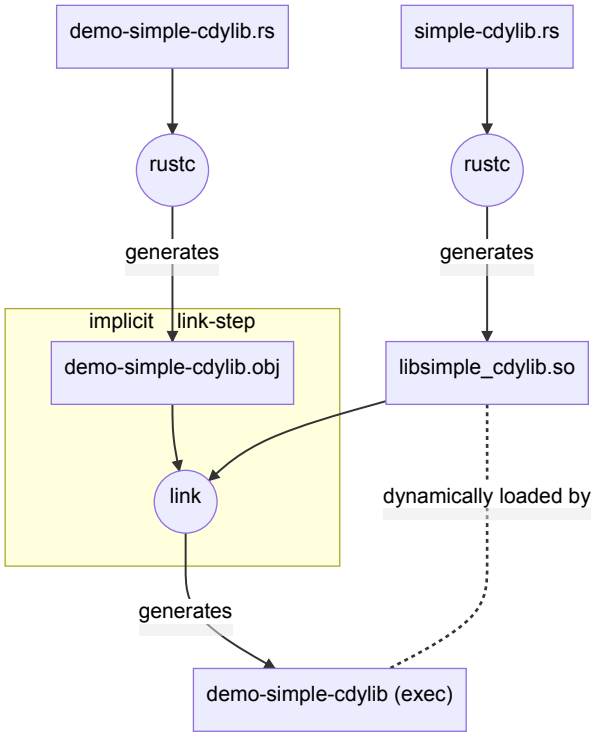
```
1 % rustc --out-dir out/ps/ss simple-staticlib.rs
2 % ls out/ps/ss
3 libsimple_staticlib.a
4 % rustc --out-dir out/bins/ps/dss demo-simple-staticlib.rs -Lout/ps/ss -lsimple_staticlib
5 % ls out/bins/ps/dss
6 demo-simple-staticlib
7 % out/bins/ps/dss/demo-simple-staticlib
8 Running staticlib_main from simple-staticlib.rs
9 %
```

Dynamic libraries for non Rust code: `cdylib`

Conceptually so far we have covered three cells in the following 2x2 matrix, and `cdylib` will finish the table.

	Linked from Rust	Linked from Non-Rust
Static	rlib	staticlib
Dynamic	dylib	cdylib

The diagram for using `cdylib` should have elements that remind you of both the table for `dylib` and the table for `staticlib`.



Namely, here we see:

1. Much like `demo-simple-staticlib`, compiling `demo-simple-cdylib.rs` does not attempt to extract metadata from the `simple-cdylib.so` or even treat it as a Rust crate; instead, one must provide the right linker flags to the compiler, and the right `extern` function signatures in the source code for `demo-simple-cdylib.rs`.
2. Much like `demo-simple-dylib`, the execution of `demo-simple-cdylib` will itself load the shared library `demo-simple-cdylib.so` and link to its code dynamically.

The source code for this demostration is just like that of `staticlib`.

```
1 // simple-cdylib.rs
2 #![crate_type="cdylib"]
3 #[no_mangle]
4 pub extern "C" fn cdylib_main() {
5     println!("Running cdylib_main from {}", file!());
6 }
```

```
1 // demo-simple-cdylib.rs
2 fn main() {
3     extern "C" { fn cdylib_main(); }
4     unsafe { cdylib_main(); }
5 }
```

The command sequence here is interesting: we are no longer forced to use `-C prefer-dynamic`.

```
1 % rustc --out-dir out/ps/sc simple-cdylib.rs
2 % ls out/ps/sc
3 libsimple_cdylib.so
4 % rustc --out-dir out/bins/ps/dsc demo-simple-cdylib.rs -Lout/ps/sc -lsimple_cdylib
5 % ls out/bins/ps/dsc
6 demo-simple-cdylib
7 % LD_LIBRARY_PATH=out/ps/sc out/bins/ps/dsc/demo-simple-cdylib
8 Running cdylib_main from simple-cdylib.rs
```

⊕ ☐ This hypothesis is one of many items that I want to follow up on in a future post. I believe this is because we end up treating the two components (`simple-cdylib.so` and `demo-simple-dylib`) as completely divorced entities: thus, they each get *their own copy* of the functions they use from the Rust standard library statically linked into them.

Conclusion

That was quite a romp!

And yet, we have not even gotten to some of the hairier stuff, like:

- mixing distinct upstream crate types into the same project,
- having multiple crate-types for the same crate available,
- using `--extern` to specify which crate type should be used for a given crate type,
- mixing crates built with and without `-Cprefer-dynamic` (how to get it to work today, and how should it work in ideal world?), or
- varying whether the program will statically or dynamically link to the platforms C runtime.

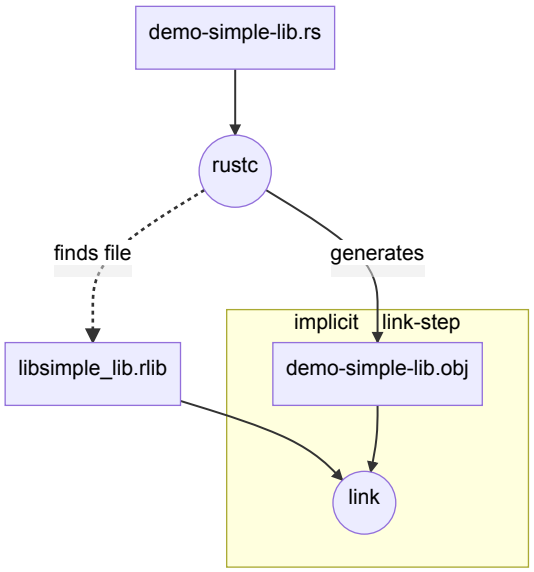
Furthermore, I did not really dig into what static linking *means*, especially when it comes to crate types like `rlib`, which explicitly *do not have* link-time dependencies. I want to elaborate on that too, preferably by demonstrating how to use `objdump` to learn things about the generated code.

So, lots of material for future posts.

(And also, lots of opportunities to try to clean up the presentation of this post.)

Appendix: Who is using that library?

Consider this diagram:



Here is an important question you should ask yourself: is `libsimple_lib.rlib` actually used by the linker? Or is it solely used as input to `rustc` (i.e., potentially used in the generation of `demo-simple-lib.obj` itself). Or is it used by both `rustc` *and* the linker?

We can test this question directly, with some slight tweaks to our commands.

Proving the link step's dependence on the library

First, we can test whether its used by the linker at all by separating the link invocation from the rest of the compiler steps, and then modifying it and running it on its own.

```
% mkdir -p out/sl out/bins/ps/dsl2step
% rustc --out-dir out/ps/sl simple-lib.rs
% ls out/ps/sl
libsimple_lib.rlib
% LINKER_COMMAND=$(rustc --out-dir out/bins/ps/dsl2step demo-simple-lib.rs -Lout/ps/sl -Csave-temps --print=link-args -Ccodegen-units=1)
% ls out/bins/ps/dsl2step
demo-simple-lib                                         demo-simple-lib.demo_simple_lib.66aa33f3-cgu.0.rcgu.o
demo-simple-lib.demo_simple_lib.66aa33f3-cgu.0.rcgu.bc  demo-simple-lib.hlsxcd1s0pxo20a.rcgu.bc
demo-simple-lib.demo_simple_lib.66aa33f3-cgu.0.rcgu.no-opt.bc  demo-simple-lib.hlsxcd1s0pxo20a.rcgu.o
% ./out/bins/ps/dsl2step/demo-simple-lib
Running main from simple-lib.rs
% rm ./out/bins/ps/dsl2step/demo-simple-lib
% eval $LINKER_COMMAND
% ./out/bins/ps/dsl2step/demo-simple-lib
Running main from simple-lib.rs
%
```

The significance of the above sequence: It runs `rustc -Csave-temps --print=link-args` which will preserve the generated object files and also print out the linker invocation it runs.

⊕ ☐ In practice when doing these kinds of experiments, you should not blindly use `eval` in the manner I have shown here, but instead echo the linker command to the screen and confirm that it is something you trust running. We can run the generated binary, delete the

binary, and then re-run that linker command ourselves (`eval $LINKER_COMMAND`), and re-run the binary again. This confirms that this linker invocation does indeed generate that binary.

With that in place, one way we could exercise the linker's use of the file: we could just delete `libsimple_lib.rlib`, and run the original linker invocation:

```
% eval $LINKER_COMMAND
% rm out/ps/sl/libsimple_lib.rlib
% ls out/ps/sl/
% eval $LINKER_COMMAND
/usr/bin/ld: cannot find /media/pnkfelix/Rust/Linking/out/ps/sl/libsimple_lib.rlib: No such file or directory
collect2: error: ld returned 1 exit status
```

Another slightly more complicated way we could expose the dependence of our object code on that file: We can remove the reference to `libsimple_lib.rlib` from the linker invocation, and run the resulting new linker invocation:

```
% NEW_COMMAND=$(echo "$LINKER_COMMAND" | sed -e 's@"-Wl,-Bstatic" .*/libsimple_lib.rlib"@@')
% eval $NEW_COMMAND
/usr/bin/ld: out/bins/ps/dsl2step/demo-simple-lib.demo_simple_lib.66aa33f3-cgu.0.rcgu.o: in function `demo_simple_lib::main':
demo_simple_lib.66aa33f3-cgu.0:(.text._ZN15demo_simple_lib4main17h9794b393d760d697E+0x3): undefined reference to `simple_lib::main'
collect2: error: ld returned 1 exit status
```

This gives you an idea of the kinds of nasty error messages you have to deal with when you start playing games with your build artifacts: The linker is rightfully complaining that the generated object code for `demo_simple_lib::main` has some reference to `simple_lib::main` (indeed,the very definition of the former is just a single invocation of the latter), and yet that reference cannot be satisfied. (It is up to the user to read that message and infer that the core problem is that the linker is no longer receiving the path to `libsimple_lib.rlib` as one of its command line arguments.)

Proving the compiler's dependence on the library

The previous section established that the linker needs the `.rlib` file in place. But, maybe that's only necessary for the link step alone, and `rustc` itself doesn't need the actual file?

We can test this theory too, in a similar form of direct experimentation on the build artifacts.

First, lets try removing the file (same as illustrated in the previous section)

```
% rm -f out/ps/sl/libsimple_lib.rlib
% ls out/ps/sl
% rustc --out-dir out/bins/ps/dsl demo-simple-lib.rs -Lout/ps/sl
error[E0463]: can't find crate for `simple_lib`
 --> demo-simple-lib.rs:1:1
  |
1 | extern crate simple_lib;
  | ^^^^^^^^^^^^^^^^^^^^^^^^ can't find crate

error: aborting due to previous error

For more information about this error, try `rustc --explain E0463`.
%
```

This shows that the compiler is *definitely* using the file `libsimple_lib.rlib` as part of its compilation of `demo-simple-lib.rs`.

If we try to force the compiler to make forward progress by giving it a dummy file, it will still complain:

```
% touch out/ps/sl/libsimple_lib.rlib
% ls -s out/ps/sl/
total 0
0 libsimple_lib.rlib
% rustc --out-dir out/bins/ps/dsl demo-simple-lib.rs -Lout/ps/sl
error[E0786]: found invalid metadata files for crate `simple_lib`
 --> demo-simple-lib.rs:1:1
  |
1 | extern crate simple_lib;
  | ^^^^^^^^^^^^^^^^^^^^^^^^
  |
  = note: failed to mmap file '/media/pnkfelix/Rust/Linking/out/ps/sl/libsimple_lib.rlib': memory map must have a non-zero length

error: aborting due to previous error

For more information about this error, try `rustc --explain E0786`.
```

Upon reflection, this all makes perfect sense: As part of compiling `demo-simple-lib.rs`, the compiler is going to reference external metadata (i.e. the function signatures and type definitions from the `simple-lib` crate).

Posted by Felix S. Klock II linkers

Tweet

« Visuals redux: Getting mermaid going

Recent Posts

- Linking Rust Crates, Part 1
- Visuals redux: Getting mermaid going
- What is Rust's Hole Purpose?
- Why I use a debugger
- Road to TurboWish part 3: Design

GitHub Repos

- cee-scape

  The cee-scape crate provides access in Rust to `setjmp` and `sigsetjmp` functionality.

- cbr-issue-284-demo

  Demo of cargo-bisect-rustc issue 284

- BinFiles

  Little scripts that my ConfigFiles and DotFiles sometimes reference. Meant to be alias of ~/bin/

- add3

  A demo project for illustrating dependency handling in Cargo

@pnkfelix on GitHub