Rust tutorials

http://adrientetar.github.io/rust-tuts

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. Rust allows the use of either green (M:N) or native (1:1) threads.
- Inspirations from Haskell and OCaml, not only for the functional programming paradigm but also for the type classes (`trait`, `impl`)
- Some programmers friendly features from Python – that being said, Rust does not compromise on performance: it has static typing (with type inference!)
- Optional tracing-GC (WIP) – the `std::gc::Gc` type recently landed on mainline

As you can see, 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 80s, 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 more 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 (the syntax is modern and not overly verbose)
- 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 afterwards.

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, which 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, i.e., it is as of now a research project and is still in early stages of development. BTW, Servo passed Acid1 recently!

2 Few things you should know

The Rust API is frequently evolving (read: improving), so this tutorial will be aimed at current master which has the latest language contents.
Note that most of the "standard" features are pretty much stabilized and should not change much. You can choose to use the latest release, but as it gets older, there will probably be numerous changes onto the latest trunk. If you have any concern related to the language or its syntax, just jump on IRC and ask the others!

The latest release is Rust 0.9, released on January 9th 2014.

2.1 Installing Rust

2.1.1 Windows

You can get snapshot builds for Windows through the NuGet Package Manager console:

```
Install-Package Rust
```

Or have a look at the build instructions if you want to build it yourself.
Note: `mingw-w64` should fix a few things compared to the regular `mingw` (it works on both 32- and 64-bit systems, despite the name).

If you are not willing to go through these steps, just download the 0.9 release.

2.1.2 Mac OS X

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

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

And to get the latest release:

```
brew install rust
```

2.1.3 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 0.9 release, use these sources:

```
curl -O http://static.rust-lang.org/dist/rust-0.9.tar.gz
tar -xzf rust-0.9.tar.gz
cd rust-0.9
./configure
make
sudo make install
```

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

2.2 Get your code editor ready

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

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

Using the `-O` switch will add the default optimization level to the code.

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

3 And so it begins!

3.1 The starting point

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

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

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

To try it out on 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! */
        spawn(proc() {
            print!("{}", c);
        });
    }
}
```

3.2 The looping point

First, notice the loop syntax: for (each) `element` in `container`, as in Python.
Here, we make use of Rust's iterator type with `.chars()` to successively host each char of the string (it used to be `.iter()`) 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.

```
// Yields an iterator that will do 'n' times, from 0 to n-1
// Note: using just 'range(n)' is not possible just yet
for i in range(0, n) {}
// This one will go from 1 to n
for _ in range(1, n+1)
```

As you can see, `range()` has exclusive upper bound, to be in line with 0-based indexing.
Notice the wildcard _ in the second example, which is used if we don't want to reference the iterator for use as a local variable in the loop.

BTW, let's look at the conditional `while` loop right now:

```
let mut cake = have_my_cake();
while cake > 0 {
    cake -= 1; // eat it
}
```

Note that for infinite loops, the `loop` keyword is to be preferred to `while true { }` (which is essentially a hack). Also, Rust doesn't have the `do { } while;` syntax; you can just use a `loop` with a breaking condition instead – `break` will exit a loop while `continue` will jump to the next iteration.

Now, before 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 unpredictably end up before the other. Anyways, spawning threads to print out the letters of a sentence is pretty silly!

Note: `proc()` is a one-shot closure, the latter being a nested function that captures its surrounding environment (i.e. one can access the variables of the containing namespace).

3.3 Types and assignment

As you can see, variables are declared via the `let` keyword and type specification is optional (~ML) because Rust has a type inference mechanism, which means that the variable type is determined at *compile time* – not to be confused with dynamic typing, as in Python, where the variable type is managed at run-time and can be dynamically modified.

Note: the convention is that variables and function names are_lowercase while type names are CamelCase.

Let's have a 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 (of fixed-length)
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.
Note: 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; // that’s the same, using a short variable suffix for convenience

let mut n = 3; // ‘n‘ is supposedly an ‘int‘ here...
n = 5u; // but the compiler will infer it as an ‘uint‘ from here
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-bit 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 will need explicit parameters and return types through. Let's look at one:

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

```
fn main() {
    let x = 3u;
    /* Now, calculate the square. */
    let y = square(x);
    println!("So this gives us {}.", y);
}
```

Ok, but this code will not typecheck because we are passing an unsigned int to a function that takes an 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 various ways to change variable types, and while they all share different characteristics, they are also not all possible with all type combinations. Here are the general rules:

cast	`as T`	ref change-no allocation	read-only	`&'a T -> &'a U`
conversion	`.to_T()` `.into_T()`	allocates new memory	“in-place” conversion, without copying	`&T -> ~U` `~T -> ~U`

You can see `as` as a sole modified reference to a value in memory, while others change the in-memory representation of the converted variable.
A research on the libstd documentation will tell you which types implement which of these methods – in the present case, we would just use a `as` cast, i.e. `x as uint`.

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 not only to return from a function but also – 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
    };
```

3.4 A last note on types and prints

Let's go back on a similar example as with the `spawn` 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);
    }
}
```

Rust supports UTF-8 (Unicode) natively through the u8 type (8-bit unsigned type); strings are essentially just ~[u8] arrays with a few specific methods on top of it.
FYI, there is a blog article that talks about types in Rust and has some interesting facts, check it out!

So Rust uses the `print!` and `println!` macros (also called syntax extensions, ending with `!`) in order to format-print your variables marked as `{}` and passed as arguments into strings.
Note that you can specify the type manually, e.g. `{:u}` means unsigned integer literal; this will permit compiler type-checking.
Also, `{:?}` will conveniently raw-print any kind of variable (in particular, arrays).

3.5 Appendix: Going back on macros

Macros are defined as 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 macros are much more powerful. Macros have a few syntax elements of their own; for example, variables get referenced with `$var`, so that you can differenciate macro variables – which serve “internally” – from actual, real Rust code.

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

3.6 Appendix: On functional programming

Iterators have methods such as `.map()`, `.zip()` and `.filter()` – a touch of functional programming which can combine and/or reuse iterator’s values in various ways. So one can write something like `iterator.map(...).zip(...)` which manages to apply a function to each of the iterator’s element (map) and append another iterator to it as a pair of elements (zip).

Here is an example using fold, which can combine an iterator’s values:

```
let xs = [14, 1, 5, 3, 12];

// this yields -35
let result = xs.iter().fold(0, |accumulator, item| accumulator - *item);
```

This approach is very practical as it keeps code concise and manages to do various kind of operations on variables quite easily.

Check out the iterators tutorial to find out more.

3.7 Appendix: Comments and documentation

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 and also has a doc comments system (librustdoc) that’s quite similar to Javadoc, for example.

	Code comments	Global doc comments	Block-level doc comments
inline	`// yuck!`	`//! foo`	`/// bar`
multiline	`/* yuck! */`	`/*! foo */`	`/** bar */`

The built-in doc generator (librustdoc) will make pretty HTML documentation out of these: check out how it looks with the libstd documentation!
All comments accept the Markdown syntax and code snippets inside backticks, like so:

```
/*
 * This file is part of my software,
 * licensed under CC0 public domain license.
 */
//! Okay, this file is about providing an
//! interface to the serial device I’m using.
//! All relevant functions are here.

/// This functions returns the current status.
/// true: ok
/// false: err
///
/// # Example
///
/// “‘
/// if !reachable() {
///     fail!(”Device not responding!”);
/// }
/// “‘
fn reachable() -> bool {
    foo
}
```

You can learn more about rustdoc intrinsics here.

Hmm, do you see that `fail!` macro in the example above?
It is part of a stack of debugging macros, we will have a look at that in the next chapter.

4 Testing, Logging, Matching… so wird's gemacht!

It might seem a bit early to talk about those topics but it is a good spot for Rust because these features are often used while writing initial code. We will also talk about error handling, which you will encounter sooner or later as it is quite different from most of today's programming languages.

This chapter will not discuss things like debugging running executables (e.g., with Valgrind).

4.1 Tests and linkage attributes

Okay so first, Rust packs a built-in testing/benching framework – an approach that might seem familiar to people coming from Ruby.

In order to use it, you will need to preappend the `#[test]` compiler attribute (notice the # and brackets syntax) to a function that will serve as test.

Here is one:

```
#[test]
fn try_fetch() {
    if external::input() == -1 {
        fail!("Couldn't retrieve the result!");
    }
}
```

These functions – just like blocks or statements marked with the `#[cfg(test)]` compiler attribute – will not be part of the default compiled code. This way, you can keep your files tidy by placing the tests relating to a specific part of your program at the bottom of the corresponding file.

4.2 Doing it wrong

Test functions are blackboxes: they have no arguments nor returns.
Besides `fail!` that halts the program and displays the error message passed as argument, tests are often made of assertions: the `assert!` macro takes a boolean expression as parameter; it does nothing if it evaluates to true and fails when it's false. `assert_eq!` is a variant takes 2 variables as parameters and asserts equality.

```
use std::ascii::OwnedStrAsciiExt;

#[test]
fn lowercase_ascii() {
    let tmp = ~"ME";
    assert_eq!(~"me", tmp.into_ascii_lower());
}
```

Now, we just have to call `rustc –test` with the name of our file, and run it.

That is how the output looks like (when everything is fine, that is):

```
running 1 test

test lowercase_ascii ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured
```

There is also `#[bench]` for benchmarking certain functions. These features are largely used in the Rust compiler itself to avoid any regression during the development process.

Note: if you are running your program in debugging mode (that is, with `RUST_LOG` cmdline parameter defined to level 1/2/3/4), `error! warn! info! debug!` will get triggered depending on the logging level.

4.3 Moving on about error handling

Rust does not have “opt-in” error handling (well, it trys to avoid opt-in safety measures when possible); that is, it works with the use of an `Option` wrapper directly in the function declaration.

Here are its possible values as defined in an `enum` (C-like cases enumerator):

```
enum Option<T> {
    Some(T),
    None
}
```

Note: if you are coming from C++, you will recognize those <> which denote polymorphism, i.e. a method that applies to various variable types (T being a lambda type). We will go back on those notions and on the `trait` concept in a later chapter.

So, an `Option` type encapsulating a certain type of variable will return `Some()` if a value gets catched or `None` if nothing passes through it.

Now let’s go about doing error handling, what we must do is destructuring those types so that we can make use of the value if we get one and also handle the case where we would not get anything returned.

If you are a C user, you are probably thinking about the use of the `switch` pattern?
Well, Rust has a more powerful tool called `match`, for pattern matching.

The mainline Rust documentation has a good example to show its most basic innards:

```
match my_number {
    x if x < 0 => println!(”something lower than zero”),
    0          => println!(”zero”),
    1 | 2      => println!(”one or two”),
    3..10       => println!("three to ten"),
    _           => println!("something else")
}
```

Something that must be noted is that `match` must handle all possible cases for it to compile.
The wildcard _ means: "for all (other) possible cases", so you will often have to use it as the last option, unless you are, for example, matching over all the individual components of an `enum`.

Here is how we would match against an `Option` value:

```
match optional_arg {
    Some(val) => val,
    None => () // 'nil', the empty type
}
```

If you remember, about assignments:

- We can assign a variable from a match statement, as shown before
- Otherwise, the pattern above will behave as an expression, i.e. as if we had written a `return` statement

We said that `match` was not the only way to destructure an `Option` type before, and there is another way involving Rust's functional capabilities.

```
let optional_arg: Option<int> = get_option_val();

// this will remove the 'Option' wrapper,
// i.e. return the value contained in 'Some()' or fail
optional_arg.unwrap();

// this will unwrap or fail with the specified message in the 'None' case
optional_arg.expect("No value returned!");
// this will return the value passed as parameter when encountering 'None'
optional_arg.unwrap_or(3);
```

Note: as you can see, error handling is not a big deal in the sense that you can simply call an `.unwrap()` if you do not care much about the failing case.

This does not give you as much possibilities as with `match` but it covers the common use case of `Option`. There is also `.is_some()` or `.is_none()` if you want to do conditionals.

4.4 Appendix: Getting decent `Results`

There is a variant to the `Option` type, called `Result`:

```
enum Result<T, E> {
    Ok(T),
    Err(E)
}
```

It is quite different from `Option`:

- As you can see, it returns an error message when it fails, which implies differences with `Option`…
- In fact, `Result` will handle cases when computation errors can occur (and even allow you to do different things depending on the error message) while `Option` handles cases when an argument can be missing, it's also used to pass NULL pointers to C, and so on…

Similarly to `Option`, `Result` has the `is_ok()` and `is_err()` methods.

4.5 Appendix: Can you `match` it?

Pattern matching is pretty powerful, noticeably when coupled with the use of tuples.
Like in Python, tuples are heterogenous lists of elements (that is, with different, unsequenced element types).

Tuples come also in handy, for example, when returning multiple values from a function:

```
fn main() {
    // type is ambiguous here and needs to be annotated
    let f: f32 = 1.0e-2;

    let (mantissa, exponent, sign) = f.integer_decode();
    println!("{}", mantissa);
}
```

As an example, here is an implementation of FizzBuzz using pattern matching:

> Print numbers from 1 to 100, but for multiples of three print “Fizz” instead and for the multiples of five print “Buzz”; for multiples of both print “FizzBuzz”.

```
fn main() {
    for i in range(0u, 101) {
        match (i % 3 == 0, i % 5 == 0) {
            (true, true)   => println!("Fizz Buzz"),
            (true, false)  => println!("Fizz"),
            (false, true)  => println!("Buzz"),
            (false, false) => println!("{}", i)
        }
    }
}
```

If you want to see more on this, there is a whole article about various ways to implement this program in Rust.

Note: you can match through `ref x` instead of `x` to get a reference instead of a value (i.e. get a pointer referencing `x`) – some functions will need referencing.

5 Guess what…

Okay, now let's do some lab work. You should (almost) have the knowledge to implement a classic litle game: guessing a secret number.

> The computer (randomly) chooses a number between 1 and 100 and the user must find it; the computer will tell whether the given number is too low/high until the user finds the right one.

Alright, let's have some prerequisities beforehand.

The first thing we will need is to have a random number in our hands.

5.1 Getting some RNG

We want an integer random number, bounded between 1 and 100. Let's print a few of them:

```
// ‘use‘ imports methods into namespace
use std::rand::{task_rng, Rng};

fn main() {
    let mut cur;
    for _ in range(0, 11) {
        // nb: range upper bound is exclusive
        cur = task_rng().gen_range(1u, 101);
        println!(”{}”, cur);
    }
}
```

Then, we will have to get input from the user. Unlike `print` and friends, input methods are not built in Rust programs by default, i.e. they are not in the prelude.

5.2 Importing from the standard library

So, we will have to import the relevant `std::io` methods manually, with the `use` keyword that is just like `import` in Python or Java. `#include` in C is quite similar too althrough C works only on a per-file basis.
Contrariwise, Rust uses the concept of module. Similarly to Java's packages, each file is a module, but not only: you can declare modules with the `mod` keyword within a file, and each one has a different namespace, which means that you will have to go through logical paths (`::`) of modules when importing methods.

Module names are lowercase by convention.

Note: modules are superseded by `crates`, which are individual compilation units (one gets compiled at a time, e.g., one library on a project makes one crate).

```
// We can select different functions with ‘{}‘.
// Note: ‘use‘ must precede ‘mod‘ statement.
use foo::{bar, baz};

mod foo {
    pub fn bar();
    pub fn baz();
}

bar();
baz();

// Now, import ‘baz()‘ fn from ‘bonus.rs‘.
mod bonus;
bonus::baz();
```

Note: the `priv` and `pub` attributes permit to allow/disallow access to a specific module of function.

So, back to the stdio things:

```
use std::io::stdin;
```

If you are looking for a specific function to import at some point, just look in libstd documentation or ask your way on IRC.

Note: if you wanted to import from external libraries you would have to link to them using `extern crate`. For example, the Rust filestack has an `extra` library for non-core things, which you could link to using `extern crate extra;` before your `use` statement.

5.3 Calling the user… err?

Alright, now let's call `read_line()` to get input from the user. This method makes use of the `Option` type we just saw.

```
use std::io::BufferedReader;
use std::io::stdin;

fn main() {
    println!(”Type something:”);

    let mut reader = BufferedReader::new(stdin());
    // Note: input will end with '\n'
    match reader.read_line() {
        // Handle empty input too
        Some(~"\n") => println!("\nI never asked for this..."),
        Some(thing) => println!("\nYou typed: {}", thing),
        None        => println!("\nWell, that's unexpected!")
    }
}
```

On a side note: if you want a message and an input on the same line, it will take a direct access to `stdout`.

```
use std::io::stdout;

stdout.write("Type something: ".as_bytes());
println!(reader.read_line().unwrap());
```

Now we want to get an int out of `stdin`, but looking at the empty case, the program could behave strangely depending on what we are doing with it. So instead we could ask the user back for input (by a recursive call to our input function, for example) or stop with a specific error message.
Here is an example that uses a `loop`:

```
use std::io::stdin;

fn number_input() {
    println!("Please type a number: ");

    loop {
        // We need to trim the EOL '\n' char to be able to convert
        match from_str::<uint>(stdin().read_line().trim_right_chars(&'\n')) {
            Some(x) => return x,
            None => println!("I'd rather have a number.")
        }
    }
}
```

Okay, that should be all you need. Just do it by yourself and the compiler outputs first before looking at the doc!

Let's add something to the game: we will count how many times the user tries and display it at the end.

5.4 The Solution (well, an implementation of it)

So here it is:

```
use std::io::BufferedReader;
use std::io::stdin;
use std::rand::{task_rng, Rng};

fn input_line() -> ~str {
    let mut reader = BufferedReader::new(stdin());
    loop {
        match reader.read_line() {
            Some(~"\n") => println!("\nUhm, please type something..."),
            Some(thing) => return thing,
            None => continue
        }
    }
}

fn input_number() -> uint {
    println!("Please type a number: ");

    loop {
        match from_str::<uint>(input_line().trim_right_chars(&'\n')) {
            Some(x) => return x,
            None => println!("I'd rather have a number.")
        }
    }
}

fn main() {
    println!("The computer is choosing a secret number, between 1 and 100...");
    let nbr = task_rng().gen_range(1u, 101);
    let mut cpt = 0u;

    println!("Can you guess it?");
    loop {
        cpt += 1;
        match input_number() {
            x if x < nbr => println!("Too low!"),
            x if x > nbr => println!("Too high!"),
            _ => { println!("Yes, you found it in {} tries!", cpt); break; }
        }
    }
}
```

We have `input_line()` that takes a non-empty stdin input and `input_number()` that ensures a non-empty converted number.

6 Getting some pointers

Rust is quite particular when it comes to pointers: it keeps a high-performance scheme while ensuring safety use patterns (no dangling pointers, etc.).

We will also talk about *ownership* and *lifetimes* in this chapter.

6.1 Introduction

First, let's look for a second at how things are laid out in memory. There are two important memory pools, the **stack** and the **heap**.

6.1.1 Simply put

Functions are individual, reusable pieces of code. If we want to directly edit a variable from the originating (caller) namespace, we need to pass its memory address, i.e. a pointer to the said variable so that the function knows where to find it and access it.

Also, when passing huge structures of data you might want to pass a pointer to it to save on memory copies but that is a quite sparse use-case as we will see.

6.1.2 On stack, and heap

Each application uses a pool of memory called the **stack**, of fixed-size, allocated when a task starts up. There are various mechanisms occuring inside the stack but it is mostly hosting all *local variables* and parameters used along the execution. When a function is called, the existing registers are saved on memory and the program jumps to the function and creates a new stack frame for it. The stack also traces the order in which functions are called so that function returns occur correctly. The default stack size of a Rust task is 2 MiB.
So, we need to pass a memory address in order to be able to edit a variable from the caller's namespace – that is the main reason why we use pointers.

A program can also dynamically request memory with the unused memory in the computer, managed by the OS: this is called the **heap**. All dynamically-sized types (DST) are stored on the heap with an OS allocation.
The heap can only be accessed via a pointer located on the stack since it is not the default memory pool, these pointers create a **box**.

So, an `int` has a fixed-size in memory and is stack-allocated. A dynamic-array `~[]`, a string `~str` or anything ~ is allocated on the heap (that is where you would use `malloc` in C or `new` in C++).

6.2 Different kinds

Rust has two primary pointer types: the borrowed reference (using the & symbol) and the owned pointer (indicated by ~).

6.2.1 Referencing

Referencing is also called borrowing because creating a reference freezes the target variable, i.e. it cannot be freed nor transfered (so no use-after-free). When the reference gets out of scope and freed, it is available again to the caller. References use the & operator, just like C pointers; or &mut with mutability, which need to point to uniquely referenced, mutable data – everything is statically checked.

This would be the most basic example:

```
let x = 3;
{
    y = &x; // ‘x‘ is now frozen and cannot be modified
    // ...but it can be read since the borrow is immutable
}
// ‘x‘ can be accessed again (‘y‘ is out of scope)
```

A borrow cannot outlive the variable it originates from: it has a **lifetime**, meaning that you will have to change your allocation pattern if a reference outlives its lifetime, like here:

```
let mut x = &3;
{
    let mut y = 4;
    x = &y;
} // ‘y‘ is freed here, but ‘x‘ still lives...
```

This pattern will be rejected, since y has a shorter lifetime than x.
The compiler enforces valid references and yields:

```
error: borrowed value does not live long enough
```

Note: there are a few cases through like when returning a reference passed to a function where you will need to add a lifetime annotation, so that it is inferred from the caller; more on that a bit later.

Referencing is the default choice as a pointer: all checks are performed at compile-time (by the borrow checker) so its footprint is that of a C pointer (which is also available in Rust as unsafe, * pointer). The unary star operator * also serves for dereferencing, like in C.

6.2.2 Unique ownership

An owned pointer owns a certain (dynamically allocated) part of the heap, i.e. the owner is the only one who can access the data – unless it transfers ownership, at which point the compiler will free the memory automatically (pointer is copied, but not the value).

```
fn take<T>(ptr: ~T) { // works for any type, ‘T‘
    ...
}
```

```
let m = ~"Hello!";
take(m);
take_again(m); // ERROR: 'm' is moved
```

Note that owned pointers can, like most objects, be borrowed. You can also copy a unique pointer using `.clone()`, but this is an expensive operation.

6.2.3 Shared pointers

"Shared" may evoke Garbage Collection to you. Well, this is partly the case.

Using only owned and borrowed pointers, ownership forms a DAG, while shared pointers allow multiple pointers to the same object and cycles. There are few of them, either GC-managed (with immutability to prevent data races), or Reference counted with specific types that allow thread-sharing or mutability, for instance. Note that Rust also has, of course, mutable shared pointer types.

Note: As you can see, these pointers serve a particular purpose. You should not have to look at them until they are needed in one of your programs.

First and foremost, please note that the new tracing, task-local Garbage Collection algorithm that will be introduced into the standard library is being worked on right now.
So, there used to be an @ managed pointer type, but it has been phased out in favor of `std::rc::Rc` and `std::gc::Gc`; which are standard library types. Right now, Gc is just a wrapper around Rc, which manually counts references, meaning less overhead than a GC algorithm (which periodically checks for pointer references) but it has a few limitations, e.g. it will leak memory on cycles (recursive data structures and the likes).

Note: task-local (or per-thread) GC is an important part of the deal because it means that you can have a task which handles low-latency jobs and is manually managed and another that can just run GC; task local also means that you can't pass these pointers between tasks, which can be desired in some cases.

Okay, let's have a look at some of these "smart pointers":

- First, if we want mutability inside of our Gc/Rc types, we will have to use `Cell` or `RefCell`, depending on whether the contained type is Pod or not (Pod is whatever type can be copied with a memcpy instruction, i.e. bool, integers, floats, char, &/* and all types that are construction of these all): Pod can use `Cell`, everything else will use `RefCell`.
  You can see Pod as every type that has a fixed size and where you do not need to dereference it in order to access the content (that includes `struct` of Pod types, tuples, etc. but not dynamic vectors for instance).
- We have said that Gc uses immutable, thread-local data. If we want to share data **across threads**, we would have to use a variant of Rc: Arc, i.e. *Atomically Reference Counted*. As the name suggests, it will make RefCount an atomic (insecable) operation (using Fetch-and-add on modern processors) so that it avoids data races where threads would access/modify the count at the same time.

- What if we want **cross-thread mutability**? `Arc<Cell<>>` is not allowed since it would break atomicity, so there is a special type for that: `MutexArc`, internally using mutexes to prevent data races.
  There is a variant called `RWArc` that uses a Readers–writer lock, making it more efficient in the case where you have lots of readers.

6.3 The Edge-cases

6.3.1 References and lifetimes

Okay, let's use a silly example involving a function return:

```
fn take(x: &int) -> &int {
    x
}

fn main() {
    let x = 4;
    println!("{}", *take(&x));
}
```

You are probably thinking that `x` outlives its lifetime; that's where it is:

```
error: cannot infer an appropriate lifetime due to conflicting requirements
```

It doesn't end there through, since we can pass a lifetime parameter from the caller to the function:

```
fn take<'a>(ptr: &'a int) -> &'a int {
    ptr
}

fn main() {
    let x = 4;
    println!("{}", *take(&x));
}
```

As you can see here, we define `'a` as a parameter (the single quotation mark prefix denoting a lifetime), and annotate it to both the value being passed and the return value. In short, the return value will inherit the lifetime of the parameter.
Since `x` is still alive until the end of `main()` – the caller function, this pattern is valid and typechecks.

So generally speaking, if you want to return a borrowed value (eventually with a condition evaluation for instance), you will have to use that.
This is particularly useful if you want to modify a variable in-place (that is, without having to pass it as heap pointer), in which case you can take a mutable borrow `&mut` with a lifetime annotation.

You can also annotate lifetime parameters to several variables, in which case the compiler will pick the lowest. This is useful when your output depends on a few variables:

```
fn max<'a>(x: &'a int, y: &'a int) -> &'a int {
    if (*x >= *y) {
        x
    } else {
        y
    }
}
```

Lastly, there is a `'static` lifetime, which you want to use outside of any brace scope and does not expire. As an example, here is how rust defines its bug report URL string:

```
static BUG_REPORT_URL: &'static str = "...url...";
```