Rust IN ACTION

T. S. McNamara

MEAP

MANNING

MEAP Edition
Manning Early Access Program
Rust in Action
Version 7

For more information on this and other Manning titles go to
www.manning.com

welcome

Dear Reader,
Thanks for taking a chance and buying this early release book on the Rust programming language and the internals of computer systems. I hope that you'll be rewarded with a fun, informative read!

Part 1: Rust Language Distinctives will provide a quick-fire introduction to the Rust language by working through projects that begin to introduce concepts that are expanded upon later in the book, such as implementing a File API.

Part 2: Systems Programming from the Ground Up (Almost) will shift your focus towards the computer. You will learn how data is represented at various levels between the application, operating system and hardware. Rust language features will be introduced as required.

A note about Chapter 6: This chapter should be considered a work in progress. It covers lots of ground. Readers progress from learning what a pointer is to benchmarking memory allocations with system utilities for Linux. Some of the explanations are terse... perhaps too terse.

We believe that the content is interesting and relevant to professional programmers, but are considering how it should be expanded and refactored to be most relevant to the book's readers. The author and the editorial team are very interested in your views about what works and what needs more work.

Part 3: Concurrent and Parallel Programming will walk through the tools that computers offer to get work done in parallel. Its primary goal will be to explain how multiple threads can cooperate and coordinate. A large example—building a graph that can be distributed in multiple ways—will be extended throughout several chapters.

Finally, Part 4: Extending Applications Up and Out will show Rust being used to build up software components to extend large software projects. These projects demonstrate systems programming "with the system" rather than systems programming "of the system". Part 4 will deal with the complexities of handling multiple CPU/OS pairs from the same code base.

As a MEAP reader, you have a unique opportunity to contribute to the book's development. All feedback is greatfully received, but please remember that there will be a human receiving the message. The best channel is the Manning forum. I'm also active on Twitter (@timClicks), Reddit (/u/timclicks) and the Rust Discourse forum (https://users.rust-lang.org/u/timclicks).

Thank you once again for choosing to purchase the book. It's a privilege to ride a short way with you along your computing journey.

—Tim McNamara

brief contents

1 Introducing Rust

This chapter covers:

- Highlighting some great features of the language and its community
- Exposing you to Rust's syntax
- Introducing the goals of the project
- Discussing where Rust might be useful and when to avoid it
- Building your first Rust program
- Explaining how Rust compares to object-orientated and wider languages

Welcome to Rust, the programming language that rewards your curiosity. Once you scratch the surface, you will find a programming language with unparalleled speed and safety that is still comfortable to use. Learning the language can be challenging, but the rewards can be immense.

Rust has established a strong following. In Stack Overflow's annual developer survey of over 50k software developers, Rust won "most loved programming language" in both 2016 & 2017.

The language has proven its ability to build powerful, reliable software.

- Dropbox rebuilt its storage backend in Rust during 2015-2016. This rewrite of a *distributed file system* lead to significant performance improvements.
- Oracle has developed a *container runtime* in Rust. Containers are lightweight isolated environments for applications that enable greater utilization of hardware than alternatives such as virtual machines. Their use was popularized within Linux environments by Docker, Inc.

- Mozilla uses Rust enhance the Firefox web browser. The browser project contains 15 million lines of code. Mozilla's first two Rust-in-Firefox projects, its MP4 metadata parser and text encoder/decoder have led to performance and stability improvements.
- Samsung Electronic's Internet of Things subsidiary SmartThings, uses Rust in its "Hub", the core of the service.

The language has also spawned lots of innovative projects. This includes operating systems, game engines, databases and drivers. Like its peer systems languages, it can be in micro-controllers through to supercomputers. Rust is one of the first programming languages to support WebAssembly, a standard for deploying apps into browsers without JavaScript. This flexibility allows you to compile and deploy your Rust project to the server, IoT devices, mobile devices and the browser.

Many of its semantic features are drawn from both the systems programming community and functional programming. From systems programming, Rust draws the notion of object lifetimes and a focus on safety. From functional programming, Rust provides efficient implementations of higher order programming, an impressive type system and first class support for generics. The language's syntax is similar to programming languages from the C programming heritage. Yet the differences are sufficiently different to minimize issues switching from one environment to the next.

So what does Rust look and feel like? Below at Listing 1.1 , you'll see an example of a JSON-based web service for providing the current time. It demonstrates a number of interesting features that help to produce effective software productively:

- High-level, expressive feel. Rust programs are able to feel ergonomic and yet achieve bare metal performance.
- No fuss type creation. Defining a new struct doesn't require compulsory methods such as constructors, attribute accessors (get/set methods) or destructors to get off the ground (lines 14-17).
- Convenient method syntax. Rust is not an object-oriented language, but does include a few syntactic features to create readable/writable code (lines 32-34).
- The ability for library writers to tweak the compiler's behavior with plugins. In the example, the web framework asks the compiler to automatically create boilerplate code (lines 1-2) and generate a JSON representation of timestamps itself (line 15).

To see the web service in action, here are some instructions to get you started.

- Install Rust via rustup.rs/
- Open a console prompt and move to the `ch1-time-api` directory
- Execute `rustup install nightly`. This enables extra features to .
- Execute `rustup run nightly cargo run`. This tells Rust that we want to use the "nightly" channel that we've just installed to execute "cargo run", which will build and run the project.

All going well, after several lines describing compiling the code's dependencies, you'll

see a report like this appear on your screen:

```
   Compiling ch1-time-api v0.1.0 (file:///path/to/ch1-time-api)
    Finished dev [unoptimized + debuginfo] target(s) in n.nn secs
     Running `target\debug\ch1-time-api.exe`
🔧  Configured for development.
    => address: localhost
    => port: 8000
    => log: normal
    => workers: 8
    => secret key: generated
    => limits: forms = 32KiB
    => tls: disabled
📡  Mounting '/':
    => GET /
    => GET /time
🚀  Rocket has launched from http://localhost:8000
```

Here is the web service code itself:

Listing 1.1. Example Rust Code - A Web API That Tells The Time (ch1-time-api/src/main.rs)

```
#![feature(plugin)]                 ❶
#![plugin(rocket_codegen)]          ❶

extern crate serde;                 ❷
extern crate chrono;                ❷
extern crate rocket;                ❷
extern crate rocket_contrib;        ❷

#[macro_use]                        ❸
extern crate serde_derive;

use chrono::prelude::*;             ❹
use rocket_contrib::{Json};         ❺

#[derive(Serialize)]                ❻
struct Timestamp {
    time: String,
}

#[get("/")]                         ❼
fn index() -> &'static str {        ❽
    "Hello, world!"                 ❾
}

#[get("/time")]
fn time_now() -> Json<Timestamp> {
    let now: DateTime<Utc> = Utc::now();
    let timestamp = Timestamp { time: now.to_rfc3339() };
    Json(timestamp)
```

```
}

fn main() {
    rocket::ignite()
            .mount("/", routes![index, time_now])
            .launch();
}}
```

1. Attributes, such as `#[feature(plugin)]` and `#[derive(Serialize)]`, customize behavior. `#[derive(Serialize)]` asks the compiler to create its own code for converting the `Timestamp` struct to a string, to be sent down the wire as JSON.
2. `extern` brings external crates (packages) into local scope
3. Attribute to indicate that we want to import macros from another crate
4. `use` and an asterisk brings all exported members of `chrono::prelude` into local scope. This crate uses `DateTime` and `Utc`
5. `use` with curly braces brings only specified members into local scope. In this case, just the single member `Json`
6. `#[derive(Serialize)]` is provided by the `serde` crate and automatically generates a string representation of this struct (which will be used as JSON down the wire later on)
7. The `get` attribute is provided by the web framework. It tells Rust to generate code on our behalf for serving HTTP requests.
8. Define a function with no arguments. Its return type is a "'static str", which can be thought of as a string type for string literals.
9. Rust returns the result of the final expression of a function. Using the `return` keyword at the end of a function is considered poor style

The preceding example was chosen as an attempt at packing as many representative features of the language into a single example as possible. Over the course of a few chapters, each of those features will be unpacked and examined. Until then, let's take a step back and consider some of the thinking behind the language and where it fits within the programming language ecosystem.

1.1 What is Rust?

Rust is a programming language that emerged from a small number of big ideas: safety, control and ergonomics. Its primary focus is "systems programming", but those small ideas are widely applicable. So is Rust.

Visitors to its website and blog posts are greeted with an impressive welcome message:

> *Rust is a systems programming language that runs blazingly fast, prevents segfaults and guarantees thread safety.*
>
> *-- Rust Homepage*

The underlying themes from that sentence are control (essential for systems programming), safety and speed. Hidden within there though is developer satisfaction. Over the course of the language's progress, it has been obvious to me that the Rust team deeply cares about making Rust a productive environment.

I argue that the three main ideas that make Rust what it is are:

- Safety
- Ergonomics
- Control

Performance is important, but I believe that it emerges from these deeper ideas.

1.1.1 Safety

Rust's overarching goal is to facilitate safe software. What is meant by the term "safety"?

Rust programs are free from:

- "dangling pointers" - live references to data that has become invalid over the course of the program
- "data races" - behavior dependent on timing
- "buffer overflow" - attempting to access the 12th element of an array of only 6 elements
- "iterator invalidation" - an issue caused by something that is being iterated over being altered mid-way through

When programs are compiled in debug mode, Rust also protects against integer overflow. Integers are fixed-width memory. Integer overflow is what happens when they hit their limit and flow over to the beginning again.

Knowing that a language is safety provides programmers with a degree of freedom. They know that their program won't implode. Within the Rust community, this has spawned the saying "fearless concurrency". At times, the compiler will also be able to run things faster due to a similar principle. In the following snippet, the local variable `v` is a read-only reference to a vector. Rust uses this information to turn off runtime bounds checking as the `for..in` loop is guaranteed not to overflow.

```
fn for_each(v: &Vec<T>) {      ❶
     for item in 0..v.len() { ❷
        work(v[things]);
     }
}
```

❶ "&Vec<T>" reads as "a reference to a vector of type T".

❷ `0..v.len()` returns an iterator from the

1.1.2 Ergonomics

Ergonomics, or programmer usability, has become an increasingly important part of the Rust community's ideals. New software projects require expressibility, in order to land new features faster. Established software projects require readability because most time will be spent in maintainence.

Ergonomics is a difficult concept to demonstrate in a single example. Among much else, Rust offers generics, sophisticated data types, pattern matching and closures.[1] Those who have worked with other ahead-of-time compilation languages are likely to appreciate Rust's build system and its comprehensive package management. For beginners to the language though, the compiler's error messages are probably the most obvious sign that the language's designers have its users in mind. Let's start there.

A common syntax error that is difficult to debug can occur when programmers use a single equals sign (=) for assignment rather than two (==) for testing equality.

```
fn main() {
    let a = 10;

    if a = 10 {
        println!("a equals ten");
    }
}
```

In Rust, the preceding code would fail to compiler. The Rust 1.17 compiler generates the following message:

```
rustc 1.17.0 (56124baa9 2017-04-24)
error[E0308]: mismatched types
 --> <anon>:4:8
  |
4 |     if a = 10 {
  |        ^^^^^^ expected bool, found ()
  |
  = note: expected type `bool`
             found type `()`

error: aborting due to previous error
```

At first, "mismatched types" might feel like a strange error message. Surely variables can be tested for equality against integers. After some thought, it becomes apparent that the `if` test is being provided with the wrong type. It requires a boolean value and is instead receiving `()`. `()` is a placeholder value that will be discussed in more depth later.

Adding a second = on line 4 results in a working program that prints `a equals ten`.

```
fn main() {
    let a = 10;

    if a == 10 {
        println!("a equals ten");
    }
}
```

[1] If these terms are unfamiliar, do keep reading. They are explained in detail further on.

1.1.3 Control

Control over memory access, memory layout and specific CPU instructions is very important when squeezing the best performance out of code. At times, it is imperative to manage how something is operating. It might matter that data is stored in the stack, rather than on the heap. At times, it might make sense to add reference counting to a shared value. Often, it makes sense to pass references to functions. Occasionally, it might be useful to create one's own type of pointer for a particular access pattern. Most of the time, there are other things to worry about. And for that, there are good defaults.

The following code snippet prints out the line `a: 10, b: 20, c: 30, d: Mutex { data: 40 }`. Each representation is another way of storing an integer. As we progress through the next few chapters, the trade offs related to each level become apparent. For the moment, the important things to remember is that the menu is comprehensive and you are welcome to choose exactly what's right for your specific use case.

```
use std::rc::Rc;
use std::sync::{Arc, Mutex};

fn main() {
    let a = 10;                         // an integer on the stack

    let b = Box::new(20);               // an integer on the heap

    let c = Rc::new(30);                // a boxed integer wrapped
                                        // behind a reference counter

    let d = Arc::new(Mutex::new(40)); // an integer protected by a
                                        // mutual exclusion lock, wrapped
                                        // in an atomic reference counter
                                        // to enable concurrent mutability

    println!("a: {}, b: {}, c: {}, d: {}", a, b, c, d);
}
```

To understand why Rust is doing something the way it is, it can be helpful to refer back to these three prinicples. Data within Rust is immutable by default. The language's first goal is safety. But it always wants to provide that safety without constraining you or imposing runtime costs.

1.2 Rust's Big Features

Our tools shape what we believe we can make. Rust seeks to enable you to build the software that you would like to, but were too scared to try. So what kind tool is Rust?

Flowing out from the three principles discussed in the last section are three overarching features of the language:

- Performance
- Concurrency

- Memory efficiency

1.2.1 Performance

Svelte programs run fast. Yet the speed of the CPU is fixed. Thus, for software to run faster, it needs to do less. To achieve this, Rust pushes the burden of its high level features onto the compiler. Famously, Rust does not need a garbage collector to ensure safety.

Rust also has some less obvious tricks. An object's methods are always dispatched statically, unless dynamic dispatch is explicitly requested. This enables the compiler to heavy optimize code, sometimes to the point of eliminating the cost of the function call entirely.

1.2.2 Concurrency

Asking a computer to do more than one thing at the same time has proven very difficult for software engineers to do. As far as an operating system is concerned, two independent threads of execution are at liberty to destroy each other if a programmer makes a serious mistake. Yet Rust has spawned the saying "fearless concurrency". Its emphasis on safety crosses the bounds of independent threads. There is no global interpreter lock (GIL) to constrain a thread's speed. We explore some of the implications of this in Part 2.

1.2.3 Memory Efficiency

Rust enables you to create programs that require minimal memory. High-level constructs such as objects can be created with minimal overhead and may be optimized away.

1.3 Downsides of Rust

It's easy to talk about this language as if it is the panacea of all software engineering. The (sometimes overstated) slogans are great. "High level syntax with low level performance!" "Concurrency without crashes!" "C with perfect safety!" For all of its merits, Rust does have disadvantages.

1.3.1 Compile Times

Rust is slower at compiling code than its peer languages. It has a complex compiler toolchain that includes multiple intermediate representations and sending lots of code to LLVM. The "unit of compilation" for a Rust program is not an individual file, but a whole package (known affectionately as a "crate"). As crates can include multiple modules, they can be very large units of compilation. This enables whole-of-crate optimization, but requires whole-of-create compilation as well.

1.3.2 Strictness

Rust is a bit of a stickler. It's impossible—well, very difficult—to be lazy when programming with it. Programs it won't start until everything is just right.

Over time, it's likely that you'll come to appreciate this feature. If you've ever programmed in a dynamic language, then surely you would have encountered the fustration of your progam crashing because of a misnamed variable. Rust brings that frustration sooner.

1.3.3 Size of the Language

The language is large. It has a type system, multiple ways to gain access to values, an ownership system that is paired with enforced object lifetimes. The language is also comprehensive. The snippet below, seen before at "Control", probably has caused a degree of being overwhelmed.

```
use std::rc::Rc;
use std::sync::{Arc, Mutex};

fn main() {
    let a = 10;                       // an integer on the stack

    let b = Box::new(20);             // an integer on the heap

    let c = Rc::new(30);              // a boxed integer wrapped
                                      // behind a reference counter

    let d = Arc::new(Mutex::new(40)); // an integer protected by a
                                      // mutual exclusion lock, wrapped
                                      // in an atomic reference counter
                                      // to enable concurrent mutability

    println!("a: {}, b: {}, c: {}, d: {}", a, b, c, d);
}
```

There are design decisions that some people will disagree with. Rust's referencing system are implemented as "fat pointers".

- The language is large. Eventually, you will encounter jargon such as affine types and monomorphization.
- The compiler is slower than other languages in its class. This may result in your code-compile-test workflow.
- The syntax is arguably ugly. It's workable and easy to parse, but not elegant.

1.3.4 Hype

"Have you considered rewriting this in Rust?" The Rust community of very wary of growing too quickly and being consumed by hype. Yet, a number of software projects have encountered a question on their mailing list or issue tracker recommending a complete rewrite in a new language.

Software written in Rust is not immune to security problems. By 2015, as Rust gained prominence, implementations of SSL/TLS, namely OpenSSL and Apple's own fork, were found to have serious security holes. Known informally as "Heartbleed" and

"goto fail", both exploits provide opportunities to test Rust's claims of memory safety.

Rust is likely to have helped in both cases, but it is still possible to write Rust code that suffers from similar issues.

HEARTBLEED

Heartbleed was caused to the re-use of a buffer, i.e. an array of bytes set aside for taking input. Buffers are re-used to minimize the number of times an application must request memory from the operating system. We can create a similar situation fairly easily in Rust that leads to similar problems.

Imagine that we wished to store information from multiple User objects. We decide, for whatever reason, it re-use a single buffer through the course of the program. If we don't reset this buffer once we have made use of it, information from earlier calls leaks to the latter ones.

Here is a précis of a program that would encounter this error:

```
let buffer = &mut[0u8; 1024]; ❶

read_secrets(&user1, buffer); ❷
store_secrets(buffer);

read_secrets(&user2, buffer); ❸
store_secrets(buffer);
```

❶ This reads "bind a reference (&) to a mutable (`mut`) array (`[…]`) that contains 1024 unsigned 8-bit integers (`u8`) intialized to 0 to the variable `buffer`."

❷ `buffer` is filled with bytes from the `user1` object

❸ `buffer` still contains data from `user1` that may or may not be overwritten by `user2`

Rust does not protect you from logical errors. Rust ensures that your data is never able to be written in two places at the same time. It does not ensure that your program is free from all security issues.

GOTO FAIL

The "goto fail" bug was caused by programmer error coupled with design issues with C and potentially its compilers for not pointing out the flaw. A function that was designed to verify a cryptographic key pair ended up skipping all checks. Here is a selected extract from the function at hand, with a fair amount of obfuscatory syntax retained.

```
static OSStatus
SSLVerifySignedServerKeyExchange(SSLContext \*ctx, bool isRsa, SSLBuffer
signedParams,
                                 uint8_t \*signature, UInt16 signatureLen)
{
    OSStatus        err;                                            ❶
```

```
    ...

    if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0) ❷
        goto fail;
    if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
        goto fail;
        goto fail;                                                  ❸
    if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
        goto fail;

    err = sslRawVerify(ctx,
                       ctx->peerPubKey,
                       dataToSign,              /* plaintext \*/
                       dataToSignLen,           /* plaintext length \*/
                       signature,
                       signatureLen);
    if(err) {
        sslErrorLog("SSLDecodeSignedServerKeyExchange: sslRawVerify "
                    "returned %d\n", (int)err);
        goto fail;
    }

fail:
    SSLFreeBuffer(&signedHashes);
    SSLFreeBuffer(&hashCtx);
    return err; ❹

}
```

❶ Initialization of the `OSStatus` object with a "pass" value, e.g. `0`
❷ A series of defensive programming checks
❸ Unconditional `goto`, meaning substantive check at `sslRawVerify()` is always skipped
❹ Return the "pass" value of `0`, even for inputs that should have failed the verification test

The issue lies between lines 11 & 13. In C, logical tests do not require brackets. C compilers interpret those three lines like this:

```
    if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0) {
        goto fail;
    }
    goto fail;
```

Would Rust have helped? Probably. In this specific case, Rust's grammar would have caught the bug. It does not allow logical tests without parentheses. Rust also issues a warning when code is unreachable. But that doesn't mean the error is made impossible in Rust. Stressed programmers under tight deadlines make mistakes. In the general case, code would compile and run.

Code with caution.

1.3.5 Single Implementation

While we are discussing potential pitfalls, allow me to touch on one other. There is only a single implementation of the Rust programming language. This exposes a degree of risk to those people who wish to remove any potential single point of failure from their software. Rust's development is sponsored by Mozilla. If Mozilla were to become insolvent or change its priorities, progress developing the language would certainly slow down or even halt.

1.4 Where Rust Fits

Rust is a general purpose language that can be successfully deployed in many areas.

Its headline domain is *systems programming*, an area most prominently occupied by C & C++. Yet the boundary of that domain is quite porous. Historically, it would have covered software such as operating systems, compilers, interpreters, file systems and device drivers. Now, it would also include web browsers and other software, even if it are user-facing. With that in mind, where does Rust shine?

1.4.1 Data Processing

Rust is extremely good at text processing and data wrangling. As of mid-2017, it touts the world's fastest regular expression engine. Its type system and memory control provide you with the ability to create high-throughput data pipelines with low and stable memory footprint. Small filter programs can be easily embedded into a larger framework via Apache Storm or Apache Kafka.

1.4.2 Extending an Application

Rust is well suited for extending programs written in a dynamic language. This enables "Ruby gems in Rust", "Python C Extensions in Rust" or "Erlang/Elixir NIFs written in Rust". C extensions are typically a scary proposition. They tend to be quite tightly integrated with the runtime. Make a mistake and you could be looking at runaway memory consumption due to a memory leak or a complete crash. Rust takes away a lot of this fear.

1.4.3 Operating in Resource-constrained Environments

C has been the domain of microcontrollers for decades. The Internet of Things is coming and that means many billions of insecure devices exposed to the network. Any input parsing code will be routinely probed for weaknesses, given how infrequently devices have their firmware are updated. Rust can play an important role by adding a layer of safety without imposing runtime costs.

1.4.4 Applications

There is nothing inherent in Rust's design that prevents it from being deployed to develop user-facing software. Servo, the web browser engine that acted as an incubator for Rust's early development, is a user-facing application.

DESKTOP

There is still a large space for applications to be written that live on people's computers. Desktop applications are often complex, difficult to engineer and difficult to support. With Rust's ergonomic approach to deployment, its rigor, it is likely to become many applications' secret sauce. To start, they will be built by small, indy developers. As Rust matures, so will the ecosystem.

MOBILE

Android, macOS and other smart phone operating systems generally provide a blessed path for developers. In the case of Android, that path is Java. In the case of macOS, developers generally program in Swift. There is however, another way.

Both platforms provide the ability for "native applications" to run on the system. This is generally intended for applications written in C++, such as games, to be able to be deployed on people's phones. Rust is able to talk to the phone via the same interface with no additional runtime cost.

WEB

As you'll be very aware, JavaScript is the language of the web. Over time though, this will change. Browser vendors are developing a standard called WebAssembly, which promises to be a compiler target for many languages. Rust is one of the first. Porting a Rust project to the browser requires two additional commands on the command line.

1.4.5 Systems Programming

In some sense, systems programming is Rust's raison d'être. Many large programs have been implemented in Rust, including compilers (Rust itself), video game engines and operating systems. The Rust community includes writers of parser generators, databases and file formats. Rust has proven to be a productive environment for programmers who share Rust's goals.

1.5 Rust's hidden feature: its community

It takes more than software to grow a programming language. One of the things that the Rust team has done extraordinarily well is to foster a positive and welcoming community around the language. Everywhere you go within the Rust world, you'll find that you'll be treated with courtesy and respect.

1.6 A Taste of the Language

This section is a chance to experience Rust face to face. We start by understanding how to use the compiler, then move on to writing full programs.

1.6.1 Cheating Your Way To "Hello, world!"

The first thing that most programmers will do when they reach for a new programming language is to learn how to print "Hello, world!" to the console. You'll do that too, but

with flair. You'll be verifying that everything is in working order before you start encountering annoying syntax errors.

If you are on Windows, please open the Rust Command Prompt that is available in the Start menu after installing Rust. Please then execute the command:

```
C:\> cd %TMP%
```

If you are running Linux or OS X, please open a Terminal Window. Once open, please enter the following:

```
$ cd /tmp
$
```

From this point onwards, the commands for all operating systems should be the same. If you have installed Rust correctly, the following three commands will produce "Hello, world!" on the screen (as well as a bunch of other output).

```
$ cargo new --bin hello
$ cd hello
$ cargo run
```

Here is an example of what the whole process looks like on Windows running Rust 1.12:

```
C:\> cd %TMP%

C:\Users\Tim\AppData\Local\Temp\> cargo new --bin hello
     Created binary (application) `hello` project

C:\Users\Tim\AppData\Local\Temp\> cd hello

C:\Users\Tim\AppData\Local\Temp\hello\> cargo run
   Compiling hello v0.1.0 (file:///C:/Users/Tim/AppData/Local/Temp/hello)
    Finished debug [unoptimized + debuginfo] target(s) in 2.36 secs
     Running `target\debug\hello.exe`
Hello, world!
```

If you have reached this far, fantastic! You have been able to run your first Rust code without needing to write any Rust.

Let's take a look at what's just happened. `cargo` a tool that provides both a build system and package manager. That is, cargo knows how to execute `rustc` (the Rust compiler) to convert your Rust code into executable binaries or shared libraries.

Cargo understands things such as the difference between debug build and a release build. It also knows how to manage 3rd party libraries, called crates in Rust. It can assist you to incorporate them into your project by automatically downloading the correct version from crates.io. One of its other miscellaneous tasks is to help you out when you create a new project by creating a boilerplate program.

The first line has four parts. The command "cargo", a subcommand "new", a parameter

"hello" and an option "--bin". These four parts together should be read as, "cargo, please create a new project called hello I can run as a binary application".

```
[todo: hedgehog diagram]
cargo new --bin hello
       |     |   \--> call the project “hello”
       |     \------> make this an executable binary
       \------------> create a new project
```

Cargo then goes and creates a project for you that follows a standard template that all Rust crates follow. After completing the `cargo new` command, your directory structure will have been changed to something that looks like this:

```
$ tree hello
hello
├── Cargo.toml
└── src
    └── main.rs

1 directory, 2 files
```

All Rust crates have the same structure. In their base directory, a file called `Cargo.toml` describes the project's metadata, such as the project's name, its version and its dependencies. Source code appears in the `src/` directory. Rust source code files use the `.rs` file extension.

The next command that you executed was `cargo run`. This line is much simpler to grasp for you, but there was actually much more work being by cargo. You asked `cargo` to run the project. As there was nothing to actually run when you invoked the command, it decided to compile the code on your behalf in debug mode to provide maximal error information. As it happens, `src/main.rs` always includes a "Hello, world!" stub. The result of that compilation was a file called `hello` (or `hello.exe`). `hello` was executed and the result was printed to your screen.

For the curious, our project's directory structure has changed a great deal. We now have a `Cargo.lock` file in the base of our project and a `target/` directory. Both that file and the directory are `cargo's domain. We won't need to touch them. They are artifacts of the compilation process. `Cargo.lock` is a file that specifies the exact version numbers of all the dependencies so that future builds are reliably built the same way until `Cargo.toml` is modified.

```
$ tree --dirsfirst hello
hello
├── src
│   └── main.rs
├── target
│   └── debug
│       ├── build
│       ├── deps
│       ├── examples
```

```
│       ├── native
│       └── hello
├── Cargo.lock
└── Cargo.toml
```

Well done for getting things up and running. Now that we've cheated our way to "Hello, World!", let's get there via the long way.

1.6.2 Your First Rust Program

We want to write a program that will output the following text:

```
Hello, world!
Grüß Gott!
ハロー・ワールド
```

You have probably seen the first line in your travels. The other two are there to highlight few of Rust's features: easy iteration and built-in support for Unicode.

We will use `cargo`, as in the previous section. To start, open a console window. Move into a temporary directory (`cd /tmp/` or `cd %TMP%` on Windows).

```
$ cargo new --bin hello2
$ cd hello2
```

Now open `hello2/src/main.rs` in a text editor. Replace the text with in that file with the following:

Listing 1.2. Hello World In Three Languages (ch1-hello2.rs)

```
fn greet_world() {
    println!("Hello, world!");  // our old friend. (1)

    let southern_germany = "Grüß Gott!";           (2)
    let japan = "ハロー・ワールド";                  (3)

    let regions = [southern_germany, japan];       (4)

    for region in regions.iter() {                 (5)
            println!("{}", &region);
    }
}

fn main() {
    greet_world();
}
```

(1) The exclamation mark here indicates the use of a macro, which we discuss shortly
(2) Assignment in Rust, more properly called variable binding, uses the `let` keyword
(3) Unicode support out of the box
(4) Array literals
(5) "Borrow" the `region` variable by taking a reference, providing fast read-only access to its contents

Now that our code is updated, run `cargo run` from the `hello2/` directory. You should be presented with three greetings.

Let's take a few moments to touch on some of the elements Listing 1.2.

One of the first things that you are likely to notice is that strings in Rust are able to include a wide range of characters. Strings are UTF-8. This means that you are able to use non-English languages with relative ease.

The one character that might look out of place is an exclamation mark after `println`. If you have programmed in Ruby, you may be used to thinking that it is used to signal a destructive operation. In Rust, it signals the use of a macro. Macros can be thought of as sort of fancy functions for now. They offer the ability to avoid boilerplate code. In the case of `println!`, there is a bunch of type detection going on under the hood so that arbitrary data types can be printed to the screen.

You are likely to have heard that Rust is a "systems" programming language. Typically defined, a systems programming language is "low level" and "close to the metal". Languages defined in these terms have tended to be fairly unwieldy, with the reward being that they're very powerful.

If you are familiar with the internals of programming languages such as Perl, Ruby, Python, Java and JavaScript, you may have come across the term "garbage collector". A garbage collector is a service that tells the operating system that one of your variables has left scope and the memory allocated to your program is able to be used by others.

Garbage collection is extremely convenient, but imposes a cost in that your programs must run more slowly as they are doing extra work to check which variables are still needed as your program runs. Rust has no garbage collector, but still offers the convenience of one.

One of the things that may be apparent if you have been reading about Rust for a long time is that we haven't added any type annotations to our example code. Rust is statically typed. That means, the behavior of all data is defined in advanced of the program being run and that behavior is well specified. Yet, Rust has a very smart compiler. That compiler doesn't always need to be told about the data types that it encounters.

1.7 Summary

This chapter you learned that:

- Many companies have successfully built large software projects in Rust.
- Software written in Rust can be compiled to the PC, the browser, the server, as well as mobile and IoT devices.
- The language is well loved by software developers. It has repeatedly won Stack Overflow's "most loved programming language" title.
- Rust allows you to experiment without fear. It provides correctness guarantees that

other tools are unable to provide without imposing runtime costs.

- There are three main command line tools to learn:
  - `cargo`, which manages a whole crate
  - `rustup`, which manages Rust installations
  - `rustc`, which manages compilation of Rust source code
- Rust projects are not immune from all bugs.