THERE IS LIFE AFTER MAIN IN RUST

_start/main, link sections, ctor/dtor
platform cautions/notes
example: building scattered map
string interning table

Every Rust binary has one thing in common: fn main(). If you come from the C world, that might be more familiar as int main(argc, argv). Some platforms might obfuscate it a bit more (Python...) but under the hood, every binary has an entrypoint.

What might not be familiar to most developers is how you get into the main function. You see: under the hood for every language is the runtime. C has one: the C runtime that you might recognize as libc. Rust also has its own runtime: the Rust standard library. And because C is the lingua franca of runtimes for most languages†, Rust has both. ← reorder

† Go is a notable exception in that it avoids the C runtime on most platforms, but Apple requires a C runtime to access syscalls.

There's an entire ecosystem of processing that happens before the function you declared as main starts up. Rust uses this time to configure parts of the language and runtime. Your platform's C library does some work as well. And by not taking advantage of the early, highly consistent environment of pre-main, you're missing out on a useful bootstrapping time.

The runtime is responsible for accepting a handoff from the OS loader. There's a platform specific hook on every OS that accepts the handoff — to some extent this is the real main. At this point the runtime has a chance to configure itself, and the way that all runtime do this is via initialization functions.

In early iterations of runtimes, bootstrapping was a static tree of function calls: initialize file I/O, initialize the allocator, etc. As runtimes became more complex, this tree of function calls became more complex, and binary sizes increased to absorb more C runtime functionality that they may or may not need.

Linkers developed the ability to discard unused code (including that of the C runtime), and with that came a need for a replacement for the static init call trees.

The most popular method of declaring init code came from GCC: __attribute__((constructor)). The way this worked was to place a list of init functions into a chunk of the binary on disk. When the C runtime started, it could walk through each of these functions and call them, allowing various bits of the C runtime to request initialization without strongly coupling subsystems, and allowing the linker to jettison unused subsystems, init code and all.

Eventually the need for constructor ordering became urgent enough that constructors could be given a priority and run in a specific order — allowing the runtime to initialize subsystems before and after each other. E.g., the malloc subsystem might be needed for buffered I/O or name resolution.

On most platforms
^ The linker was called in to do the priority work†: each platform ended up with a way to prioritize the order in which data gets written to sections, which allowed for the C runtime to end up with a well-ordered list of function pointers. ← Add note re: reserved priority

† macOS does not supports this
†† AIX uses symbol naming (sinit/sterm)

[Example with ctor]

The process in which constructors are linked isn't specific magic, though. In fact, C compilers allow you to name the location in the binary you want to place any of your data and/or code. And by extension, Rust allows this as well. The challenge, as we will see, is making use of this organizational feature.

Linkers have been the key to C's ability to target any form of binary for some time. Most linkers allow for developers to provide linker scripts — text files that live alongside your source code (which is compiled to object files) and instruct the linker on how those object files are assembled. Using a linker script, a single C file might become a Linux executable, or a block of raw assembly that lives in the boot sector of a hard drive.

Linker scripts also allow for defining virtual symbols — that is symbols that don't exist in any source file but can be used by C code to access pointers to the underlying data in the loaded binary.

[C linker script example]

To make developer's lives easier, linkers added features where C symbols would be magically defined to point to the beginning and ends of sections. This allows a developer to both group data and code and access that data/code without touching a linker script.

[Rust example]

Given all of these tools, it turns out we can do quite a bit before main even starts.

One advantage we have in doing work before main is that it is well-behaved. No threads are running unless we start them.

We have also developed the ability to distribute or scatter "registration" throughout an entire binary. In the same way that the C runtime registers init functions, we can register chunks of constant data as complex as we need them to be.

[Registration example]

But we don't need to stop at immutable data registration. That data is just that: data. And we can mutate that data, or even reorder it! ← clarify why not

As we build more complex examples, we need to be careful about how we are accessing data to avoid UB in Rust. As such we'll switch to using the link-section crate that handles much of this complexity.

We can try to build something more complex: a string interning pool, defined entirely at link and compile time. Normally, a string pool built at runtime requires O(...). We can build a pool that allows for O(log log N) lookup with just a single O(...) sort at program start.

[String example]

With some creative use of link sections we can even build more complex data structures. For example, HashMap in Rust cannot be created as a static variable without making it Lazy, because it uses internal allocation for storing the buckets and indexes. What if we could create a map with zero allocations and do a small bit of work at startup to "finish" it?

[Map example]