Engineer

Rust

a compiled language that checks memory for you

What it is

Rust is a programming language for the work traditionally done in C and C++, where a program manages its own memory and is often linked into software written in other languages. Its compiler produces native machine code ahead of time. Version 1.0 was announced on 15 May 2015 as a language combining low-level control over performance with safety guarantees, reached without a garbage collector or a runtime, which is what lets a Rust library stand in for a C one.

Memory is managed through ownership, a set of rules the compiler checks. Each value has one owner at a time, and once the owner leaves its scope, the value is dropped and its memory freed, at a point fixed when the program is compiled rather than whenever a collector next looks. Code that needs a value without taking it borrows a reference, and the borrowing rules allow, at any moment, either one mutable reference or any number of immutable ones, and require every reference to be valid. A program that breaks a rule does not compile. The same rules exclude data races, so two threads writing to the same data without synchronisation is a compile error rather than an intermittent fault.

Rust has no exceptions. A function that can fail in an expected way returns a `Result`, holding either a value or an error, and the compiler warns by default if the caller leaves the result unused; a bug the program cannot recover from, such as reading past the end of an array, goes through the `panic!` macro instead.

Static analysis is conservative, so the compiler rejects some programs that would have been fine, and some work, such as driving hardware or calling another language, cannot be checked at all. For those cases there is `unsafe`, which unlocks five extra abilities, among them dereferencing a raw pointer and calling an unsafe function, which by default includes any function declared from C. It

does not switch off the borrow checker; it marks the lines where the programmer, not the compiler, vouches for memory safety, so the safety of those operations depends on the programmer maintaining their contracts, including in surrounding safe code. The Rust Book's advice is to keep those blocks small and wrap them in a safe interface, which is how parts of the standard library are built.

A new compiler is released every six weeks. Changes that would break existing code wait for an edition, which arrives every three years or so. There are four, 2015, 2018, 2021 and 2024, and the newest became stable on 20 February 2025 with Rust 1.85.0. Each package names its edition in `Cargo.toml`, every compiler builds all the editions that came before it, and packages of different editions link together, so a project moves up when it chooses. Cargo, the dependency manager and build tool, is included with the language, rustfmt keeps the code style consistent, and third-party libraries arrive as packages called crates.

Use the concept in a decision

Put "What is Rust?" to work in a real decision. We can help relate the concept to the system you are building or buying. The part users never see: where the data lives, what the rules are, and how other systems are allowed to talk to it.

Explore our services Get in touch →

Test the assumptions behind the term

Use the explanation and linked work to frame questions for your project. We can examine the assumptions with you before a tool is chosen. Designed and ran 6 ETL/ELT pipelines — 6 ETL pipelines across three database engines

Designed and ran 6 ETL/ELT pipelines Get in touch →

Why it matters

In C and C++, a pointer used after its memory has been freed, or two threads writing the same data, compiles without complaint and fails later, intermittently and often far from the cause. Rust turns those programs into compile errors, and because the checks run at build time, the finished program pays nothing for them.

With no garbage collector and no runtime to carry along, Rust code can sit underneath software in other languages. Anything that can call C can call a Rust core through a C interface, so one well-tested implementation can serve clients on several platforms. The Rust Book lists command-line tools, web services, embedded devices and major parts of the Firefox web browser among its production uses.

The cost is paid up front. Code has to be shaped so that the compiler can see it is safe, which often means deciding who owns a value instead of passing references around freely, and the Book itself says that ownership takes time to get used to. When the compiler refuses a program it cannot verify, the answer is usually a different structure rather than a way around the check.

Stability is the other half of the bargain. Since 1.0, breaking changes have been largely ruled out, so a change that could break old code arrives only through an edition that a package opts into.

What it looks like in practice

In the portfolio, Rust is the core underneath a native macOS messaging client: the protocol's well-tested core is written in Rust and exposes a C interface, and the client, 11,141 lines of Swift across 53 files, links it as a static archive compiled from a pinned upstream revision. Pinning matters because the C interface is the contract: a moving core changes the contract without changing a line of Swift. Everything the C interface returns carries an ownership rule, and getting one wrong produces a leak or a crash with no compiler warning in between, so a thin wrapper owns every pointer and every string that crosses the boundary, and no layer above it sees a raw pointer.

Linking that core into an application means shipping everything it depends on, which is the subject of a licence audit of 644 Rust crates. Each crate declares its licence as an expression, and the audit checks every expression in the full dependency graph against an allow list on every build, reading OR as a choice between licences and AND as an obligation to honour both. Anything unmatched fails the build. Because a check that has never failed looks the same as one that cannot, it was made to fail twice on purpose: once by rewriting a crate's licence to a term the list rejects, and once by moving the pinned core revision without regenerating the audit. It reads declared metadata only and is not legal advice.

This site has no Rust source of its own; it meets the language at one remove, through Ruff, which is written in Rust and checks every Python script a commit touches.

Where it goes wrong

Most Rust trouble sits at the edges of what the compiler checks: arithmetic in an optimised build, the boundary with another language, and code that arrives with a dependency.

Arithmetic wraps in a release build. A counter that panicked with an overflow during testing turns silently into a small number in production. In debug builds Rust checks integer arithmetic for overflow and panics; a build with `--release` leaves the check out and wraps around, so 256 in a `u8` becomes 0, and the Book treats relying on that wrap as an error. Where a value can overflow, choose the behaviour explicitly: `checked_*` returns `None`, `saturating_*` stops at the limit and `wrapping_*` wraps on purpose.

```
fn add_stock(on_hand: u8, delivered: u8) -> Option<u8> {
    // None when the sum does not fit, in debug and release alike
    on_hand.checked_add(delivered)
}
```

Memory bugs come back at the C boundary. A program whose Rust side builds without a single warning leaks memory or crashes inside a foreign call. Functions declared in an `extern` block are unsafe to call because the other language enforces none of Rust's rules, so who frees each pointer is a convention the compiler cannot see. Put one wrapper in charge of every pointer and string that crosses, write each ownership rule down beside the call, and expose only a safe interface above it.

A panic reaches C and the process aborts. A Rust function called from C panics, and instead of an error the whole program stops. Since Rust 1.81, released on 5 September 2024, a panic that tries to leave a function declared with the plain `"C"` interface aborts the process. Turn failures into return values before they reach the boundary, and declare a function with `"C-unwind"` only where the code on the other side expects to unwind.

A routine update changes the licence of the whole build. A version bump three levels down brings in a crate whose licence the product cannot accept, and nothing in the build says so. Each crate declares its own licence, and the

obligations add up across the whole graph. Check every crate's expression against an allow list on every build, fail on anything unmatched, and break the check on purpose once to prove that it fires.

Further reading

- The Rust Programming Language — the official book, from installation through ownership, error handling and unsafe code.
- What Is Ownership? — the three ownership rules and how memory is freed without a garbage collector.
- Unsafe Rust — the five unsafe abilities, raw pointers and calling C through `extern`.
- The Manifest Format — the `edition`, `rust-version` and `license` fields of `Cargo.toml`.
- Announcing Rust 1.0 — the original aims and the stability promise that began with 1.0.



Glossary

Choose for your actual constraints

Describe the workflow, the existing interfaces and the users who need the change. We can define a testable delivery and a handover that lets your team maintain it.

See what it costs Get in touch →

Make the terminology practical

Ask how “What is Rust?” applies to your organisation. A concrete example from your current system is enough to start the discussion.

Explore our services

welcome@engineer.company +45 50 10 61 84 Copenhagen, Denmark

Take this page with you

Contact card

Carefully Crafted With Love by D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Copyright © 2025 – present · Engineer ApS · All rights reserved.

Printed from https://engineer.company/glossary/rust/