(https://github.com/rust-lang/rust)

(/en-US/index.html)

Documentation (/en-US/documentation.html) Install (/en-US/install.html) Community (/en-US/community.html) Contribute (/en-US/contribute.html)

Frequently Asked Questions

This page exists to answer common questions about the Rust programming language. It is not a complete guide to the language, nor is it a tool for teaching the language. It is a reference to answer oft-repeated questions people in the Rust community encounter, and to clarify the reasoning behind some of Rust's design decisions.

If there is some common or important question you feel is wrongly left unanswered here, feel free to help us fix it (https://github.com/rust-lang/rust-www/blob/master/CONTRIBUTING.md).

Table of Contents

(#toggle-toc)

1. The Rust Project (#project)
2. Performance (#performance)
3. Syntax (#syntax)
4. Numerics (#numerics)
5. Strings (#strings)
6. Collections (#collections)
7. Ownership (#ownership)
8. Lifetimes (#lifetimes)
9. Generics (#generics)
10. Input / Output (#input-output)
11. Error Handling (#error-handling)
12. Concurrency (#concurrency)
13. Macros (#macros)
14. Debugging and Tooling (#debugging)
15. Low-Level (#low-level)
16. Cross-Platform (#cross-platform)
17. Modules and Crates (#modules-and-crates)
18. Libraries (#libraries)
19. Design Patterns (#design-patterns)
20. Other Languages (#other-languages)
21. Documentation (#documentation)

The Rust Project

What is this project's goal? (#what-is-this-projects-goal)

To design and implement a safe, concurrent, practical systems language.

Rust exists because other languages at this level of abstraction and efficiency are unsatisfactory. In particular:

1. There is too little attention paid to safety.

2. They have poor concurrency support.
3. There is a lack of practical affordances.
4. They offer limited control over resources.

Rust exists as an alternative that provides both efficient code and a comfortable level of abstraction, while improving on all four of these points.

Is this project controlled by Mozilla? (#is-this-project-controlled-by-mozilla)

No. Rust started as Graydon Hoare’s part-time side project in 2006 and remained so for over 3 years. Mozilla got involved in 2009 once the language was mature enough to run basic tests and demonstrate its core concepts. Though it remains sponsored by Mozilla, Rust is developed by a diverse community of enthusiasts from many different places around the world. The Rust Team (https://www.rust-lang.org/team.html) is composed of both Mozilla and non-Mozilla members, and `rust` on GitHub has had over 1,900 unique contributors (https://github.com/rust-lang/rust/) so far.

As far as project governance (https://github.com/rust-lang/rfcs/blob/master/text/1068-rust-governance.md) goes, Rust is managed by a core team that sets the vision and priorities for the project, guiding it from a global perspective. There are also subteams to guide and foster development of particular areas of interest, including the core language, the compiler, Rust libraries, Rust tools, and moderation of the official Rust communities. Designs in each of these areas are advanced through an RFC process (https://github.com/rust-lang/rfcs). For changes which do not require an RFC, decisions are made through pull requests on the `rustc` repository (https://github.com/rust-lang/rust).

What are some non-goals of Rust? (#what-are-some-non-goals)

1. We do not employ any particularly cutting-edge technologies. Old, established techniques are better.
2. We do not prize expressiveness, minimalism or elegance above other goals. These are desirable but subordinate goals.
3. We do not intend to cover the complete feature-set of C++, or any other language. Rust should provide majority-case features.
4. We do not intend to be 100% static, 100% safe, 100% reflective, or too dogmatic in any other sense. Trade-offs exist.
5. We do not demand that Rust run on “every possible platform”. It must eventually work without unnecessary compromises on widely-used hardware and software platforms.

In which projects is Mozilla using Rust? (#how-does-mozilla-use-rust)

The main project is Servo (https://github.com/servo/servo), an experimental browser engine Mozilla is working on. They are also working to integrate Rust components (https://bugzilla.mozilla.org/show_bug.cgi?id=1135640) into Firefox.

What examples are there of large Rust projects? (#what-examples-are-there-of-large-rust-projects)

The two biggest open source Rust projects right now are Servo (https://github.com/servo/servo) and the Rust compiler (https://github.com/rust-lang/rust) itself.

Who else is using Rust? (#who-else-is-using-rust)

A growing number of organizations! (friends.html)

How can I try Rust easily? (#how-can-i-try-rust-easily)

The easiest way to try Rust is through the playpen (https://play.rust-lang.org/), an online app for writing and running Rust code. If you want to try Rust on your system, install it (https://www.rust-lang.org/install.html) and go through the Guessing Game (https://doc.rust-lang.org/stable/book/second-edition/ch02-00-guessing-game-tutorial.html) tutorial in the book.

How do I get help with Rust issues? (#how-do-i-get-help-with-rust-issues)

There are several ways. You can:

- Post in users.rust-lang.org (https://users.rust-lang.org/), the official Rust users forum
- Ask in the official Rust IRC channel (https://chat.mibbit.com/?server=irc.mozilla.org&channel=%23rust) (#rust on irc.mozilla.org)
- Ask on Stack Overflow (https://stackoverflow.com/questions/tagged/rust) with the “rust” tag
- Post in /r/rust (https://www.reddit.com/r/rust), the unofficial Rust subreddit

Why has Rust changed so much over time? (#why-has-rust-changed-so-much)

Rust started with a goal of creating a safe but usable systems programming language. In pursuit of this goal it explored a lot of ideas, some of which it kept (lifetimes, traits) while others were discarded (the typestate system, green threading). Also, in the run up to 1.0 a lot of the standard library was rewritten as early designs were updated to best use Rust’s features and provide quality, consistent cross-platform APIs. Now that Rust has reached 1.0, the language is guaranteed to be “stable”; and while it may continue to evolve, code which works on current Rust should continue to work on future releases.

How does Rust language versioning work? (#how-does-rust-language-versioning-work)

Rust’s language versioning follows SemVer (http://semver.org/), with backwards incompatible changes of stable APIs only allowed in minor versions if those changes fix compiler bugs, patch safety holes, or change dispatch or type inference to require additional annotation. More detailed guidelines for minor version changes are available as approved RFCs for both the language (https://github.com/rust-lang/rfcs/blob/master/text/1122-language-semver.md) and standard library (https://github.com/rust-lang/rfcs/blob/master/text/1105-api-evolution.md).

Rust maintains three “release channels”: stable, beta, and nightly. Stable and beta are updated every six weeks, with the current nightly becoming the new beta, and the current beta becoming the new stable. Language and standard library features marked unstable or hidden behind feature gates may only be used on the nightly release channel. New features land as unstable, and are “ungated” once approved by the core team and relevant subteams. This approach allows for experimentation while providing strong backwards-compatibility guarantees for the stable channel.

For additional details, read the Rust blog post “Stability as a Deliverable.” (http://blog.rust-lang.org/2014/10/30/Stability.html)

Can I use unstable features on the beta or stable channel? (#can-i-use-unstable-features-on-the-beta-or-stable-channel)

No, you cannot. Rust works hard to provide strong guarantees about the stability of the features provided on the beta and stable channels. When something is unstable, it means that we can’t provide those guarantees for it yet, and don’t want people relying on it staying the same. This gives us the opportunity to try changes in the wild on the nightly release channel, while still maintaining strong guarantees for people seeking stability.

Things stabilize all the time, and the beta and stable channels update every six weeks, with occasional fixes accepted into beta at other times. If you’re waiting for a feature to be available without using the nightly release channel, you can locate its tracking issue by checking the `B-unstable` (https://github.com/rust-lang/rust/issues?q=is%3Aissue+is%3Aopen+tracking+label%3AB-unstable) tag on the issue tracker.

What are "Feature Gates"? (#what-are-feature-gates)

“Feature gates” are the mechanism Rust uses to stabilize features of the compiler, language, and standard library. A feature that is “gated” is accessible only on the nightly release channel, and then only when it has been explicitly enabled through `#[feature]` attributes or the `-Z unstable-options` command line argument. When a feature is stabilized it becomes available on the stable release channel, and does not need to be explicitly enabled. At that point the feature is considered “ungated”. Feature gates allow developers to test experimental features while they are under development, before they are available in the stable language.

Why a dual MIT/ASL2 License? (#why-a-dual-mit-asl2-license)

The Apache license includes important protection against patent aggression, but it is not compatible with the GPL, version 2. To avoid problems using Rust with GPL2, it is alternately MIT licensed.

Why a BSD-style permissive license rather than MPL or tri-license? (#why-a-permissive-license)

This is partly due to preference of the original developer (Graydon), and partly due to the fact that languages tend to have a wider audience and more diverse set of possible embeddings and end-uses than products such as web browsers. We’d like to appeal to as many of those potential contributors as possible.

Performance

How fast is Rust? (#how-fast-is-rust)

Fast! Rust is already competitive with idiomatic C and C++ in a number of benchmarks (like the Benchmarks Game (https://benchmarksgame.alioth.debian.org/u64q/compare.php?lang=rust&lang2=gpp) and others (https://github.com/kostya/benchmarks)).

Like C++, Rust takes zero-cost abstractions (http://blog.rust-lang.org/2015/05/11/traits.html) as one of its core principles: none of Rust’s abstractions impose a global performance penalty, nor is there overhead from any runtime system in the traditional sense.

Given that Rust is built on LLVM and strives to resemble Clang from LLVM’s perspective, any LLVM performance improvements also help Rust. In the long run, the richer information in Rust’s type system should also enable optimizations that are difficult or impossible for C/C++ code.

Is Rust garbage collected? (#is-rust-garbage-collected)

No. One of Rust’s key innovations is guaranteeing memory safety (no segfaults) *without* requiring garbage collection.

By avoiding GC, Rust can offer numerous benefits: predictable cleanup of resources, lower overhead for memory management, and essentially no runtime system. All of these traits make Rust lean and easy to embed into arbitrary contexts, and make it much easier to integrate Rust code with languages that have a GC (http://calculist.org/blog/2015/12/23/neon-node-rust/).

Rust avoids the need for GC through its system of ownership and borrowing, but that same system helps with a host of other problems, including resource management in general (http://blog.skylight.io/rust-means-never-having-to-close-a-socket/) and concurrency (http://blog.rust-lang.org/2015/04/10/Fearless-Concurrency.html).

For when single ownership does not suffice, Rust programs rely on the standard reference-counting smart pointer type, `Rc` (https://doc.rust-lang.org/std/rc/struct.Rc.html), and its thread-safe counterpart, `Arc` (https://doc.rust-lang.org/std/sync/struct.Arc.html), instead of GC.

We are however investigating *optional* garbage collection as a future extension. The goal is to enable smooth integration with garbage-collected runtimes, such as those offered by the Spidermonkey (https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey) and V8 (https://developers.google.com/v8/?hl=en) JavaScript engines. Finally, some people have investigated implementing pure Rust garbage collectors (https://manishearth.github.io/blog/2015/09/01/designing-a-gc-in-rust/) without compiler support.

Why is my program slow? (#why-is-my-program-slow)

The Rust compiler doesn't compile with optimizations unless asked to, as optimizations slow down compilation and are usually undesirable during development (https://users.rust-lang.org/t/why-does-cargo-build-not-optimise-by-default/4150/3).

If you compile with `cargo`, use the `--release` flag. If you compile with `rustc` directly, use the `-O` flag. Either of these will turn on optimizations.

Rust compilation seems slow. Why is that? (#why-is-rustc-slow)

Code translation and optimizations. Rust provides high level abstractions that compile down into efficient machine code, and those translations take time to run, especially when optimizing.

But Rust's compilation time is not as bad as it may seem, and there is reason to believe it will improve. When comparing projects of similar size between C++ and Rust, compilation time of the entire project is generally believed to be comparable. The common perception that Rust compilation is slow is in large part due to the differences in the *compilation model* between C++ and Rust: C++'s compilation unit is the file, while Rust's is the crate, composed of many files. Thus, during development, modifying a single C++ file can result in much less recompilation than in Rust. There is a major effort underway to refactor the compiler to introduce incremental compilation (https://github.com/rust-lang/rfcs/blob/master/text/1298-incremental-compilation.md), which will provide Rust the compile time benefits of C++'s model.

Aside from the compilation model, there are several other aspects of Rust's language design and compiler implementation that affect compile-time performance.

First, Rust has a moderately-complex type system, and must spend a non-negligible amount of compile time enforcing the constraints that make Rust safe at runtime.

Secondly, the Rust compiler suffers from long-standing technical debt, and notably generates poor-quality LLVM IR which LLVM must spend time "fixing." The addition of a new internal representation called MIR (https://github.com/rust-lang/rfcs/blob/master/text/1211-mir.md) to the Rust compiler offers the potential to perform more optimizations and improve the quality of LLVM IR generated, but this work has not yet occured.

Thirdly, Rust's use of LLVM for code generation is a double-edged sword: while it enables Rust to have world-class runtime performance, LLVM is a large framework that is not focused on compile-time performance, particularly when working with poor-quality inputs.

Finally, while Rust's preferred strategy of monomorphising generics (ala C++) produces fast code, it demands that significantly more code be generated than other translation strategies. Rust programmers can use trait objects to trade away this code bloat by using dynamic dispatch instead.

Why are Rust's `HashMap`s slow? (#why-are-rusts-hashmaps-slow)

By default, Rust's `HashMap` (https://doc.rust-lang.org/stable/std/collections/struct.HashMap.html) uses the SipHash (https://131002.net/siphash/) hashing algorithm, which is designed to prevent hash table collision attacks (http://programmingisterrible.com/post/40620375793/hash-table-denial-of-service-attacks-revisited) while providing reasonable performance on a variety of workloads (https://www.reddit.com/r/rust/comments/3hw9zf/rust_hasher_comparisons/cub4oh6).

While SipHash demonstrates competitive performance (http://cglab.ca/%7Eabeinges/blah/hash-rs/) in many cases, one case where it is notably slower than other hashing algorithms is with short keys, such as integers. This is why Rust programmers often observe slow performance with `HashMap` (https://doc.rust-lang.org/stable/std/collections/struct.HashMap.html). The FNV hasher (https://crates.io/crates/fnv) is frequently recommended for these cases, but be aware that it does not have the same collision-resistance properties as SipHash.

Why is there no integrated benchmarking infrastructure? (#why-is-there-no-integrated-benchmarking)

There is, but it's only available on the nightly release channel. We ultimately plan to build a pluggable system for integrated benchmarks, but in the meantime, the current system is considered unstable (https://github.com/rust-lang/rust/issues/29553).

Does Rust do tail-call optimization? (#does-rust-do-tail-call-optimization)

Not generally, no. Tail-call optimization may be done in limited circumstances (http://llvm.org/docs/CodeGenerator.html#sibling-call-optimization), but is not guaranteed (https://mail.mozilla.org/pipermail/rust-dev/2013-April/003557.html). As the feature has always been desired, Rust has a keyword (`become`) reserved, though it is not clear yet whether it is technically possible, nor whether it will be implemented. There was a proposed extension (https://github.com/rust-lang/rfcs/pull/81) that would allow tail-call elimination in certain contexts, but it is currently postponed.

Does Rust have a runtime? (#does-rust-have-a-runtime)

Not in the typical sense used by languages such as Java, but parts of the Rust standard library can be considered a "runtime", providing a heap, backtraces, unwinding, and stack guards. There is a small amount of initialization code (https://github.com/rust-lang/rust/blob/33916307780495fe311fe9c080b330d266f35bfb/src/libstd/rt.rs#L43) that runs before the user's `main` function. The Rust standard library additionally links to the C standard library, which does similar runtime initialization (http://www.embecosm.com/appnotes/ean9/html/ch05s02.html). Rust code can be compiled without the standard library, in which case the runtime is roughly equivalent to C's.

Syntax

Why curly brackets? Why can't Rust's syntax be like Haskell's or Python's? (#why-curly-brackets)

Use of curly brackets to denote blocks is a common design choice in a variety of programming languages, and Rust's consistency is useful for people already familiar with the style.

Curly brackets also allow for more flexible syntax for the programmer and a simpler parser in the compiler.

I can leave out parentheses on `if` conditions, so why do I have to put curly brackets around single line blocks? Why is the C style not allowed? (#why-curly-brackets-around-blocks)

Whereas C requires mandatory parentheses for `if` -statement conditionals but leaves curly brackets optional, Rust makes the opposite choice for its `if` -expressions. This keeps the conditional clearly separate from the body and avoids the hazard of optional curly brackets, which can lead to easy-to-miss errors during refactoring, like Apple's goto fail (https://gotofail.com/) bug.

Why is there no literal syntax for dictionaries? (#why-no-literal-syntax-for-dictionaries)

Rust's overall design preference is for limiting the size of the *language* while enabling powerful *libraries*. While Rust does provide initialization syntax for arrays and string literals, these are the only collection types built into the language. Other library-defined types, including the ubiquitous `Vec` (https://doc.rust-lang.org/stable/std/vec/struct.Vec.html) collection type, use macros for initialization like the `vec!` (https://doc.rust-lang.org/stable/std/macro.vec!.html) macro.

This design choice of using Rust's macro facilities to initialize collections will likely be extended generically to other collections in the future, enabling simple initialization of not only `HashMap` (https://doc.rust-lang.org/stable/std/collections/struct.HashMap.html) and `Vec` (https://doc.rust-lang.org/stable/std/vec/struct.Vec.html), but also other collection types such as `BTreeMap` (https://doc.rust-lang.org/stable/std/collections/struct.BTreeMap.html). In the meantime, if you want a more convenient syntax for initializing collections, you can create your own macro (https://stackoverflow.com/questions/27582739/how-do-i-create-a-hashmap-literal) to provide it.

When should I use an implicit return? (#when-should-i-use-an-implicit-return)

Rust is a very expression-oriented language, and "implicit returns" are part of that design. Constructs like `if`s, `match`es, and normal blocks are all expressions in Rust. For example, the following code checks if an `i64` (https://doc.rust-lang.org/stable/std/primitive.i64.html) is odd, returning the result by simply yielding it as a value:

```
fn is_odd(x: i64) -> bool {
    if x % 2 != 0 { true } else { false }
}
```

Although it can be simplified even further like so:

```
fn is_odd(x: i64) -> bool {
    x % 2 != 0
}
```

In each example, the last line of the function is the return value of that function. It is important to note that if a function ends in a semicolon, its return type will be `()`, indicating no returned value. Implicit returns must omit the semicolon to work.

Explicit returns are only used if an implicit return is impossible because you are returning before the end of the function's body. While each of the above functions could have been written with a `return` keyword and semicolon, doing so would be unnecessarily verbose, and inconsistent with the conventions of Rust code.

Why aren't function signatures inferred? (#why-arent-function-signatures-inferred)

In Rust, declarations tend to come with explicit types, while actual code has its types inferred. There are several reasons for this design:

- Mandatory declaration signatures help enforce interface stability at both the module and crate level.
- Signatures improve code comprehension for the programmer, eliminating the need for an IDE running an inference algorithm across an entire crate to be able to guess at a function's argument types; it's always explicit and nearby.
- Mechanically, it simplifies the inference algorithm, as inference only requires looking at one function at a time.

Why does `match` have to be exhaustive? (#why-does-match-have-to-be-exhaustive)

To aid in refactoring and clarity.

First, if every possibility is covered by the `match`, adding variants to the `enum` in the future will cause a compilation failure, rather than an error at runtime. This type of compiler assistance makes fearless refactoring possible in Rust.

Second, exhaustive checking makes the semantics of the default case explicit: in general, the only safe way to have a non-exhaustive `match` would be to panic the thread if nothing is matched. Early versions of Rust did not require `match` cases to be exhaustive and it was found to be a great source of bugs.

It is easy to ignore all unspecified cases by using the `_` wildcard:

```
match val.do_something() {
    Cat(a) => { /* ... */ }
    _      => { /* ... */ }
}
```

Numerics

Which of `f32` and `f64` should I prefer for floating-point math? (#which-type-of-float-should-i-use)

The choice of which to use is dependent on the purpose of the program.

If you are interested in the greatest degree of precision with your floating point numbers, then prefer `f64` (https://doc.rust-lang.org/stable/std/primitive.f64.html). If you are more interested in keeping the size of the value small or being maximally efficient, and are not concerned about the associated inaccuracy of having fewer bits per value, then `f32` (https://doc.rust-lang.org/stable/std/primitive.f32.html) is better. Operations on `f32` (https://doc.rust-lang.org/stable/std/primitive.f32.html) are usually faster, even on 64-bit hardware. As a common example, graphics programming typically uses `f32` (https://doc.rust-lang.org/stable/std/primitive.f32.html) because it requires high performance, and 32-bit floats are sufficient for representing pixels on the screen.

If in doubt, choose `f64` (https://doc.rust-lang.org/stable/std/primitive.f64.html) for the greater precision.

Why can't I compare floats or use them as `HashMap` or `BTreeMap` keys? (#why-cant-i-compare-floats)

Floats can be compared with the `==`, `!=`, `<`, `<=`, `>`, and `>=` operators, and with the `partial_cmp()` function. `==` and `!=` are part of the `PartialEq` (https://doc.rust-lang.org/stable/std/cmp/trait.PartialEq.html) trait, while `<`, `<=`, `>`, `>=`, and `partial_cmp()` are part of the `PartialOrd` (https://doc.rust-lang.org/stable/std/cmp/trait.PartialOrd.html) trait.

Floats cannot be compared with the `cmp()` function, which is part of the `Ord` (https://doc.rust-lang.org/stable/std/cmp/trait.Ord.html) trait, as there is no total ordering for floats. Furthermore, there is no total equality relation for floats, and so they also do not implement the `Eq` (https://doc.rust-lang.org/stable/std/cmp/trait.Eq.html) trait.

There is no total ordering or equality on floats because the floating-point value `NaN` (https://en.wikipedia.org/wiki/NaN) is not less than, greater than, or equal to any other floating-point value or itself.

Because floats do not implement `Eq` (https://doc.rust-lang.org/stable/std/cmp/trait.Eq.html) or `Ord` (https://doc.rust-lang.org/stable/std/cmp/trait.Ord.html), they may not be used in types whose trait bounds require those traits, such as `BTreeMap` (https://doc.rust-lang.org/stable/std/collections/struct.BTreeMap.html) or `HashMap` (https://doc.rust-lang.org/stable/std/collections/struct.HashMap.html). This is important because these types *assume* their keys provide a total ordering or total equality relation, and will malfunction otherwise.

There is a crate (https://crates.io/crates/ordered-float) that wraps `f32` (https://doc.rust-lang.org/stable/std/primitive.f32.html) and `f64` (https://doc.rust-lang.org/stable/std/primitive.f64.html) to provide `Ord` (https://doc.rust-lang.org/stable/std/cmp/trait.Ord.html) and `Eq` (https://doc.rust-lang.org/stable/std/cmp/trait.Eq.html) implementations, which may be useful in certain cases.

How can I convert between numeric types? (#how-can-i-convert-between-numeric-types)

There are two ways: the `as` keyword, which does simple casting for primitive types, and the `Into` (https://doc.rust-lang.org/stable/std/convert/trait.Into.html) and `From` (https://doc.rust-lang.org/stable/std/convert/trait.From.html) traits, which are implemented for a number of type conversions (and which you can implement for your own types). The `Into` (https://doc.rust-lang.org/stable/std/convert/trait.Into.html) and `From` (https://doc.rust-lang.org/stable/std/convert/trait.From.html) traits are only implemented in cases where conversions are lossless, so for example, `f64::from(0f32)` will compile while `f32::from(0f64)` will not. On the other hand, `as` will convert between any two primitive types, truncating values as necessary.

Why doesn't Rust have increment and decrement operators? (#why-doesnt-rust-have-increment-and-decrement-operators)

Preincrement and postincrement (and the decrement equivalents), while convenient, are also fairly complex. They require knowledge of evaluation order, and often lead to subtle bugs and undefined behavior in C and C++. `x = x + 1` or `x += 1` is only slightly longer, but unambiguous.

Strings

How can I convert a `String` or `Vec<T>` to a slice (`&str` and `&[T]`)? (#how-to-convert-string-or-vec-to-slice)

Usually, you can pass a reference to a `String` or `Vec<T>` wherever a slice is expected. Using Deref coercions (https://doc.rust-lang.org/stable/book/second-edition/ch15-02-deref.html), `String`s (https://doc.rust-lang.org/stable/std/string/struct.String.html) and `Vec`s (https://doc.rust-lang.org/stable/std/vec/struct.Vec.html) will automatically coerce to their respective slices when passed by reference with `&` or `& mut`.

Methods implemented on `&str` and `&[T]` can be accessed directly on `String` and `Vec<T>`. For example, `some_string.trim()` will work even though `trim` is a method on `&str` and `some_string` is a `String`.

In some cases, such as generic code, it’s necessary to convert manually. Manual conversions can be achieved using the slicing operator, like so: `&my_vec[..]`.

How can I convert from `&str` to `String` or the other way around? (#how-to-convert-between-str-and-string)

The `to_string()` (https://doc.rust-lang.org/stable/std/string/trait.ToString.html#tymethod.to_string) method converts from a `&str` (https://doc.rust-lang.org/stable/std/primitive.str.html) into a `String` (https://doc.rust-lang.org/stable/std/string/struct.String.html), and `String`s (https://doc.rust-lang.org/stable/std/string/struct.String.html) are automatically converted into `&str` (https://doc.rust-lang.org/stable/std/primitive.str.html) when you borrow a reference to them. Both are demonstrated in the following example:

```
fn main() {
    let s = "Jane Doe".to_string();
    say_hello(&s);
}

fn say_hello(name: &str) {
    println!("Hello {}!", name);
}
```

What are the differences between the two different string types? (#what-are-the-differences-between-str-and-string)

`String` (https://doc.rust-lang.org/stable/std/string/struct.String.html) is an owned buffer of UTF-8 bytes allocated on the heap. Mutable `String`s (https://doc.rust-lang.org/stable/std/string/struct.String.html) can be modified, growing their capacity as needed. `&str` (https://doc.rust-lang.org/stable/std/primitive.str.html) is a fixed-capacity “view” into a `String` (https://doc.rust-lang.org/stable/std/string/struct.String.html) allocated elsewhere, commonly on the heap, in the case of slices dereferenced from `String`s (https://doc.rust-lang.org/stable/std/string/struct.String.html), or in static memory, in the case of string literals.

`&str` (https://doc.rust-lang.org/stable/std/primitive.str.html) is a primitive type implemented by the Rust language, while `String` (https://doc.rust-lang.org/stable/std/string/struct.String.html) is implemented in the standard library.

How do I do O(1) character access in a `String`? (#how-do-i-do-o1-character-access-in-a-string)

You cannot. At least not without a firm understanding of what you mean by “character”, and preprocessing the string to find the index of the desired character.

Rust strings are UTF-8 encoded. A single visual character in UTF-8 is not necessarily a single byte as it would be in an ASCII-encoded string. Each byte is called a “code unit” (in UTF-16, code units are 2 bytes; in UTF-32 they are 4 bytes). “Code points” are composed of one or more code units, and combine in “grapheme clusters” which most closely approximate characters.

Thus, even though you may index on bytes in a UTF-8 string, you can’t access the `i`th code point or grapheme cluster in constant time. However, if you know at which byte that desired code point or grapheme cluster begins, then you *can* access it in constant time. Functions including `str::find()` (https://doc.rust-lang.org/stable/std/primitive.str.html#method.find) and regex matches return byte indices, facilitating this sort of access.

Why are strings UTF-8 by default? (#why-are-strings-utf-8)

The `str` (https://doc.rust-lang.org/stable/std/primitive.str.html) type is UTF-8 because we observe more text in the wild in this encoding – particularly in network transmissions, which are endian-agnostic – and we think it’s best that the default treatment of I/O not involve having to recode codepoints in each direction.

This does mean that locating a particular Unicode codepoint inside a string is an O(n) operation, although if the starting byte index is already known then they can be accessed in O(1) as expected. On the one hand, this is clearly undesirable; on the other hand, this problem is full of trade-offs and we’d like to point out a few important qualifications:

Scanning a `str` (https://doc.rust-lang.org/stable/std/primitive.str.html) for ASCII-range codepoints can still be done safely byte-at-a-time. If you use `.as_bytes()` (https://doc.rust-lang.org/stable/std/primitive.str.html#method.as_bytes), pulling out a `u8` (https://doc.rust-lang.org/stable/std/primitive.u8.html) costs only `O(1)` and produces a value that can be cast and compared to an ASCII-range `char` (https://doc.rust-lang.org/stable/std/primitive.char.html). So if you’re (say) line-breaking on `'\n'`, byte-based treatment still works. UTF-8 was well-designed this way.

Most “character oriented” operations on text only work under very restricted language assumptions such as “ASCII-range codepoints only”. Outside ASCII-range, you tend to have to use a complex (non-constant-time) algorithm for determining linguistic-unit (glyph, word, paragraph) boundaries anyway. We recommend using an “honest” linguistically-aware, Unicode-approved algorithm.

The `char` (https://doc.rust-lang.org/stable/std/primitive.char.html) type is UTF-32. If you are sure you need to do a codepoint-at-a-time algorithm, it’s trivial to write a `type wstr = [char]`, and unpack a `str` (https://doc.rust-lang.org/stable/std/primitive.str.html) into it in a single pass, then work with the `wstr`. In other words: the fact that the language is not “decoding to UTF32 by default” shouldn’t stop you from decoding (or re-encoding any other way) if you need to work with that encoding.

For a more in-depth explanation of why UTF-8 is usually preferable over UTF-16 or UTF-32, read the UTF-8 Everywhere manifesto (http://utf8everywhere.org/).

What string type should I use? (#what-string-type-should-i-use)

Rust has four pairs of string types, each serving a distinct purpose (http://www.suspectsemantics.com/blog/2016/03/27/string-types-in-rust/). In each pair, there is an “owned” string type, and a “slice” string type. The organization looks like this:

	“Slice” type	**“Owned” type**
UTF-8	`str`	`String`
OS-compatible	`OsStr`	`OsString`
C-compatible	`CStr`	`CString`
System path	`Path`	`PathBuf`

Rust’s different string types serve different purposes. `String` and `str` are UTF-8 encoded general-purpose strings. `OsString` and `OsStr` are encoded according to the current platform, and are used when interacting with the operating system. `CString` and `CStr` are the Rust equivalent of strings in C, and are used in FFI code, and `PathBuf` and `Path` are convenience wrappers around `OsString` and `OsStr`, providing methods specific to path manipulation.

How can I write a function that accepts both `&str` and `String`? (#why-are-there-multiple-types-of-strings)

There are several options, depending on the needs of the function:

- If the function needs an owned string, but wants to accept any type of string, use an `Into<String>` bound.
- If the function needs a string slice, but wants to accept any type of string, use an `AsRef<str>` bound.
- If the function does not care about the string type, and wants to handle the two possibilities uniformly, use `Cow<str>` as the input type.

Using `Into<String>`

In this example, the function will accept both owned strings and string slices, either doing nothing or converting the input into an owned string within the function body. Note that the conversion needs to be done explicitly, and will not happen otherwise.

```
fn accepts_both<S: Into<String>>(s: S) {
    let s = s.into();   // This will convert s into a `String`.
    // ... the rest of the function
}
```

Using `AsRef<str>`

In this example, the function will accept both owned strings and string slices, either doing nothing or converting the input into a string slice. This can be done automatically by taking the input by reference, like so:

```
fn accepts_both<S: AsRef<str>>(s: &S) {
    // ... the body of the function
}
```

Using `Cow<str>`

In this example, the function takes in a `Cow<str>`, which is not a generic type but a container, containing either an owned string or string slice as needed.

```
fn accepts_cow(s: Cow<str>) {
    // ... the body of the function
}
```

Collections

Can I implement data structures like vectors and linked lists efficiently in Rust? (#can-i-implement-linked-lists-in-rust)

If your reason for implementing these data structures is to use them for other programs, there's no need, as efficient implementations of these data structures are provided by the standard library.

If, however, your reason is simply to learn (http://cglab.ca/~abeinges/blah/too-many-lists/book/), then you will likely need to dip into unsafe code. While these data structures *can* be implemented entirely in safe Rust, the performance is likely to be worse than it would be with the use of unsafe code. The simple reason for this is that data structures like vectors and linked lists rely on pointer and memory operations that are disallowed in safe Rust.

For example, a doubly-linked list requires that there be two mutable references to each node, but this violates Rust's mutable reference aliasing rules. You can solve this using `Weak<T>` (https://doc.rust-lang.org/stable/std/rc/struct.Weak.html), but the performance will be poorer than you likely want. With unsafe code you can bypass the mutable reference aliasing rule restriction, but must manually verify that your code introduces no memory safety violations.

How can I iterate over a collection without moving/consuming it? (#how-can-i-iterate-over-a-collection-without-consuming-it)

The easiest way is by using the collection's `IntoIterator` (https://doc.rust-lang.org/stable/std/iter/trait.IntoIterator.html) implementation. Here is an example for `&Vec` (https://doc.rust-lang.org/stable/std/vec/struct.Vec.html):

```
let v = vec![1,2,3,4,5];
for item in &v {
    print!("{} ", item);
}
println!("\nLength: {}", v.len());
```

Rust `for` loops call `into_iter()` (defined on the `IntoIterator` (https://doc.rust-lang.org/stable/std/iter/trait.IntoIterator.html) trait) for whatever they're iterating over. Anything implementing the `IntoIterator` (https://doc.rust-lang.org/stable/std/iter/trait.IntoIterator.html) trait may be looped over with a `for` loop. `IntoIterator` (https://doc.rust-lang.org/stable/std/iter/trait.IntoIterator.html) is implemented for `&Vec` (https://doc.rust-lang.org/stable/std/vec/struct.Vec.html) and `&mut Vec` (https://doc.rust-lang.org/stable/std/vec/struct.Vec.html), causing the iterator from `into_iter()` to borrow the contents of the collection, rather than moving/consuming them. The same is true for other standard collections as well.

If a moving/consuming iterator is desired, write the `for` loop without `&` or `&mut` in the iteration.

If you need direct access to a borrowing iterator, you can usually get it by calling the `iter()` method.

Why do I need to type the array size in the array declaration? (#why-do-i-need-to-type-the-array-size-in-the-array-declaration)

You don't necessarily have to. If you're declaring an array directly, the size is inferred based on the number of elements. But if you're declaring a function that takes a fixed-size array, the compiler has to know how big that array will be.

One thing to note is that currently Rust doesn't offer generics over arrays of different size. If you'd like to accept a contiguous container of a variable number of values, use a `Vec` (https://doc.rust-lang.org/stable/std/vec

/struct.Vec.html) or slice (depending on whether you need ownership).

Ownership

How can I implement a graph or other data structure that contains cycles? (#how-can-i-implement-a-data-structure-that-contains-cycles)

There are at least four options (discussed at length in Too Many Linked Lists (http://cglab.ca/~abeinges/blah/too-many-lists/book/)):

- You can implement it using `Rc` (https://doc.rust-lang.org/stable/std/rc/struct.Rc.html) and `Weak` (https://doc.rust-lang.org/stable/std/rc/struct.Weak.html) to allow shared ownership of nodes, although this approach pays the cost of memory management.
- You can implement it using `unsafe` code using raw pointers. This will be more efficient, but bypasses Rust's safety guarantees.
- Using vectors and indices into those vectors. There are several (http://smallcultfollowing.com/babysteps/blog/2015/04/06/modeling-graphs-in-rust-using-vector-indices/) available (https://featherweightmusings.blogspot.com/2015/04/graphs-in-rust.html) examples and explanations of this approach.
- Using borrowed references with `UnsafeCell` (https://doc.rust-lang.org/stable/std/cell/struct.UnsafeCell.html). There are explanations and code (https://github.com/nrc/r4cppp/blob/master/graphs/README.md#node-and-unsafecell) available for this approach.

How can I define a struct that contains a reference to one of its own fields? (#how-can-i-define-a-struct-that-contains-a-reference-to-one-of-its-own-fields)

It's possible, but useless to do so. The struct becomes permanently borrowed by itself and therefore can't be moved. Here is some code illustrating this:

```
use std::cell::Cell;

#[derive(Debug)]
struct Unmovable<'a> {
    x: u32,
    y: Cell<Option<&'a u32>>,
}


fn main() {
    let test = Unmovable { x: 42, y: Cell::new(None) };
    test.y.set(Some(&test.x));

    println!("{:?}", test);
}
```

What is the difference between passing by value, consuming, moving, and transferring ownership? (#what-is-the-difference-between-consuming-and-moving)

These are different terms for the same thing. In all cases, it means the value has been moved to another owner, and moved out of the possession of the original owner, who can no longer use it. If a type implements the `Copy` trait, the original owner's value won't be invalidated, and can still be used.

Why can values of some types be used after passing them to a function, while reuse of values of other types results in an error? (#why-can-values-of-some-types-by-reused-while-others-are-consumed)

If a type implements the `Copy` (https://doc.rust-lang.org/stable/std/ptr/fn.copy.html) trait, then it will be copied when passed to a function. All numeric types in Rust implement `Copy` (https://doc.rust-lang.org/stable/std/ptr/fn.copy.html), but struct types do not implement `Copy` (https://doc.rust-lang.org/stable/std/ptr/fn.copy.html) by default, so they are moved instead. This means that the struct can no longer be used elsewhere, unless it is moved back out of the function via the return.

How do you deal with a "use of moved value" error? (#how-do-you-deal-with-a-use-of-moved-value-error)

This error means that the value you're trying to use has been moved to a new owner. The first thing to check is whether the move in question was necessary: if it moved into a function, it may be possible to rewrite the function to use a reference, rather than moving. Otherwise if the type being moved implements `Clone` (https://doc.rust-lang.org/stable/std/clone/trait.Clone.html#tymethod.clone), then calling `clone()` on it before moving will move a copy of it, leaving the original still available for further use. Note though that cloning a value should typically be the last resort since cloning can be expensive, causing further allocations.

If the moved value is of your own custom type, consider implementing `Copy` (https://doc.rust-lang.org/stable/std/ptr/fn.copy.html) (for implicit copying, rather than moving) or `Clone` (https://doc.rust-lang.org/stable/std/clone/trait.Clone.html#tymethod.clone) (explicit copying). `Copy` (https://doc.rust-lang.org/stable/std/ptr/fn.copy.html) is most commonly implemented with `#[derive(Copy, Clone)]` (`Copy` (https://doc.rust-lang.org/stable/std/ptr/fn.copy.html) requires `Clone` (https://doc.rust-lang.org/stable/std/clone/trait.Clone.html#tymethod.clone)), and `Clone` (https://doc.rust-lang.org/stable/std/clone/trait.Clone.html#tymethod.clone) with `#[derive(Clone)]`.

If none of these are possible, you may want to modify the function that acquired ownership to return ownership of the value when the function exits.

What are the rules for using `self`, `&self`, or `&mut self` in a method declaration? (#what-are-the-rules-for-different-self-types-in-methods)

- Use `self` when a function needs to consume the value
- Use `&self` when a function only needs a read-only reference to the value
- Use `&mut self` when a function needs to mutate the value without consuming it

How can I understand the borrow checker? (#how-can-i-understand-the-borrow-checker)

The borrow checker applies only a few rules, which can be found in the Rust book's section on borrowing (https://doc.rust-lang.org/stable/book/second-edition/ch15-02-deref.html), when evaluating Rust code. These rules are:

> First, any borrow must last for a scope no greater than that of the owner. Second, you may have one or the other of these two kinds of borrows, but not both at the same time:
> - one or more references (&T) to a resource.
> - exactly one mutable reference (&mut T)

While the rules themselves are simple, following them consistently is not, particularly for those unaccustomed to reasoning about lifetimes and ownership.

The first step in understanding the borrow checker is reading the errors it produces. A lot of work has been put into making sure the borrow checker provides quality assistance in resolving the issues it identifies. When you encounter a borrow checker problem, the first step is to slowly and carefully read the error reported, and to only approach the code after you understand the error being described.

The second step is to become familiar with the ownership and mutability-related container types provided by the Rust standard library, including `Cell` (https://doc.rust-lang.org/stable/std/cell/struct.Cell.html), `RefCell` (https://doc.rust-lang.org/stable/std/cell/struct.RefCell.html), and `Cow` (https://doc.rust-lang.org/stable/std/borrow/enum.Cow.html). These are useful and necessary tools for expressing certain ownership and mutability situations, and have been written to be of minimal performance cost.

The single most important part of understanding the borrow checker is practice. Rust's strong static analyses guarantees are strict and quite different from what many programmers have worked with before. It will take some time to become completely comfortable with everything.

If you find yourself struggling with the borrow checker, or running out of patience, always feel free to reach out to the Rust community (community.html) for help.

When is `Rc` useful? (#when-is-rc-useful)

This is covered in the official documentation for `Rc` (https://doc.rust-lang.org/stable/std/rc/struct.Rc.html), Rust's non-atomically reference-counted pointer type. In short, `Rc` (https://doc.rust-lang.org/stable/std/rc/struct.Rc.html) and its thread-safe cousin `Arc` (https://doc.rust-lang.org/stable/std/sync/struct.Arc.html) are useful to express shared ownership, and have the system automatically deallocate the associated memory when no one has access to it.

How do I return a closure from a function? (#how-do-i-return-a-closure-from-a-function)

To return a closure from a function, it must be a "move closure", meaning that the closure is declared with the `move` keyword. As explained in the Rust book (https://doc.rust-lang.org/book/closures.html#move-closures), this gives the closure its own copy of the captured variables, independent of its parent stack frame. Otherwise, returning a closure would be unsafe, as it would allow access to variables that are no longer valid; put another way: it would allow reading potentially invalid memory. The closure must also be wrapped in a `Box` (https://doc.rust-lang.org/stable/std/boxed/struct.Box.html), so that it is allocated on the heap. Read more about this in the book (https://doc.rust-lang.org/book/closures.html#returning-closures).

What is a deref coercion and how does it work? (#what-are-deref-coercions)

A deref coercion (https://doc.rust-lang.org/book/deref-coercions.html) is a handy coercion that automatically converts references to pointers (e.g., `&Rc<T>` or `&Box<T>`) into references to their contents (e.g., `&T`). Deref coercions exist to make using Rust more ergonomic, and are implemented via the `Deref` (https://doc.rust-lang.org/stable/std/ops/trait.Deref.html) trait.

A Deref implementation indicates that the implementing type may be converted into a target by a call to the `deref` method, which takes an immutable reference to the calling type and returns a reference (of the same lifetime) to the target type. The `*` prefix operator is shorthand for the `deref` method.

They're called "coercions" because of the following rule, quoted here from the Rust book (https://doc.rust-lang.org/stable/book/second-edition/ch15-02-deref.html):

> If you have a type `U`, and it implements `Deref<Target=T>`, values of `&U` will automatically coerce to a `&T`.

For example, if you have a `&Rc<String>`, it will coerce via this rule into a `&String`, which then coerces to a `&str` in the same way. So if a function takes a `&str` parameter, you can pass in a `&Rc<String>` directly, with all coercions handled automatically via the `Deref` trait.

The most common sorts of deref coercions are:

- `&Rc<T>` to `&T`
- `&Box<T>` to `&T`
- `&Arc<T>` to `&T`
- `&Vec<T>` to `&[T]`
- `&String` to `&str`

Lifetimes

Why lifetimes? (#why-lifetimes)

Lifetimes are Rust's answer to the question of memory safety. They allow Rust to ensure memory safety without the performance costs of garbage collection. They are based on a variety of academic work, which can be found in the Rust book (https://doc.rust-lang.org/stable/book/first-edition/bibliography.html#type-system).

Why is the lifetime syntax the way it is? (#why-is-the-lifetime-syntax-the-way-it-is)

The `'a` syntax comes from the ML family of programming languages, where `'a` is used to indicate a generic type parameter. For Rust, the syntax had to be something that was unambiguous, noticeable, and fit nicely in a type declaration right alongside traits and references. Alternative syntaxes have been discussed, but no alternative syntax has been demonstrated to be clearly better.

How do I return a borrow to something I created from a function? (#how-do-i-return-a-borrow-to-something-i-created-from-a-function)

You need to ensure that the borrowed item will outlive the function. This can be done by binding the output lifetime to some input lifetime like so:

```
type Pool = TypedArena<Thing>;

// (the lifetime below is only written explicitly for
// expository purposes; it can be omitted via the
// elision rules described in a later FAQ entry)
fn create_borrowed<'a>(pool: &'a Pool,
                       x: i32,
                       y: i32) -> &'a Thing {
    pool.alloc(Thing { x: x, y: y })
}
```

An alternative is to eliminate the references entirely by returning an owning type like `String` (https://doc.rust-lang.org/stable/std/string/struct.String.html):

```
fn happy_birthday(name: &str, age: i64) -> String {
    format!("Hello {}! You're {} years old!", name, age)
}
```

This approach is simpler, but often results in unnecessary allocations.

Why do some references have lifetimes, like `&'a T`, and some do not, like `&T`? (#when-are-lifetimes-required-to-be-explicit)

In fact, *all* reference types have a lifetime, but most of the time you do not have to write it explicitly. The rules are as follows:

1. Within a function body, you never have to write a lifetime explicitly; the correct value should always be inferred.
2. Within a function *signature* (for example, in the types of its arguments, or its return type), you *may* have to write a lifetime explicitly. Lifetimes there use a simple defaulting scheme called "lifetime elision"

(https://doc.rust-lang.org/book/lifetimes.html#lifetime-elision), which consists of the following three rules:

- Each elided lifetime in a function's arguments becomes a distinct lifetime parameter.
- If there is exactly one input lifetime, elided or not, that lifetime is assigned to all elided lifetimes in the return values of that function.
- If there are multiple input lifetimes, but one of them is &self or &mut self, the lifetime of self is assigned to all elided output lifetimes.

3. Finally, in a `struct` or `enum` definition, all lifetimes must be explicitly declared.

If these rules result in compilation errors, the Rust compiler will provide an error message indicating the error caused, and suggesting a potential solution based on which step of the inference process caused the error.

How can Rust guarantee "no null pointers" and "no dangling pointers"? (#how-can-rust-guarantee-no-null-pointers)

The only way to construct a value of type `&Foo` or `&mut Foo` is to specify an existing value of type `Foo` that the reference points to. The reference "borrows" the original value for a given region of code (the lifetime of the reference), and the value being borrowed from cannot be moved or destroyed for the duration of the borrow.

How do I express the absence of a value without `null`? (#how-do-i-express-the-absense-of-a-value-without-null)

You can do that with the `Option` (https://doc.rust-lang.org/stable/std/option/enum.Option.html) type, which can either be `Some(T)` or `None`. `Some(T)` indicates that a value of type `T` is contained within, while `None` indicates the absence of a value.

Generics

What is "monomorphisation"? (#what-is-monomorphisation)

Monomorphisation specializes each use of a generic function (or structure) with specific instance, based on the parameter types of calls to that function (or uses of the structure).

During monomorphisation a new copy of the generic function is translated for each unique set of types the function is instantiated with. This is the same strategy used by C++. It results in fast code that is specialized for every call-site and statically dispatched, with the tradeoff that functions instantiated with many different types can cause "code bloat", where multiple function instances result in larger binaries than would be created with other translation strategies.

Functions that accept trait objects (https://doc.rust-lang.org/book/trait-objects.html) instead of type parameters do not undergo monomorphisation. Instead, methods on the trait objects are dispatched dynamically at runtime.

What's the difference between a function and a closure that doesn't capture any variables? (#whats-the-difference-between-a-function-and-a-closure-that-doesnt-capture)

Functions and closures are operationally equivalent, but have different runtime representations due to their differing implementations.

Functions are a built-in primitive of the language, while closures are essentially syntactic sugar for one of three traits: `Fn` (https://doc.rust-lang.org/stable/std/ops/trait.Fn.html), `FnMut` (https://doc.rust-lang.org/stable/std/ops/trait.FnMut.html), and `FnOnce` (https://doc.rust-lang.org/stable/std/ops/trait.FnOnce.html). When you make a closure, the Rust compiler automatically creates a struct implementing the appropriate trait of those three and containing the captured environment variables as members, and makes it so the struct can be called as a function. Bare functions can not capture an environment.

The big difference between these traits is how they take the `self` parameter. `Fn` (https://doc.rust-lang.org/stable/std/ops/trait.Fn.html) takes `&self`, `FnMut` (https://doc.rust-lang.org/stable/std/ops/trait.FnMut.html) takes `&mut self`, and `FnOnce` (https://doc.rust-lang.org/stable/std/ops/trait.FnOnce.html) takes `self`.

Even if a closure does not capture any environment variables, it is represented at runtime as two pointers, the same as any other closure.

What are higher-kinded types, why would I want them, and why doesn't Rust have them? (#what-are-higher-kinded-types)

Higher-kinded types are types with unfilled parameters. Type constructors, like `Vec` (https://doc.rust-lang.org/stable/std/vec/struct.Vec.html), `Result` (https://doc.rust-lang.org/stable/std/result/enum.Result.html), and `HashMap` (https://doc.rust-lang.org/stable/std/collections/struct.HashMap.html) are all examples of higher-kinded types: each requires some additional type parameters in order to actually denote a specific type, like `Vec<u32>`. Support for higher-kinded types means these "incomplete" types may be used anywhere "complete" types can be used, including as generics for functions.

Any complete type, like `i32` (https://doc.rust-lang.org/stable/std/primitive.i32.html), `bool` (https://doc.rust-lang.org/stable/std/primitive.bool.html), or `char` (https://doc.rust-lang.org/stable/std/primitive.char.html) is of kind `*` (this notation comes from the field of type theory). A type with one parameter, like `Vec<T>` (https://doc.rust-lang.org/stable/std/vec/struct.Vec.html) is of kind `* -> *`, meaning that `Vec<T>` (https://doc.rust-lang.org/stable/std/vec/struct.Vec.html) takes in a complete type like `i32` (https://doc.rust-lang.org/stable/std/primitive.i32.html) and returns a complete type `Vec<i32>`. A type with three parameters, like `HashMap<K, V, S>` (https://doc.rust-lang.org/stable/std/collections/struct.HashMap.html) is of kind `* -> * -> * -> *`, and takes in three complete types (like `i32` (https://doc.rust-lang.org/stable/std/primitive.i32.html), `String` (https://doc.rust-lang.org/stable/std/string/struct.String.html), and `RandomState` (https://doc.rust-lang.org/stable/std/collections/hash_map/struct.RandomState.html)) to produce a new complete type `HashMap<i32, String, RandomState>`.

In addition to these examples, type constructors can take *lifetime* arguments, which we'll denote as `Lt`. For example, `slice::Iter` has kind `Lt -> * -> *`, because it must be instantiated like `Iter<'a, u32>`.

The lack of support for higher-kinded types makes it difficult to write certain kinds of generic code. It's particularly problematic for abstracting over concepts like iterators, since iterators are often parameterized over a lifetime at least. That in turn has prevented the creation of traits abstracting over Rust's collections.

Another common example is concepts like functors or monads, both of which are type constructors, rather than single types.

Rust doesn't currently have support for higher-kinded types because it hasn't been a priority compared to other improvements we want to make. Since the design is a major, cross-cutting change, we also want to approach it carefully. But there's no inherent reason for the current lack of support.

What do named type parameters like `<T=Foo>` in generic types mean? (#what-do-named-type-parameters-in-generic-types-mean)

These are called associated types (https://doc.rust-lang.org/stable/book/second-edition/ch19-03-advanced-traits.html), and they allow for the expression of trait bounds that can't be expressed with a `where` clause. For example, a generic bound `X: Bar<T=Foo>` means "`X` must implement the trait `Bar`, and in that implementation of `Bar`, `X` must choose `Foo` for `Bar`'s associated type, `T`." Examples of where such a constraint cannot be expressed via a `where` clause include trait objects like `Box<Bar<T=Foo>>`.

Associated types exist because generics often involve families of types, where one type determines all of the others in a family. For example, a trait for graphs might have as its `Self` type the graph itself, and have associated types for nodes and for edges. Each graph type uniquely determines the associated types. Using associated types makes it much more concise to work with these families of types, and also provides better type inference in many cases.

Can I overload operators? Which ones and how? (#how-do-i-overload-operators)

You can provide custom implementations for a variety of operators using their associated traits: `Add` (https://doc.rust-lang.org/stable/std/ops/trait.Add.html) for `+`, `Mul` (https://doc.rust-lang.org/stable/std/ops/trait.Mul.html) for `*`, and so on. It looks like this:

```
use std::ops::Add;

struct Foo;

impl Add for Foo {
    type Output = Foo;
    fn add(self, rhs: Foo) -> Self::Output {
        println!("Adding!");
        self
    }
}
```

The following operators can be overloaded:

Operation	Trait
`+`	`Add` (https://doc.rust-lang.org/stable/std/ops/trait.Add.html)
`+=`	`AddAssign` (https://doc.rust-lang.org/stable/std/ops/trait.AddAssign.html)
`binary -`	`Sub` (https://doc.rust-lang.org/stable/std/ops/trait.Sub.html)
`-=`	`SubAssign` (https://doc.rust-lang.org/stable/std/ops/trait.SubAssign.html)
`*`	`Mul` (https://doc.rust-lang.org/stable/std/ops/trait.Mul.html)
`*=`	`MulAssign` (https://doc.rust-lang.org/stable/std/ops/trait.MulAssign.html)
`/`	`Div` (https://doc.rust-lang.org/stable/std/ops/trait.Div.html)
`/=`	`DivAssign` (https://doc.rust-lang.org/stable/std/ops/trait.DivAssign.html)
`unary -`	`Neg` (https://doc.rust-lang.org/stable/std/ops/trait.Neg.html)
`%`	`Rem` (https://doc.rust-lang.org/stable/std/ops/trait.Rem.html)
`%=`	`RemAssign` (https://doc.rust-lang.org/stable/std/ops/trait.RemAssign.html)
`&`	`BitAnd` (https://doc.rust-lang.org/stable/std/ops/trait.BitAnd.html)
`|`	`BitOr` (https://doc.rust-lang.org/stable/std/ops/trait.BitOr.html)
`|=`	`BitOrAssign` (https://doc.rust-lang.org/stable/std/ops/trait.BitOrAssign.html)
`^`	`BitXor` (https://doc.rust-lang.org/stable/std/ops/trait.BitXor.html)
`^=`	`BitXorAssign` (https://doc.rust-lang.org/stable/std/ops/trait.BitXorAssign.html)
`!`	`Not` (https://doc.rust-lang.org/stable/std/ops/trait.Not.html)
`<<`	`Shl` (https://doc.rust-lang.org/stable/std/ops/trait.Shl.html)
`<<=`	`ShlAssign` (https://doc.rust-lang.org/stable/std/ops/trait.ShlAssign.html)
`>>`	`Shr` (https://doc.rust-lang.org/stable/std/ops/trait.Shr.html)
`>>=`	`ShrAssign` (https://doc.rust-lang.org/stable/std/ops/trait.ShrAssign.html)
`*`	`Deref` (https://doc.rust-lang.org/stable/std/ops/trait.Deref.html)
`mut *`	`DerefMut` (https://doc.rust-lang.org/stable/std/ops/trait.DerefMut.html)
`[]`	`Index` (https://doc.rust-lang.org/stable/std/ops/trait.Index.html)

Operation	Trait
`mut []`	`IndexMut` (https://doc.rust-lang.org/stable/std/ops/trait.IndexMut.html)

Why the split between `Eq` / `PartialEq` and `Ord` / `PartialOrd` ? (#why-the-split-between-eq-partialeq-and-ord-partialord)

There are some types in Rust whose values are only partially ordered, or have only partial equality. Partial ordering means that there may be values of the given type that are neither less than nor greater than each other. Partial equality means that there may be values of the given type that are not equal to themselves.

Floating point types (`f32` (https://doc.rust-lang.org/stable/std/primitive.f32.html) and `f64` (https://doc.rust-lang.org/stable/std/primitive.f64.html)) are good examples of each. Any floating point type may have the value `NaN` (meaning "not a number"). `NaN` is not equal to itself (`NaN == NaN` is false), and not less than or greater than any other floating point value. As such, both `f32` (https://doc.rust-lang.org/stable/std/primitive.f32.html) and `f64` (https://doc.rust-lang.org/stable/std/primitive.f64.html) implement `PartialOrd` (https://doc.rust-lang.org/stable/std/cmp/trait.PartialOrd.html) and `PartialEq` (https://doc.rust-lang.org/stable/std/cmp/trait.PartialEq.html) but not `Ord` (https://doc.rust-lang.org/stable/std/cmp/trait.Ord.html) and not `Eq` (https://doc.rust-lang.org/stable/std/cmp/trait.Eq.html).

As explained in the earlier question on floats (#why-cant-i-compare-floats), these distinctions are important because some collections rely on total orderings/equality in order to give correct results.

Input / Output

How do I read a file into a `String` ? (#how-do-i-read-a-file-into-a-string)

Using the `read_to_string()` (https://doc.rust-lang.org/stable/std/io/trait.Read.html#method.read_to_string) method, which is defined on the `Read` (https://doc.rust-lang.org/stable/std/io/trait.Read.html) trait in `std::io` (https://doc.rust-lang.org/stable/std/io/index.html).

```
use std::io::Read;
use std::fs::File;

fn read_file(path: &str) -> Result<String, std::io::Error> {
    let mut s = String::new();
    let _ = File::open(path)?.read_to_string(&mut s);  // `s` contains the contents of "
    Ok(s)
}

fn main() {
    match read_file("foo.txt") {
        Ok(_) => println!("Got file contents!"),
        Err(err) => println!("Getting file contents failed with error: {}", err)
    };
}
```

How do I read file input efficiently? (#how-do-i-read-file-input-efficiently)

The `File` (https://doc.rust-lang.org/stable/std/fs/struct.File.html) type implements the `Read` (https://doc.rust-lang.org/stable/std/io/trait.Read.html) trait, which has a variety of functions for reading and writing data, including `read()` (https://doc.rust-lang.org/stable/std/io/trait.Read.html#tymethod.read), `read_to_end()` (https://doc.rust-lang.org/stable/std/io/trait.Read.html#method.read_to_end), `bytes()`

(https://doc.rust-lang.org/stable/std/io/trait.Read.html#method.bytes), `chars()` (https://doc.rust-lang.org/stable/std/io/trait.Read.html#method.chars), and `take()` (https://doc.rust-lang.org/stable/std/io/trait.Read.html#method.take). Each of these functions reads a certain amount of input from a given file. `read()` (https://doc.rust-lang.org/stable/std/io/trait.Read.html#tymethod.read) reads as much input as the underlying system will provide in a single call. `read_to_end()` (https://doc.rust-lang.org/stable/std/io/trait.Read.html#method.read_to_end) reads the entire buffer into a vector, allocating as much space as is needed. `bytes()` (https://doc.rust-lang.org/stable/std/io/trait.Read.html#method.bytes) and `chars()` (https://doc.rust-lang.org/stable/std/io/trait.Read.html#method.chars) allow you to iterate over the bytes and characters of the file, respectively. Finally, `take()` (https://doc.rust-lang.org/stable/std/io/trait.Read.html#method.take) allows you to read up to an arbitrary number of bytes from the file. Collectively, these should allow you to efficiently read in any data you need.

For buffered reads, use the `BufReader` (https://doc.rust-lang.org/stable/std/io/struct.BufReader.html) struct, which helps to reduce the number of system calls when reading.

How do I do asynchronous input / output in Rust? (#how-do-i-do-asynchronous-input-output-in-rust)

There are several libraries providing asynchronous input / output in Rust, including mio (https://github.com/carllerche/mio), tokio (https://github.com/tokio-rs/tokio-core), mioco (https://github.com/dpc/mioco), coio-rs (https://github.com/zonyitoo/coio-rs), and rotor (https://github.com/tailhook/rotor).

How do I get command line arguments in Rust? (#how-do-i-get-command-line-arguments)

The easiest way is to use `Args` (https://doc.rust-lang.org/stable/std/env/struct.Args.html), which provides an iterator over the input arguments.

If you're looking for something more powerful, there are a number of options on crates.io (https://crates.io/keywords/argument).

Error Handling

Why doesn't Rust have exceptions? (#why-doesnt-rust-have-exceptions)

Exceptions complicate understanding of control-flow, they express validity/invalidity outside of the type system, and they interoperate poorly with multithreaded code (a major focus of Rust).

Rust prefers a type-based approach to error handling, which is covered at length in the book (https://doc.rust-lang.org/stable/book/second-edition/ch09-00-error-handling.html). This fits more nicely with Rust's control flow, concurrency, and everything else.

What's the deal with `unwrap()` everywhere? (#whats-the-deal-with-unwrap)

`unwrap()` is a function that extracts the value inside an `Option` (https://doc.rust-lang.org/stable/std/option/enum.Option.html) or `Result` (https://doc.rust-lang.org/stable/std/result/enum.Result.html) and panics if no value is present.

`unwrap()` shouldn't be your default way to handle errors you expect to arise, such as incorrect user input. In production code, it should be treated like an assertion that the value is non-empty, which will crash the program if violated.

It's also useful for quick prototypes where you don't want to handle an error yet, or blog posts where error handling would distract from the main point.

Why do I get an error when I try to run example code that uses the `try!` macro? (#why-do-i-get-errors-with-try)

It's probably an issue with the function's return type. The `try!` (https://doc.rust-lang.org/stable/std/macro.try!.html) macro either extracts the value from a `Result` (https://doc.rust-lang.org/stable/std/result/enum.Result.html), or returns early with the error `Result` (https://doc.rust-lang.org/stable/std/result/enum.Result.html) is carrying. This means that `try` (https://doc.rust-lang.org/stable/std/macro.try!.html) only works for functions that return `Result` (https://doc.rust-lang.org/stable/std/result/enum.Result.html) themselves, where the `Err`-constructed type implements `From::from(err)`. In particular, this means that the `try!` (https://doc.rust-lang.org/stable/std/macro.try!.html) macro cannot work inside the `main` function.

Is there an easier way to do error handling than having `Result`s everywhere? (#error-handling-without-result)

If you're looking for a way to avoid handling `Result`s (https://doc.rust-lang.org/stable/std/result/enum.Result.html) in other people's code, there's always `unwrap()` (https://doc.rust-lang.org/stable/core/option/enum.Option.html#method.unwrap), but it's probably not what you want. `Result` (https://doc.rust-lang.org/stable/std/result/enum.Result.html) is an indicator that some computation may or may not complete successfully. Requiring you to handle these failures explicitly is one of the ways that Rust encourages robustness. Rust provides tools like the `try!` macro (https://doc.rust-lang.org/stable/std/macro.try!.html) to make handling failures ergonomic.

If you really don't want to handle an error, use `unwrap()` (https://doc.rust-lang.org/stable/core/option/enum.Option.html#method.unwrap), but know that doing so means that the code panics on failure, which usually results in a shutting down the process.

Concurrency

Can I use static values across threads without an `unsafe` block? (#can-i-use-static-values-across-threads-without-an-unsafe-block)

Mutation is safe if it's synchronized. Mutating a static `Mutex` (https://doc.rust-lang.org/stable/std/sync/struct.Mutex.html) (lazily initialized via the lazy-static (https://crates.io/crates/lazy_static/) crate) does not require an `unsafe` block, nor does mutating a static `AtomicUsize` (https://doc.rust-lang.org/stable/std/sync/atomic/struct.AtomicUsize.html) (which can be initialized without lazy_static).

More generally, if a type implements `Sync` (https://doc.rust-lang.org/stable/std/marker/trait.Sync.html) and does not implement `Drop` (https://doc.rust-lang.org/stable/std/ops/trait.Drop.html), it can be used in a `static` (https://doc.rust-lang.org/book/const-and-static.html#static).

Macros

Can I write a macro to generate identifiers? (#can-i-write-a-macro-to-generate-identifiers)

Not currently. Rust macros are "hygienic macros" (https://en.wikipedia.org/wiki/Hygienic_macro), which intentionally avoid capturing or creating identifiers that may cause unexpected collisions with other identifiers. Their capabilities are significantly different than the style of macros commonly associated with the C preprocessor. Macro invocations can only appear in places where they are explicitly supported: items, method declarations, statements, expressions, and patterns. Here, "method declarations" means a blank space where a method can be put. They can't be used to complete a partial method declaration. By the same logic, they can't be used to complete a partial variable declaration.

Debugging and Tooling

How do I debug Rust programs? (#how-do-i-debug-rust-programs)

Rust programs can be debugged using gdb (https://sourceware.org/gdb/current/onlinedocs/gdb/) or lldb (http://lldb.llvm.org/tutorial.html), the same as C and C++. In fact, every Rust installation comes with one or both of rust-gdb and rust-lldb (depending on platform support). These are wrappers over gdb and lldb with Rust pretty-printing enabled.

`rustc` said a panic occurred in standard library code. How do I locate the mistake in my code? (#how-do-i-locate-a-panic)

This error is usually caused by `unwrap()`ing (https://doc.rust-lang.org/stable/core/option/enum.Option.html#method.unwrap) a `None` or `Err` in client code. Enabling backtraces by setting the environment variable `RUST_BACKTRACE=1` helps with getting more information. Compiling in debug mode (the default for `cargo build`) is also helpful. Using a debugger like the provided `rust-gdb` or `rust-lldb` is also helpful.

What IDE should I use? (#what-ide-should-i-use)

There are a number of options for development environment with Rust, all of which are detailed on the official IDE support page (https://forge.rust-lang.org/ides.html).

`gofmt` is great. Where's `rustfmt`? (#wheres-rustfmt)

`rustfmt` is right here (https://github.com/rust-lang-nursery/rustfmt), and is being actively developed to make reading Rust code as easy and predictable as possible.

Low-Level

How do I `memcpy` bytes? (#how-do-i-memcpy-bytes)

If you want to clone an existing slice safely, you can use `clone_from_slice` (https://doc.rust-lang.org/stable/std/primitive.slice.html#method.clone_from_slice).

To copy potentially overlapping bytes, use `copy` (https://doc.rust-lang.org/stable/std/ptr/fn.copy.html). To copy nonoverlapping bytes, use `copy_nonoverlapping` (https://doc.rust-lang.org/stable/std/ptr/fn.copy_nonoverlapping.html). Both of these functions are `unsafe`, as both can be used to subvert the language's safety guarantees. Take care when using them.

Can Rust function reasonably without the standard library? (#does-rust-work-without-the-standard-library)

Absolutely. Rust programs can be set to not load the standard library using the `#![no_std]` attribute. With this attribute set, you can continue to use the Rust core library, which is nothing but the platform-agnostic primitives. As such, it doesn't include IO, concurrency, heap allocation, etc.

Can I write an operating system in Rust? (#can-i-write-an-operating-system-in-rust)

Yes! In fact there are several projects underway doing just that (http://wiki.osdev.org/Rust).

How can I read or write numeric types like `i32` or `f64` in big-endian or little-endian format in a file or other

byte stream? (#how-can-i-write-endian-independent-values)

You should check out the byteorder crate (http://burntsushi.net/rustdoc/byteorder/), which provides utilities for exactly that.

Does Rust guarantee a specific data layout? (#does-rust-guarantee-data-layout)

Not by default. In the general case, `enum` and `struct` layouts are undefined. This allows the compiler to potentially do optimizations like re-using padding for the discriminant, compacting variants of nested `enum`s, reordering fields to remove padding, etc. `enums` which carry no data ("C-like") are eligible to have a defined representation. Such `enums` are easily distinguished in that they are simply a list of names that carry no data:

```
enum CLike {
    A,
    B = 32,
    C = 34,
    D
}
```

The `#[repr(C)]` attribute can be applied to such `enums` to give them the same representation they would have in equivalent C code. This allows using Rust `enum`s in FFI code where `C enum`s are also used, for most use cases. The attribute can also be applied to `struct`s to get the same layout as a `C struct` would.

Cross-Platform

What's the idiomatic way to express platform-specific behavior in Rust? (#how-do-i-express-platform-specific-behavior)

Platform-specific behavior can be expressed using conditional compilation attributes (https://doc.rust-lang.org/reference/attributes.html#conditional-compilation) such as `target_os`, `target_family`, `target_endian`, etc.

Can Rust be used for Android/iOS programming? (#can-rust-be-used-for-android-ios-programs)

Yes it can! There are already examples of using Rust for both Android (https://github.com/tomaka/android-rs-glue) and iOS (https://www.bignerdranch.com/blog/building-an-ios-app-in-rust-part-1/). It does require a bit of work to set up, but Rust functions fine on both platforms.

Can I run my Rust program in a web browser? (#can-i-run-my-rust-program-in-a-web-browser)

Possibly. Rust has experimental support (https://davidmcneil.gitbooks.io/the-rusty-web/) for both asm.js (http://asmjs.org/) and WebAssembly (http://webassembly.org/).

How do I cross-compile in Rust? (#how-do-i-cross-compile-rust)

Cross compilation is possible in Rust, but it requires a bit of work (https://github.com/japaric/rust-cross/blob/master/README.md) to set up. Every Rust compiler is a cross-compiler, but libraries need to be cross-compiled for the target platform.

Rust does distribute copies of the standard library (https://static.rust-lang.org/dist/index.html) for each of the supported platforms, which are contained in the `rust-std-*` files for each of the build directories found on the distribution page, but there are not yet automated ways to install them.

Modules and Crates

What is the relationship between a module and a crate? (#what-is-the-relationship-between-a-module-and-a-crate)

- A crate is a compilation unit, which is the smallest amount of code that the Rust compiler can operate on.
- A module is a (possibly nested) unit of code organization inside a crate.
- A crate contains an implicit, un-named top-level module.
- Recursive definitions can span modules, but not crates.

Why can't the Rust compiler find this library I'm `use`ing? (#why-cant-the-rust-compiler-find-a-library-im-using)

There are a number of possible answers, but a common mistake is not realizing that `use` declarations are relative to the crate root. Try rewriting your declarations to use the paths they would use if defined in the root file of your project and see if that fixes the problem.

There are also `self` and `super`, which disambiguate `use` paths as being relative to the current module or parent module, respectively.

For complete information on `use`ing libraries, read the Rust book's chapter "Crates and Modules" (https://doc.rust-lang.org/stable/book/second-edition/ch07-00-modules.html).

Why do I have to declare module files with `mod` at the top level of the crate, instead of just `use`ing them? (#why-do-i-have-to-declare-modules-with-mod)

There are two ways to declare modules in Rust, inline or in another file. Here is an example of each:

```
// In main.rs
mod hello {
    pub fn f() {
        println!("hello!");
    }
}

fn main() {
    hello::f();
}
```

```
// In main.rs
mod hello;

fn main() {
    hello::f();
}

// In hello.rs
pub fn f() {
    println!("hello!");
}
```

In the first example, the module is defined in the same file it's used. In the second example, the module declaration in the main file tells the compiler to look for either `hello.rs` or `hello/mod.rs`, and to load that file.

Note the difference between `mod` and `use`: `mod` declares that a module exists, whereas `use` references a module declared elsewhere, bringing its contents into scope within the current module.

How do I configure Cargo to use a proxy? (#how-do-i-configure-cargo-to-use-a-proxy)

As explained on the Cargo configuration documentation (http://doc.crates.io/config.html), you can set Cargo to use a proxy by setting the "proxy" variable under `[http]` in the configuration file.

Why can't the compiler find the method implementation even though I'm already `use`ing the crate? (#why-cant-the-compile-find-method-implementations)

For methods defined on a trait, you have to explicitly import the trait declaration. This means it's not enough to import a module where a struct implements the trait, you must also import the trait itself.

Why can't the compiler infer `use` declarations for me? (#why-cant-the-compiler-infer-use-statements)

It probably could, but you also don't want it to. While in many cases it is likely that the compiler could determine the correct module to import by simply looking for where a given identifier is defined, this may not be the case in general. Any decision rule in `rustc` for choosing between competing options would likely cause surprise and confusion in some cases, and Rust prefers to be explicit about where names are coming from.

For example, the compiler could say that in the case of competing identifier definitions the definition from the earliest imported module is chosen. So if both module `foo` and module `bar` define the identifier `baz`, but `foo` is the first registered module, the compiler would insert `use foo::baz;`.

```
mod foo;
mod bar;

// use foo::baz  // to be inserted by the compiler.

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

If you know this is going to happen, perhaps it saves a small number of keystrokes, but it also greatly increases the possibility for surprising error messages when you actually meant for `baz()` to be `bar::baz()`, and it decreases the readability of the code by making the meaning of a function call dependent on module declaration. These are not tradeoffs we are willing to make.

However, in the future, an IDE could help manage declarations, which gives you the best of both worlds: machine assistance for pulling in names, but explicit declarations about where those names are coming from.

How do I do dynamic Rust library loading? (#how-do-i-do-dynamic-rust-library-loading)

Import dynamic libraries in Rust with libloading (https://crates.io/crates/libloading), which provides a cross-platform system for dynamic linking.

Why doesn't crates.io have namespaces? (#why-doesnt-crates-io-have-namespaces)

Quoting the official explanation (https://internals.rust-lang.org/t/crates-io-package-policies/1041) of https://crates.io (https://crates.io)'s design:

> In the first month with crates.io, a number of people have asked us about the possibility of introducing namespaced packages (https://github.com/rust-lang/crates.io/issues/58).
>
> While namespaced packages allow multiple authors to use a single, generic name, they add complexity to how packages are referenced in Rust code and in human communication about packages. At first glance, they allow multiple authors to claim names like `http`, but that simply means that people will need to refer to those packages as `wycats' http` or `reem's http`, offering little benefit over package names like `wycats-http` or `reem-http`.
>
> When we looked at package ecosystems without namespacing, we found that people tended to go with more creative names (like `nokogiri` instead of "tenderlove's libxml2"). These creative names tend to be short and memorable, in part because of the lack of any hierarchy. They make it easier to communicate concisely and unambiguously about packages. They create exciting brands. And we've seen the success of several 10,000+ package ecosystems like NPM and RubyGems whose communities are prospering within a single namespace.
>
> In short, we don't think the Cargo ecosystem would be better off if Piston chose a name like `bvssvni/game-engine` (allowing other users to choose `wycats/game-engine`) instead of simply `piston`.
>
> Because namespaces are strictly more complicated in a number of ways, and because they can be added compatibly in the future should they become necessary, we're going to stick with a single shared namespace.

Libraries

How can I make an HTTP request? (#how-can-i-make-an-http-request)

The standard library does not include an implementation of HTTP, so you will want to use an external crate. reqwest (http://docs.rs/reqwest) is the simplest. It is built on hyper (https://github.com/hyperium/hyper), and written in Rust, but there are a number of others as well (https://crates.io/keywords/http). The curl (https://docs.rs/curl) crate is widely used and provides bindings to the curl library.

How can I write a GUI application in Rust? (#how-can-i-write-a-gui-application)

There are a variety of ways to write GUI applications in Rust. Just check out this list of GUI frameworks (https://github.com/kud1ing/awesome-rust#gui).

How can I parse JSON/XML? (#how-can-i-parse-json-xml)

Serde (https://github.com/serde-rs/serde) is the recommended library for serialization and deserialization of

Rust data to and from a number of different formats.

Is there a standard 2D+ vector and shape crate? (#is-there-a-standard-2d-vector-crate)

Not yet! Want to write one?

How do I write an OpenGL app in Rust? (#how-do-i-write-an-opengl-app)

Glium (https://github.com/tomaka/glium) is the major library for OpenGL programming in Rust. GLFW (https://github.com/bjz/glfw-rs) is also a solid option.

Can I write a video game in Rust? (#can-i-write-a-video-game-in-rust)

Yes you can! The major game programming library for Rust is Piston (http://www.piston.rs/), and there's both a subreddit for game programming in Rust (https://www.reddit.com/r/rust_gamedev/) and an IRC channel (`#rust-gamedev` on Mozilla IRC (https://wiki.mozilla.org/IRC)) as well.

Design Patterns

Is Rust object oriented? (#is-rust-object-oriented)

It is multi-paradigm. Many things you can do in OO languages you can do in Rust, but not everything, and not always using the same abstraction you're accustomed to.

How do I map object-oriented concepts to Rust? (#how-do-i-map-object-oriented-concepts-to-rust)

That depends. There *are* ways of translating object-oriented concepts like multiple inheritance (https://www.reddit.com/r/rust/comments/2sryuw/ideaquestion_about_multiple_inheritence/) to Rust, but as Rust is not object-oriented the result of the translation may look substantially different from its appearance in an OO language.

How do I handle configuration of a struct with optional parameters? (#how-do-i-configure-a-struct-with-optional-parameters)

The easiest way is to use the `Option` (https://doc.rust-lang.org/stable/std/option/enum.Option.html) type in whatever function you're using to construct instances of the struct (usually `new()`). Another way is to use the builder pattern (https://doc.rust-lang.org/stable/book/first-edition/method-syntax.html#builder-pattern), where only certain functions instantiating member variables must be called before the construction of the built type.

How do I do global variables in Rust? (#how-do-i-do-global-variables)

Globals in Rust can be done using `const` declarations for compile-time computed global constants, while `static` can be used for mutable globals. Note that modifying a `static mut` variable requires the use of `unsafe`, as it allows for data races, one of the things guaranteed not to happen in safe Rust. One important distinction between `const` and `static` values is that you can take references to `static` values, but not references to `const` values, which don't have a specified memory location. For more information on `const` vs. `static`, read the Rust book (https://doc.rust-lang.org/book/const-and-static.html).

How can I set compile-time constants that are defined procedurally? (#how-can-i-set-compile-time-constants-that-are-defined-procedurally)

Rust currently has limited support for compile time constants. You can define primitives using `const` declarations (similar to `static`, but immutable and without a specified location in memory) as well as define `const` functions and inherent methods.

To define procedural constants that can't be defined via these mechanisms, use the `lazy-static` (https://github.com/rust-lang-nursery/lazy-static.rs) crate, which emulates compile-time evaluation by automatically evaluating the constant at first use.

Can I run initialization code that happens before main? (#can-i-run-code-before-main)

Rust has no concept of "life before `main`". The closest you'll see can be done through the `lazy-static` (https://github.com/Kimundi/lazy-static.rs) crate, which simulates a "before main" by lazily initializing static variables at their first usage.

Does Rust allow non-constant-expression values for globals? (#does-rust-allow-non-constant-expression-values-for-globals)

No. Globals cannot have a non-constant-expression constructor and cannot have a destructor at all. Static constructors are undesirable because portably ensuring a static initialization order is difficult. Life before main is often considered a misfeature, so Rust does not allow it.

See the C++ FQA (http://yosefk.com/c++fqa/ctors.html#fqa-10.12) about the "static initialization order fiasco", and Eric Lippert's blog (https://ericlippert.com/2013/02/06/static-constructors-part-one/) for the challenges in C#, which also has this feature.

You can approximate non-constant-expression globals with the lazy-static (https://crates.io/crates/lazy_static/) crate.

Other Languages

How can I implement something like C's `struct X { static int X; };` in Rust? (#how-can-i-use-static-fields)

Rust does not have `static` fields as shown in the code snippet above. Instead, you can declare a `static` variable in a given module, which is kept private to that module.

How can I convert a C-style enum to an integer, or vice-versa? (#how-can-i-convert-a-c-style-enum-to-an-integer)

Converting a C-style enum to an integer can be done with an `as` expression, like `e as i64` (where `e` is some enum).

Converting in the other direction can be done with a `match` statement, which maps different numeric values to different potential values for the enum.

Why do Rust programs have larger binary sizes than C programs? (#why-do-rust-programs-have-larger-binary-sizes-than-C-programs)

There are several factors that contribute to Rust programs having, by default, larger binary sizes than functionally-equivalent C programs. In general, Rust's preference is to optimize for the performance of real-world programs, not the size of small programs.

Monomorphization

Rust monomorphizes generics, meaning that a new version of a generic function or type is generated for each concrete type it's used with in the program. This is similar to how templates work in C++. For example, in the following program:

```
fn foo<T>(t: T) {
    // ... do something
}

fn main() {
    foo(10);       // i32
    foo("hello");  // &str
}
```

Two distinct versions of `foo` will be in the final binary, one specialized to an `i32` input, one specialized to a `&str` input. This enables efficient static dispatch of the generic function, but at the cost of a larger binary.

Debug symbols

Rust programs compile with some debug symbols retained, even when compiling in release mode. These are used for providing backtraces on panics, and can be removed with `strip`, or another debug symbol removal tool. It is also useful to note that compiling in release mode with Cargo is equivalent to setting optimization level 3 with rustc. An alternative optimization level (called `s` or `z`) has recently landed (https://github.com/rust-lang/rust/pull/32386) and tells the compiler to optimize for size rather than performance.

Jemalloc

Rust uses jemalloc as the default allocator, which adds some size to compiled Rust binaries. Jemalloc is chosen because it is a consistent, quality allocator that has preferable performance characteristics compared to a number of common system-provided allocators. There is work being done to make it easier to use custom allocators (https://github.com/rust-lang/rust/issues/32838), but that work is not yet finished.

Link-time optimization

Rust does not do link-time optimization by default, but can be instructed to do so. This increases the amount of optimization that the Rust compiler can potentially do, and can have a small effect on binary size. This effect is likely larger in combination with the previously mentioned size optimizing mode.

Standard library

The Rust standard library includes libbacktrace and libunwind, which may be undesirable in some programs. Using `#![no_std]` can thus result in smaller binaries, but will also usually result in substantial changes to the sort of Rust code you're writing. Note that using Rust without the standard library is often functionally closer to the equivalent C code.

As an example, the following C program reads in a name and says "hello" to the person with that name.

```
#include <stdio.h>

int main(void) {
    printf("What's your name?\n");
    char input[100] = {0};
    scanf("%s", input);
    printf("Hello %s!\n", input);
    return 0;
}
```

Rewriting this in Rust, you may get something like the following:

```
use std::io;

fn main() {
    println!("What's your name?");
    let mut input = String::new();
    io::stdin().read_line(&mut input).unwrap();
    println!("Hello {}!", input);
}
```

This program, when compiled and compared against the C program, will have a larger binary and use more memory. But this program is not exactly equivalent to the above C code. The equivalent Rust code would instead look something like this:

```
#![feature(lang_items)]
#![feature(libc)]
#![feature(no_std)]
#![feature(start)]
#![no_std]

extern crate libc;

extern "C" {
    fn printf(fmt: *const u8, ...) -> i32;
    fn scanf(fmt: *const u8, ...) -> i32;
}

#[start]
fn start(_argc: isize, _argv: *const *const u8) -> isize {
    unsafe {
        printf(b"What's your name?\n\0".as_ptr());
        let mut input = [0u8; 100];
        scanf(b"%s\0".as_ptr(), &mut input);
        printf(b"Hello %s!\n\0".as_ptr(), &input);
        0
    }
}

#[lang="eh_personality"] extern fn eh_personality() {}
#[lang="panic_fmt"] fn panic_fmt() -> ! { loop {} }
#[lang="stack_exhausted"] extern fn stack_exhausted() {}
```

Which should indeed roughly match C in memory usage, at the expense of more programmer complexity, and a lack of static guarantees usually provided by Rust (avoided here with the use of `unsafe`).

Why does Rust not have a stable ABI like C does, and why do I have to annotate things with extern? (#why-no-stable-abi)

Committing to an ABI is a big decision that can limit potentially advantageous language changes in the future. Given that Rust only hit 1.0 in May of 2015, it is still too early to make a commitment as big as a stable ABI. This does not mean that one won’t happen in the future, though. (Though C++ has managed to go for many years without specifying a stable ABI.)

The `extern` keyword allows Rust to use specific ABI’s, such as the well-defined C ABI, for interop with other languages.

Can Rust code call C code? (#can-rust-code-call-c-code)

Yes. Calling C code from Rust is designed to be as efficient as calling C code from C++.

Can C code call Rust code? (#can-c-code-call-rust-code)

Yes. The Rust code has to be exposed via an `extern` declaration, which makes it C-ABI compatible. Such a function can be passed to C code as a function pointer or, if given the `#[no_mangle]` attribute to disable symbol mangling, can be called directly from C code.

I already write perfect C++. What does Rust give me? (#why-rust-vs-cxx)

Modern C++ includes many features that make writing safe and correct code less error-prone, but it's not perfect, and it's still easy to introduce unsafety. This is something the C++ core developers are working to overcome, but C++ is limited by a long history that predates a lot of the ideas they are now trying to implement.

Rust was designed from day one to be a safe systems programming language, which means it's not limited by historic design decisions that make getting safety right in C++ so complicated. In C++, safety is achieved by careful personal discipline, and is very easy to get wrong. In Rust, safety is the default. It gives you the ability to work in a team that includes people less perfect than you are, without having to spend your time double-checking their code for safety bugs.

How do I do the equivalent of C++ template specialization in Rust? (#how-to-get-cxx-style-template-specialization)

Rust doesn't currently have an exact equivalent to template specialization, but it is being worked on (https://github.com/rust-lang/rfcs/pull/1210) and will hopefully be added soon. However, similar effects can be achieved via associated types (https://doc.rust-lang.org/stable/book/second-edition/ch19-04-advanced-types.html).

How does Rust's ownership system relate to move semantics in C++? (#how-does-ownership-relate-to-cxx-move-semantics)

The underlying concepts are similar, but the two systems work very differently in practice. In both systems, "moving" a value is a way to transfer ownership of its underlying resources. For example, moving a string would transfer the string's buffer rather than copying it.

In Rust, ownership transfer is the default behavior. For example, if I write a function that takes a `String` as argument, this function will take ownership of the `String` value supplied by its caller:

```
fn process(s: String) { }

fn caller() {
    let s = String::from("Hello, world!");
    process(s); // Transfers ownership of `s` to `process`
    process(s); // Error! ownership already transferred.
}
```

As you can see in the snippet above, in the function `caller`, the first call to `process` transfers ownership of the variable `s`. The compiler tracks ownership, so the second call to `process` results in an error, because it is illegal to give away ownership of the same value twice. Rust will also prevent you from moving a value if there is an outstanding reference into that value.

C++ takes a different approach. In C++, the default is to copy a value (to invoke the copy constructor, more specifically). However, callees can declare their arguments using an "rvalue reference", like `string&&`, to indicate that they will take ownership of some of the resources owned by that argument (in this case, the string's internal buffer). The caller then must either pass a temporary expression or make an explicit move using `std::move`. The rough equivalent to the function `process` above, then, would be:

```
void process(string&& s) { }

void caller() {
    string s("Hello, world!");
    process(std::move(s));
    process(std::move(s));
}
```

C++ compilers are not obligated to track moves. For example, the code above compiles without a warning or error, at least using the default settings on clang. Moreover, in C++ ownership of the string `s` itself (if not its internal buffer) remains with `caller`, and so the destructor for `s` will run when `caller` returns, even though it has been moved (in Rust, in contrast, moved values are dropped only by their new owners).

How can I interoperate with C++ from Rust, or with Rust from C++? (#how-to-interoperate-with-cxx)

Rust and C++ can interoperate through C. Both Rust and C++ provide a foreign function interface (https://doc.rust-lang.org/book/ffi.html) for C, and can use that to communicate between each other. If writing C bindings is too tedious, you can always use rust-bindgen (https://github.com/servo/rust-bindgen) to help automatically generate workable C bindings.

Does Rust have C++-style constructors? (#does-rust-have-cxx-style-constructors)

No. Functions serve the same purpose as constructors without adding language complexity. The usual name for the constructor-equivalent function in Rust is `new()`, although this is just a convention rather than a language rule. The `new()` function in fact is just like any other function. An example of it looks like so:

```
struct Foo {
    a: i32,
    b: f64,
    c: bool,
}

impl Foo {
    fn new() -> Foo {
        Foo {
            a: 0,
            b: 0.0,
            c: false,
        }
    }
}
```

Does Rust have copy constructors? (#does-rust-have-copy-constructors)

Not exactly. Types which implement `Copy` will do a standard C-like "shallow copy" with no extra work (similar to trivially copyable types in C++). It is impossible to implement `Copy` types that require custom copy behavior. Instead, in Rust "copy constructors" are created by implementing the `Clone` trait, and explicitly calling the `clone` method. Making user-defined copy operators explicit surfaces the underlying complexity, making it easier for the developer to identify potentially expensive operations.

Does Rust have move constructors? (#does-rust-have-move-constructors)

No. Values of all types are moved via `memcpy`. This makes writing generic unsafe code much simpler since assignment, passing and returning are known to never have a side effect like unwinding.

How are Go and Rust similar, and how are they different? (#compare-go-and-rust)

Rust and Go have substantially different design goals. The following differences are not the only ones (which are too numerous to list), but are a few of the more important ones:

- Rust is lower level than Go. For example, Rust does not require a garbage collector, whereas Go does. In general, Rust affords a level of control that is comparable to C or C++.
- Rust's focus is on ensuring safety and efficiency while also providing high-level affordances, while Go's is on being a small, simple language which compiles quickly and can work nicely with a variety of tools.
- Rust has strong support for generics, which Go does not.
- Rust has strong influences from the world of functional programming, including a type system which draws from Haskell's typeclasses. Go has a simpler type system, using interfaces for basic generic programming.

How do Rust traits compare to Haskell typeclasses? (#how-do-rust-traits-compare-to-haskell-typeclasses)

Rust traits are similar to Haskell typeclasses, but are currently not as powerful, as Rust cannot express higher-kinded types. Rust's associated types are equivalent to Haskell type families.

Some specific difference between Haskell typeclasses and Rust traits include:

- Rust traits have an implicit first parameter called `Self`. `trait Bar` in Rust corresponds to `class Bar self` in Haskell, and `trait Bar<Foo>` in Rust corresponds to `class Bar foo self` in Haskell.
- "Supertraits" or "superclass constraints" in Rust are written `trait Sub: Super`, compared to `class Super self => Sub self` in Haskell.
- Rust forbids orphan instances, resulting in different coherence rules in Rust compared to Haskell.
- Rust's `impl` resolution considers the relevant `where` clauses and trait bounds when deciding whether two `impl`s overlap, or choosing between potential `impl`s. Haskell only considers the constraints in the `instance` declaration, disregarding any constraints provided elsewhere.
- A subset of Rust's traits (the "object safe" (https://github.com/rust-lang/rfcs/blob/master/text/0255-object-safety.md) ones) can be used for dynamic dispatch via trait objects. The same feature is available in Haskell via GHC's `ExistentialQuantification`.

Documentation

Why are so many Rust answers on Stack Overflow wrong? (#why-are-so-many-rust-answers-on-stackoverflow-wrong)

The Rust language has been around for a number of years, and only reached version 1.0 in May of 2015. In the time before then the language changed significantly, and a number of Stack Overflow answers were given at the time of older versions of the language.

Over time more and more answers will be offered for the current version, thus improving this issue as the proportion of out-of-date answers is reduced.

Where do I report issues in the Rust documentation? (#where-do-i-report-issues-in-the-rust-documentation)

You can report issues in the Rust documentation on the Rust compiler issue tracker (https://github.com/rust-lang/rust/issues). Make sure to read the contributing guidelines (https://github.com/rust-lang/rust/blob/master/CONTRIBUTING.md#writing-documentation) first.

How do I view rustdoc documentation for a library my project depends on? (#how-do-i-view-rustdoc-documentation-for-a-library-my-project-depends-on)

When you use `cargo doc` to generate documentation for your own project, it also generates docs for the active dependency versions. These are put into the `target/doc` directory of your project. Use `cargo doc --open` to open the docs after building them, or just open up `target/doc/index.html` yourself.

Our site in other languages: **Deutsch (/de-DE/)**, **English (/en-US/)**, **Español (/es-ES/)**, **Français (/fr-FR/)**, **Bahasa Indonesia (/id-ID/)**, **Italiano (/it-IT/)**, **日本語 (/ja-JP/)**, **한국어 (/ko-KR/)**, **Polski (/pl-PL/)**, **Português (/pt-BR/)**, **Русский (/ru-RU/)**, **Svenska (/sv-SE/)**, **Tiếng việt (/vi-VN/)**, **简体中文 (/zh-CN/)**