Chapter 01

1 Tell me more about Rust!

1.1 A bit of history

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 in 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 Clang).

1.2 The Goodness

So what is so good about Rust? We talked about C++ right at the beginning and you might think that since Mozilla is actively leading its 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 it does not 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

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. This is a big deal for today's programming languages with the rise of multicore processors; in fact, some sort of multithreading is already there on C++11 (2011) and will probably get extended in C++14. Actually, the actor implementation of multithreading consists of explicitly sending message from a lightweight thread (`chan`) and receiving it from another (`port`); it is a great 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 does not compromise on safety: it has static typing (with type inference!)

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

Rust does not 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 concepts from the rich history of programming languages, rather than building up the wheel over again. ;) ~ as you might have seen, that's partly where the name "Rust" comes from!

> I was fiddling with a number of names. Primarily it's named after the pathogenic fungi which are just amazing multi-lifecycle, multi-host parasites, fascinating creatures.
>
> But the name "rust" has other nice features. It fits with our theme of being "close to the metal" and "using old well-worn language technology". It's also a substring of "trust" and "robust". What's not to like?
>
> – Graydon Hoare

Actually, Mozilla is already using it in an experimental browser rendering engine that is 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 not to worry, it is converging towards a stable state which will ultimately happen in 1.0).

There have been other attempts at providing such a new System programming language (e.g. D, Nimrod)—Go also gets mentioned sometimes but it has been developed for apps/webapps and is oriented towards simplicity and fast compilation-speed.
Right now, it looks like Rust has a big chance to set itself apart because:

- it focuses on safety without compromising performance (D requires GC for safety, and Nimrod does not ensure thread safety), it has low-level features but also lots of sugar for the developer

- it is pretty innovative! —seeing how it reuses a lot of features that have been developed independently, but make sense together

- it is backed by Mozilla, not bad if you are trying to set a standard!

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 is one of the Top Five languages because it has been fast at a 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 is always less time consuming and more convenient to just avoid these programming derps than debugging your program down the road.

Rust tries to ensure safety with some additional syntax markup (which actually makes you to think unambiguously about your desired behavior) that allows for a way more fine-grained static analysis by the compiler and can lead to additional compiler optimizations—indeed, some unsafe code that would pass as valid in most of today's most used languages would get caught at compile-time in Rust. Instructions that are not safe or cannot be statically checked will have to be located inside an `unsafe` code block (that is mostly for the FFI, that allows you to link your programs to C).

1.4 Appendix: Elaborating on Servo

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

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.)
Its development started in early-2012 (they like contributions too!) and it is 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 April 2013, Mozilla and Samsung are collaborating in the project development.

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

% Chapter 02

2 Few things you should know

The Rust API is frequently evolving. While the use of the 0.8 release is recommended on Windows (unless you're willing to build it), this tutorial will be aimed at current master which has the latest language contents.
Note that much of the "standard" features are pretty much stabilized and shouldn't change much. If you have any concern related to the language or its syntax, just jump on IRC and ask the others!

3 Installing Rust

3.1 Windows

Checkout the building instructions.
Note: `mingw-w64` support is incoming on latest trunk.

If you're not willing to go through these steps, just download the 0.8 installer.

3.2 Mac OS X

The installation process on Mac OS is fairly easy with the Homebrew package manager, for the latest trunk:

```
brew install --HEAD rust
```

And to get the latest release:

```
brew install rust
```

3.3 Linux

Okay Linux users, let's build the beast:

```
git clone https://github.com/mozilla/rust.git
cd rust
./configure
make
sudo make install
```

And for the 0.8 release, use these sources:

```
curl -O http://static.rust-lang.org/dist/rust-0.8.tar.gz
tar -xzf rust-0.8.tar.gz
cd rust-0.8
./configure
make
sudo make install
```

Note that there are repositories with nightly packages for Arch and Ubuntu.

4 Get your code editor ready

You are about to write your first Rust program; first, just create a `.rs` file with your favorite text editor. You are going to build it with `rustc`, the Rust compiler.

```
# create a new file
touch hello.rs
# now compile it...
rustc hello.rs
# ...and run!
./hello
```

Note: on Windows, you will have to call `rustc.exe` for now due to issue #3319.

BTW, you can get syntax highlighting for vim, gedit or other highlighting solutions on the main repo.

5 And so it begins!

5.1 The starting point

So, you are learning a programming language... you know what this means?

```
fn main() {
    println("Hello World!");
}
```

Yes, it is that simple. As you can see, functions are declared using the generic `fn` keyword (for the C habitués, the return type–if any–is specified as in ML, after an ASCII arrow e.g. `fn random() -> int { ... }`).

To out your own, save this code snippet and call `rustc`!

Now let's try something funny:

```
fn main() {
    let out = "Hello World!";
    for c in out.chars() {
        /* spawn concurrent tasks! */
        do spawn {
            print!("{}", c);
        }
    }
}
```

5.2 The looping point

First, notice the loop syntax: for (each) `element` in `container`, as in Python. `element` should be `_` if you are not using it (no referencing).
Here, we make use of Rust's functional paradigm by using `.chars()` to iterate over the chars of our string (it used to be `.iter()`): our iterator will successively host each char of the string, so we can print those directly.
What if we wanted to simply iterate a said number of times? Well, for that we have a special container, the `range()` iterator.

```
for i in range(n) /// Creates an iterator that will do `n` times, from 0 to n-1
for _ in range(1,n+1) /// This one will go from 1 to n
/// Notice the joker `_`, that's if we don't want to reference the iterator for use in t
```

Now, before first looking at the type syntax itself, let's look at the output:

```
Hllo World!
e
```

Well, we did `spawn` a few concurrent tasks. Things are done asynchronously, which means that one of the tasks may unpredictibly end-up before the other. Anyways, spawning threads to print out the letters of a sentence is pretty silly!

BTW, speaking of loops: `while` uses the usual syntax: `while <condition> {}` and the `loop` keyword is to be preferred to `while true {}`, which is essentially a hack.
Rust doesn't have the `do {} while;` syntax, we just use a `while` with a condition instead.

5.3 Variables and types

As you can see, variables are declared via the `let` keyword and you do not need to specify the type (ML anyone?) because Rust has type inference, which means that the variable type is determined automatically at *compile time* —not to be confused with dynamic typing, as in Python, where the variable type gets determined at run-time and can be modified.
Let's look at some of the primitive types:

```
let a = 3; // considered as `int` by default
let b = "code"; // that would be an `str` - string
let c = [1, 2, 3]; // and that's a `vec` - array
let d = true; // bool
```

Rust variables are immutable by default in order to avoid some common errors, so you need to append the mutable `mut` keyword to `let` in order to reassign a value. The compiler will warn you if there are unused `mut`s in your code.
BTW: unlike C for example, Rust does not let you edit an immutable variable via its pointer.

Speaking of type assignment, you can force the use of a particular type:

```
let a: uint = 3; // unsigned int
let a = 3u; // same as the last line, this is a short variable suffix for convenience

let mut n = 3; // `n` is supposedly an `int` from here...
n = 5u; // but is assigned an `uint` here, so the compiler will consider `n` as an `uint
let mut h = ~[]; // unknown-type `vec` - won't compile as such
h = [1u, 2, 3]; // a `vec` of `uint`s
```

There are other suffixes like `i32` (32-bits integer), `i64`, `f32` (32-bit float), `f64`... the empty type is called nil and annotated `()`. But remember, you cannot change the type of a variable during it's lifetime—static typing!

Functions need explicit parameters and return types. Let's look at one:

```
fn abs(x: int) -> uint {
    if x > 0 { x }
    -x
}

fn square(x: int) -> int {
    x*x
}

fn main() {
    let mut x = 3u;
    /* Now, calculate the square. */
    x = square(x);
```

```
    println(x);
}
```

Ok, but this code will warn because we are passing an unsigned int to a function that takes int as parameter; if we were in a situation where we cannot change the type annotation, we would have to just cast the value of `x`. Rust has 2 ways of casting: `x.to_targetType();` and `x as targetType;`. The `to_*` method is to be preferred for performance.
Additionally, there's a third way of casting, into.

In Rust, instructions that do not end with semicolons are expressions (not statements), meaning that they will return their value to the higher-level block.
This allows you no only to return from a function but—for example to allocate a value based on a test's result. This is valid Rust code:

```
let hype =
    if cake == "KitKat" {
        10
    } else if cake == "Jelly Bean" {
        7
    } else {
        5
    };
```

5.4 A last note on types and prints

Let's go back on a similar example as with the `spawns` before, this time using a `vec` of `chars`:

```
fn main() {
    let out = ['H', 'e', 'l', 'l', 'o'];
    for c in out.iter() { // iterate over `out`
        print!("{}", c);
    }
}
```

Strings and pack of char's are essentially the same in memory, but strings have specific methods and are plain better at handling a bunch of `chars`.
FYI, there's a small article that talks about types in Rust and has interesting facts, check it out!

So Rust has a few printing functions, the standards `print/println` which want a `string` as input but also `print!/println!` macros (or syntax-entensions—ending with `!`) which will format-print your variables marked as `{}` and passed as arguments into strings.
BTW, you can specify the type manually, e.g. `{:u}` says this is an unsigned integer literal.

5.5 Appendix: Going back on macros

About the macros: they are syntax extensions in the sense that they are like scripts, but are evaluated at compile-time. So they serve only the developer by simplifying/automating things. Think about `#define` in C, but much more powerful. BTW, macros have a few syntax elements of their own, for example, variables get referenced as `$var`.

Not elaborating further on it for now but you can have a look at the mainline macro tutorial for some code examples.

5.6 Appendix: Code comments

You will probably start quickly adding comments to your code (well, you should anyways!) so let's speak about that now.
Rust uses C-style comments: inline with `// yuck!` and multiline with `/* yuck! */`. Rust also has a built-in doc generator (librustdoc); doc comments get referenced as such with either `///` or `/** **/`; check out how it works for the standard library!

You can learn more about rustdoc here.

5.7 Appendix: on functional programming

The functional capatibilities of Rust are powerful as it makes use of type-specific operations.

```
let my_vec = [-1, 0, 1];
for i in my_vec.iter().invert() {
    print(i.to_str());
}
```