# Productive Rust

By

Marvin Hansen

# Contents

# 1 Getting Started

## 1.1 Requirements

To install Rust, you need a C compiler on your system and the command line tool *rustup* that manages and updates your Rust installation and Rust toolchain.

### 1.1.1 Windows

On Windows, you install a C compiler with Visual Studio from the official Microsoft website:

*https://visualstudio.microsoft.com/downloads/*

During installation, when asked which workloads to install, include:

- Desktop Development with C++
- The Windows 10 or 11 SDK
- The English language pack component

Once Visual Studio has been installed, you download rustup from the official website:

*https://rustup.rs/*

Download and run *rustup-init.exe* then follow the onscreen instructions.

### 1.1.2 Linux, Unix, BSD

Linux and Unix systems provide a C compiler such as GCC or Clang via package manager. Please consult the distribution documentation of how to install GCC. Next, you install rustup via your terminal using the command below:

```
curl --proto '=https' --tlsv1.3 https://sh.rustup.rs -sSf | sh
```

### 1.1.3 Mac

On MacOS, you install Clang and then rustup via terminal:

```
xcode-select --install
curl --proto '=https' --tlsv1.3 https://sh.rustup.rs -sSf | sh
```

## 1.2 Rustup

By design, rustup is a toolchain multiplexer. It installs and manages many Rust toolchains and presents them all through a single set of tools. That means, rustup installs the Rust Programming Language and other standard tools from the officially released channels, enabling you to switch between stable, beta, and nightly compilers and keeps them all updated.

### 1.2.1 Concepts

- **Channel:** Rust releases to three channels: Stable, Beta, and Nightly
- **Toolchain:** A "toolchain" is a complete installation of the Rust compiler and all related tools. Note, each release has its own toolchain.
- **Target:** The "target" specifies the platform for which the compiler generates code. By default, cargo and rustc use the host toolchain's platform as the target. To build for a different target, usually the target's standard library needs to be installed first via the rustup target command.
- **Component:** Each release of Rust includes several "components", some of which are required (like rustc) and some that are optional (like clippy)
- **Profile:** To simplify working with components, a "profile" defines a grouping of components.

### 1.2.2 Components

Each toolchain has several "components", some of which are required (like rustc) and some that are optional tools (like clippy). The rustup component command is used to manage the installed components. For example, run rustup component list to see a list of available components for all channels and platforms. Components can be added for an existing toolchain. For example:

```
    rustup component add rust-docs
```

The set of available components may vary with different releases and toolchains. The following is an overview of the different components:

- rustc — The Rust compiler and Rustdoc.
- cargo — Cargo is a package manager and build tool.
- rustfmt — Rustfmt is a tool for automatically formatting code.
- rust-std — This is the Rust standard library. There is a separate rust-std component for each target that rustc supports.

- rust-docs — This is a local copy of the Rust documentation. Use the rustup doc command to open the documentation in a web browser.
- rls - RLS is a language server that provides support for editors and IDEs.
- clippy — Clippy is a lint tool that provides extra checks for common mistakes and stylistic choices.
- rust-src — This is a local copy of the source code of the Rust standard library. This can be used by some tools, such as RLS, to provide auto-completion for functions within the standard library
- rust-analysis — Metadata about the standard library, used by tools like RLS.

Not all components are available for all toolchains. Especially on the nightly channel, some components may not be included if they are in a broken state. Consult the nightly status page to check the availability of components:

*https://rust-lang.github.io/rustup-components-history*

### 1.2.3 Cross-compilation

Rust supports many different platforms and rustup gives easy access to all of them. When you first install a toolchain, rustup installs only the standard library for your host platform - that is, the architecture and operating system you are presently running. To show all currently installed targets' toolchains, run:

```
    rustup show
```

If you're currently on a stable toolchain and wish to try out a newly introduced feature that is available in the nightly version you can easily switch to the nightly toolchain:

```
    rustup default
    rustup default nightly
```

To compile to other platforms you must install other target platforms. To list all available targets, run:

```
    rustup target list
```

Suppose you want to cross-compile for Android on Arm. Install the generic arm toolchain for Android, run:

```
rustup target add arm-linux-androideabi
```

You can then build for Android with Cargo by passing the –target flag:

```
    cargo build --target=arm-linux-androideabi
```

When you don't need a specific target anymore, you can remove it:

```
rustup target remove arm-linux-androideabi
```

### 1.2.4 Updates

To update rustup itself, run:

```
rustup self update
```

This checks if there is a newer version of rustup available and if so, installs it.
To update rust, the rust compiler and all tools, run

```
rustup update
```

## 1.3 Cargo

For the most part, you are using cargo to create, build, and run your project. To create a new hello-world project use cargo to create a skeleton:

```
cargo new hello-world
cd hello-world
```

Open src/main.rs in your favorite editor and add the following rust code to print out a simple hello-world:

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

### 1.3.1 Build

In Rust, you barely use the compiler directly. Rather, cargo builds and runs your project. Make sure to run cargo in your working directory, hello-world.

```
cargo build
cargo run
```

Notice, by default cargo builds for the development release, which means the resulting binary is non-optimized and contains debug information. The build process is bit faster, but obviously not meant for production. To build a binary for production usage, you have to set the –release flag (1).

```
cargo build --release
cargo run --release
```

### 1.3.2 Dependencies

You should have a new hello-world folder with a Cargo.toml file. The Cargo configuration file serves as a single point of truth for the project. By default, a minimal Cargo.toml only shows the project name, version, and rust edition.

```toml
[package]
name = "hello-world"
version = "0.1.0"
edition = "2021"
```

Note, adding dependencies to the project can be done in one of two ways:

1. Editing the Cargo.toml file manually
2. Using Cargo to add a dependency

In the first case, one needs to specify the name and version of each dependency as shown below:

```toml
[package]
name    = "hello-world"
version = "0.1.0"
edition = "2021"

[dependencies]
time  = "0.1.12"
regex = "0.1.41"
```

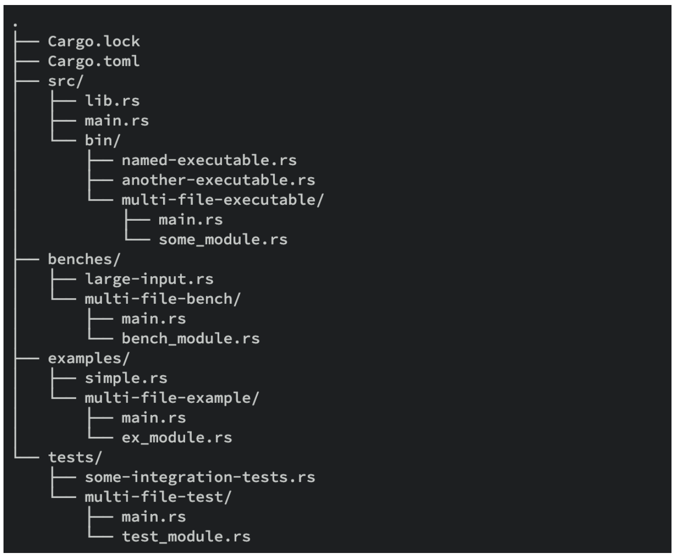
Following a cargo build, time and regex can be used within the project. Cargo build generates a Cargo.lock file which contains exact version information for each dependency used in the project. Note, do not edit Cargo.lock because it is a purely generated file and all modifications will be overwritten.

In the second case, using cargo add would result in adding the most recent version of the dependency to the Cargo.toml file. Note, cargo add is useful when developing without strict dependency version requirements. However, in case only a very specific version of a dependency can be used, edit the Cargo.toml manually.

```
cargo add regex
cargo add time
```

By convention, each new cargo project already is a git repository so you only need to add your remote and you ready to start writing Rust.

```
git add .
git commit -am "Project skeleton"
git remote add origin git@github.com:GitHubName/hello-world.git
git push -u origin main
```



```
.
├── Cargo.lock
├── Cargo.toml
├── src/
│   ├── lib.rs
│   ├── main.rs
│   └── bin/
│       ├── named-executable.rs
│       ├── another-executable.rs
│       └── multi-file-executable/
│           ├── main.rs
│           └── some_module.rs
├── benches/
│   ├── large-input.rs
│   └── multi-file-bench/
│       ├── main.rs
│       └── bench_module.rs
├── examples/
│   ├── simple.rs
│   └── multi-file-example/
│       ├── main.rs
│       └── ex_module.rs
└── tests/
    ├── some-integration-tests.rs
    └── multi-file-test/
        ├── main.rs
        └── test_module.rs
```

Figure 1.1: Project folder structure

### 1.3.3 Conventions

Cargo uses conventions for file placement to make it easy to navigate across different projects as shown in 1.1. The standard files and folders are:

- Cargo.toml and Cargo.lock are stored in the root of your package (package root).
- Source code goes in the src directory.
- The default library file is src/lib.rs.
- The default executable file is src/main.rs.
- Other executables can be placed in src/bin/.
- Benchmarks go in the benches directory.
- Examples go in the examples directory.
- Integration tests go in the tests directory.

If a binary, example, bench, or integration test consists of multiple source files, place a main.rs file along with the extra modules within a subdirectory of the src/bin, examples, benches, or tests directory. The name of the executable will be the directory name.

## 1.4 Inner Loop

According to Luca Palmieri (2), the inner development loop consists of four steps:

1. Make a code change
2. Compile the code
3. Run tests
4. Run the application

The speed of your inner development loop limits the number of iterations that you can complete in a unit of time. That means, the faster the inner loop, the more iterations you can complete per hour. There are a number of ways you can accelerate your inner loop:

### 1.4.1 Ai code assistance

Ai auto-complete tools such as Tabnine or GitHub Copilot have increasingly become more mature and useful and may accelerate the time it takes to write or change code. GitHub Copilot has been trained to use the vast codebase of Github which may or may not suit your specific project, but at least it is worth trying. Tabnine, on the other hand, supports training the ML model specifically on your own code base. This comes in handy, especially in organizations that enforce uniform code standards and formatting. Depending on project requirements, it may be worth running an A/B test to determine whether engineers with Ai code assistance can achieve a faster task completion time compared to those without.

### 1.4.2 Code templates

Effective code templates are an often-overlooked benefit of uniform code standards. Specifically, when the engineering team agrees on a certain blueprint for standardized deliverables (CLI, services, plugins etc.), the next obvious step is to codify all applicable best practices in a single template versioned in a Git repo accessible to the entire team or wider organization. The upside, really, is that a lot of boilerplate code can either be pre-written or generated in a uniform way which then saves valuable time when starting a new project. The author repeatedly gained the experience that in certain kinds of projects, roughly 60% of all committed code came from templates and generators, leaving only the remaining 40% up to manual coding. This, with the help of Ai coding assistance, became relatively straight forward, thus leaving much more time to work on harder problems.

### 1.4.3 Continuous compilation

Since Rust 1.52.1, the Rust compiler has worked incrementally and in parallel, which drastically accelerates re-compile time when working continuously on one project. Going one step further, cargo-watch monitors all files and re-compiles automatically on every code change, thus giving instant feedback. Cargo watch supports command chaining, allowing you to check, test, and run your code continuously.

```
    cargo watch -x check -x test -x run
```

When working on a library using Test Driven Development (TDD), cargo watch helps tremendously to sustain the development flow of designing, implementing, and fixing the test. When working on implementing a microservice template, cargo watch again becomes invaluable in focusing on making mostly correct code changes to keep the compile and test loop running.

## 1.5 Loop Safeguard

A common misconception prevalent in Continuous Integration (CI) is the idea to add increasingly more checks to the CI pipeline to catch what engineers may have missed in an increasingly large and complex code base. The premise that people overlook things, especially when under time pressure, remains valid. The conclusion to offload code checks to CI, however, is not.

The problem with CI linting and code checks really is that they tend to fail quite often with the result of an aborted build resulting in an interruption of the outer loop thus slowing down total time to market.

Rather, linting, security checks, code checks, code-rewrite, and code generation have to happen before pushing into the repo. Instead of failing on the CI, all potential code issues have to be rejected prior to a push or merge into the working branch. It is not worth running these checks on every commit because that would slow down the inner loop quite substantially. Instead, between the inner and outer loop, a comprehensive check is needed to prevent the propagation of potentially problematic code by basically rejecting every push that fails the checks.

In the absence of an established term, I have dubbed the practice "Loop Safeguard" because, on the one hand, it prevents unnecessary CI stops, but on the other hand it also returns all problematic code to its origin. The reason for that is simple. The engineer who wrote the code in question is the person in the team who can fix the detected issues the fastest.

Rust already comes with a set of tools to establish a meaningful Loop safeguard:

- clippy - code linting that checks unidiomatic code, overly-complex constructs and common mistakes/inefficiencies.
- rustfmt - code formatter
- audit - checks if vulnerabilities have been reported for any of the crates in the dependency tree of your project.

To install all three components, run:

```
rustup component add clippy
rustup component add rustfmt
cargo install cargo-audit
cargo install cargo-outdated
```

Designing a loop safeguard requires you to make priorities explicit. For example, does your project value security? Add more static security checks. Speaking of security checks, cargo audit supports an experimental feature to automatically update Cargo.toml to fix vulnerable dependencies.

```
cargo install cargo-audit --features=fix
```

Does your organisation value performance? Add checks that test for slow code. However, requirements vary from project to project even within the same company and, to address that reality, I use bash scripts to run checks on various stage with the overall idea to keep the steps in a workflow the same, but customizing the scripts in each step.

For the loop safeguard, lets add linting, formatting, and security checks to cover the basics in form of a blueprint bash script by creating a script called 'pre-check.sh' and adding the content shown in listing 1.1. As shown in the script, you may want to customize the linting, whether cargo audit fixes dependencies automatically, or add any other custom checks specific to your project. Once your script is ready, add it to a githook executed prior to any push to ensure that only verified code makes it into your CI pipeline.

Listing 1.1: Script to run code checks before push

```

#!/usr/bin/env bash
set -o errexit  # exit when a command fails.
set -o nounset  # exit when using an undeclared variables.
set -o pipefail # exit when anything fails.

# Check for unformatted code.
# https://github.com/rust-lang/rustfmt
fmt -- --check

# Security checks
# https://lib.rs/crates/cargo-audit
# https://github.com/rustsec/rustsec
cargo-audit audit

# Use fix when you want to update your Cargo.toml automatically
# cargo-audit fix

# Linting
# https://github.com/rust-lang/rust-clippy#configuration
cargo clippy -- -D warnings

# This allows all lints, but warns only in specific cases
# cargo clippy -- -A clippy::all -W clippy::bool_comparison

# Dependecny checks similar to GH Dependabot
cargo outdated
```

# 2 Memory

About 70% of the vulnerabilities Microsoft assigns a CVE each year continue to be memory safety issues[1]. Even more staggering, the majority of vulnerabilities fixed are caused by developers inadvertently inserting memory corruption bugs into their C and C++ code. These findings are quite common, as shown in various industry statistics for memory related vulnerabilities:

- Two-thirds of Linux kernel vulnerabilities come from memory safety issues[2].
- Multiple studies on Apple's operating systems document that 70% to 80% of vulnerabilities in MacOS are memory safety vulnerabilities (2019[3], 2020[4], 2021[5]).
- Google estimated that 90% of Android vulnerabilities are memory safety issues[6].
- 70% of all Chrome security bugs are memory safety issues.
- An analysis of 0-days exploit in the wild found that more than 80% of the exploited vulnerabilities were memory safety issues[7].

Microsoft concludes that there is a real need for a language considered safe from memory corruption that also offers performance on par with C or C++. Rather than providing guidance and tools for addressing flaws, we should use programming languages that prevent the developer from introducing the flaws in the first place. According to the MSRC, Rust fits the requirements well by offering comprehensive memory protection, fast and deterministic execution speed while preventing programmers from inadvertently inserting memory corruption bugs. Writing secure and reliable software, however, requires understanding the underlying hardware and the standard memory model underlying most modern programming languages.

[1]https://msrc-blog.microsoft.com/2019/07/16/a-proactive-approach-to-more-secure-code/
[2]https://static.sched.com/hosted_files/lssna19/d6/kernel-modules-in-rust-lssna2019.pdf
[3]https://langui.sh/2019/07/23/apple-memory-safety/
[4]https://langui.sh/2020/07/24/apple-memory-safety/
[5]https://langui.sh/2021/12/13/apple-memory-safety/
[6]https://security.googleblog.com/2019/05/queue-hardening-enhancements.html
[7]https://twitter.com/LazyFishBarrel/status/1129000965741404160

## 2.1 Hardware

Modern memory follows a multi-tier hierarchy that leads to a trade-off between small and fast memory (usually CPU caches) or large and slow memory (Ram). As seen in table2.1, the smaller the memory tier, the faster the access time as measured in latency. However, as modern operating systems and applications increasingly render graphics on the GPU, the actual bottleneck is not memory itself, but rather the PCI Express (PCI-E) interconnect that sits between CPU and GPU.

| Type | Size | Latency | Throughput |
|---|---|---|---|
| CPU Registers | 4 - 8 Bytes | 1 ns | ~ |
| CPU L1 - L3 Cache | KB - MB | 1 - 4 ns | 2.3 TB/s (L1) - 370 GB/s (l3) |
| RAM | GB | 100 ns | 51.2 GB/s (DDR5) |
| Disk | GB - TB | 16,000- 4,000,000ns | 80 MB/s (HDD) - 5000 MB/s (SSD) |

Table 2.1: Memory performance

PCI-E v4, for example, provides a maximum throughput of 31.508 GB/s in 16 lane configuration[8] that is commonly used for high-end GPU. While this may sound plenty, it is significantly below the maximum throughput that the DDR5 memory achieves, not to mention future generations of even faster memory.

More profoundly, for each render GPU operation, memory needs to be copied from the system's main memory to the GPU memory and after the completion back from the GPU memory to the main memory.GPU vendors are keenly aware of this limitation and actively pushing the PCI Express standard towards more throughput with each new version. However, when programming anything that renders or calculates on a GPU, the PCI-E induced latency and overhead needs to be considered when optimizing performance.

As seen in table 2.1, both memory latency and throughput determine the overall performance and responsiveness of any software running on a system. Semiconductor manufactures are keenly aware of this trade-off of and, especially, the reality that higher tier memory is incredibly fast but also incredibly expensive per byte.In response, modern CPU design usually try to find a solid middle ground between economics and performance. Optimization, however, varies by CPU architecture. Intel and AMD tend to optimize the x64 architecture, for example, by increasing L3 cache while keeping CPU register size roughly the same size.

[8]https://en.wikipedia.org/wiki/PCI_Express

ARM64 often merges CPU and GPU on a single System on a Chip (SoC) with the implication that increasing interconnect speed often increases performance at a reasonable cost without increasing expensive CPU registers. One particularly interesting example of ARM64 is the M1 architecture Apple introduced in 2021. Apple follows the ARM design of combining CPU and GPU on one single SoC, but also combines CPU and GPU memory in what Apple calls "unified Memory" that eliminates the PCI-E bottleneck altogether. By implication, GPU rendering on M1 works without any memory copying that allows more efficient optimizations. Because of the direct connection between the M1 SoC and the unified memory, Apply can increases system performance by simply increasing the interconnect speed and, in fact, the only tangible differentiator of the M1 product line up is the actual speed of the unified memory.

Effective software performance optimization requires a solid understanding of the underlying hardware to achieve mechanical sympathy, a term coined by Martin Thompson[9]. Mechanical sympathy refers to the practice of writing code that leverages the memory hierarchy by performing chunks of work on data that is co-located, and processing in a predictable pattern. Initially, the CPU cache miss rate needs to be assessed by using standard tools like perf[10]. Then, the most commonly used data structure needs to fit in the l1 or L2 cache. To do so, it is paramount to calculate the actual struct size including padding to ensure efficient cache utilization while leaving space for other code. Then, processing collections needs to be batched in chunks that fit in the L2 / L3 cache of the CPU. When the loader feeds a complete batch of data into, say, the L3 cache and each item fits nicely in the L1 / L2 cache, then the cache rate will go down substantially and increase performance by multiple orders of magnitude.

## 2.2 Memory Model

Most modern programming languages apply a memory model similar to C; therefore, the C model serves as a reference. A typical memory model of a C program consists of a text segment that stores instructions, initialized and uninitialized data segments, the heap, and the stack.

A text segment is a program's segment in memory containing executable instructions. The text segment memory region is usually stored below the heap or the stack address space to prevent memory overwriting. Furthermore, the text segment is often read-only to prevent instructions modifications; a critical security best practice because it prevents stack overflow attacks that would allow malicious code injections.s.

The initialized data segment is a portion of the virtual address space of a program containing the global and static variables. This memory segment can be further divided into the initialized read-only area and the initialized read-write area. The read-only area

[9]https://mechanical-sympathy.blogspot.com/2012/08/memory-access-patterns-are-important.html
[10]https://perf.wiki.kernel.org/index.php/Tutorial

stores global constants that are known to be invariant. The read-write area contains all other variables that may change during runtime.

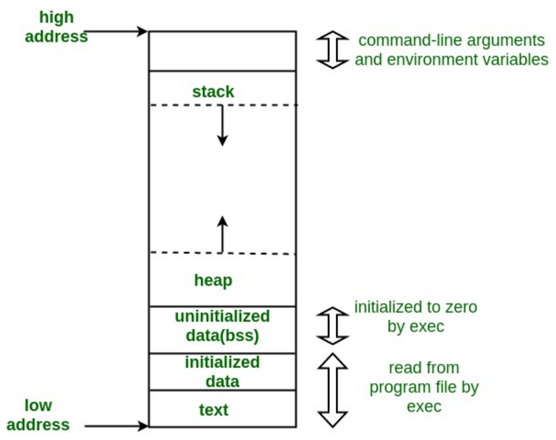

high address
command-line arguments and environment variables
stack
heap
uninitialized data(bss)
initialized to zero by exec
initialized data
read from program file by exec
text
low address

Figure 2.1: memory layout

The uninitialized data segment contains all global variables and static variables that the kernel initializes to zero in case they do not have explicit initialization in the source code. Accessing an uninitialized variable causes the infamous NullPointerException at runtime because the operating system kernel is assigned null by default. A compiler warning should be issued by default to prevent runtime access of non-initialized variables.

The stack area is located next to the heap area and grows in the opposite direction, which means that when the stack pointer met the heap pointer, all available free memory was exhausted. The stack area is a LIFO structure, typically located in the higher parts of memory. On the standard PC x86 computer architecture, it grows towards address zero, as shown in figure 2.1.

A "stack pointer" tracks the top of the stack, which means it is adjusted each time a value is added, a.k.a. "pushed" onto the stack. Conversely, a removed value is "popped." Stack values are pushed in "Last In, First Out" (LIFO) order which means all values are removed in reverse order.

Stack allocation happens on contiguous blocks of memory because the allocation occurs in the function call stack of the program. The size of stack-allocated memory is known to the compiler, and whenever a function call occurs, its variables get memory allocated on the stack. Whenever the function call ends, the memory for the variables is de-allocated. Stack memory allocation is also known as temporary memory allocation because once a function call completes its execution, all data belonging to it flushes out from the stack automatically. Stack memory de-allocation is automatic because the compiler inserts predefined code routines to de-allocate stack memory after usage.

Heap is the segment where dynamic memory allocation usually takes place. The heap begins at the end of the uninitialized data segment and grows dynamically in size. The memory is allocated dynamically during the execution of instructions written by programmers. It is called heap memory because it is a pile of memory space available to programmers to allocate and de-allocate during program runtime.

Each time a program creates an object, the memory for the object is heap-allocated, but the reference to the object is stack allocated. The stack only accepts fixed-sized values; therefore, a reference to the object stored in the heap memory is placed on the stack from which the program accesses the object.

Heap memory allocation isn't as safe as stack memory allocation because the data stored in this space is visible to all threads. In contrast, stack-allocated memory is only visible to a function call-stack. Also, the stack memory gets de-allocated automatically, whereas heap memory needs to be de-allocated explicitly by the programmer or a runtime.

## 2.3 Memory Safety

Rust moved several memory-safety features that previously required either specialized tooling or runtime checks into its compiler enabling static memory checks. As seen in figure 2.2, Rust enables both stack and heap memory safety by default.
Rust accomplishes static memory safety verification through an affine linear type system. Linear types ensure that objects are used at most once, allowing safe de-allocation. In that sense, the Rust compiler operates similarly to an automated theorem prover because it applies logic rules to the program to conclude whether all memory access conforms to specified rules. Therefore, safe memory access becomes fully decidable in Rust, which means a significant subset of exploitable code vulnerabilities result in compiler errors.

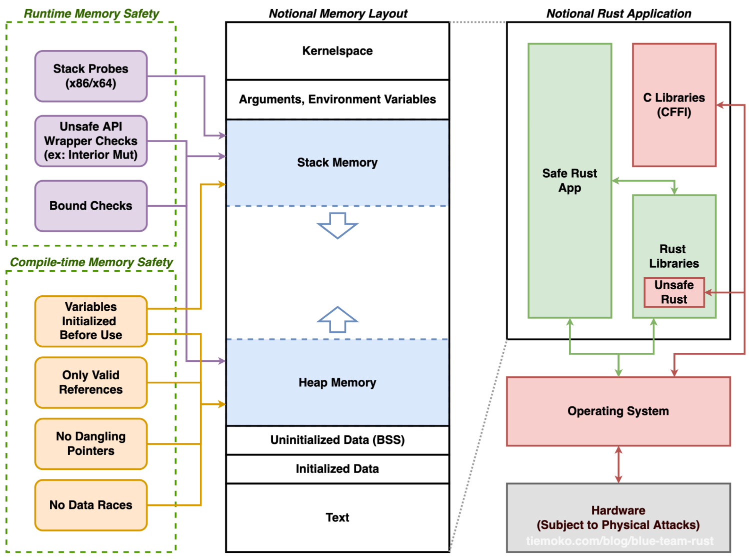


Figure 2.2: Rust memory safety

### 2.3.1 Stack protection

Any attempt to write past the end of a buffer in Rust will trigger a panic instead of leading to stack overflow. The resulting panic prevents attacker-controlled stack corruption, so your program won't fall victim to Arbitrary Code Execution exploits[11]. Likewise, Rust stops any attempt to read past the end of a buffer, preventing so-called Buffer-Overread exploits such as the infamous Heartbleed[12] vulnerability, which rendered millions of websites vulnerable.

[11]https://en.wikipedia.org/wiki/Arbitrary_code_execution
[12]https://heartbleed.com/

The Heartbleed bug would not have been possible if OpenSSL were implemented in Rust because Rust's memory management prevents Buffer-Over-Read, the origin[13] of the Heartbleed vulnerability. With Rust, no disastrous leakage of sensitive data can occur as it is impossible to read beyond any data structure (3).

### 2.3.2 Heap protection

Bounds checks and end-of-buffer panic also apply to heap-allocated objects. Furthermore, Rust's ownership paradigm eliminates dangling pointers, preventing Use-After-Free[15] and Double-Free[16] vulnerabilities. Ownership rules are enforced statically during compilation covering all possible dynamic executions. Therefore, a safe Rust program cannot enter an invalid memory state (3).

### 2.3.3 Reference protection

Rust does not allow manipulation of raw pointers, ensuring that pointer dereferences are valid. By implication, that means no pointer arithmetic, no null dereference, and no pointer manipulation for arbitrary read/write is possible. Similarly, the compiler prevents the infamous NullPointerException by ensuring access only to initialized variables.

The problem with null is that it conveys different meanings depending on the context. This conceptual overloading means that null has more than one meaning, violating the principle of single responsibility. For example, in Golang, nil may refer to the zero value of pointers or interfaces, but it also means a non-initialized object or the absence of an error. How do you know? Tony Hoare, the inventor of null, calls this his billion-dollar mistake[17] because countless programming bugs resulted from incorrect or missing null checks leading to numerous software crashes (3).

Rust disentangles the many meanings null may convey and represents each concept separately. For example, Rust uses an enum Option[18] to encode the idea of a value being present (some) or absent (None). Notice the difference; None is an enum you can check before accessing the value it represents; thus, it is impossible to throw a NullPointerException at runtime.

1314

[15]https://cwe.mitre.org/data/definitions/416.html
[16]https://cwe.mitre.org/data/definitions/415.html
[17]https://hackernoon.com/null-the-billion-dollar-mistake-8t5z32d6
[18]https://doc.rust-lang.org/book/ch06-01-defining-an-enum.html

Similarly, Rust represents the situation that a function call may or may not return with an enum Result[19], a type that represents either success (Ok) or failure (Err). Like the Option type, you already know upfront whether the function call returns a value you can access; if not, you know that you have to handle the returned error. With the Result type, it is impossible to throw a NullPointerException at runtime.

Rust allows delayed (lazy) initialization on immutable fields as long as all branches of the delayed initialization are exhaustive. Again, even for lazy initialization, it is impossible to trigger a NullPointerException at runtime. A very similar mechanism applies to constant initialization using const functions which ensures whatever the initialization function might be, it must be deterministic (always returning the same result) and not null. No value in Rust can be null in the sense of a null pointer because Rust guarantees consistently valid dereferencing once code compiles.

### 2.3.4 Data race protection

Rust's ownership system ensures that any given variable can only have one writer (or mutable reference) at any given time but an unlimited number of readers. Thus, Rust security guarantees an absence of data races because it is impossible to alias a mutable reference. Therefore it is impossible to perform a data race. However, Rust cannot prevent general race conditions because this would be impossible for any programming language, given that the underlying operating system or access to shared hardware can lead to deadlocks and race conditions regardless of the programming language[20].

[19]https://doc.rust-lang.org/book/ch09-02-recoverable-errors-with-result.html
[20]http://web.mit.edu/rust-lang_v1.25/arch/amd64_ubuntu1404/share/doc/rust/html/nomicon/races.html

## 2.4 Memory Management

In programming languages, there are three fundamental ways to manage memory. The first is manual memory management, where the programmer explicitly defines memory usage. The second is automated memory management, often through a virtual machine that uses a so-called garbage collector to free unused memory. The third one is compiler-added memory management, also known as Automatic Reference Counting (ARC).

Manual memory management is more efficient and often results in better performance, although it is complex, error-prone, and can lead to significant security problems. While it is relatively easy to write C code that runs fast, it is tough to write robust C or C++ code without major memory bugs because of the many things allowed in a C program that will go wrong. However, decades of experience have led to sophisticated tooling and best practices that aid engineers in writing secure C code.

Programming languages that rely on automated memory management are easier to learn and less complex but, at the same time, less efficient because the runtime requires more memory. Also, designing a programming language for a runtime like the JVM generally enables improved memory safety and a higher level of abstraction.

The third paradigm is the concept of the compiler adding implicit memory management code so that a programmer focuses on only writing application code. This idea was pioneered in Objective-C with Automatic Reference Counting (ARC) to retain, release, or auto-release memory. ARC became central to Swift, which became the de-facto successor to Objective-C as the primary programming language for the entire Apple platform. Those familiar with Swift recognize its simple but elegant syntax, free of memory management but also free of any runtime. And that naturally raises the question of how ARC works.

The compiler adds ARC by analyzing code and then emitting deterministic memory management code that handles all values and references correctly. Because ARC relies on compiler-generated memory management code, it simplifies the programming language and enables increasing optimization through compiler updates not available to manually maintained code. ARC generally leads to fewer memory leaks. Every time you create a new class instance, ARC allocates a chunk of memory to store information about that instance. ARC automatically counts how many properties, constants, and variables are currently referring to each class instance. ARC will not deallocate an instance as long as at least one active reference is present. Once the reference count drops to zero, the class and its memory get deallocated.

Swift accomplishes this feat by automatic code generation and re-writing during compilation. Specifically, the Swift compiler adds generated resource count and de-allocation code to each class to ensure memory management works without a runtime while offering modern programming concepts such as Async, Traits, and Protocols.

By default, Swift prevents unsafe practices. For example, Swift checks that variables are initialized before usage, memory is not accessed after de-allocation, and array bounds are checked. Swift makes a convincing case for a modern yet efficient programming language. Considering the generally positive impact of ARC, it is surprising that more modern programming languages are not using it. However, there are a couple of less obvious problems:

1. Slow compiler time.
2. Performance becomes non-deterministic.
3. Optimization becomes complex.

The Swift compiler is already notoriously slow on a relatively small code base. Larger Swift projects often use more advanced build tools such as Bazel to leverage incremental compilation, primarily to avoid excessively long compile times. Because of the code generation and code-rewrite required by ARC, compiler optimization would become significantly harder than it already is. For the same reason, the performance of Swift programs varies widely. Well-written Swift code can get close to the execution speed of C, but these instances are relatively rare. More often, the performance of Swift programs trails C by at least an order of magnitude.

Automatically generated ARC code incurs a particular overhead during execution. Especially when reference counts have to be updated often, but in a non-predictive pattern, it leads to non-deterministic performance, one way or the other. However, optimizing Swift code for performance is non-trivial because of the ARC management generated under the hood. In all fairness, the Swift documentation is unambiguous that Swift exposes complexity gradually, meaning that topics as complex as performance optimization require more expertise to master.

Rust, however, aims for a different goal: Safe, reliable, and performant system programming. Most system programming these days is still happening in C with all the implied complexity of manual memory management. Rust offers a compelling alternative for the first time by combining the low-level performance similar to C with high-level abstraction similar to Swift or Haskell while also improving static memory safety.

Rust expands the third paradigm further by applying rule-based static checks on memory safety in addition to the established type and safety checks a compiler already performs. By doing so, Rust moves into the compiler what used to be the primary motivation of building a run-time like the JVM. Rust also eliminates a large part of the ARC generated code which results in deterministic performance. Furthermore, Rust provides low level programming similar to C, but also high level zero cost abstraction syntax similar to Swift, Scala or Haskell.

## 2.5 Rust memory management

Rust solves the seemingly intractable problem of knowing ahead of time how long a value or reference is going to stay in memory so it can safely be removed afterwards. Unlike ARC, Rust does not generate reference counting code, but rather Rust's ownership paradigm implements an affine type system that enables safety without sacrificing predictable performance. To do so, Rust developed a comprehensive type system around memory usage across five dimensions:

1. Mutable vs. immutable data
2. Value vs. reference type
3. Single vs. shared ownership
4. Read vs. write access
5. Single vs. multi-threaded processing

### 2.5.1 Mutable vs. Immutable

In Rust, a value is declared with the *let* keyword and immutable by default. Type annotations are optional because in most instances these can be inferred automatically. Mutability must be declared explicitly with the with the *mut* keyword which enables the compiler to verify read/write memory access efficiently. Constants in Rust are always immutable, declared with the *const* keyword, and the type of the constant value must be annotated. Rust's naming convention for constants follows the C convention to use all uppercase with underscores between words. One particularity is that a constant in Rust can evaluate a limited number of operations and const function at compile time thus allowing a convenient syntax. Listing 2.1 shows the Rust syntax for declaring constants, immutable and mutable values.

Listing 2.1: Mutability

```rust
const THREE_HOURS: u32 = 3;
const THREE_HOURS_IN_MINUTES: u32 = 60 * 3;

fn main() {
    let x = 5;
    println!("The value of x is: {x}");
    // x = 6; //  Throws: cannot assign twice to immutable variable

    let mut y = 5;
    println!("The value of y is: {y}");
    y = 6;
    println!("The value of y is: {y}");
}
```

### 2.5.2 Values vs. Reference

For a value, its lifetime is self-evident: a value exists when it comes into scope and will be de-allocated the moment it scopes ends. Values are moved between different owners through assignment or passing a value as a function parameter. Functions taking ownership of a parameter are said to consume the value after which the value is no longer accessible anymore. A violation of this principle leads to the compiler error "value used here after move". In most cases, the Rust compiler correctly suggests to "borrow" the value, which means passing a reference. This leads to a very important distinction:

**In Rust, values are moved; references are borrowed.**

When you declare a variable in Rust, the variable "owns" the data. When a variable owns something it can move it to other variables using assignment. After giving away its data, the old variable cannot access the value anymore, and the new variable is the new owner. That means, when you move a value to a different function, you cannot use that value after it has been moved to a function. Conversely, if you create a value within a function, you can only return a value a.ka. handing it over to a new owner.

To illustrate the application of the ownership rules, consider the code shown in listing 2.2. The function print_string takes a value and because of that it also takes ownership. That means, whatever value gets passed into the function also implies ownership gets transferred to the print_string function. When the function ends, in line 3, the owner goes out of scope and with it the value of s. Therefore, s cannot be accessed after the completion of the print_string function.

Listing 2.2: Value ownership

```rust
1 fn print_string(s: String) {
2     println!("{}", s);
3 } // owner and s goe out of scope
4
5 fn main() {
6     let s = String::from("Hello, World");
7     print_string(s);  // ownership of s moved to print_string
8     // another print_string(s); would result in an error
9 }
```

The moment a value's owner goes of scope, the value goes will be gone. There is a catch, though, what if you want to process your data by more than one function and then re-use the result? In this case, you need to use references. A reference is simply an address pointing to a value in memory. A reference is similar to a pointer in C because it is address we can follow to access the data stored at that address. Unlike a pointer in C, a reference is guaranteed to point to a valid value of, as discussed in section 2.3.3.

When you create an immutable value *let a = 5;*, that value will be stored in memory. Depending on the type, the value is either stored on the stack (primitive types such as numbers or bool) or on the heap (String or objects). When you create a second value b that is also 5, you end up with two values in memory. This adds up over time. Conversely, when you take a reference that points to a, you only store a pointer, which is significantly smaller and is always stored on the stack. Figure 2.3 shows the schema of a reference in Rust for a pointer *s1* to a string with value "Hello". The pointer *s1* stores only a few data, among it the length of the referenced data.

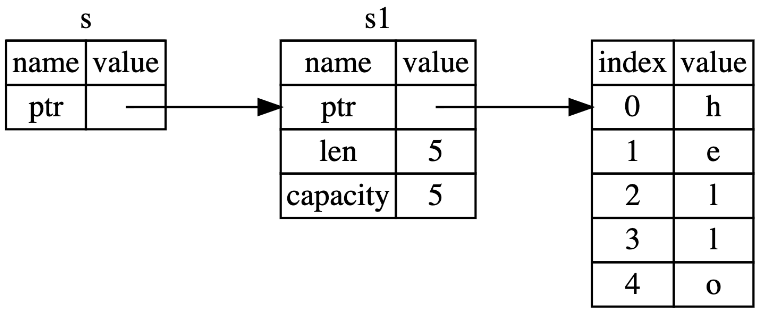


Figure 2.3: Rust reference

Remember, in Rust, values are moved; references are borrowed. That means, borrowing allows another variable to temporarily borrow the data in your variable and gives it back when its done. Rust allows you to have two types of references:

- Immutable reference: "&" indicates read-only data access
- Mutable reference: "&mut" indicates and write data access.

The ampersand symbol represents a reference and allows you to refer to some value without taking ownership of it. The opposite of referencing is dereferencing, which is accomplished with the dereference operator *, which returns the actual value.

Per the above rules, an owner must live longer than any borrowed reference. Rust's memory safety depends to a large degree on the following rules:

- There may be many immutable references
- Or only one mutable reference.
- You either hold immutable or mutable references but never both at the same time.
- A mutable reference cannot be aliased

### 2.5.3 Ownership

Ownership allows Rust to be completely memory-safe and efficient, while avoiding garbage collection. Rust manages memory through a set of rules that the compiler checks. If any of the rules are violated, the program won't compile. None of the features of ownership will impact runtime performance.At a high-level, the rules determine two pieces of information for every value in a program:

1. Valid scope: Where the value is valid and thus can be used.
2. Access type: read-only or writable.

These rules apply to both values and references, albeit in different ways. The Rust compiler enforces three ownership rules to ascertain memory safety:

1. **Each value in Rust has one owner.**
2. **There can only be one owner at a time.**
3. **When the owner goes out of scope, the value will be dropped.**

# Bibliography

[1] S. Klabnik and C. Nichols, *The Rust Programming Language.* No Starch Press, 2019.

[2] L. Palmieri, *Zero To Production In Rust.* Gumroad, first edition ed., 3 2022.

[3] A. J. Tiemoko Ballo, Moumine Ballo, *High Assurance Rust: Developing Secure and Robust Software.* Github, first edition ed., 10 2022.