Chapter 01

1 Tell me more about Rust!

1.1 A bit of history (skip it)

Initially, Rust started out in 2006 as a personal project from Graydon Hoare—a language designer "by trade" to quote his own words. Developed in his free time during the first three years, Rust was then presented to his employers at Mozilla, which went on and started to actively support it. Indeed, Mozilla Research caught interest on it as something which could do away in the future with various recurrent bugs found in Gecko, Firefox's rendering engine written in C++.

An original compiler has been written in OCaml until a bootstrapped compiler written in Rust itself could take over, in 2010. Right now the compilation process uses an LLVM backend (which you might have seen in action inside the Clang/LLVM C Compiler).

1.2 The Goodness

So what's so good about Rust? We talked about C++ right at the beginning and you might think that since Mozilla is actively leading it's development, it should actually have some significant advantages compared to it.

First, what are people looking for when they want to use C++?

- A System programming language, which means something that doesn't compromise on performance and allows for low-level access to the available resources (pointers, manual memory management, etc.)
- A language with a fairly high-level of abstraction and a large feature-set (OO-programming is one of them) that stays fast and has broad compatibility — along with a large feature set

Now, what are the main features that Rust brings on top of that?

- Safety (and mostly: safe pointer types thanks to a very efficient memory model, no void* heresy, no use-after-free, mem leaks,…)

- Native actor-based concurrency inspired from Erlang. Yup, that's a big deal of today's programming languages with the rise of multicore processors; in fact, some sort of multithreading is already there on C++ Boost and maybe in C++14 at a later date! —actually, the actor implementation of multithreading consists of explicitly sending message from a lightweight thread (`chan`) and receiving it from another (`port`); its a safety/efficiency compromise

- Inspirations from Haskell and OCaml, not only for the functional programming paradigm but also for the type classes (`type`, `impl`)

- Some easier to use functionalities from Python —that being said, Rust doesn't comprise on safety: it has static typing (with type inference!)

- Optional tracing-GC (WIP) —right now the `std::gc::Gc` type recently landed on mainline

Rust doesn't try to be a new fancy thing; rather, it builds up on top of proven useful technology and merges features from some existings languages, noticeably: Erlang's green threads, Haskell trait system and overall functional programming elements of a few languages (it has pattern matching!), some syntax sugar from Python (the `range` iterator is one of them) and it also resurrects a few features from some American research languages developed in the 90s, amongst others. As you can see, along with some features that definitely set it in today's rollup of programming languages, Rust is also the result of a true willingness to reuse what's been left by the rich history of programming languages, rather been building up the wheel over again. ;) ~ as you might have seen, that's where the name "Rust" comes from!

Actually, Mozilla is already using it in an experimental browser rendering engine that's being developed simultaneously to Rust and is at the core of the team's pragmatic development approach, just like D's—that is, the developers implement features and sometimes change them based on their experience with it; besides, they also take feedback (and contributions!) from the community.
Remember, C—today's most successful language has been built in concert with the UNIX OS, allowing adaptations from what happened during the OS development process. Right now, Rust is still in alpha state which means that developers can still make breaking changes to the API (but don't worry, it's converging towards a stable state which will ultimately happen in 1.0).

You can have a shot at Rust's performance evolution right here.

There have been other attempts at providing such a new System programming language (e.g. D, Nimrod, Go—although that one doesn't really follows a

low-level scheme†).
Right now, it looks like Rust has a big chance to set itself apart because:
- it's pretty innovative! —seeing how it reuses a lot of features that have been developed independently, but make sense together
- it doesn't force you to use GC, it has low-level features but also lots of sugar for the developer
- it's backed by Mozilla, not bad if you're trying to set a standard!

†Go has no pointers and mandatory GC which isn't necessarily a bad thing, it just serves a different purpose. Google originally developed it for creating web-apps but it can be useful for all kind of things.

1.3 Appendix: Elaborating on C++ and safety

You might think: why were these even allowed in the first place? To quote Tony Hoare, the `null_ptr` precursor:

> I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.
> — Tony Hoare

C++ is a mixed-breed: it has been developed from the beginning to build up on top of C with new features that were emerging in the field of programming. Today, it's one of the Top Five languages because it's been fast at an higher level of abstraction (also complexity!) than the others.
Looking back, we realized that the time spared by relaxed programming APIs was often creating much bigger problems down the road—because it's always less time consuming and more convenient to just avoid these programming derps than debugging your program down the road.

Rust trys to ensure safety with some additional syntax markup (which actually makes you to think unambiguously about your desired behavior—for example, it has 2 safe pointer types) that allows for a way more fine-grained static analysis by the compiler and can lead to additional compiler optimizations—yea, some unsafe code that would pass as valid in most of todays's most used languages will get catched at compile-time, otherwise some instructions that aren't safe or cannot be statically checked will have to be located inside an `unsafe` code block (that's mostly for the FFI, that allows you to link your programs to C).

1.4 Appendix: Elaborating on Servo

What's Servo? Well—nothing to do with mechanics: it's Mozilla's experimental browser rendering engine that's been mentioned before.

So, in a way, you could say that it's their biggest hope and biggest motivation for supporting Rust's development. Safety. Parallelism. (And Performance.) It's development started in early-2012 (they like contributions too!) and it's evolving along with Rust.

Servo is built on a modular architecture and each engine component is separated from the other in an effort to have parallelized and easy-to-maintain code. Since the last 3rd April, Mozilla and Samsung are collaborating in the project development. Samsung's work is essentially maintaining ARM/Android support, for both Rust and Servo.

Note that right now, Mozilla has no plans to integrate Servo onto Firefox, ie. it is as of now a research project. Well it's still in early stages of development anyways. BTW, Servo passed Acid1 recently!