rmfeeder — Reading Bundle

Collected Articles

November 18, 2025

Contents

Rust (programming language)

Rust

Paradigms	• Concurrent • functional • generic • imperative • structured
Developer	The Rust Team
First appeared	January 19, 2012; 13 years ago
Stable release	1.91.1[1] / November 10, 2025; 8 days ago
Typing discipline	• Affine • inferred • nominal • static • strong
Implementation language	OCaml (2006–2011) Rust (2012–present)
Platform	Cross-platform[note 1]
OS	Cross-platform[note 2]
License	MIT, Apache 2.0[note 3]
Filename extensions	`.rs`, `.rlib`

Website	rust-lang.org

Influenced by

- Alef
- BETA
- CLU
- C#
- C++
- Cyclone
- Elm
- Erlang
- Haskell
- Hermes
- Limbo
- Mesa
- Napier
- Newsqueak
- NIL[note 4]
- OCaml
- Ruby
- Sather
- Scheme
- Standard ML
- Swift[7][8]

Influenced

- Idris[9]
- Project Verona[10]
- SPARK[11]
- Swift[12]
- V[13]
- Zig[14]

Rust is a general-purpose programming language. It is noted for its emphasis on performance, type safety, concurrency, and memory safety.

Rust supports multiple programming paradigms. It was influenced by ideas from functional programming, including immutability, higher-order functions, algebraic data types, and pattern matching. It also supports object-oriented programming via structs, enums, traits, and methods. Rust is noted for enforcing memory safety (i.e., that all references point to valid memory) without a conventional garbage collector; instead, memory safety errors and data races are prevented by the "borrow checker", which tracks the object lifetime of references at compile time.

Software developer Graydon Hoare created Rust in 2006 while working at Mozilla, which officially sponsored the project in 2009. The first stable release, Rust 1.0, was published in May 2015. Following a layoff of Mozilla employees in August 2020, four other companies joined Mozilla in sponsoring Rust through the creation of the Rust Foundation in February 2021.

Rust has been adopted by many software projects, especially web services and system software, and is the first language other than C and assembly to be supported in the development of the Linux kernel. It has been studied academically and has a growing community of developers.

2006–2009: Early years

[edit]

Mozilla Foundation headquarters, 650 Castro Street in Mountain View, California, June 2009

Rust began as a personal project by Mozilla employee Graydon Hoare in 2006. According to *MIT Technology Review*, he started the project due to his frustration with a broken elevator in his apartment building,[15] and named the language after the group of fungi of the same name that is "over-engineered for survival".[15] During the time period between 2006 and 2009, Rust was not publicized to others at

Mozilla and was written in Hoare's free time;[16]:7:50 Hoare began speaking about the language around 2009 after a small group at Mozilla became interested in the project.[17] Hoare cited languages from the 1970s, 1980s, and 1990s as influences — including CLU, BETA, Mesa, NIL,[note 4] Erlang, Newsqueak, Napier, Hermes, Sather, Alef, and Limbo.[17] He described the language as "technology from the past come to save the future from itself."[16]:8:17[17] Early Rust developer Manish Goregaokar similarly described Rust as being based on "mostly decades-old research."[15]

During the early years, the Rust compiler was written in about 38,000 lines of OCaml.[16]:15:34[18] Features of early Rust that were later removed include explicit object-oriented programming via an `obj` keyword,[16]:10:08 and a typestates system for variable state changes (such as going from uninitialized to initialized).[16]:13:12

Mozilla officially sponsored the Rust project in 2009.[15] Brendan Eich and other executives, intrigued by the possibility of using Rust for a safe web browser engine, placed engineers on the project including Patrick Walton, Niko Matsakis, Felix Klock, and Manish Goregaokar.[15] A conference room taken by the project developers was dubbed "the nerd cave," with a sign placed outside the door.[15]

During this time period, work had shifted from the initial OCaml compiler to a self-hosting compiler (*i.e.*, written in Rust) targetting LLVM.[19][note 5] The ownership system was in place by 2010.[15] The Rust logo was developed in 2011 based on a bicycle chainring.[21]

Rust 0.1 became the first public release on January 20, 2012[22] for Windows, Linux, and MacOS.[23] The early 2010s witnessed an increasing number of full-time engineers at Mozilla, as well as increasing involvement from open source volunteers outside of Mozilla and outside of the United States.[15]

2012–2015: Evolution

[edit]

The years from 2012 to 2015 were marked by substantial changes to the Rust type system.[16]:18:36[15] Memory management through the ownership system was gradually consolidated and expanded. By 2013, the garbage collector was rarely used, and was removed in favor of the ownership system.[15] Other features were removed in order to simplify the language, including typestates, the `pure` keyword,[24] various specialized pointer types, and syntax support for channels.[16]:22:32

According to Steve Klabnik, Rust was influenced during this period by developers coming from C++ (e.g., low-level performance of features), scripting languages (e.g., Cargo and package management), and functional programming (e.g., type systems development).[16]:30:50

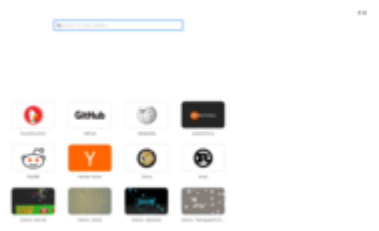
Graydon Hoare stepped down from Rust in 2013.[15] After Hoare's departure, it evolved organically under a federated governance structure, with a "core team" of initially six people,[16]:21:45 and around 30-40 developers total across various other teams.[16]:22:22 A Request for Comments (RFC) process for new language features was added in March 2014.[16]:33:47 The core team would grow to nine people by 2016[16]:21:45 with over 1600 RFCs.[16]:34:08

According to Andrew Binstock for *Dr. Dobb's Journal* in January 2014, while Rust was "widely viewed as a remarkably elegant language", adoption slowed because it radically changed from version to version.[25] Rust development at this time focused on finalizing features for version 1.0 so that it could begin promising backward compatibility.[16]:41:26

Six years after Mozilla's sponsorship, Rust 1.0 was published and became the first stable release on May 15, 2015.[15] A year after, the Rust compiler had accumulated over 1,400 contributors and there were over 5,000 third-party libraries published on the Rust package management website Crates.io.[16]:43:15

2015–2020: Servo and early adoption

[edit]



Early homepage of Mozilla's Servo browser engine

The development of the Servo browser engine continued in parallel with Rust, jointly funded by Mozilla and Samsung.[26] The teams behind the two projects worked in close collaboration; new features in Rust were tested out by the Servo team, and new features in Servo were used to give feedback back to the Rust team.[16]:5:41 The first version of Servo was released in 2016.[15] The Firefox web browser shipped with Rust code as of 2016 (version 45),[16]:53:30[27] but components of Servo did not appear in Firefox until September 2017 (version 57) as part of the Gecko and Quantum projects.[28]

Improvements were made to the Rust toolchain ecosystem during the years following 1.0 including Rustfmt, integrated development environment integration,[16]:44:56 and a regular compiler testing and release cycle.[16]:46:48 Rust gained a community code of conduct and an IRC chat for community discussion.[16]:50:36

The earliest known adoption outside of Mozilla was by individual projects at Samsung, Facebook (now Meta Platforms), Dropbox, and Tilde, Inc., the company behind ember.js.[16]:55:44[15] Amazon Web Services followed in 2020.[15] Engineers cited performance, lack of a garbage collector, safety, and pleasantness of working in the language as reasons for the adoption. Amazon developers cited a finding by Portuguese researchers that Rust code used less energy compared to similar code written in Java.[15][29]

2020–present: Mozilla layoffs and Rust Foundation

[edit]

In August 2020, Mozilla laid off 250 of its 1,000 employees worldwide, as part of a corporate restructuring caused by the COVID-19 pandemic.[30][31] The team behind Servo was disbanded. The event raised concerns about the future of Rust.[32] In the following week, the Rust Core Team acknowledged the severe impact of the layoffs and announced that plans for a Rust foundation were underway. The first goal of the foundation would be to take ownership of all trademarks and domain names and to take financial responsibility for their costs.[33]

On February 8, 2021, the formation of the Rust Foundation was announced by five founding companies: Amazon Web Services, Google, Huawei, Microsoft, and Mozilla.[34][35] The foundation would provide financial support for Rust developers in the form of grants and server funding.[15] In a blog post published on April 6, 2021, Google announced support for Rust within the Android Open Source Project as an alternative to C/C++.[36]

On November 22, 2021, the Moderation Team, which was responsible for enforcing the community code of conduct, announced their resignation "in protest of the Core Team placing themselves unaccountable to anyone but themselves".[37] In May 2022, members of the Rust leadership council posted a public response to the incident.[38]

The Rust Foundation posted a draft for a new trademark policy on April 6, 2023, which resulted in widespread negative reactions from Rust users and contributors.[39] The trademark policy included rules for how the Rust logo and name could be used.[39]

On February 26, 2024, the U.S. White House Office of the National Cyber Director released a 19-page press report urging software development to move away from C and C++ to memory-safe languages like C#, Go, Java, Ruby, Swift, and Rust.[40][41][42]

Syntax and features

[edit]

Rust's syntax is similar to that of C and C++,[43][44] although many of its features were influenced by functional programming languages such as OCaml.[45] Hoare has described Rust as targeted at frustrated C++ developers and emphasized safety, control of memory layout, and concurrency.[17]

Hello World program

[edit]

Below is a "Hello, World!" program in Rust. The `fn` keyword denotes a function, and the `println!` macro (see § Macros) prints the message to standard output.[46] Statements in Rust are separated by semicolons.

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

Variables in Rust are defined through the `let` keyword.[47] The example below assigns a value to the variable with name `foo` of type `i32` and outputs its value; the type annotation `: i32` can be omitted.

```
fn main() {
    let foo: i32 = 10;
    println!("The value of foo is {foo}");
}
```

Variables are immutable by default, unless the `mut` keyword is added.[48] The following example uses `//`, which denotes the start of a comment.[49]

```
fn main() {
    // This code would not compile without adding "mut".
    let mut foo = 10;
    println!("The value of foo is {foo}");
    foo = 20;
    println!("The value of foo is {foo}");
}
```

Multiple `let` expressions can define multiple variables with the same name, known as variable shadowing. Variable shadowing allows transforming variables without having to name the variables differently.[50] The example below declares a new variable with the same name that is double the original value:

```
fn main() {
    let foo = 10;
    // This will output "The value of foo is 10"
    println!("The value of foo is {foo}");
    let foo = foo * 2;
    // This will output "The value of foo is 20"
    println!("The value of foo is {foo}");
}
```

Variable shadowing is also possible for values of different types. For example, going from a string to its length:

```
fn main() {
    let letters = "abc";
    let letters = letters.len();
}
```

Block expressions and control flow

[edit]

A *block expression* is delimited by curly brackets. When the last expression inside a block does not end with a semicolon, the block evaluates to the value of that trailing expression:[51]

```
fn main() {
    let x = {
        println!("this is inside the block");
        1 + 2
    };
    println!("1 + 2 = {x}");
}
```

Trailing expressions of function bodies are used as the return value:[52]

```
fn add_two(x: i32) -> i32 {
    x + 2
}
```

An `if` conditional expression executes code based on whether the given value is `true`. `else` can be used for when the value evaluates to `false`, and `else if` can be used for combining multiple expressions.[53]

```
fn main() {
    let x = 10;
    if x > 5 {
        println!("value is greater than five");
    }

    if x % 7 == 0 {
        println!("value is divisible by 7");
    } else if x % 5 == 0 {
        println!("value is divisible by 5");
    } else {
        println!("value is not divisible by 7 or 5");
    }
}
```

`if` and `else` blocks can evaluate to a value, which can then be assigned to a variable:[53]

```
fn main() {
    let x = 10;
    let new_x = if x % 2 == 0 { x / 2 } else { 3 * x + 1 };
    println!("{new_x}");
}
```

`while` can be used to repeat a block of code while a condition is met.[54]

```
fn main() {
    // Iterate over all integers from 4 to 10
    let mut value = 4;
    while value <= 10 {
         println!("value = {value}");
         value += 1;
    }
}
```

`for` loops and iterators

[edit]

For loops in Rust loop over elements of a collection.[55] `for` expressions work over any iterator type.

```
fn main() {
    // Using `for` with range syntax for the same functionality as
above
```

```
    // The syntax 4..=10 means the range from 4 to 10, up to and
including 10.
    for value in 4..=10 {
        println!("value = {value}");
    }
}
```

In the above code, `4..=10` is a value of type `Range` which implements the `Iterator` trait. The code within the curly braces is applied to each element returned by the iterator.

Iterators can be combined with functions over iterators like `map`, `filter`, and `sum`. For example, the following adds up all numbers between 1 and 100 that are multiples of 3:

```
(1..=100).filter(|x: &u32| -> bool { x % 3 == 0 }).sum()
```

`loop` and `break` statements

[edit]

More generally, the `loop` keyword allows repeating a portion of code until a `break` occurs. `break` may optionally exit the loop with a value. In the case of nested loops, labels denoted by `'label_name` can be used to break an outer loop rather than the innermost loop.[56]

```
fn main() {
    let value = 456;
    let mut x = 1;
    let y = loop {
        x *= 10;
        if x > value {
            break x / 10;
        }
    };
    println!("largest power of ten that is smaller than or equal to
value: {y}");

    let mut up = 1;
    'outer: loop {
        let mut down = 120;
        loop {
            if up > 100 {
                break 'outer;
            }

            if down < 4 {
                break;
            }

            down /= 2;
            up += 1;
            println!("up: {up}, down: {down}");
        }
        up *= 2;
    }
}
```

The `match` and `if let` expressions can be used for pattern matching. For example, `match` can be used to double an optional integer value if present, and return zero otherwise:[57]

```
fn double(x: Option<u64>) -> u64 {
    match x {
        Some(y) => y * 2,
        None => 0,
    }
}
```

Equivalently, this can be written with `if let` and `else`:

```
fn double(x: Option<u64>) -> u64 {
    if let Some(y) = x {
        y * 2
    } else {
        0
    }
}
```

Rust is strongly typed and statically typed, meaning that the types of all variables must be known at compilation time. Assigning a value of a particular type to a differently typed variable causes a compilation error. Type inference is used to determine the type of variables if unspecified.[58]

The type `()`, called the "unit type" in Rust, is a concrete type that has exactly one value. It occupies no memory (as it represents the absence of value). All functions that do not have an indicated return type implicitly return `()`. It is similar to `void` in other C-style languages, however `void` denotes the absence of a type and cannot have any value.

The default integer type is `i32`, and the default floating point type is `f64`. If the type of a literal number is not explicitly provided, it is either inferred from the context or the default type is used.[59]

Integer types in Rust are named based on the signedness and the number of bits the type takes. For example, `i32` is a signed integer that takes 32 bits of storage, whereas `u8` is unsigned and only takes 8 bits of storage. `isize` and `usize` take storage depending on the memory address bus width of the compilation target. For example, when building for 32-bit targets, both types will take up 32 bits of space.[60][61]

By default, integer literals are in base-10, but different radices are supported with prefixes, for example, `0b11` for binary numbers, `0o567` for octals, and `0xDB` for hexadecimals. By default, integer literals default to `i32` as its type. Suffixes such as `4u32` can be used to explicitly set the type of a literal.[62] Byte literals such as `b'X'` are available to represent the ASCII value (as a `u8`) of a specific character.[63]

The Boolean type is referred to as `bool` which can take a value of either `true` or `false`. A `char` takes up 32 bits of space and represents a Unicode scalar value:[64] a Unicode codepoint that is not a surrogate.[65] IEEE 754 floating point numbers are supported with `f32` for single precision floats and `f64` for double precision floats.[66]

Compound types can contain multiple values. Tuples are fixed-size lists that can contain values whose types can be different. Arrays are fixed-size lists whose values are of the same type. Expressions of the tuple and array types can be written through listing the values, and can be accessed with `.index` or `[index]`:[67]

```
let tuple: (u32, bool) = (3, true);
let array: [i8; 5] = [1, 2, 3, 4, 5];
let value = tuple.1; // true
let value = array[2]; // 3
```

Arrays can also be constructed through copying a single value a number of times: [68]

```
let array2: [char; 10] = [' '; 10];
```

Ownership and references

[edit]

Rust's ownership system consists of rules that ensure memory safety without using a garbage collector. At compile time, each value must be attached to a variable called the *owner* of that value, and every value must have exactly one owner.[69] Values are moved between different owners through assignment or passing a value as a function parameter. Values can also be *borrowed,* meaning they are temporarily passed to a different function before being returned to the owner.[70] With these rules, Rust can prevent the creation and use of dangling pointers:[70][71]

```
fn print_string(s: String) {
    println!("{}", s);
}

fn main() {
    let s = String::from("Hello, World");
    print_string(s); // s consumed by print_string
    // s has been moved, so cannot be used any more
    // another print_string(s); would result in a compile error
}
```

The function `print_string` takes ownership over the `String` value passed in; Alternatively, `&` can be used to indicate a reference type (in `&String`) and to create a reference (in `&s`):[72]

```
fn print_string(s: &String) {
    println!("{}", s);
}

fn main() {
    let s = String::from("Hello, World");
    print_string(&s); // s borrowed by print_string
    print_string(&s); // s has not been consumed; we can call the
function many times
}
```

Because of these ownership rules, Rust types are known as *linear* or *affine* types, meaning each value can be used exactly once. This enforces a form of software fault isolation as the owner of a value is solely responsible for its correctness and deallocation.[73]

When a value goes out of scope, it is *dropped* by running its destructor. The destructor may be programmatically defined through implementing the `Drop` trait. This helps manage resources such as file handles, network sockets, and locks, since when objects are dropped, the resources associated with them are closed or released automatically.[74]

Object lifetime refers to the period of time during which a reference is valid; that is, the time between the object creation and destruction.[75] These *lifetimes* are implicitly associated with all Rust reference types. While often inferred, they can also be indicated explicitly with named lifetime parameters (often denoted `'a`, `'b`, and so on).[76]

Lifetimes in Rust can be thought of as lexically scoped, meaning that the duration of an object lifetime is inferred from the set of locations in the source code (i.e., function, line, and column numbers) for which a variable is valid.[77] For example, a reference to a local variable has a lifetime corresponding to the block it is defined in: [77]

```
fn main() {
    let x = 5;                // ------------------+-- Lifetime 'a
                              //                   |
    let r = &x;               // -+-- Lifetime 'b  |
                              //  |                |
    println!("r: {}", r);     //  |                |
                              //  |                |
                              // -+                |
}                             // ------------------+
```

The borrow checker in the Rust compiler then enforces that references are only used in the locations of the source code where the associated lifetime is valid.[78][79] In the example above, storing a reference to variable `x` in `r` is valid, as variable `x` has a longer lifetime (`'a`) than variable `r` (`'b`). However, when `x` has a shorter lifetime, the borrow checker would reject the program:

```
fn main() {
    let r;                    // ------------------+- Lifetime 'a
                              //                   |
    {                         //                   |
        let x = 5;            // -+-- Lifetime 'b  |
        r = &x; // ERROR: x does  |                |
    }           // not live long -|                |
                // enough                          |
    println!("r: {}", r);     //                   |
}                             // ------------------+
```

Since the lifetime of the referenced variable (`'b`) is shorter than the lifetime of the variable holding the reference (`'a`), the borrow checker errors, preventing `x` from being used from outside its scope.[80]

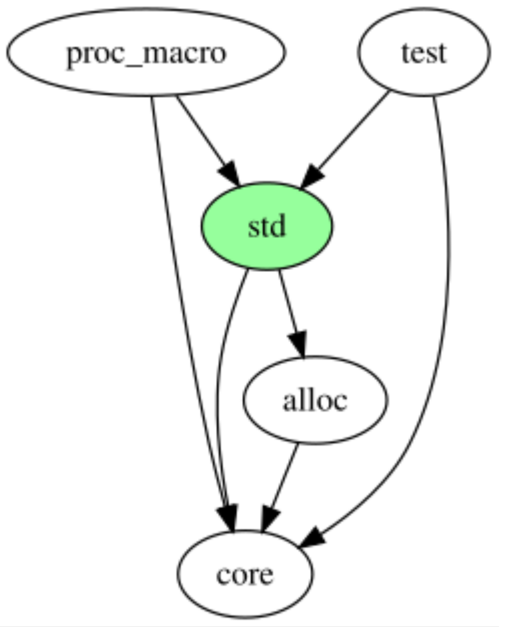
Lifetimes can be indicated using explicit *lifetime parameters* on function arguments. For example, the following code specifies that the reference returned by the function has the same lifetime as `original` (and *not* necessarily the same lifetime as `prefix`): [81]

```
fn remove_prefix<'a>(mut original: &'a str, prefix: &str) -> &'a str
{
    if original.starts_with(prefix) {
        original = original[prefix.len()..];
    }
    original
}
```

In the compiler, ownership and lifetimes work together to prevent memory safety issues such as dangling pointers.[82][83]

User-defined types are created with the `struct` or `enum` keywords. The `struct` keyword is used to denote a record type that groups multiple related values.[84] `enum`s can take on different variants at runtime, with its capabilities similar to algebraic data types found in functional programming languages.[85] Both records and enum variants can contain fields with different types.[86] Alternative names, or aliases, for the same type can be defined with the `type` keyword.[87]

The `impl` keyword can define methods for a user-defined type. Data and functions are defined separately. Implementations fulfill a role similar to that of classes within other languages.[88]



A diagram of the dependencies between the standard library modules of Rust.

The Rust standard library defines and implements many widely used custom data types, including core data structures such as `Vec`, `Option`, and `HashMap`, as well as smart pointer types. Rust provides a way to exclude most of the standard library using the attribute `#![no_std]`, for applications such as embedded devices. Internally, the standard library is divided into three parts, `core`, `alloc`, and `std`, where `std` and `alloc` are excluded by `#![no_std]`.[89]

Rust uses the option type `Option<T>` to define optional values, which can be matched using `if let` or `match` to access the inner value:[90]

```
fn main() {
    let name1: Option<&str> = None;
    // In this case, nothing will be printed out
    if let Some(name) = name1 {
        println!("{name}");
    }

    let name2: Option<&str> = Some("Matthew");
    // In this case, the word "Matthew" will be printed out
    if let Some(name) = name2 {
        println!("{name}");
    }
}
```

Similarly, Rust's result type `Result<T, E>` holds either a successfully computed value (the `Ok` variant) or an error (the `Err` variant).[91] Like `Option`, the use of `Result` means that the inner value cannot be used directly; programmers must use a `match` expression, syntactic sugar such as `?` (the "try" operator), or an explicit `unwrap` assertion to access it. Both `Option` and `Result` are used throughout the standard library and are a fundamental part of Rust's explicit approach to handling errors and missing data.

The `&` and `&mut` reference types are guaranteed to not be null and point to valid memory.[92] The raw pointer types `*const` and `*mut` opt out of the safety guarantees, thus they may be null or invalid; however, it is impossible to dereference them unless the code is explicitly declared unsafe through the use of an `unsafe` block.[93] Unlike dereferencing, the creation of raw pointers is allowed inside safe Rust code.[94]

Rust provides no implicit type conversion (coercion) between most primitive types. But, explicit type conversion (casting) can be performed using the `as` keyword.[95]

```
let x: i32 = 1000;
println!("1000 as a u16 is: {}", x as u16);
```

A presentation on Rust by Emily Dunham from Mozilla's Rust team (linux.conf.au conference, Hobart, 2017)

Rust supports polymorphism through traits, generic functions, and trait objects.[96]

Common behavior between types is declared using traits and `impl` blocks:[97]

```
trait Zero: Sized {
    fn zero() -> Self;
    fn is_zero(&self) -> bool
    where
        Self: PartialEq,
    {
        self == &Zero::zero()
    }
}

impl Zero for u32 {
    fn zero() -> u32 { 0 }
}

impl Zero for f32 {
    fn zero() -> Self { 0.0 }
}
```

The example above includes a method `is_zero` which provides a default implementation that may be overridden when implementing the trait.[97]

A function can be made generic by adding type parameters inside angle brackets (`<Num>`), which only allow types that implement the trait:

```
// zero is a generic function with one type parameter, Num
fn zero<Num: Zero>() -> Num {
    Num::zero()
}

fn main() {
    let a: u32 = zero();
    let b: f32 = zero();
    assert!(a.is_zero() && b.is_zero());
}
```

In the examples above, `Num: Zero` as well as `where Self: PartialEq` are trait bounds that constrain the type to only allow types that implement `Zero` or `PartialEq`.[97] Within a trait or impl, `Self` refers to the type that the code is implementing.[98]

Generics can be used in functions to allow implementing a behavior for different types without repeating the same code (see bounded parametric polymorphism). Generic functions can be written in relation to other generics, without knowing the actual type.[99]

By default, traits use static dispatch: the compiler monomorphizes the function for each concrete type instance, yielding performance equivalent to type-specific code at the cost of longer compile times and larger binaries.[100]

When the exact type is not known at compile time, Rust provides trait objects `&dyn Trait` and `Box<dyn Trait>`.[101] Trait object calls use dynamic dispatch via a lookup table; a trait object is a "fat pointer" carrying both a data pointer and a method table pointer.[100] This indirection adds a small runtime cost, but it keeps a single copy of the code and reduces binary size. Only "object-safe" traits are eligible to be used as trait objects.[102]

This approach is similar to duck typing, where all data types that implement a given trait can be treated as functionally interchangeable.[103] The following example creates a list of objects where each object implements the `Display` trait:

```
use std::fmt::Display;

let v: Vec<Box<dyn Display>> = vec![
    Box::new(3),
    Box::new(5.0),
    Box::new("hi"),
];

for x in v {
    println!("{x}");
}
```

If an element in the list does not implement the `Display` trait, it will cause a compile-time error.[104]

Rust does not use garbage collection. Memory and other resources are instead managed through the "resource acquisition is initialization" convention,[105] with optional reference counting. Rust provides deterministic management of resources, with very low overhead.[106] Values are allocated on the stack by default, and all dynamic allocations must be explicit.[107]

The built-in reference types using the `&` symbol do not involve run-time reference counting. The safety and validity of the underlying pointers is verified at compile time, preventing dangling pointers and other forms of undefined behavior.[108]

Rust's type system separates shared, immutable references of the form `&T` from unique, mutable references of the form `&mut T`. A mutable reference can be coerced to an immutable reference, but not vice versa.[109]

Rust's memory safety checks (See #Safety) may be circumvented through the use of `unsafe` blocks. This allows programmers to dereference arbitrary raw pointers, call external code, or perform other low-level functionality not allowed by safe Rust.[110] Some low-level functionality enabled in this way includes volatile memory access, architecture-specific intrinsics, type punning, and inline assembly.[111]

Unsafe code is sometimes needed to implement complex data structures.[112] A frequently cited example is that it is difficult or impossible to implement doubly linked lists in safe Rust.[113][114][115][116]

Programmers using unsafe Rust are considered responsible for upholding Rust's memory and type safety requirements, for example, that no two mutable references exist pointing to the same location.[117] If programmers write code which violates these requirements, this results in undefined behavior.[117] The Rust documentation includes a list of behavior considered undefined, including accessing dangling or misaligned pointers, or breaking the aliasing rules for references.[118]

Macros allow generation and transformation of Rust code to reduce repetition. Macros come in two forms, with *declarative macros* defined through `macro_rules!`, and *procedural macros*, which are defined in separate crates.[119][120]

A declarative macro (also called a "macro by example") is a macro, defined using the `macro_rules!` keyword, that uses pattern matching to determine its expansion.[121][122] Below is an example that sums over all its arguments:

```
macro_rules! sum {
    ( $initial:expr $(, $expr:expr )* $(,)? ) => {
        $initial $(+ $expr)*
    }
}

fn main() {
    let x = sum!(1, 2, 3);
    println!("{x}"); // prints 6
}
```

Procedural macros are Rust functions that run and modify the compiler's input token stream, before any other components are compiled. They are generally more flexible than declarative macros, but are more difficult to maintain due to their complexity.[123][124]

Procedural macros come in three flavors:

- Function-like macros `custom!(...)`
- Derive macros `#[derive(CustomDerive)]`
- Attribute macros `#[custom_attribute]`

Interface with C and C++

[edit]

Rust supports the creation of foreign function interfaces (FFI) through the `extern` keyword. A function that uses the C calling convention can be written using `extern "C" fn`. Symbols can be exported from Rust to other languages through the `#[unsafe(no_mangle)]` attribute, and symbols can be imported into Rust through `extern` blocks:[note 6][126]

```
#[unsafe(no_mangle)]
pub extern "C" fn exported_from_rust(x: i32) -> i32 { x + 1 }
unsafe extern "C" {
    fn imported_into_rust(x: i32) -> i32;
}
```

The `#[repr(C)]` attribute enables deterministic memory layouts for `struct`s and `enum`s for use across FFI boundaries.[126] External libraries such as `bindgen` and `cxx` can generate Rust bindings for C/C++.[126][127]

Safety properties guaranteed by Rust include memory safety, type safety, and data race freedom. As described above, these guarantees can be circumvented by using the `unsafe` keyword.

Memory safety includes the absence of dereferences to null, dangling, and misaligned pointers, and the absence of buffer overflows and double free errors.[128][129][130][131] Data values can be initialized only through a fixed set of forms, all of which require their inputs to be already initialized.[132]

Memory leaks are possible in safe Rust.[133][134]

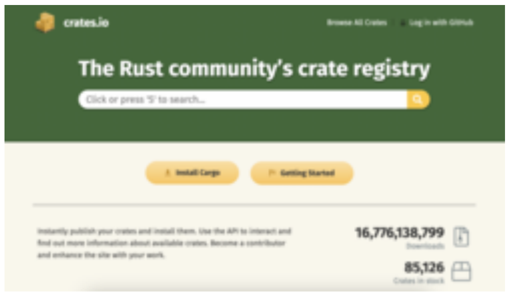
Some users have pointed out that Rust's safety guarantees can be circumvented by exploiting unsoundness in the Rust compiler.[135]

Compiling a Rust program with Cargo

The Rust ecosystem includes its compiler, its standard library, and additional components for software development. Component installation is typically managed by `rustup`, a Rust toolchain installer developed by the Rust project.[136]

The Rust compiler, `rustc`, compiles Rust code into binaries. First, the compiler parses the source code into an AST. Next, this AST is lowered to IR. The compiler backend is then invoked as a subcomponent to apply optimizations and translate the resulting IR into object code. Finally, a linker is used to combine the object(s) into a single executable image.[137]

rustc uses LLVM as its compiler backend by default, but it also supports using alternative backends such as GCC and Cranelift.[138] The intention of those alternative backends is to increase platform coverage of Rust or to improve compilation times.[139][140]

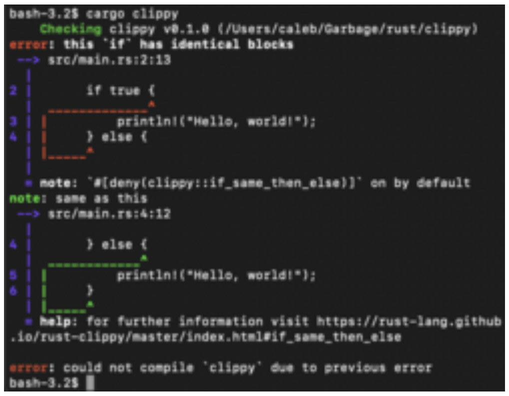

crates.io
The Rust community's crate registry
Click or press 'S' to search...
Install Cargo
Getting Started
16,776,138,799
85,126

Screenshot of crates.io in June 2022

Cargo is Rust's build system and package manager. It downloads, compiles, distributes, and uploads packages—called *crates*—that are maintained in an official registry. It also acts as a front-end for Clippy and other Rust components.[141]

By default, Cargo sources its dependencies from the user-contributed registry *crates.io*, but Git repositories, crates in the local filesystem, and other external sources can also be specified as dependencies.[142]

Rustfmt is a code formatter for Rust. It formats whitespace and indentation to produce code in accordance with a common style, unless otherwise specified. It can be invoked as a standalone program, or from a Rust project through Cargo.[143]



```
bash-3.2$ cargo clippy
    Checking clippy v0.1.0 (/Users/caleb/Garbage/rust/clippy)
error: this `if` has identical blocks
 --> src/main.rs:2:13
  |
2 |         if true {
  |  _____________^
3 | |           println!("Hello, world!");
4 | |       } else {
  | |_____^
  |
  = note: `#[deny(clippy::if_same_then_else)]` on by default
note: same as this
 --> src/main.rs:4:12
  |
4 |       } else {
  |  ____________^
5 | |           println!("Hello, world!");
6 | |       }
  | |_____^
  = help: for further information visit https://rust-lang.github
.io/rust-clippy/master/index.html#if_same_then_else

error: could not compile `clippy` due to previous error
bash-3.2$
```

Example output of Clippy on a hello world Rust program

Clippy is Rust's built-in linting tool to improve the correctness, performance, and readability of Rust code. As of 2025, it has 795 rules.[144][145]

Following Rust 1.0, new features are developed in *nightly* versions which are released daily. During each six-week release cycle, changes to nightly versions are released to beta, while changes from the previous beta version are released to a new stable version.[146]

Every two or three years, a new "edition" is produced. Editions are released to allow making limited breaking changes, such as promoting `await` to a keyword to support async/await features. Crates targeting different editions can interoperate with each other, so a crate can upgrade to a new edition even if its callers or its dependencies still target older editions. Migration to a new edition can be assisted with automated tooling.[147]

rust-analyzer is a set of utilities that provides integrated development environments (IDEs) and text editors with information about a Rust project through the Language Server Protocol. This enables features including autocomplete, and compilation error display, while editing code.[148]

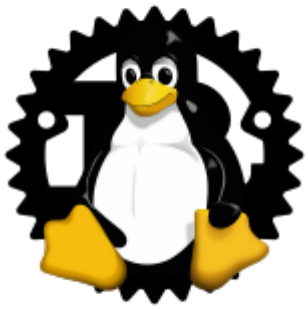
Since it performs no garbage collection, Rust is often faster than other memory-safe languages.[149][73][150] Most of Rust's memory safety guarantees impose no runtime overhead,[151] with the exception of array indexing which is checked at runtime by default.[152] The performance impact of array indexing bounds checks varies, but can be significant in some cases.[152]

Many of Rust's features are so-called *zero-cost abstractions*, meaning they are optimized away at compile time and incur no runtime penalty.[153] The ownership and borrowing system permits zero-copy implementations for some performance-sensitive tasks, such as parsing.[154] Static dispatch is used by default to eliminate method calls, except for methods called on dynamic trait objects.[155] The compiler uses inline expansion to eliminate function calls and statically-dispatched method invocations.[156]

Since Rust uses LLVM, all performance improvements in LLVM apply to Rust also.[157] Unlike C and C++, Rust allows the compiler to reorder struct and enum elements unless a `#[repr(C)]` representation attribute is applied.[158] This allows the compiler to optimize for memory footprint, alignment, and padding, which can be used to produce more efficient code in some cases.[159]

Firefox has components written in Rust as part of the underlying Gecko browser engine.

In web services, OpenDNS, a DNS resolution service owned by Cisco, uses Rust internally.[160][161] Amazon Web Services uses Rust in "performance-sensitive components" of its several services. In 2019, AWS open-sourced Firecracker, a virtualization solution primarily written in Rust.[162] Microsoft Azure IoT Edge, a platform used to run Azure services on IoT devices, has components implemented in Rust.[163] Microsoft also uses Rust to run containerized modules with WebAssembly and Kubernetes.[164] Cloudflare, a company providing content delivery network services, used Rust to build a new web proxy named Pingora for increased performance and efficiency.[165] The npm package manager used Rust for its production authentication service in 2019.[166][167][168]



The Rust for Linux project has been supported in the Linux kernel since 2022.

In operating systems, the Rust for Linux project, launched in 2020, merged initial support into the Linux kernel version 6.1 in late 2022.[169][170][171] The project is active with a team of 6–7 developers, and has added more Rust code with kernel releases from 2022 to 2024,[172] aiming to demonstrate the minimum viability of the project and resolve key compatibility blockers.[169][173] The first drivers written in Rust were merged into the kernel for version 6.8.[169] The Android developers used Rust in 2021 to rewrite existing components.[174][175] Microsoft has rewritten parts of Windows in Rust.[176] The r9 project aims to re-implement Plan 9 from Bell Labs in Rust.[177] Rust has been used in the development of new operating systems such as Redox, a "Unix-like" operating system and microkernel,[178] Theseus, an experimental operating system with modular state management,[179][180] and most of Fuchsia.[181] Rust is used for command-line tools and operating system components such as stratisd, a file system manager[182][183] and COSMIC, a desktop environment by System76.[184]

In web development, Deno, a secure runtime for JavaScript and TypeScript, is built on top of V8 using Rust and Tokio.[185] Other notable adoptions in this space include Ruffle, an open-source SWF emulator,[186] and Polkadot, an open source blockchain and cryptocurrency platform.[187] Components from the Servo browser engine (funded by Mozilla and Samsung) were incorporated in the Gecko browser engine underlying Firefox.[188] In January 2023, Google (Alphabet) announced support for using third party Rust libraries in Chromium.[189][190]

In other uses, Discord, an instant messaging software company, rewrote parts of its system in Rust for increased performance in 2020. In the same year, Dropbox announced that its file synchronization had been rewritten in Rust. Facebook (Meta) used Rust to redesign its system that manages source code for internal projects.[15]

In the 2025 Stack Overflow Developer Survey, 14.8% of respondents had recently done extensive development in Rust.[191] The survey named Rust the "most admired programming language" annually from 2016 to 2025 (inclusive), as measured by the number of existing developers interested in continuing to work in the language.[192][note 7] In 2025, 29.2% of developers not currently working in Rust expressed an interest in doing so.[191]

In academic research

[edit]

Rust's safety and performance have been investigated in programming languages research.[193][112][194]

In other fields, a journal article published to *Proceedings of the International Astronomical Union* used Rust to simulate multi-planet systems.[195] An article published in *Nature* shared stories of bioinformaticians using Rust.[141] Both articles cited Rust's performance and safety as advantages, and the learning curve as being a primary drawback to Rust adoption.

The 2025 DARPA project TRACTOR aims to automatically translate C to Rust using techniques such as static analysis, dynamic analysis, and large language models.[196]

Some Rust users refer to themselves as Rustaceans (similar to the word crustacean) and have adopted an orange crab, Ferris, as their unofficial mascot.[197][198]

According to the *MIT Technology Review*, the Rust community has been seen as "unusually friendly" to newcomers and particularly attracted people from the queer community, partly due to its code of conduct.[15] Inclusiveness has been cited as an important factor for some Rust developers.[141] The official Rust blog collects and publishes demographic data each year.[199]

Rust Foundation

Formation	February 8, 2021; 4 years ago
Founders	• Amazon Web Services • Google • Huawei • Microsoft • Mozilla Foundation
Type	Nonprofit organization
Location	• United States
Chairperson	Shane Miller
Executive Director	Rebecca Rumbul

Website	foundation.rust-lang.org

The **Rust Foundation** is a non-profit membership organization incorporated in United States; it manages the Rust trademark, infrastructure, and assets.[200][44]

It was established on February 8, 2021, with five founding corporate members (Amazon Web Services, Huawei, Google, Microsoft, and Mozilla).[201] The foundation's board was chaired by Shane Miller,[202] with Ashley Williams as interim executive director.[44] In late 2021, Rebecca Rumbul became Executive Director and CEO.[203]

The Rust project is composed of *teams* that are responsible for different subareas of development, including the compiler team and language team. The Rust project website lists 6 top-level teams as of July 2024.[204] Representatives among teams form the Leadership council, which oversees the Rust project as a whole.[205]

- Comparison of programming languages
- History of programming languages
- List of programming languages
- List of programming languages by type
- List of Rust software and tools
- Outline of the Rust programming language

1. ^ Including build tools, host tools, and standard library support for x86-64, ARM, MIPS, RISC-V, WebAssembly, i686, AArch64, PowerPC, and s390x.[2]
2. ^ Including Windows, Linux, macOS, FreeBSD, NetBSD, and Illumos. Host build tools on Android, iOS, Haiku, Redox, and Fuchsia are not officially shipped; these operating systems are supported as targets.[2]
3. ^ Third-party dependencies, e.g., LLVM or MSVC, are subject to their own licenses.[3][4]
4. ^ ***a*** ***b*** NIL is cited as an influence for Rust in multiple sources; this likely refers to Network Implementation Language developed by Robert Strom and others at IBM, which pioneered typestate analysis,[5][6] not to be confused with New Implementation of LISP.
5. ^ The list of Rust compiler versions (referred to as a bootstrapping chain) has history going back to 2012.[20]
6. ^ wrapping `no_mangle` with `unsafe` as well as prefacing the `extern "C"` block with `unsafe` are required in the 2024 edition or later.[125]

7. ^ That is, among respondents who have done "extensive development work [with Rust] in over the past year" (14.8%), Rust had the largest percentage who also expressed interest to "work in [Rust] over the next year" (72.4%).[191]

- *Gjengset, Jon (2021). Rust for Rustaceans (1st ed.). No Starch Press. ISBN 9781718501850. OCLC 1277511986.*
- *Klabnik, Steve; Nichols, Carol (2019-08-12). The Rust Programming Language (Covers Rust 2018). No Starch Press. ISBN 978-1-7185-0044-0.*
- *Blandy, Jim; Orendorff, Jason; Tindall, Leonora F. S. (2021). Programming Rust: Fast, Safe Systems Development (2nd ed.). O'Reilly Media. ISBN 978-1-4920-5254-8. OCLC 1289839504.*
- *McNamara, Tim (2021). Rust in Action. Manning Publications. ISBN 978-1-6172-9455-6. OCLC 1153044639.*
- *Klabnik, Steve; Nichols, Carol (2023). The Rust programming language (2nd ed.). No Starch Press. ISBN 978-1-7185-0310-6. OCLC 1363816350.*

1. ^ *"Announcing Rust 1.91.1". 2025-11-10. Retrieved 2025-11-11.*
2. ^ *a b "Platform Support". The rustc book. Archived from the original on 2022-06-30. Retrieved 2022-06-27.*
3. ^ *"Copyright". GitHub. The Rust Programming Language. 2022-10-19. Archived from the original on 2023-07-22. Retrieved 2022-10-19.*
4. ^ *"Licenses". The Rust Programming Language. Archived from the original on 2025-02-23. Retrieved 2025-03-07.*
5. ^ *Strom, Robert E. (1983). "Mechanisms for compile-time enforcement of security". Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages - POPL '83. pp. 276–284. doi:10.1145/567067.567093. ISBN 0897910907. S2CID 6630704.*
6. ^ *Strom, Robert E.; Yemini, Shaula (1986). "Typestate: A programming language concept for enhancing software reliability" (PDF). IEEE Transactions on Software Engineering.* ***12****. IEEE: 157–171. doi:10.1109/tse.1986.6312929. S2CID 15575346.*
7. ^ *"Uniqueness Types". Rust Blog. Archived from the original on 2016-09-15. Retrieved 2016-10-08. "Those of you familiar with the Elm style may recognize that the updated --explain messages draw heavy inspiration from the Elm approach."*
8. ^ *"Influences". The Rust Reference. Archived from the original on 2023-11-26. Retrieved 2023-12-31.*
9. ^ *"Uniqueness Types". Idris 1.3.3 documentation. Archived from the original on 2018-11-21. Retrieved 2022-07-14. "They are inspired by ... ownership types and borrowed pointers in the Rust programming language."*

10. ^ Tung, Liam. "Microsoft opens up Rust-inspired Project Verona programming language on GitHub". ZDNET. Archived from the original on 2020-01-17. Retrieved 2020-01-17.
11. ^ Jaloyan, Georges-Axel (2017-10-19). "Safe Pointers in SPARK 2014". arXiv:1710.07047 [cs.PL].
12. ^ Lattner, Chris. "Chris Lattner's Homepage". Nondot. Archived from the original on 2018-12-25. Retrieved 2019-05-14.
13. ^ "V documentation (Introduction)". GitHub. The V Programming Language. Retrieved 2023-11-04.
14. ^ Yegulalp, Serdar (2016-08-29). "New challenger joins Rust to topple C language". InfoWorld. Archived from the original on 2021-11-25. Retrieved 2022-10-19.
15. ^ a b c d e f g h i j k l m n o p q r s Thompson, Clive (2023-02-14). "How Rust went from a side project to the world's most-loved programming language". MIT Technology Review. Archived from the original on 2024-09-19. Retrieved 2023-02-23.
16. ^ a b c d e f g h i j k l m n o p q r s t u Klabnik, Steve (2016-06-02). "The History of Rust". Applicative 2016. New York, NY, USA: Association for Computing Machinery. p. 80. doi:10.1145/2959689.2960081. ISBN 978-1-4503-4464-7.
17. ^ a b c d Hoare, Graydon (July 2010). Project Servo: Technology from the past come to save the future from itself (PDF). Mozilla Annual Summit. Archived from the original (PDF) on 2021-12-26. Retrieved 2024-10-29.
18. ^ Hoare, Graydon (November 2016). "Rust Prehistory (Archive of the original Rust OCaml compiler source code)". GitHub. Retrieved 2024-10-29.
19. ^ "0.1 first supported public release Milestone · rust-lang/rust". GitHub. Retrieved 2024-10-29.
20. ^ Nelson, Jynn (2022-08-05). RustConf 2022 - Bootstrapping: The once and future compiler. Portland, Oregon: Rust Team. Retrieved 2024-10-29 – via YouTube.
21. ^ "Rust logo". Bugzilla. Archived from the original on 2024-02-02. Retrieved 2024-02-02.
22. ^ Anderson, Brian (2012-01-24). "[rust-dev] The Rust compiler 0.1 is unleashed". rust-dev (Mailing list). Archived from the original on 2012-01-24. Retrieved 2025-01-07.
23. ^ Anthony, Sebastian (2012-01-24). "Mozilla releases Rust 0.1, the language that will eventually usurp Firefox's C++". ExtremeTech. Retrieved 2025-01-07.
24. ^ "Purity by pcwalton · Pull Request #5412 · rust-lang/rust". GitHub. Retrieved 2024-10-29.

25. ^ *Binstock, Andrew (2014-01-07). "The Rise And Fall of Languages in 2013". Dr. Dobb's Journal. Archived from the original on 2016-08-07. Retrieved 2022-11-20.*
26. ^ *Lardinois, Frederic (2015-04-03). "Mozilla And Samsung Team Up To Develop Servo, Mozilla's Next-Gen Browser Engine For Multicore Processors". TechCrunch. Archived from the original on 2016-09-10. Retrieved 2017-06-25.*
27. ^ *"Firefox 45.0, See All New Features, Updates and Fixes". Mozilla. Archived from the original on 2016-03-17. Retrieved 2024-10-31.*
28. ^ *Lardinois, Frederic (2017-09-29). "It's time to give Firefox another chance". TechCrunch. Archived from the original on 2023-08-15. Retrieved 2023-08-15.*
29. ^ *Pereira, Rui; Couto, Marco; Ribeiro, Francisco; Rua, Rui; Cunha, Jácome; Fernandes, João Paulo; Saraiva, João (2017-10-23). "Energy efficiency across programming languages: How do energy, time, and memory relate?". Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering. SLE 2017. New York, NY, USA: Association for Computing Machinery. pp. 256–267. doi:10.1145/3136014.3136031. hdl:1822/65359. ISBN 978-1-4503-5525-4.*
30. ^ *Cimpanu, Catalin (2020-08-11). "Mozilla lays off 250 employees while it refocuses on commercial products". ZDNET. Archived from the original on 2022-03-18. Retrieved 2020-12-02.*
31. ^ *Cooper, Daniel (2020-08-11). "Mozilla lays off 250 employees due to the pandemic". Engadget. Archived from the original on 2020-12-13. Retrieved 2020-12-02.*
32. ^ *Tung, Liam (2020-08-21). "Programming language Rust: Mozilla job cuts have hit us badly but here's how we'll survive". ZDNET. Archived from the original on 2022-04-21. Retrieved 2022-04-21.*
33. ^ *"Laying the foundation for Rust's future". Rust Blog. 2020-08-18. Archived from the original on 2020-12-02. Retrieved 2020-12-02.*
34. ^ *"Hello World!". Rust Foundation. 2020-02-08. Archived from the original on 2022-04-19. Retrieved 2022-06-04.*
35. ^ *"Mozilla Welcomes the Rust Foundation". Mozilla Blog. 2021-02-09. Archived from the original on 2021-02-08. Retrieved 2021-02-09.*
36. ^ *Amadeo, Ron (2021-04-07). "Google is now writing low-level Android code in Rust". Ars Technica. Archived from the original on 2021-04-08. Retrieved 2021-04-08.*
37. ^ *Anderson, Tim (2021-11-23). "Entire Rust moderation team resigns". The Register. Archived from the original on 2022-07-14. Retrieved 2022-08-04.*
38. ^ *Levick, Ryan; Bos, Mara. "Governance Update". Inside Rust Blog. Archived from the original on 2022-10-27. Retrieved 2022-10-27.*

39. ^ a b Claburn, Thomas (2023-04-17). "Rust Foundation apologizes for trademark policy confusion". The Register. Archived from the original on 2023-05-07. Retrieved 2023-05-07.
40. ^ Gross, Grant (2024-02-27). "White House urges developers to dump C and C++". InfoWorld. Retrieved 2025-01-26.
41. ^ Warminsky, Joe (2024-02-27). "After decades of memory-related software bugs, White House calls on industry to act". The Record. Retrieved 2025-01-26.
42. ^ "Press Release: Future Software Should Be Memory Safe". The White House. 2024-02-26. Archived from the original on 2025-01-18. Retrieved 2025-01-26.
43. ^ Proven, Liam (2019-11-27). "Rebecca Rumbul named new CEO of The Rust Foundation". The Register. Archived from the original on 2022-07-14. Retrieved 2022-07-14. "Both are curly bracket languages, with C-like syntax that makes them unintimidating for C programmers."
44. ^ a b c Vigliarolo, Brandon (2021-02-10). "The Rust programming language now has its own independent foundation". TechRepublic. Archived from the original on 2023-03-20. Retrieved 2022-07-14.
45. ^ Klabnik & Nichols 2019, p. 263.
46. ^ Klabnik & Nichols 2019, pp. 5–6.
47. ^ Klabnik & Nichols 2023, p. 32.
48. ^ Klabnik & Nichols 2023, pp. 32–33.
49. ^ Klabnik & Nichols 2023, pp. 49–50.
50. ^ Klabnik & Nichols 2023, pp. 34–36.
51. ^ Klabnik & Nichols 2023, pp. 6, 47, 53.
52. ^ Klabnik & Nichols 2023, pp. 47–48.
53. ^ a b Klabnik & Nichols 2023, pp. 50–53.
54. ^ Klabnik & Nichols 2023, p. 56.
55. ^ Klabnik & Nichols 2023, pp. 57–58.
56. ^ Klabnik & Nichols 2023, pp. 54–56.
57. ^ Klabnik & Nichols 2019, pp. 104–109.
58. ^ Klabnik & Nichols 2019, pp. 24.
59. ^ Klabnik & Nichols 2019, pp. 36–38.
60. ^ "isize". doc.rust-lang.org. Retrieved 2025-09-28.
61. ^ "usize". doc.rust-lang.org. Retrieved 2025-09-28.
62. ^ Klabnik & Nichols 2023, pp. 36–38.
63. ^ Klabnik & Nichols 2023, p. 502.
64. ^ "Primitive Type char". The Rust Standard Library documentation. Retrieved 2025-09-07.
65. ^ "Glossary of Unicode Terms". Unicode Consortium. Archived from the original on 2018-09-24. Retrieved 2024-07-30.

66. ^ Klabnik & Nichols 2019, pp. 38–40.
67. ^ Klabnik & Nichols 2023, pp. 40–42.
68. ^ Klabnik & Nichols 2023, p. 42.
69. ^ Klabnik & Nichols 2019, pp. 59–61.
70. ^ *a* *b* Klabnik & Nichols 2019, pp. 63–68.
71. ^ Klabnik & Nichols 2019, pp. 74–75.
72. ^ Klabnik & Nichols 2023, pp. 71–72.
73. ^ *a* *b* *Balasubramanian, Abhiram; Baranowski, Marek S.; Burtsev, Anton; Panda, Aurojit; Rakamarić, Zvonimir; Ryzhyk, Leonid (2017-05-07). "System Programming in Rust". Proceedings of the 16th Workshop on Hot Topics in Operating Systems. HotOS '17. New York, NY, US: Association for Computing Machinery. pp. 156–161. doi:10.1145/3102980.3103006. ISBN 978-1-4503-5068-6. S2CID 24100599. Archived from the original on 2022-06-11. Retrieved 2022-06-01.*
74. ^ Klabnik & Nichols 2023, pp. 327–30.
75. ^ *"Lifetimes". Rust by Example. Archived from the original on 2024-11-16. Retrieved 2024-10-29.*
76. ^ *"Explicit annotation". Rust by Example. Retrieved 2024-10-29.*
77. ^ *a* *b* Klabnik & Nichols 2019, p. 194.
78. ^ Klabnik & Nichols 2019, pp. 75, 134.
79. ^ *Shamrell-Harrington, Nell (2022-04-15). "The Rust Borrow Checker – a Deep Dive". InfoQ. Archived from the original on 2022-06-25. Retrieved 2022-06-25.*
80. ^ Klabnik & Nichols 2019, pp. 194–195.
81. ^ Klabnik & Nichols 2023, pp. 208–12.
82. ^ Klabnik & Nichols 2023, 4.2. References and Borrowing.
83. ^ *Pearce, David (2021-04-17). "A Lightweight Formalism for Reference Lifetimes and Borrowing in Rust". ACM Transactions on Programming Languages and Systems. **43**: 1–73. doi:10.1145/3443420. Archived from the original on 2024-04-15. Retrieved 2024-12-11.*
84. ^ Klabnik & Nichols 2019, p. 83.
85. ^ Klabnik & Nichols 2019, p. 97.
86. ^ Klabnik & Nichols 2019, pp. 98–101.
87. ^ Klabnik & Nichols 2019, pp. 438–440.
88. ^ Klabnik & Nichols 2019, pp. 93.
89. ^ Gjengset 2021, pp. 213–215.
90. ^ Klabnik & Nichols 2023, pp. 108–110, 113–114, 116–117.
91. ^ *"Rust error handling is perfect actually". Bitfield Consulting. Archived from the original on 2025-08-07. Retrieved 2025-09-15.*
92. ^ Gjengset 2021, p. 155-156.

93. ^ Klabnik & Nichols 2023, pp. 421–423.
94. ^ Klabnik & Nichols 2019, pp. 418–427.
95. ^ *"Casting". Rust by Example. Retrieved 2025-04-01.*
96. ^ Klabnik & Nichols 2023, p. 378.
97. ^ *a b c* Klabnik & Nichols 2023, pp. 192–198.
98. ^ Klabnik & Nichols 2023, p. 98.
99. ^ Klabnik & Nichols 2019, pp. 171–172, 205.
100. ^ *a b* Klabnik & Nichols 2023, pp. 191–192.
101. ^ Klabnik & Nichols 2019, pp. 441–442.
102. ^ Gjengset 2021, p. 25.
103. ^ Klabnik & Nichols 2023, 18.2. Using Trait Objects That Allow for Values of Different Types.
104. ^ Klabnik & Nichols 2019, pp. 379–380.
105. ^ *"RAII". Rust by Example. Archived from the original on 2019-04-21. Retrieved 2020-11-22.*
106. ^ *"Abstraction without overhead: traits in Rust". Rust Blog. Archived from the original on 2021-09-23. Retrieved 2021-10-19.*
107. ^ *"Box, stack and heap". Rust by Example. Archived from the original on 2022-05-31. Retrieved 2022-06-13.*
108. ^ Klabnik & Nichols 2019, pp. 70–75.
109. ^ Klabnik & Nichols 2019, p. 323.
110. ^ Klabnik & Nichols 2023, pp. 420–429.
111. ^ McNamara 2021, p. 139, 376–379, 395.
112. ^ *a b Astrauskas, Vytautas; Matheja, Christoph; Poli, Federico; Müller, Peter; Summers, Alexander J. (2020-11-13). "How do programmers use unsafe rust?". Proceedings of the ACM on Programming Languages.* **4***: 1–27. doi:10.1145/3428204. hdl:20.500.11850/465785. ISSN 2475-1421.*
113. ^ *Lattuada, Andrea; Hance, Travis; Cho, Chanhee; Brun, Matthias; Subasinghe, Isitha; Zhou, Yi; Howell, Jon; Parno, Bryan; Hawblitzel, Chris (2023-04-06). "Verus: Verifying Rust Programs using Linear Ghost Types". Proceedings of the ACM on Programming Languages.* **7***: 85:286–85:315. doi:10.1145/3586037. hdl:20.500.11850/610518.*
114. ^ *Milano, Mae; Turcotti, Julia; Myers, Andrew C. (2022-06-09). "A flexible type system for fearless concurrency". Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation. New York, NY, USA: Association for Computing Machinery. pp. 458–473. doi:10.1145/3519939.3523443. ISBN 978-1-4503-9265-5.*
115. ^ *"Introduction - Learning Rust With Entirely Too Many Linked Lists". rust-unofficial.github.io. Retrieved 2025-08-06.*

116. ^ *Noble, James; Mackay, Julian; Wrigstad, Tobias (2023-10-16). "Rusty Links in Local Chains∗". Proceedings of the 24th ACM International Workshop on Formal Techniques for Java-like Programs. New York, NY, USA: Association for Computing Machinery. pp. 1–3. doi:10.1145/3611096.3611097. ISBN 979-8-4007-0784-1.*
117. ^ a b *Evans, Ana Nora; Campbell, Bradford; Soffa, Mary Lou (2020-10-01). "Is rust used safely by software developers?". Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering. New York, NY, USA: Association for Computing Machinery. pp. 246–257. arXiv:2007.00752. doi:10.1145/3377811.3380413. ISBN 978-1-4503-7121-6.*
118. ^ *"Behavior considered undefined". The Rust Reference. Retrieved 2025-08-06.*
119. ^ Klabnik & Nichols 2023, pp. 449–455.
120. ^ Gjengset 2021, pp. 101–102.
121. ^ *"Macros By Example". The Rust Reference. Archived from the original on 2023-04-21. Retrieved 2023-04-21.*
122. ^ Klabnik & Nichols 2019, pp. 446–448.
123. ^ *"Procedural Macros". The Rust Programming Language Reference. Archived from the original on 2020-11-07. Retrieved 2021-03-23.*
124. ^ Klabnik & Nichols 2019, pp. 449–455.
125. ^ *Baumgartner, Stefan (2025-05-23). "Programming language: Rust 2024 is the most comprehensive edition to date". heise online. Retrieved 2025-06-28.*
126. ^ a b c Gjengset 2021, pp. 193–209.
127. ^ *"Safe Interoperability between Rust and C++ with CXX". InfoQ. 2020-12-06. Archived from the original on 2021-01-22. Retrieved 2021-01-03.*
128. ^ *Rosenblatt, Seth (2013-04-03). "Samsung joins Mozilla's quest for Rust". CNET. Archived from the original on 2013-04-04. Retrieved 2013-04-05.*
129. ^ *Brown, Neil (2013-04-17). "A taste of Rust". LWN.net. Archived from the original on 2013-04-26. Retrieved 2013-04-25.*
130. ^ *"Races". The Rustonomicon. Archived from the original on 2017-07-10. Retrieved 2017-07-03.*
131. ^ *Vandervelden, Thibaut; De Smet, Ruben; Deac, Diana; Steenhaut, Kris; Braeken, An (2024-09-07). "Overview of Embedded Rust Operating Systems and Frameworks". Sensors.* ***24*** *(17): 5818. Bibcode:2024Senso..24.5818V. doi:10.3390/s24175818. PMC 11398098. PMID 39275729.*
132. ^ *"The Rust Language FAQ". The Rust Programming Language. 2015. Archived from the original on 2015-04-20. Retrieved 2017-04-24.*
133. ^ *"Reference Cycles Can Leak Memory - The Rust Programming Language". doc.rust-lang.org. Retrieved 2025-11-16.*
134. ^ *"Is it possible to cause a memory leak in Rust?". Stack Overflow. Retrieved 2025-11-16.*

135. ^ *Speykious (2025-11-15), Speykious/cve-rs: Blazingly 🔥 fast 🚀 memory vulnerabilities, written in 100% safe Rust. 🦀, retrieved 2025-11-16*
136. ^ Blandy, Orendorff & Tindall 2021, pp. 6–8.
137. ^ *"Overview of the compiler". Rust Compiler Development Guide. Rust project contributors. Archived from the original on 2023-05-31. Retrieved 2024-11-07.*
138. ^ *"Code Generation". Rust Compiler Development Guide. Rust project contributors. Retrieved 2024-03-03.*
139. ^ *"rust-lang/rustc_codegen_gcc". GitHub. The Rust Programming Language. 2024-03-02. Retrieved 2024-03-03.*
140. ^ *"rust-lang/rustc_codegen_cranelift". GitHub. The Rust Programming Language. 2024-03-02. Retrieved 2024-03-03.*
141. ^ a b c *Perkel, Jeffrey M. (2020-12-01). "Why scientists are turning to Rust". Nature. **588** (7836): 185–186. Bibcode:2020Natur.588..185P. doi:10.1038/d41586-020-03382-2. PMID 33262490. S2CID 227251258. Archived from the original on 2022-05-06. Retrieved 2022-05-15.*
142. ^ *Simone, Sergio De (2019-04-18). "Rust 1.34 Introduces Alternative Registries for Non-Public Crates". InfoQ. Archived from the original on 2022-07-14. Retrieved 2022-07-14.*
143. ^ Klabnik & Nichols 2019, pp. 511–512.
144. ^ *"Clippy". GitHub. The Rust Programming Language. 2023-11-30. Archived from the original on 2021-05-23. Retrieved 2025-10-09.*
145. ^ *"Clippy Lints". The Rust Programming Language. Retrieved 2023-11-30.*
146. ^ Klabnik & Nichols 2019, Appendix G – How Rust is Made and "Nightly Rust"
147. ^ Blandy, Orendorff & Tindall 2021, pp. 176–177.
148. ^ Klabnik & Nichols 2023, p. 623.
149. ^ *Anderson, Tim (2021-11-30). "Can Rust save the planet? Why, and why not". The Register. Archived from the original on 2022-07-11. Retrieved 2022-07-11.*
150. ^ *Yegulalp, Serdar (2021-10-06). "What is the Rust language? Safe, fast, and easy software development". InfoWorld. Archived from the original on 2022-06-24. Retrieved 2022-06-25.*
151. ^ McNamara 2021, p. 11.
152. ^ a b *Popescu, Natalie; Xu, Ziyang; Apostolakis, Sotiris; August, David I.; Levy, Amit (2021-10-15). "Safer at any speed: automatic context-aware safety enhancement for Rust". Proceedings of the ACM on Programming Languages. **5** (OOPSLA). Section 2. doi:10.1145/3485480. S2CID 238212612. p. 5: "We observe a large variance in the overheads of checked indexing: 23.6% of benchmarks do report significant performance hits from checked indexing, but 64.5% report little-to-no impact and, surprisingly, 11.8% report improved*

performance ... Ultimately, while unchecked indexing can improve performance, most of the time it does not."

153. ^ McNamara 2021, p. 19, 27.
154. ^ *Couprie, Geoffroy (2015). "Nom, A Byte oriented, streaming, Zero copy, Parser Combinators Library in Rust". 2015 IEEE Security and Privacy Workshops. pp. 142–148. doi:10.1109/SPW.2015.31. ISBN 978-1-4799-9933-0. S2CID 16608844.*
155. ^ McNamara 2021, p. 20.
156. ^ *"Code generation". The Rust Reference. Archived from the original on 2022-10-09. Retrieved 2022-10-09.*
157. ^ *"How Fast Is Rust?". The Rust Programming Language FAQ. Archived from the original on 2020-10-28. Retrieved 2019-04-11.*
158. ^ *Farshin, Alireza; Barbette, Tom; Roozbeh, Amir; Maguire, Gerald Q. Jr; Kostić, Dejan (2021). "PacketMill: Toward per-Core 100-GBPS networking". Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 1–17. doi:10.1145/3445814.3446724. ISBN 9781450383172. S2CID 231949599. Archived from the original on 2022-07-12. Retrieved 2022-07-12. "... While some compilers (e.g., Rust) support structure reordering [82], C & C++ compilers are forbidden to reorder data structures (e.g., struct or class) [74] ..."*
159. ^ Gjengset 2021, p. 22.
160. ^ *Shankland, Stephen (2016-07-12). "Firefox will get overhaul in bid to get you interested again". CNET. Archived from the original on 2022-07-14. Retrieved 2022-07-14.*
161. ^ *Security Research Team (2013-10-04). "ZeroMQ: Helping us Block Malicious Domains in Real Time". Cisco Umbrella. Archived from the original on 2023-05-13. Retrieved 2023-05-13.*
162. ^ *Cimpanu, Catalin (2019-10-15). "AWS to sponsor Rust project". ZDNET. Retrieved 2024-07-17.*
163. ^ *Nichols, Shaun (2018-06-27). "Microsoft's next trick? Kicking things out of the cloud to Azure IoT Edge". The Register. Archived from the original on 2019-09-27. Retrieved 2019-09-27.*
164. ^ *Tung, Liam (2020-04-30). "Microsoft: Why we used programming language Rust over Go for WebAssembly on Kubernetes app". ZDNET. Archived from the original on 2022-04-21. Retrieved 2022-04-21.*
165. ^ *Claburn, Thomas (2022-09-20). "In Rust We Trust: Microsoft Azure CTO shuns C and C++". The Register. Retrieved 2024-07-07.*
166. ^ *Simone, Sergio De (2019-03-10). "NPM Adopted Rust to Remove Performance Bottlenecks". InfoQ. Archived from the original on 2023-11-19. Retrieved 2023-11-20.*

167. ^ *Lyu, Shing (2020). "Welcome to the World of Rust". In Lyu, Shing (ed.). Practical Rust Projects: Building Game, Physical Computing, and Machine Learning Applications. Berkeley, CA: Apress. pp. 1–8. doi:10.1007/978-1-4842-5599-5_1. ISBN 978-1-4842-5599-5. Retrieved 2023-11-29.*
168. ^ *Lyu, Shing (2021). "Rust in the Web World". In Lyu, Shing (ed.). Practical Rust Web Projects: Building Cloud and Web-Based Applications. Berkeley, CA: Apress. pp. 1–7. doi:10.1007/978-1-4842-6589-5_1. ISBN 978-1-4842-6589-5. Retrieved 2023-11-29.*
169. ^ a b c *Li, Hongyu; Guo, Liwei; Yang, Yexuan; Wang, Shangguang; Xu, Mengwei (2024-06-30). "An Empirical Study of Rust-for-Linux: The Success, Dissatisfaction, and Compromise". USENIX. Retrieved 2024-11-28.*
170. ^ *Corbet, Jonathan (2022-10-13). "A first look at Rust in the 6.1 kernel". LWN.net. Archived from the original on 2023-11-17. Retrieved 2023-11-11.*
171. ^ *Vaughan-Nichols, Steven (2021-12-07). "Rust takes a major step forward as Linux's second official language". ZDNET. Retrieved 2024-11-27.*
172. ^ *Corbet, Jonathan (2022-11-17). "Rust in the 6.2 kernel". LWN.net. Retrieved 2024-11-28.*
173. ^ *Corbet, Jonathan (2024-09-24). "Committing to Rust in the kernel". LWN.net. Retrieved 2024-11-28.*
174. ^ *Amadeo, Ron (2021-04-07). "Google is now writing low-level Android code in Rust". Ars Technica. Archived from the original on 2021-04-08. Retrieved 2022-04-21.*
175. ^ *Darkcrizt (2021-04-02). "Google Develops New Bluetooth Stack for Android, Written in Rust". Desde Linux. Archived from the original on 2021-08-25. Retrieved 2024-08-31.*
176. ^ *Claburn, Thomas (2023-04-27). "Microsoft is rewriting core Windows libraries in Rust". The Register. Archived from the original on 2023-05-13. Retrieved 2023-05-13.*
177. ^ *Proven, Liam (2023-12-01). "Small but mighty, 9Front's 'Humanbiologics' is here for the truly curious". The Register. Retrieved 2024-03-07.*
178. ^ *Yegulalp, Serdar (2016-03-21). "Rust's Redox OS could show Linux a few new tricks". InfoWorld. Archived from the original on 2016-03-21. Retrieved 2016-03-21.*
179. ^ *Anderson, Tim (2021-01-14). "Another Rust-y OS: Theseus joins Redox in pursuit of safer, more resilient systems". The Register. Archived from the original on 2022-07-14. Retrieved 2022-07-14.*
180. ^ *Boos, Kevin; Liyanage, Namitha; Ijaz, Ramla; Zhong, Lin (2020). Theseus: an Experiment in Operating System Structure and State Management. pp. 1–19.*

ISBN 978-1-939133-19-9. Archived from the original on 2023-05-13. Retrieved 2023-05-13.

181. ^ Zhang, HanDong (2023-01-31). "2022 Review | The adoption of Rust in Business". Rust Magazine. Retrieved 2023-02-07.
182. ^ Sei, Mark (2018-10-10). "Fedora 29 new features: Startis now officially in Fedora". Marksei, Weekly sysadmin pills. Archived from the original on 2019-04-13. Retrieved 2019-05-13.
183. ^ Proven, Liam (2022-07-12). "Oracle Linux 9 released, with some interesting additions". The Register. Archived from the original on 2022-07-14. Retrieved 2022-07-14.
184. ^ Proven, Liam (2023-02-02). "System76 teases features coming in homegrown Rust-based desktop COSMIC". The Register. Archived from the original on 2024-07-17. Retrieved 2024-07-17.
185. ^ Hu, Vivian (2020-06-12). "Deno Is Ready for Production". InfoQ. Archived from the original on 2020-07-01. Retrieved 2022-07-14.
186. ^ Abrams, Lawrence (2021-02-06). "This Flash Player emulator lets you securely play your old games". Bleeping Computer. Archived from the original on 2021-12-25. Retrieved 2021-12-25.
187. ^ Kharif, Olga (2020-10-17). "Ethereum Blockchain Killer Goes By Unassuming Name of Polkadot". Bloomberg News. Bloomberg L.P. Archived from the original on 2020-10-17. Retrieved 2021-07-14.
188. ^ Keizer, Gregg (2016-10-31). "Mozilla plans to rejuvenate Firefox in 2017". Computerworld. Archived from the original on 2023-05-13. Retrieved 2023-05-13.
189. ^ Claburn, Thomas (2023-01-12). "Google polishes Chromium code with a layer of Rust". The Register. Retrieved 2024-07-17.
190. ^ Jansens, Dana (2023-01-12). "Supporting the Use of Rust in the Chromium Project". Google Online Security Blog. Archived from the original on 2023-01-13. Retrieved 2023-11-12.
191. ^ a b c "2025 Stack Overflow Developer Survey – Technology". Stack Overflow. Retrieved 2025-08-09.
192. ^ Claburn, Thomas (2022-06-23). "Linus Torvalds says Rust is coming to the Linux kernel". The Register. Archived from the original on 2022-07-28. Retrieved 2022-07-15.
193. ^ Jung, Ralf; Jourdan, Jacques-Henri; Krebbers, Robbert; Dreyer, Derek (2017-12-27). "RustBelt: securing the foundations of the Rust programming language". Proceedings of the ACM on Programming Languages. **2** (POPL): 1–34. doi:10.1145/3158154. hdl:21.11116/0000-0003-34C6-3. ISSN 2475-1421.

194. ^ *Popescu, Natalie; Xu, Ziyang; Apostolakis, Sotiris; August, David I.; Levy, Amit (2021-10-20). "Safer at any speed: automatic context-aware safety enhancement for Rust". Proceedings of the ACM on Programming Languages.* **5** *(OOPSLA): 1–23. doi:10.1145/3485480. ISSN 2475-1421.*
195. ^ *Blanco-Cuaresma, Sergi; Bolmont, Emeline (2017-05-30). "What can the programming language Rust do for astrophysics?". Proceedings of the International Astronomical Union.* **12** *(S325): 341–344. arXiv:1702.02951. Bibcode:2017IAUS..325..341B. doi:10.1017/S1743921316013168. ISSN 1743-9213. S2CID 7857871. Archived from the original on 2022-06-25. Retrieved 2022-06-25.*
196. ^ *Wallach, Dan. "TRACTOR: Translating All C to Rust". DARPA. Retrieved 2025-08-03.*
197. ^ Klabnik & Nichols 2019, p. 4.
198. ^ *"Getting Started". The Rust Programming Language. Archived from the original on 2020-11-01. Retrieved 2020-10-11.*
199. ^ *The Rust Survey Team (2025-02-13). "2024 State of Rust Survey Results". The Rust Programming Language. Retrieved 2025-09-07.*
200. ^ *Tung, Liam (2021-02-08). "The Rust programming language just took a huge step forwards". ZDNET. Archived from the original on 2022-07-14. Retrieved 2022-07-14.*
201. ^ *Krill, Paul (2021-02-09). "Rust language moves to independent foundation". InfoWorld. Archived from the original on 2021-04-10. Retrieved 2021-04-10.*
202. ^ *Vaughan-Nichols, Steven J. (2021-04-09). "AWS's Shane Miller to head the newly created Rust Foundation". ZDNET. Archived from the original on 2021-04-10. Retrieved 2021-04-10.*
203. ^ *Vaughan-Nichols, Steven J. (2021-11-17). "Rust Foundation appoints Rebecca Rumbul as executive director". ZDNET. Archived from the original on 2021-11-18. Retrieved 2021-11-18.*
204. ^ *"Governance". The Rust Programming Language. Archived from the original on 2022-05-10. Retrieved 2024-07-18.*
205. ^ *"Introducing the Rust Leadership Council". Rust Blog. 2023-06-20. Retrieved 2024-01-12.*

- Official website

- Source code on GitHub
- Documentation

Back to TOC

Go (programming language)

Go

Paradigm	Multi-paradigm: concurrent, imperative, functional,[1] object-oriented[2][3]
Designed by	Robert Griesemer Rob Pike Ken Thompson[4]
Developer	The Go Authors[5]
First appeared	November 10, 2009; 16 years ago
Stable release	1.25.4 / 5 November 2025; 11 days ago
Typing discipline	Inferred, static, strong,[6] structural,[7][8] nominal
Memory management	Garbage collection
Implementation language	Go, Assembly language (gc); C++ (gofrontend)
OS	DragonFly BSD, FreeBSD, Linux, macOS, NetBSD, OpenBSD,[9] Plan 9,[10] Solaris, Windows
License	3-clause BSD[5] + patent grant[11]
Filename extensions	.go
Website	go.dev

Major implementations

gc, gofrontend, gold

Influenced by

C, Oberon-2, Limbo, Active Oberon, communicating sequential processes, Pascal, Oberon, Smalltalk, Newsqueak, Modula-2, Alef, APL, BCPL, Modula, occam

Influenced

Crystal, V

Go is a high-level, general-purpose programming language that is statically-typed and compiled. It is known for the simplicity of its syntax and the efficiency of development that it enables through the inclusion of a large standard library supplying many needs for common projects.[12] It was designed at Google[13] in 2007 by Robert Griesemer, Rob Pike, and Ken Thompson, and publicly announced in November 2009.[4] It is syntactically similar to C, but also has garbage collection, structural typing,[7] and CSP-style concurrency.[14] It is often referred to as **Golang** to avoid ambiguity and because of its former domain name, `golang.org`, but its proper name is Go.[15]

There are two major implementations:

- The original, self-hosting[16] compiler toolchain, initially developed inside Google;[17]
- A frontend written in C++, called gofrontend,[18] originally a GCC frontend, providing gccgo, a GCC-based Go compiler;[19] later extended to also support LLVM, providing an LLVM-based Go compiler called gollvm.[20]

A third-party source-to-source compiler, GopherJS,[21] transpiles Go to JavaScript for front-end web development.

Go was designed at Google in 2007 to improve programming productivity in an era of multicore, networked machines and large codebases.[22] The designers wanted to address criticisms of other languages in use at Google, but keep their useful characteristics:[23]

- Static typing and run-time efficiency (like C)
- Readability and usability (like Python)[24]
- High-performance networking and multiprocessing

Its designers were primarily motivated by their shared dislike of C++.[25][26][27]

Go was publicly announced in November 2009,[28] and version 1.0 was released in March 2012.[29][30] Go is widely used in production at Google[31] and in many other organizations and open-source projects.

In retrospect the Go authors judged Go to be successful due to the overall engineering work around the language, including the runtime support for the language's concurrency feature.

> Although the design of most languages concentrates on innovations in syntax, semantics, or typing, Go is focused on the software development process itself. ... The principal unusual property of the language itself—concurrency—addressed problems that arose with the proliferation of multicore CPUs in the 2010s. But more significant was the early work that established fundamentals for packaging, dependencies, build, test, deployment, and other workaday tasks of the software development world, aspects that are not usually foremost in language design.[32]

Branding and styling

[edit]

Go's mascot is a cartoon gopher.

The gopher mascot was introduced in 2009 for the open source launch of the language. Renée French, who had designed the rabbit mascot for Plan 9, adapted the gopher from an earlier WFMU T-shirt design.[33]

In November 2016, the Go and Go Mono fonts were released by type designers Charles Bigelow and Kris Holmes specifically for use by the Go project. Go is a humanist sans-serif resembling Lucida Grande, and Go Mono is monospaced. Both fonts adhere to the WGL4 character set and were designed to be legible with a large x-height and distinct letterforms. Both Go and Go Mono adhere to the DIN 1450 standard by having a slashed zero, lowercase `l` with a tail, and an uppercase `I` with serifs.[34][35]

In April 2018, the original logo was redesigned by brand designer Adam Smith. The new logo is a modern, stylized GO slanting right with trailing streamlines. The gopher mascot remained the same.[36]

The lack of support for generic programming in initial versions of Go drew considerable criticism.[37] The designers expressed an openness to generic programming and noted that built-in functions *were* in fact type-generic, but are treated as special cases; Pike called this a weakness that might be changed at some point.[38] The Google team built at least one compiler for an experimental Go dialect with generics, but did not release it.[39]

In August 2018, the Go principal contributors published draft designs for generic programming and error handling and asked users to submit feedback.[40][41] However, the error handling proposal was eventually abandoned.[42]

In June 2020, a new draft design document[43] was published that would add the necessary syntax to Go for declaring generic functions and types. A code translation tool, *go2go*, was provided to allow users to try the new syntax, along with a generics-enabled version of the online Go Playground.[44]

Generics were finally added to Go in version 1.18 on March 15, 2022.[45]

Go 1 guarantees compatibility[46] for the language specification and major parts of the standard library. All versions up through the current Go 1.24 release[47] have maintained this promise.

Go uses a `go1.[major].[patch]` versioning format, such as `go1.24.0` and each major Go release is supported until there are two newer major releases. Unlike most software, Go calls the second number in a version the major, i.e., in `go1.24.0` the `24` is the major version.[48] This is because Go plans to never reach 2.0, prioritizing backwards compatibility over potential breaking changes.[49]

2015 lecture of Rob Pike (one of the Go creators)

Go is influenced by C (especially the Plan 9 dialect[50][*failed verification* – *see discussion*]), but with an emphasis on greater simplicity and safety. It consists of:

- A syntax and environment adopting patterns more common in dynamic languages:[51]
    - Optional concise variable declaration and initialization through type inference (`x := 0` instead of `var x int = 0;` or `var x = 0;`)
    - Fast compilation[52]

    - Remote package management (`go get`)[53] and online package documentation[54]
- Distinctive approaches to particular problems:
    - Built-in concurrency primitives: light-weight processes (goroutines), channels, and the `select` statement
    - An interface system in place of virtual inheritance, and type embedding instead of non-virtual inheritance
    - A toolchain that, by default, produces statically linked native binaries without external Go dependencies
- A desire to keep the language specification simple enough to hold in a programmer's head,[55] in part by omitting features that are common in similar languages.
- 25 reserved words

Go's syntax includes changes from C aimed at keeping code concise and readable. A combined declaration/initialization operator was introduced that allows the programmer to write `i := 3` or `s := "Hello, world!"`, without specifying the types of variables used. This contrasts with C's `int i = 3;` and `const char s[] = "Hello, world!";` (though since C23 type inference has been supported using `auto`, like C++). Go also removes the requirement to use parentheses in if statement conditions.

Semicolons still terminate statements;[a] but are implicit when the end of a line occurs.[b]

Methods may return multiple values, and returning a `result, err` pair is the conventional way a method indicates an error to its caller in Go.[c] Go adds literal syntaxes for initializing struct parameters by name and for initializing maps and slices. As an alternative to C's three-statement `for` loop, Go's `range` expressions allow concise iteration over arrays, slices, strings, maps, and channels.[58]

Go contains the following keywords, of which there are 25:[59]

- `break`
- `case`
- `chan`
- `const`
- `continue`
- `default`

- defer
- else
- fallthrough
- for
- func
- go
- goto
- if
- import
- interface
- map
- package
- range
- return
- select
- struct
- switch
- type
- var

Go has a number of built-in types, including numeric ones (byte, int64, float32, etc.), Booleans, and byte strings (string). Strings are immutable; built-in operators and keywords (rather than functions) provide concatenation, comparison, and UTF-8 encoding/decoding.[60] Record types can be defined with the struct keyword.[61]

Go contains the following primitives:[62]

- bool
- int8
- uint8
- int16
- uint16
- int32
- uint32
- int64
- uint64
- int
- uint
- uintptr

- `float32`
- `float64`
- `complex64`
- `complex128`
- `string`

Note that `byte` is an alias for `uint8` and `rune` is an alias for `int32`.

For each type `T` and each non-negative integer constant `n`, there is an array type denoted `[n]T`; arrays of differing lengths are thus of different types. Dynamic arrays are available as "slices", denoted `[]T` for some type `T` (compare to other languages like C/C++ and Java, where instead the arrays are denoted `T[]`) These have a length and a *capacity* specifying when new memory needs to be allocated to expand the array. Several slices may share their underlying memory.[38][63][64]

Pointers are available for all types, and the pointer-to-`T` type is denoted `*T` (similar to Rust; compare to other languages like C/C++ and C#, where pointers are denoted `T*`). Address-taking and indirection use the `&` and `*` operators, as in C, or happen implicitly through the method call or attribute access syntax.[65][66] There is no pointer arithmetic,[d] except via the special `unsafe.Pointer` type in the standard library.[67]

For a pair of types `K`, `V`, the type `map[K]V` is the type mapping type-`K` keys to type-`V` values, which can be thought of as equivalent to `Map<K, V>` in other languages. The Go Programming Language specification does not give any performance guarantees or implementation requirements for map types, though it is usually implemented as a hash table (equivalent to `HashMap<K, V>` in other languages). Hash tables are built into the language, with special syntax and built-in functions. `chan T` is a *channel* that allows sending values of type `T` between concurrent Go processes.[68]

Aside from its support for interfaces, Go's type system is nominal: the `type` keyword can be used to define a new *named type*, which is distinct from other named types that have the same layout (in the case of a `struct`, the same members in the same order). Some conversions between types (e.g., between the various integer types) are pre-defined and adding a new type may define additional conversions, but conversions between named types must always be invoked explicitly.[69] For example, the `type` keyword can be used to define a type for IPv4 addresses, based on 32-bit unsigned integers as follows:

```
type ipv4addr uint32
```

With this type definition, `ipv4addr(x)` interprets the `uint32` value `x` as an IP address. Simply assigning `x` to a variable of type `ipv4addr` is a type error.[70]

Constant expressions may be either typed or "untyped"; they are given a type when assigned to a typed variable if the value they represent passes a compile-time check.[71]

Function types are indicated by the `func` keyword; they take zero or more parameters and return zero or more values, all of which are typed. The parameter and return values determine a function type; thus, `func(string, int32) (int, error)` is the type of functions that take a `string` and a 32-bit signed integer, and return a signed integer (of default width) and a value of the built-in interface type `error`.[72]

Any named type has a method set associated with it. The IP address example above can be extended with a method for checking whether its value is a known standard:

```
// ZeroBroadcast reports whether addr is 255.255.255.255.
func (addr ipv4addr) ZeroBroadcast() bool {
    return addr == 0xFFFFFFFF
}
```

Due to nominal typing, this method definition adds a method to `ipv4addr`, but not on `uint32`. While methods have special definition and call syntax, there is no distinct method type.[73]

Go provides two features that replace class inheritance.[*citation needed*]

The first is *embedding*, which can be viewed as an automated form of composition.[74]

The second are its *interfaces*, which provides runtime polymorphism.[75]:266 Interfaces are a class of types and provide a limited form of structural typing in the otherwise nominal type system of Go. An object which is of an interface type is also of another type, much like C++ objects being simultaneously of a base and derived class. The design of Go interfaces was inspired by protocols from the Smalltalk programming language.[76] Multiple sources use the term duck typing when describing Go interfaces.[77][78] Although the term duck typing is not precisely defined and therefore not wrong, it usually implies that type conformance is not statically checked. Because conformance to a Go interface is checked statically by the Go compiler (except when performing a type assertion), the Go authors prefer the term *structural typing*.[79]

The definition of an interface type lists required methods by name and type. Any object of type T for which functions exist matching all the required methods of interface type I is an object of type I as well. The definition of type T need not (and cannot) identify type I. For example, if `Shape`, `Square` and `Circle` are defined as

```
import "math"

type Shape interface {
    Area() float64
}

// Note: no "implements" declaration
type Square struct {
    side float64
}

func (sq Square) Area() float64 {
    return sq.side * sq.side
}

// No "implements" declaration here either
type Circle struct {
    radius float64
}

func (c Circle) Area() float64 {
    return math.Pi * math.Pow(c.radius, 2)
}
```

then both a `Square` and a `Circle` are implicitly a `Shape` and can be assigned to a `Shape`-typed variable.[75]:263–268 In formal language, Go's interface system provides structural rather than nominal typing. Interfaces can embed other interfaces with the effect of creating a combined interface that is satisfied by exactly the types that implement the embedded interface and any methods that the newly defined interface adds.[75]:270

The Go standard library uses interfaces to provide genericity in several places, including the input/output system that is based on the concepts of `Reader` and `Writer`.[75]:282–283

Besides calling methods via interfaces, Go allows converting interface values to other types with a run-time type check. The language constructs to do so are the *type assertion*,[80] which checks against a single potential type:

```
var shp Shape = Square{5}
square, ok := shp.(Square) // Asserts Square type on shp, should work
if ok {
	fmt.Printf("%#v\n", square)
} else {
	fmt.Println("Can't print shape as Square")
}
```

and the *type switch*,[81] which checks against multiple types:[*citation needed*]

```
func (sq Square) Diagonal() float64 { return sq.side * math.Sqrt2 }
func (c Circle) Diameter() float64 { return 2 * c.radius }

func LongestContainedLine(shp Shape) float64 {
	switch v := shp.(type) {
	case Square:
		return v.Diagonal() // Or, with type assertion, shp.
(Square).Diagonal()
	case Circle:
		return v.Diameter() // Or, with type assertion, shp.
(Circle).Diameter()
	default:
		return 0 // In practice, this should be handled with
errors
	}
}
```

The *empty interface* `interface{}` is an important base case because it can refer to an item of *any* concrete type. It is similar to the `Object` class in Java or C# or `void*` in C or `Any` in C++ and Rust and is satisfied by any type, including built-in types like `int`.[75]:284 Code using the empty interface cannot simply call methods (or built-in operators) on the referred-to object, but it can store the `interface{}` value, try to convert it to a more useful type via a type assertion or type switch, or inspect it with

Go's `reflect` package.[82] Because `interface{}` can refer to any value, it is a limited way to escape the restrictions of static typing, like `void*` in C but with additional run-time type checks.[*citation needed*]

The `interface{}` type can be used to model structured data of any arbitrary schema in Go, such as JSON or YAML data, by representing it as a `map[string]interface{}` (map of string to empty interface). This recursively describes data in the form of a dictionary with string keys and values of any type.[83]

Interface values are implemented using pointer to data and a second pointer to run-time type information.[84] Like some other types implemented using pointers in Go, interface values are `nil` if uninitialized.[85]

Generic code using parameterized types

[edit]

Since version 1.18, Go supports generic code using parameterized types.[86]

Functions and types now have the ability to be generic using type parameters. These type parameters are specified within square brackets, right after the function or type name.[87] The compiler transforms the generic function or type into non-generic by substituting *type arguments* for the type parameters provided, either explicitly by the user or type inference by the compiler.[88] This transformation process is referred to as type instantiation.[89]

Interfaces now can define a set of types (known as type set) using `|` (Union) operator, as well as a set of methods. These changes were made to support type constraints in generics code. For a generic function or type, a constraint can be thought of as the type of the type argument: a meta-type. This new `~T` syntax will be the first use of ~ as a token in Go. `~T` means the set of all types whose underlying type is `T`.[90]

```
type Number interface {
        ~int | ~float64 | ~float32 | ~int32 | ~int64
}

func Add[T Number](nums ...T) T {
        var sum T
        for _, v := range nums {
                sum += v
        }
        return sum
}

func main() {
        add := Add[int]             // Type instantiation
        println(add(1, 2, 3, 4, 5)) // 15

        res := Add(1.1, 2.2, 3.3, 4.4, 5.5) // Type Inference
        println(res)                        // +1.650000e+001
}
```

Go uses the `iota` keyword to create enumerated constants.[91][92]

```
type ByteSize int

const (
    _           = iota // ignore first value by assigning to blank
identifier; 0
    KB ByteSize = 1 << (10 * iota) // 1 << (10 * 1) == 1 << 10 ==
1024; in binary 10000000000
    MB // 1 << (10 * 2) ==
1048576;                                    in binary
100000000000000000000
    GB // 1 << (10 * 3) ==
1073741824;                                 in binary
1000000000000000000000000000000
)
```

In Go's package system, each package has a path (e.g., "`compress/bzip2`" or "`golang.org/x/net/html`") and a name (e.g., `bzip2` or `html`). By default other packages' definitions must *always* be prefixed with the other package's name. However the name used can be changed from the package name, and if imported as `_`, then no package prefix is required. Only the *capitalized* names from other packages are accessible: `io.Reader` is public but `bzip2.reader` is not.[93] The `go get` command can retrieve packages stored in a remote repository[94] and developers are encouraged to develop packages inside a base path corresponding to a source repository (such as example.com/user_name/package_name) to reduce the likelihood of name collision with future additions to the standard library or other external libraries.[95]

Concurrency: goroutines and channels

[edit]

DotGo 2015 - Matt Aimonetti - Applied concurrency in Go

The Go language has built-in facilities, as well as library support, for writing concurrent programs. The runtime is asynchronous: program execution that performs, for example, a network read will be suspended until data is available to process, allowing other parts of the program to perform other work. This is built into the runtime and does not require any changes in program code. The go runtime also automatically schedules concurrent operations (goroutines) across multiple CPUs; this can achieve parallelism for a properly written program.[96]

The primary concurrency construct is the *goroutine*, a type of green thread.[97]:280–281 A function call prefixed with the `go` keyword starts a function in a new goroutine. The language specification does not specify how goroutines should be implemented, but current implementations multiplex a Go process's goroutines onto a smaller set of operating-system threads, similar to the scheduling performed in Erlang and Haskell's GHC runtime implementation.[98]:10

While a standard library package featuring most of the classical concurrency control structures (mutex locks, etc.) is available,[98]:151–152 idiomatic concurrent programs instead prefer *channels*, which send messages between goroutines.[99] Optional buffers store messages in FIFO order[100]:43 and allow sending goroutines to proceed before their messages are received.[97]:233

Channels are typed, so that a channel of type `chan T` can only be used to transfer messages of type `T`. Special syntax is used to operate on them; `<-ch` is an expression that causes the executing goroutine to block until a value comes in over the channel `ch`, while `ch <- x` sends the value `x` (possibly blocking until another goroutine receives the value). The built-in `switch`-like `select` statement can be used to implement non-blocking communication on multiple channels; see below for an example. Go has a memory model describing how goroutines must use channels or other operations to safely share data.[101]

The existence of channels does not by itself set Go apart from actor model-style concurrent languages like Erlang, where messages are addressed directly to actors (corresponding to goroutines). In the actor model, channels are themselves actors, therefore addressing a channel just means to address an actor. The actor style can be simulated in Go by maintaining a one-to-one correspondence between goroutines and channels, but the language allows multiple goroutines to share a channel or a single goroutine to send and receive on multiple channels.[98]: 147

From these tools one can build concurrent constructs like worker pools, pipelines (in which, say, a file is decompressed and parsed as it downloads), background calls with timeout, "fan-out" parallel calls to a set of services, and others.[102] Channels have also found uses further from the usual notion of interprocess communication, like serving as a concurrency-safe list of recycled buffers,[103] implementing coroutines (which helped inspire the name *goroutine*),[104] and implementing iterators.[105]

Concurrency-related structural conventions of Go (channels and alternative channel inputs) are derived from Tony Hoare's communicating sequential processes model. Unlike previous concurrent programming languages such as Occam or Limbo (a language on which Go co-designer Rob Pike worked),[106] Go does not provide any built-in notion of safe or verifiable concurrency.[107] While the communicating-processes model is favored in Go, it is not the only one: all goroutines in a program share a single address space. This means that mutable objects and pointers can be shared between goroutines; see § Lack of data race safety, below.

Suitability for parallel programming

[edit]

Although Go's concurrency features are not aimed primarily at parallel processing,[96] they can be used to program shared-memory multi-processor machines. Various studies have been done into the effectiveness of this approach.[108] One of these studies compared the size (in lines of code) and speed of programs written by a seasoned programmer not familiar with the language and corrections to these programs by a Go expert (from Google's development team), doing the same for Chapel, Cilk and Intel TBB. The study found that the non-expert tended to write divide-and-conquer algorithms with one `go` statement per recursion, while the expert wrote distribute-work-synchronize programs using one goroutine per processor core. The expert's programs were usually faster, but also longer.[109]

Lack of data race safety

[edit]

Go's approach to concurrency can be summarized as "don't communicate by sharing memory; share memory by communicating".[110] There are no restrictions on how goroutines access shared data, making data races possible. Specifically, unless a program explicitly synchronizes via channels or other means, writes from one goroutine might be partly, entirely, or not at all visible to another, often with no guarantees about ordering of writes.[107] Furthermore, Go's *internal data structures* like interface values, slice headers, hash tables, and string headers are not immune to data races, so type and memory safety can be violated in multithreaded programs that modify shared instances of those types without synchronization.[111][112] Instead of language support, safe concurrent programming thus relies on conventions; for example, Chisnall recommends an idiom called "aliases xor mutable", meaning that passing a mutable value (or pointer) over a channel signals a transfer of ownership over the value to its receiver.[98]:155 The gc toolchain has an optional data race detector that can check for unsynchronized access to shared memory during runtime since version 1.1,[113] additionally a best-effort race detector is also included by default since version 1.6 of the gc runtime for access to the `map` data type.[114]

The linker in the gc toolchain creates statically linked binaries by default; therefore all Go binaries include the Go runtime.[115][116]

Go deliberately omits certain features common in other languages, including (implementation) inheritance, assertions,[e] pointer arithmetic,[d] implicit type conversions, untagged unions,[f] and tagged unions.[g] The designers added only those facilities that all three agreed on.[119]

Of the omitted language features, the designers explicitly argue against assertions and pointer arithmetic, while defending the choice to omit type inheritance as giving a more useful language, encouraging instead the use of interfaces to achieve dynamic dispatch[h] and composition to reuse code. Composition and delegation are in fact largely automated by `struct` embedding; according to researchers Schmager *et al.*, this feature "has many of the drawbacks of inheritance: it affects the public interface of objects, it is not fine-grained (i.e, no method-level control over embedding), methods of embedded objects cannot be hidden, and it is static", making it "not obvious" whether programmers will overuse it to the extent that programmers in other languages are reputed to overuse inheritance.[74]

Exception handling was initially omitted in Go due to lack of a "design that gives value proportionate to the complexity".[120] An exception-like `panic`/`recover` mechanism that avoids the usual `try-catch` control structure was proposed[121] and released in the March 30, 2010 snapshot.[122] The Go authors advise using it for unrecoverable errors such as those that should halt an entire program or server request, or as a shortcut to propagate errors up the stack within a package.[123][124] Across package boundaries, Go includes a canonical error type, and multi-value returns using this type are the standard idiom.[4]

The Go authors put substantial effort into influencing the style of Go programs:

- Indentation, spacing, and other surface-level details of code are automatically standardized by the `gofmt` tool. It uses tabs for indentation and blanks for alignment. Alignment assumes that an editor is using a fixed-width font.[125] `golint` does additional style checks automatically, but has been deprecated and archived by the Go maintainers.[126]
- Tools and libraries distributed with Go suggest standard approaches to things like API documentation (`godoc`),[127] testing (`go test`), building (`go build`), package management (`go get`), and so on.
- Go enforces rules that are recommendations in other languages, for example banning cyclic dependencies, unused variables[128] or imports,[129] and implicit type conversions.
- The *omission* of certain features (for example, functional-programming shortcuts like `map` and Java-style `try`/`finally` blocks) tends to encourage a particular explicit, concrete, and imperative programming style.
- On day one the Go team published a collection of Go idioms,[127] and later also collected code review comments,[130] talks,[131] and official blog posts[132] to teach Go style and coding philosophy.

The main Go distribution includes tools for building, testing, and analyzing code:

- `go build`, which builds Go binaries using only information in the source files themselves, no separate makefiles
- `go test`, for unit testing and microbenchmarks as well as fuzzing

- `go fmt`, for formatting code
- `go install`, for retrieving and installing remote packages
- `go vet`, a static analyzer looking for potential errors in code
- `go run`, a shortcut for building and executing code
- `go doc`, for displaying documentation
- `go generate`, a standard way to invoke code generators
- `go mod`, for creating a new module, adding dependencies, upgrading dependencies, etc.
- `go tool`, for invoking developer tools (added in Go version 1.24)

It also includes profiling and debugging support, fuzzing capabilities to detect bugs, runtime instrumentation (for example, to track garbage collection pauses), and a data race detector.

Another tool maintained by the Go team but is not included in Go distributions is `gopls`, a language server that provides IDE features such as intelligent code completion to Language Server Protocol compatible editors.[133]

An ecosystem of third-party tools adds to the standard distribution, such as `gocode`, which enables code autocompletion in many text editors, `goimports`, which automatically adds/removes package imports as needed, and `errcheck`, which detects code that might unintentionally ignore errors.

```
package main

import "fmt"

func main() {
        fmt.Println("hello world")
}
```

where "fmt" is the package for *formatted I/O*, similar to C's <stdio.h> or C++ <print>.[134]

The following simple program demonstrates Go's concurrency features to implement an asynchronous program. It launches two lightweight threads ("goroutines"): one waits for the user to type some text, while the other implements a timeout. The select statement waits for either of these goroutines to send a message to the main routine, and acts on the first message to arrive (example adapted from David Chisnall's book).[98]: 152

```
package main

import (
    "fmt"
    "time"
)

func readword(ch chan string) {
    fmt.Println("Type a word, then hit Enter.")
    var word string
    fmt.Scanf("%s", &word)
    ch <- word
}

func timeout(t chan bool) {
    time.Sleep(5 * time.Second)
    t <- false
}

func main() {
    t := make(chan bool)
    go timeout(t)

    ch := make(chan string)
    go readword(ch)

    select {
    case word := <-ch:
        fmt.Println("Received", word)
    case <-t:
        fmt.Println("Timeout.")
    }
}
```

The testing package provides support for automated testing of go packages.[135] Target function example:

```
func ExtractUsername(email string) string {
        at := strings.Index(email, "@")
        return email[:at]
}
```

Test code (note that **assert** keyword is missing in Go; tests live in <filename>_test.go at the same package):

```
import (
    "testing"
)

func TestExtractUsername(t *testing.T) {
        t.Run("withoutDot", func(t *testing.T) {
                username := ExtractUsername("r@google.com")
                if username != "r" {
                        t.Fatalf("Got: %v\n", username)
                }
        })

        t.Run("withDot", func(t *testing.T) {
                username := ExtractUsername("jonh.smith@example.com")
                if username != "jonh.smith" {
                        t.Fatalf("Got: %v\n", username)
                }
        })
}
```

It is possible to run tests in parallel.

The net/http[136] package provides support for creating web applications.

This example would show "Hello world!" when localhost:8080 is visited.

```
package main

import (
    "fmt"
    "log"
    "net/http"
)

func helloFunc(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Hello world!")
}

func main() {
    http.HandleFunc("/", helloFunc)
    log.Fatal(http.ListenAndServe(":8080", nil))
}
```

Go has found widespread adoption in various domains due to its robust standard library and ease of use.[137]

Popular applications include:

- Caddy — a web server that automates the process of setting up HTTPS[138]
- Docker — a platform for containerization, aiming to ease the complexities of software development and deployment[139]
- Kubernetes — automates the deployment, scaling, and management of containerized applications[140]
- CockroachDB — a distributed SQL database engineered for scalability and strong consistency[141]
- Hugo — a static site generator that prioritizes speed and flexibility, allowing developers to create websites efficiently[142]

The interface system, and the deliberate omission of inheritance, were praised by Michele Simionato, who likened these characteristics to those of Standard ML, calling it "a shame that no popular language has followed [this] particular route".[143]

Dave Astels at Engine Yard wrote in 2009:[144]

> Go is extremely easy to dive into. There are a minimal number of fundamental language concepts and the syntax is clean and designed to be clear and unambiguous. Go *is* still experimental and still a little rough around the edges.

Go was named Programming Language of the Year by the TIOBE Programming Community Index in its first year, 2009, for having a larger 12-month increase in popularity (in only 2 months, after its introduction in November) than any other language that year, and reached 13th place by January 2010,[145] surpassing established languages like Pascal. By June 2015, its ranking had dropped to below 50th in the index, placing it lower than COBOL and Fortran.[146] But as of January 2017, its ranking had surged to 13th, indicating significant growth in popularity and adoption. Go was again awarded TIOBE Programming Language of the Year in 2016.[147]

Bruce Eckel has stated:[148]

> The complexity of C++ (even more complexity has been added in the new C++), and the resulting impact on productivity, is no longer justified. All the hoops that the C++ programmer had to jump through in order to use a C-compatible language make no sense anymore -- they're just a waste of time and effort. Go makes much more sense for the class of problems that C++ was originally intended to solve.

A 2011 evaluation of the language and its gc implementation in comparison to C++ (GCC), Java and Scala by a Google engineer found:

> Go offers interesting language features, which also allow for a concise and standardized notation. The compilers for this language are still immature, which reflects in both performance and binary sizes.

—R. Hundt[149]

The evaluation got a rebuttal from the Go development team. Ian Lance Taylor, who had improved the Go code for Hundt's paper, had not been aware of the intention to publish his code, and says that his version was "never intended to be an example of idiomatic or efficient Go"; Russ Cox then optimized the Go code, as well as the C++ code, and got the Go code to run almost as fast as the C++ version and more than an order of magnitude faster than the code in the paper.[150]

- Go's nil combined with the lack of algebraic types leads to difficulty handling failures and base cases.[151][152]
- Go does not allow an opening brace to appear on its own line, which forces all Go programmers to use the same brace style.[153]
- Go has been criticized for focusing on simplicity of implementation rather than correctness and flexibility; as an example, the language uses POSIX file semantics on all platforms, and therefore provides incorrect information on platforms such as Windows (which do not follow the aforementioned standard).[154][155]
- A study showed that it is as easy to make concurrency bugs with message passing as with shared memory, sometimes even more.[156]

On November 10, 2009, the day of the general release of the language, Francis McCabe, developer of the Go! programming language (note the exclamation point), requested a name change of Google's language to prevent confusion with his language, which he had spent 10 years developing.[157] McCabe raised concerns that "the 'big guy' will end up steam-rollering over" him, and this concern resonated with the more than 120 developers who commented on Google's official issues thread saying they should change the name, with some[158] even saying the issue contradicts Google's motto of: Don't be evil.[159]

On October 12, 2010, the filed public issue ticket was closed by Google developer Russ Cox (@rsc) with the custom label "Unfortunate" accompanied by the following comment:

> "There are many computing products and services named Go. In the 11 months since our release, there has been minimal confusion of the two languages."[159]

- Fat pointer
- Fyne (software) — widget toolkit for creating GUIs with Go
- Comparison of programming languages

1. ^ But "To allow complex statements to occupy a single line, a semicolon may be omitted before a closing) or }".[56]
2. ^ "if the newline comes after a token that could end a statement, [the lexer will] insert a semicolon".[57]
3. ^ Usually, exactly one of the result and error values has a value other than the type's zero value; sometimes both do, as when a read or write can only be partially completed, and sometimes neither, as when a read returns 0 bytes. See Semipredicate problem: Multivalued return.
4. ^ *a* *b* Language FAQ "Why is there no pointer arithmetic? Safety ... never derive an illegal address that succeeds incorrectly ... using array indices can be as efficient as ... pointer arithmetic ... simplify the implementation of the garbage collector...."[4]
5. ^ Language FAQ "Why does Go not have assertions? ...our experience has been that programmers use them as a crutch to avoid thinking about proper error handling and reporting...."[4]
6. ^ Language FAQ "Why are there no untagged unions...? [they] would violate Go's memory safety guarantees."[4]
7. ^ Language FAQ "Why does Go not have variant types? ... We considered [them but] they overlap in confusing ways with interfaces.... [S]ome of what variant types address is already covered, ... although not as elegantly."[4] (The tag of an interface type[117] is accessed with a type assertion[118]).
8. ^ Questions "How do I get dynamic dispatch of methods?" and "Why is there no type inheritance?" in the language FAQ.[4]

1. ^ *"Codewalk: First-Class Functions in Go". "Go supports first class functions, higher-order functions, user-defined function types, function literals, closures, and multiple return values. This rich feature set supports a functional programming style in a strongly typed language."*
2. ^ *"Is Go an object-oriented language?". Retrieved April 13, 2019. "Although Go has types and methods and allows an object-oriented style of programming, there is no type hierarchy."*
3. ^ *"Go: code that grows with grace". Retrieved June 24, 2018. "Go is Object Oriented, but not in the usual way."*
4. ^ a b c d e f g h *"Language Design FAQ". The Go Programming Language. January 16, 2010. Archived from the original on July 28, 2025. Retrieved February 27, 2010.*
5. ^ a b *"Text file LICENSE". The Go Programming Language. Archived from the original on July 16, 2025. Retrieved October 5, 2012.*
6. ^ *"The Go Programming Language Specification". The Go Programming Language. Archived from the original on August 13, 2025.*
7. ^ a b *"Why doesn't Go have "implements" declarations?". The Go Programming Language. Archived from the original on July 28, 2025. Retrieved October 1, 2015.*
8. ^ *Pike, Rob [@rob_pike] (December 22, 2014). "Go has structural typing, not duck typing. Full interface satisfaction is checked and required" (Tweet). Archived from the original on April 7, 2022. Retrieved March 13, 2016 – via Twitter.*
9. ^ *"lang/go: go-1.4". OpenBSD ports. December 23, 2014. Retrieved January 19, 2015.*
10. ^ *"Go Porting Efforts". Go Language Resources. cat-v. January 12, 2010. Retrieved January 18, 2010.*
11. ^ *"Additional IP Rights Grant". The Go Programming Language. Archived from the original on March 30, 2025. Retrieved October 5, 2012.*
12. ^ *"Go Introduction". www.w3schools.com. Retrieved November 23, 2024.*
13. ^ *Kincaid, Jason (November 10, 2009). "Google's Go: A New Programming Language That's Python Meets C++". TechCrunch. Retrieved January 18, 2010.*
14. ^ *Metz, Cade (May 5, 2011). "Google Go boldly goes where no code has gone before". The Register. Archived from the original on July 28, 2025.*
15. ^ *"Is the language called Go or Golang?". Retrieved March 16, 2022. "The language is called Go."*

16. ^ *"Go 1.5 Release Notes". Retrieved January 28, 2016. "The compiler and runtime are now implemented in Go and assembler, without C."*
17. ^ *"The Go programming language". GitHub. Retrieved November 1, 2024.*
18. ^ *"gofrontend". GitHub. Retrieved November 1, 2024.*
19. ^ *"gccgo". Retrieved November 1, 2024. "gccgo, the GNU compiler for the Go programming language"*
20. ^ *"Gollvm". Retrieved November 1, 2024. "Gollvm is an LLVM-based Go compiler."*
21. ^ *"A compiler from Go to JavaScript for running Go code in a browser: Gopherjs/ Gopherjs". GitHub. Archived from the original on December 12, 2023.*
22. ^ *"Go at Google: Language Design in the Service of Software Engineering". Retrieved October 8, 2018.*
23. ^ *Pike, Rob (April 28, 2010). "Another Go at Language Design". Stanford EE Computer Systems Colloquium. Stanford University.* Video available.
24. ^ *"Frequently Asked Questions (FAQ) - The Go Programming Language". The Go Programming Language. Retrieved February 26, 2016.*
25. ^ *Binstock, Andrew (May 18, 2011). "Dr. Dobb's: Interview with Ken Thompson". Dr. Dobb's. Archived from* the original *on January 5, 2013. Retrieved February 7, 2014.*
26. ^ *Pike, Rob (2012). "Less is exponentially more".*
27. ^ *Griesemer, Robert (2015). "The Evolution of Go".*
28. ^ *Griesemer, Robert; Pike, Rob; Thompson, Ken; Taylor, Ian; Cox, Russ; Kim, Jini; Langley, Adam. "Hey! Ho! Let's Go!". Google Open Source. Retrieved May 17, 2018.*
29. ^ *Shankland, Stephen (March 30, 2012). "Google's Go language turns one, wins a spot at YouTube: The lower-level programming language has matured enough to sport the 1.0 version number. And it's being used for real work at Google". News. CNet. CBS Interactive Inc. Retrieved August 6, 2017. "Google has released version 1 of its Go programming language, an ambitious attempt to improve upon giants of the lower-level programming world such as C and C++."*
30. ^ *"Release History". The Go Programming Language.*
31. ^ *"Go FAQ: Is Google using Go internally?". Retrieved March 9, 2013.*
32. ^ The Go Programming Language and Environment. Communications of the ACM. https://dl.acm.org/doi/pdf/10.1145/3488716
33. ^ *"The Go Gopher - The Go Programming Language". go.dev. Retrieved February 9, 2023.*
34. ^ *"Go fonts". Go. November 16, 2016. Retrieved March 12, 2019.*
35. ^ *"Go Font TTFs". GitHub. Retrieved April 2, 2019.*
36. ^ *"Go's New Brand". The Go Blog. Retrieved November 9, 2018.*
37. ^ *Merrick, Alice (March 9, 2021). "Go Developer Survey 2020 Results". Go Programming Language. Retrieved March 16, 2022.*

38. ^ a b *Pike, Rob (September 26, 2013). "Arrays, slices (and strings): The mechanics of 'append'". The Go Blog. Retrieved March 7, 2015.*
39. ^ *"E2E: Erik Meijer and Robert Griesemer". Channel 9. Microsoft. May 7, 2012.*
40. ^ *"Go 2 Draft Designs". Retrieved September 12, 2018.*
41. ^ *"The Go Blog: Go 2 Draft Designs". August 28, 2018.*
42. ^ *"Proposal: A built-in Go error check function, "try"". Go repository on GitHub. Retrieved March 16, 2022.*
43. ^ *"Type Parameters — Draft Design". go.googlesource.com.*
44. ^ *"Generics in Go". bitfieldconsulting.com. December 17, 2021.*
45. ^ *"Go 1.18 is released!". Go Programming Language. March 15, 2022. Retrieved March 16, 2022.*
46. ^ *"Go 1 and the Future of Go Programs". The Go Programming Language.*
47. ^ *"Go 1.24 Release Notes". The Go Programming Language.*
48. ^ *"Release History". The Go Programming Language.*
49. ^ *"Backward Compatibility, Go 1.21, and Go 2". The Go Programming Language.*
50. ^ *"A Quick Guide to Go's Assembler". go.dev. Retrieved December 31, 2021.*
51. ^ *Pike, Rob (November 10, 2009). "The Go Programming Language". YouTube. Retrieved July 1, 2011.*
52. ^ *Pike, Rob (November 10, 2009). The Go Programming Language (flv) (Tech talk). Google. Event occurs at 8:53.*
53. ^ *"Download and install packages and dependencies".* See godoc.org for addresses and documentation of some packages.
54. ^ *"GoDoc". godoc.org.*
55. ^ *Pike, Rob. "The Changelog" (Podcast). Archived from the original on October 20, 2013. Retrieved October 7, 2013.*
56. ^ *"Go Programming Language Specification, §Semicolons". The Go Programming Language.*
57. ^ *"Effective Go, §Semicolons". The Go Programming Language.*
58. ^ *"The Go Programming Language Specification". The Go Programming Language.*
59. ^ *"Keywords and Identifiers in Go". go101.org. Retrieved October 10, 2025.*
60. ^ *Pike, Rob (October 23, 2013). "Strings, bytes, runes and characters in Go".*
61. ^ *Doxsey, Caleb. "Structs and Interfaces — An Introduction to Programming in Go". www.golang-book.com. Retrieved October 15, 2018.*
62. ^ *"Go Type System Overview". go101.org. Retrieved October 10, 2025.*
63. ^ *Gerrand, Andrew. "Go Slices: usage and internals".*
64. ^ *The Go Authors. "Effective Go: Slices".*
65. ^ *The Go authors. "Selectors".*
66. ^ *The Go authors. "Calls".*

67. ^ "Go Programming Language Specification, §Package unsafe". The Go Programming Language.
68. ^ "The Go Programming Language Specification". go.dev. Retrieved December 31, 2021.
69. ^ "The Go Programming Language Specification". The Go Programming Language.
70. ^ "A tour of go". go.dev.
71. ^ "The Go Programming Language Specification". The Go Programming Language.
72. ^ "The Go Programming Language Specification". go.dev. Retrieved December 31, 2021.
73. ^ "The Go Programming Language Specification". The Go Programming Language.
74. ^ a b Schmager, Frank; Cameron, Nicholas; Noble, James (2010). GoHotDraw: evaluating the Go programming language with design patterns. Evaluation and Usability of Programming Languages and Tools. ACM.
75. ^ a b c d e Balbaert, Ivo (2012). The Way to Go: A Thorough Introduction to the Go Programming Language. iUniverse.
76. ^ "The Evolution of Go". talks.golang.org. Retrieved March 13, 2016.
77. ^ Diggins, Christopher (November 24, 2009). "Duck Typing and the Go Programming Language". Dr. Dobb's, The world of software development. Retrieved March 10, 2016.
78. ^ Ryer, Mat (December 1, 2015). "Duck typing in Go". Retrieved March 10, 2016.
79. ^ "Frequently Asked Questions (FAQ) - The Go Programming Language". The Go Programming Language.
80. ^ "The Go Programming Language Specification". The Go Programming Language.
81. ^ "The Go Programming Language Specification". The Go Programming Language.
82. ^ "reflect package". pkg.go.dev.
83. ^ "map[string]interface{} in Go". bitfieldconsulting.com. June 6, 2020.
84. ^ "Go Data Structures: Interfaces". Retrieved November 15, 2012.
85. ^ "The Go Programming Language Specification". The Go Programming Language.
86. ^ "Go 1.18 Release Notes: Generics". Go Programming Language. March 15, 2022. Retrieved March 16, 2022.
87. ^ "Type Parameters Proposal". go.googlesource.com. Retrieved June 25, 2023.
88. ^ "The Go Programming Language Specification - The Go Programming Language". go.dev. Retrieved June 25, 2023.

89. ^ "An Introduction To Generics - The Go Programming Language". go.dev. Retrieved June 25, 2023.
90. ^ "Type Parameters Proposal". go.googlesource.com. Retrieved June 25, 2023.
91. ^ "Effective Go". golang.org. The Go Authors. Retrieved May 13, 2014.
92. ^ "Go Wiki: Iota - The Go Programming Language". go.dev. Retrieved May 15, 2025. "Go's iota identifier is used in const declarations to simplify definitions of incrementing numbers. Because it can be used in expressions, it provides a generality beyond that of simple enumerations"
93. ^ "A Tutorial for the Go Programming Language". The Go Programming Language. Retrieved March 10, 2013. "In Go the rule about visibility of information is simple: if a name (of a top-level type, function, method, constant or variable, or of a structure field or method) is capitalized, users of the package may see it. Otherwise, the name and hence the thing being named is visible only inside the package in which it is declared."
94. ^ "go". The Go Programming Language.
95. ^ "How to Write Go Code". The Go Programming Language. "The packages from the standard library are given short import paths such as "fmt" and "net/http". For your own packages, you must choose a base path that is unlikely to collide with future additions to the standard library or other external libraries. If you keep your code in a source repository somewhere, then you should use the root of that source repository as your base path. For instance, if you have an Example account at example.com/user, that should be your base path"
96. ^ a b Pike, Rob (September 18, 2012). "Concurrency is not Parallelism".
97. ^ a b Donovan, Alan A. A.; Kernighan, Brian W. (2016). The Go programming language. Addison-Wesley professional computing series. New York, Munich: Addison-Wesley. ISBN 978-0-13-419044-0.
98. ^ a b c d e Chisnall, David (2012). The Go Programming Language Phrasebook. Addison-Wesley. ISBN 9780132919005.
99. ^ "Effective Go". The Go Programming Language.
100. ^ Summerfield, Mark (2012). Programming in Go: Creating Applications for the 21st Century. Addison-Wesley.
101. ^ "The Go Memory Model". Retrieved April 10, 2017.
102. ^ "Go Concurrency Patterns". The Go Programming Language.
103. ^ Graham-Cumming, John (August 24, 2013). "Recycling Memory Buffers in Go". The Cloudflare Blog.
104. ^ "tree.go".
105. ^ Cheslack-Postava, Ewen. "Iterators in Go".
106. ^ Kernighan, Brian W. "A Descent Into Limbo".
107. ^ a b "The Go Memory Model". Retrieved January 5, 2011.

108. ^ *Tang, Peiyi (2010). Multi-core parallel programming in Go (PDF). Proc. First International Conference on Advanced Computing and Communications. Archived from the original (PDF) on September 9, 2016. Retrieved May 14, 2015.*
109. ^ *Nanz, Sebastian; West, Scott; Soares Da Silveira, Kaue. Examining the expert gap in parallel programming (PDF). Euro-Par 2013. CiteSeerX 10.1.1.368.6137.*
110. ^ *Go Authors. "Share Memory By Communicating".*
111. ^ *Cox, Russ. "Off to the Races".*
112. ^ *Pike, Rob (October 25, 2012). "Go at Google: Language Design in the Service of Software Engineering". Google, Inc.* "There is one important caveat: Go is not purely memory safe in the presence of concurrency."
113. ^ *"Introducing the Go Race Detector". The Go Blog. Retrieved June 26, 2013.*
114. ^ *"Go 1.6 Release Notes - The Go Programming Language". go.dev. Retrieved November 17, 2023.*
115. ^ *"Frequently Asked Questions (FAQ) - the Go Programming Language".*
116. ^ *"A Story of a Fat Go Binary". September 21, 2018.*
117. ^ *"Go Programming Language Specification, §Interface types". The Go Programming Language.*
118. ^ *"Go Programming Language Specification, §Type assertions". The Go Programming Language.*
119. ^ *"All Systems Are Go". informIT (Interview). August 17, 2010. Retrieved June 21, 2018.*
120. ^ *"Language Design FAQ". November 13, 2009. Archived from the original on November 13, 2009.*
121. ^ *"Proposal for an exception-like mechanism". golang-nuts. March 25, 2010. Retrieved March 25, 2010.*
122. ^ *"Weekly Snapshot History". The Go Programming Language.*
123. ^ *"Panic And Recover". Go wiki.*
124. ^ *"Effective Go". The Go Programming Language.*
125. ^ *"gofmt". The Go Programming Language. Retrieved February 5, 2021.*
126. ^ *"golang/lint public archive". github.com. November 30, 2022.*
127. ^ ***a*** ***b*** *"Effective Go". The Go Programming Language.*
128. ^ *"Unused local variables". yourbasic.org. Retrieved February 11, 2021.*
129. ^ *"Unused package imports". yourbasic.org. Retrieved February 11, 2021.*
130. ^ *"Code Review Comments". GitHub. Retrieved July 3, 2018.*
131. ^ *"Talks". Retrieved July 3, 2018.*
132. ^ *"Errors Are Values". Retrieved July 3, 2018.*
133. ^ *"tools/gopls/README.md at master · golang/tools". GitHub. Retrieved November 17, 2023.*
134. ^ *"fmt". The Go Programming Language. Retrieved April 8, 2019.*

135. ^ "testing". The Go Programming Language. Retrieved December 27, 2020.
136. ^ "http package - net/http - Go Packages". pkg.go.dev. Retrieved November 23, 2024.
137. ^ Lee, Wei-Meng (November 24, 2022). "Introduction to the Go Programming Language". Component Developer Magazine. Archived from the original on June 5, 2023. Retrieved September 8, 2023.
138. ^ Hoffmann, Frank; Neumeyer, Mandy (August 2018). "Simply Secure". Linux Magazine. No. 213. Archived from the original on May 28, 2023. Retrieved September 8, 2023.
139. ^ Lee, Wei-Meng (August 31, 2022). "Introduction to Containerization Using Docker". CODE Magazine. Archived from the original on May 30, 2023. Retrieved September 8, 2023.
140. ^ Pirker, Alexander (February 24, 2023). "Kubernetes Security for Starters". CODE Magazine. Archived from the original on April 1, 2023. Retrieved September 8, 2023.
141. ^ Taft, Rebecca; Sharif, Irfan; Matei, Andrei; Van Benschoten, Nathan; Lewis, Jordan; Grieger, Tobias; Niemi, Kai; Woods, Andy; Birzin, Anne; Poss, Raphael; Bardea, Paul; Ranade, Amruta; Darnell, Ben; Gruneir, Bram; Jaffray, Justin; Zhang, Lucy; Mattis, Peter (June 11, 2020). "CockroachDB: The Resilient Geo-Distributed SQL Database". Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data. SIGMOD '20. pp. 1493–1509. doi:10.1145/3318464.3386134. ISBN 978-1-4503-6735-6.
142. ^ Hopkins, Brandon (September 13, 2022). "Static Site Generation with Hugo". Linux Journal. Archived from the original on April 8, 2023. Retrieved September 8, 2023.
143. ^ Simionato, Michele (November 15, 2009). "Interfaces vs Inheritance (or, watch out for Go!)". artima. Retrieved November 15, 2009.
144. ^ Astels, Dave (November 9, 2009). "Ready, Set, Go!". engineyard. Archived from the original on October 19, 2018. Retrieved November 9, 2009.
145. ^ jt (January 11, 2010). "Google's Go Wins Programming Language Of The Year Award". jaxenter. Retrieved December 5, 2012.
146. ^ "TIOBE Programming Community Index for June 2015". TIOBE Software. June 2015. Retrieved July 5, 2015.
147. ^ "TIOBE Index". TIOBE. Retrieved July 15, 2024.
148. ^ Eckel, Bruce (August 27, 2011). "Calling Go from Python via JSON-RPC". Retrieved August 29, 2011.
149. ^ Hundt, Robert (2011). Loop recognition in C++/Java/Go/Scala (PDF). Scala Days.
150. ^ Metz, Cade (July 1, 2011). "Google Go strikes back with C++ bake-off". The Register.

151. ^ Yager, Will. "Why Go is not Good". Retrieved November 4, 2018.
152. ^ Dobronszki, Janos. "Everyday Hassles in Go". Retrieved November 4, 2018.
153. ^ "Why are there braces but no semicolons? And why can't I put the opening brace on the next line?". Retrieved March 26, 2020. “The advantages of a single, programmatically mandated format for all Go programs greatly outweigh any perceived disadvantages of the particular style.”
154. ^ "I want off Mr. Golang's Wild Ride". February 28, 2020. Retrieved November 17, 2020.
155. ^ "proposal: os: Create/Open/OpenFile() set FILE_SHARE_DELETE on windows #32088". GitHub. May 16, 2019. Retrieved November 17, 2020.
156. ^ Tu, Tengfei (2019). "Understanding Real-World Concurrency Bugs in Go" (PDF). “For example, around 58% of blocking bugs are caused by message passing. In addition to the violation of Go's channel usage rules (e.g., waiting on a channel that no one sends data to or close), many concurrency bugs are caused by the mixed usage of message passing and other new semantics and new libraries in Go, which can easily be overlooked but hard to detect”
157. ^ Brownlee, John (November 13, 2009). "Google didn't google "Go" before naming their programming language"". Archived from the original on December 8, 2015. Retrieved May 26, 2016.
158. ^ Claburn, Thomas (November 11, 2009). "Google 'Go' Name Brings Accusations Of Evil'". InformationWeek. Archived from the original on July 22, 2010. Retrieved January 18, 2010.
159. ^ a b "Issue 9 - go — I have already used the name for *MY* programming language". Github. Google Inc. Retrieved October 12, 2010.

- Donovan, Alan; Kernighan, Brian (October 2015). The Go Programming Language (1st ed.). Addison-Wesley Professional. p. 400. ISBN 978-0-13-419044-0.
- Bodner, Jon (March 2021). Learning Go (1st ed.). O'Reilly. p. 352. ISBN 9781492077213.

- Official website

Back to TOC

Comparative Analysis of Open-Source News Crawlers

Summary of Key Findings

This report evaluates six open-source news crawlers—**news-please**, **Fundus**, **news-crawler**, **news-crawl**, **Trafilatura**, and **newspaper4k**—focusing on extraction accuracy, supported sites, and ease of use. Fundus and Trafilatura lead in precision and recall for text extraction, while newspaper4k excels in multilingual support and NLP integration. News-please and news-crawl are optimized for large-scale archival, with trade-offs in speed and configurability. Below, we dissect each tool's strengths, weaknesses, and ideal use cases.

News-Please

Overview

news-please is a Python-based crawler designed for large-scale news extraction, integrating with CommonCrawl's archive for historical data retrieval[12].

Pros

- **CommonCrawl Integration**: Efficiently extracts articles from CommonCrawl's vast archive, ideal for longitudinal studies[12].
- **Structured Metadata**: Extracts titles, authors, publication dates, and multilingual content with 80+ language support[12].
- **Flexible Storage**: Supports JSON, PostgreSQL, Elasticsearch, and Redis[23].

Cons

- **Speed**: Slower processing (61x baseline in benchmarks) due to comprehensive metadata extraction[45].
- **IP Blocking**: Prone to throttling when scraping large sites like CNN[63].

- **Setup Complexity**: Requires manual configuration for Elasticsearch/Redis[23].

Conclusion: Best for researchers needing historical news data from CommonCrawl, but less suited for real-time scraping.

Fundus

Overview

Fundus uses **bespoke parsers** tailored to individual news sites, prioritizing extraction quality over quantity[789].

Pros

- **Highest Accuracy**: Achieves F1-scores of 97.69% in benchmarks, outperforming Trafilatura (93.62%) and news-please (93.39%)[685].
- **Structured Output**: Preserves article formatting (paragraphs, subheadings) and extracts meta-attributes like topics[78].
- **CommonCrawl Optimization**: Efficiently processes CC-NEWS datasets with multi-core support[82].

Cons

- **Limited Coverage**: Supports only predefined publishers (e.g., AP News, Reuters), restricting scalability[82].
- **Static Crawling**: Lacks real-time dynamic content handling[62].

Conclusion: Ideal for projects requiring artifact-free text from high-quality sources, but not for dynamic or unsupported sites.

News-Crawler (LuChang-CS)

Overview

A Python-based tool targeting major outlets like BBC and Reuters[10].

Pros

- **Ease of Use**: Simple CLI and Python API for small-scale scraping[10].
- **Versioning**: Tracks article changes over time, useful for longitudinal analysis[10].

Cons

- **Limited Benchmarking**: No public performance metrics compared to alternatives[10].
- **Resource-Intensive**: Struggles with large-scale crawls due to single-threaded design[10].

Conclusion: Suitable for academic projects with limited scope, but lacks enterprise-grade scalability.

News-Crawl (CommonCrawl)

Overview

A StormCrawler-based system producing WARC files for archival[11][10].

Pros

- **Archival Focus**: Generates WARC files compatible with CommonCrawl's AWS Open Dataset[11].
- **RSS/Sitemap Support**: Discovers articles via feeds, ensuring comprehensive coverage[11].

Cons

- **Complex Setup**: Requires Elasticsearch and Apache Storm, increasing deployment overhead[11].
- **No Content Extraction**: Stores raw HTML without text/metadata extraction[11].

Conclusion: Tailored for developers building news archives, not for direct content analysis.

Trafilatura

Overview

A Python/CLI tool optimized for precision and multilingual extraction[12135].

Pros

- **Benchmark Leader**: Outperforms Goose3, Boilerpipe, and Readability with 90.2% F1-score[135].
- **Lightweight**: Processes HTML 4.8x faster than news-please[125].
- **Metadata Retention**: Extracts publish dates, authors, and languages consistently[1314].

Cons

- **Speed vs. Recall**: Precision mode reduces recall by 3%[5].
- **Dynamic Content**: Struggles with JavaScript-rendered pages without Playwright integration[14].

Conclusion: The best all-rounder for most use cases, balancing speed, accuracy, and ease of use.

Newspaper4k

Overview

A revived fork of Newspaper3k with enhanced NLP and multithreading[615].

Pros

- **NLP Integration**: Generates summaries/extracts keywords, ideal for content curation[615].
- **Multithreading**: Downloads articles 15x faster than single-threaded tools[615].
- **Backward Compatibility**: Seamless migration from Newspaper3k[615].

Cons

- **Dependency Hell**: Requires manual installation of libxml2, Pillow, etc.[615].
- **Incomplete Fixes**: 180+ open GitHub issues, including inconsistent date parsing[615].

Conclusion: Optimal for developers needing NLP features and Google News scraping, despite setup hurdles.

Final Recommendations

By Use Case

1. **Highest Accuracy**: **Fundus** for academic/labelled datasets[78].
2. **General-Purpose**: **Trafilatura** for multilingual, precision-focused extraction[12514].
3. **NLP/Summarization**: **Newspaper4k** for keyword extraction and metadata[615].
4. **Historical Archives**: **news-please** or **news-crawl** for CommonCrawl integration[111].

Summary Table

Tool	Accuracy (F1)	Speed	Ease of Use	Best For
Fundus	97.69%[8]	Medium	Moderate	High-quality, predefined publishers
Trafilatura	90.2%[5]	High	High	Multilingual, general-purpose
Newspaper4k	94.6%[6]	High	Moderate	NLP features, Google News
news-please	85.81%[5]	Low	Low	CommonCrawl historical data

Note: Metrics derived from cited benchmarks.

Critical Considerations

- **Dynamic Content**: None of the tools natively handle JavaScript-heavy sites; pair with Playwright/Selenium[1415].

- **Legal Compliance**: Adhere to robots.txt and rate limits to avoid IP blocks[16][17].

By aligning tool capabilities with project requirements, users can optimize extraction quality and efficiency effectively[4][7][8][5].

⁂

Footnotes

1. https://github.com/fhamborg/news-please ↩ ↩2 ↩3 ↩4
2. https://github.com/free-news-api/news-crawlers ↩ ↩2 ↩3 ↩4 ↩5 ↩6 ↩7 ↩8
3. https://github.com/free-news-api/news-crawlers ↩ ↩2 ↩3
4. https://htmldate.readthedocs.io/en/latest/evaluation.html ↩ ↩2
5. https://trafilatura.readthedocs.io/en/latest/evaluation.html ↩ ↩2 ↩3 ↩4 ↩5 ↩6 ↩7 ↩8 ↩9 ↩10
6. https://github.com/free-news-api/news-crawlers ↩ ↩2 ↩3 ↩4 ↩5 ↩6 ↩7 ↩8 ↩9 ↩10 ↩11
7. https://arxiv.org/html/2403.15279 ↩ ↩2 ↩3 ↩4
8. https://aclanthology.org/2024.acl-demos.29.pdf ↩ ↩2 ↩3 ↩4 ↩5 ↩6 ↩7 ↩8
9. https://aclanthology.org/2024.acl-demos.29/ ↩
10. https://github.com/free-news-api/news-crawlers ↩ ↩2 ↩3 ↩4 ↩5 ↩6
11. https://github.com/commoncrawl/news-crawl ↩ ↩2 ↩3 ↩4 ↩5 ↩6
12. https://github.com/markusmobius/go-trafilatura ↩ ↩2 ↩3
13. https://trafilatura.readthedocs.io ↩ ↩2 ↩3

14. https://www.reddit.com/r/LangChain/comments/1ef12q6/the_rag_engineers_guide_to_document_parsing/ ↩ ↩2 ↩3 ↩4

15. https://www.reddit.com/r/Python/comments/1bmtdy0/i_forked_newspaper3k_fixed_bugs_and_improved_its/ ↩ ↩2 ↩3 ↩4 ↩5 ↩6 ↩7 ↩8

16. https://forage.ai/blog/introduction-to-news-crawling/ ↩

17. https://forage.ai/blog/introduction-to-news-crawling/ ↩

Back to TOC

Claude Code: Best Practices and Pro Tips

This guide provides tips and tricks for effectively using Claude Code, a command-line tool for agentic coding.

Using Claude Code as a Bash CLI

Claude Code (often invoked as `claude` or `cc`) can be used similarly to other bash-based command-line interfaces.

1. **Use CC as a bash CLI** You can perform many standard command-line operations. For example, to checkout a new branch and lint the project:

   ```
   claude "checkout a new branch and lint this project"
   ```

2. **Pass command line arguments to CC** Arguments passed to `claude` on startup will be run. For instance:

   ```
   claude "How does turnManager.js work?"
   ```

3. **Use `claude -p` for headless mode** The `-p` flag allows Claude Code to run in headless mode, meaning it will output the result directly to the terminal without entering the interactive interface.

   ```
   claude -p "How many files are in this project?"
   ```

   Output might be:

   ```
   834 files
   ```

4. **Chain CC with other CLIs** You can pipe the output of other commands into Claude Code or vice-versa.

5. **Pipe data into CC** For example, to analyze a CSV file:

```
cat data.csv | claude -p "Who won the most games?"
```

Output might be:

```
Based on the data, Rusty won the most games with 3 wins.
```

6. **Run instances in parallel** You can have multiple Claude Code instances running simultaneously, perhaps in different terminal tabs or windows, each working on different tasks or parts of a project.

7. **Ask CC to run "subagents" (Tasks)** Claude Code can launch instances of itself to perform sub-tasks. This is often indicated by a `Task(...)` in the Claude Code output. For example, if you ask Claude to find where `gameState.direction` is modified, it might spin up a subagent:

```
Task(Find where gameState.direction is modified)...
```

Claude Code + Images

Claude Code has powerful capabilities for working with images.

1. **Drag Images into CC** You can drag an image file directly into the terminal window where Claude Code is running. Claude will then be able to "see" and analyze the image.

```
> [Image #1]
```

2. **Copy-paste images** On macOS, you can copy a screenshot to your clipboard using `Shift + Command + Control + 4`, then select an area. Paste it into Claude Code using `Control + V` (not `Command + V`).

```
> [Image #1]
```

3. **Give CC mockups** You can provide Claude Code with image mockups of a UI and ask it to build the interface. Paste the mockup image and then prompt:

```
> [Image #1]
> Build the HTML and CSS for this UI mockup.
```

4. **Use screenshots as feedback** To iterate on a design or fix a UI bug:

    - Ask Claude Code to build something.
    - Open what it built in a browser.
    - Take a screenshot.
    - Paste the screenshot into Claude Code.
    - Provide feedback, e.g., “The button in this screenshot [Image #1] should be blue.”

5. **Automate feedback with Puppeteer** You can set up a Puppeteer MCP (Model Context Protocol) server locally. Then, Claude Code can use Puppeteer to open a web page, take a screenshot, and save it. First, ensure your Puppeteer MCP server is running. Then, in Claude Code:

```
> /mcp
MCP Server Status
* Puppeteer: connected

> Open http://localhost:1234 and take a screenshot named 'welcome-screen'.
```

   Claude Code might respond with:

```
puppeteer:puppeteer_navigate (MCP){url: "http://localhost:1234"}...
Navigated to http://localhost:1234.
puppeteer:puppeteer_screenshot (MCP){name: "welcome-screen", width: 1024,
height: 768}...
Screenshot 'welcome-screen' taken at 1024x768
[Image]
```

Claude Code + MCP (Model Context Protocol)

MCP allows Claude Code to interact with various tools and services.

1. **Use CC as an MCP *server*** Claude Code can function as an MCP server, allowing other agents or tools to interact with it.

2. **Use CC as an MCP *client*** Claude Code can also act as an MCP client, connecting to various MCP servers to leverage their capabilities. There are many available MCP servers for different services.

3. **Connect your DB with MCP** You can use a Postgres MCP server (or other database MCP servers) to allow Claude Code to interact directly with your database. After configuring the Postgres MCP server:

   ```
   > Can you see my database?
   ```

   Claude might respond with:

   ```
   * postgres:query (MCP){sql: "SELECT COUNT(*) as total_rows FROM
   netflix_shows;"}...
   [
     {
       "total_rows": "8807"
     }
   ]
   * postgres:query (MCP){sql: "SELECT * FROM netflix_shows LIMIT 5;"}...
   [
     {
       "show_id": "s1",
       "type": "Movie",
       "title": "Dick Johnson Is Dead",
       ...
     }
   ]
   Your netflix_shows table contains 8,807 Netflix titles with these columns:
   - show_id (text, required) - Unique identifier
   ...
   ```

4. **Connect APIs with MCP** MCP servers can act as wrappers around APIs, like the Stripe API. After configuring the Stripe MCP server:

   ```
   > what's my current balance on stripe?
   ```

   Claude might respond with:

   ```
   * stripe:retrieve_balance (MCP)
   {
     "object": "balance",
   ```

```
  "available": [
    {
      "amount": 111618,
      "currency": "usd",
      ...
    }
  ],
  ...
}
You have $1,116.18 available in your Stripe account.
```

5. **Import docs with MCP** Some services, like Cloudflare, provide MCP servers to give Claude access to their up-to-date documentation.

   ```
   claude mcp add --transport sse sse-server https://docs.mcp.cloudflare.com/sse
   ```

   Then in Claude Code:

   ```
   > how do I deploy this app on cloudflare workers. use the up to date
   cloudflare docs.
   ```

   Claude might use the MCP server:

   ```
   * sse-server:search_cloudflare_documentation (MCP){query: "deploy static site
   cloudflare workers pages"}...
   [result]
   <url>https://developers.cloudflare.com/pages/migrations/migrating-from-
   workers/</url>
   <text>
   # Pages > Migration guides > Migrating from Workers Sites to Pages
   ...
   </text>
   [/result]
   ```

6. **Import docs with URLs** If an MCP server for documentation isn't available, you can paste a URL to the documentation directly into Claude Code.

   ```
   > Build the hello world with pydanticai using: https://ai.pydantic.dev/
   ```

   Claude will fetch and use the content from the URL:

```
* Fetch(https://ai.pydantic.dev/)...
Received 69.8KB (200 OK)
* Modeling...
...
```

7. **Import misc knowledge with URLs** You can use the fetch capability to provide Claude with any knowledge from a URL that it might need for a task. For example, to build a game based on specific rules:

~~~
> write pseudo code to describe the rules of uno based on here: https://
www.unorules.com/
```    Claude will fetch the rules:
~~~

   - Fetch(https://www.unorules.com/)… Received 123.8KB (200 OK)
   - Write(file_path: uno_rules.txt)_ // UNO GAME RULES - PSEUDO CODE // Setup: // players = 2 to 10 (ages 7+) // FOR EACH player: // deal 7 cards …

```

```

Using `CLAUDE.md` Files

`CLAUDE.md` is a special file that Claude Code automatically pulls into context when starting a conversation.

1. **Use `CLAUDE.md` files** This file is an ideal place for documenting:

   - Common bash commands
   - Core files and utility functions
   - Code style guidelines
   - Testing instructions
   - Repository etiquette (e.g., branch naming, merge vs. rebase)
   - Developer environment setup (e.g., pyenv use, which compilers work)
   - Any unexpected behaviors or warnings particular to the project
   - Other information you want Claude to remember

   Example `CLAUDE.md` content:

```
# Bash commands
npm run build: Build the project
npm run typecheck: Run the typechecker
```

```
# Code style
- Use ES modules (import/export) syntax, not CommonJS (require)
- Destructure imports when possible (eg. import { foo } from 'bar')

# Workflow
- Be sure to typecheck when you're done making a series of code changes
- Prefer running single tests, and not the whole test suite, for performance
```

You can place `CLAUDE.md` files in:

- The root of your repo.
- Any parent of the directory where you run `claude`.
- Any child of the directory where you run `claude`.

2. **`/init` creates `CLAUDE.md`** Typing `/init` after launching Claude Code in a project directory will prompt Claude to analyze your codebase and create a `CLAUDE.md` file with essential information.

```
> /init
* /init is analyzing your codebase...
* I'll analyze this Ruby on Rails codebase and create a CLAUDE.md file with
the essential information for future Claude Code instances.
* Task(Analyze Rails codebase structure)_...
Done (16 tool uses * 22.2k tokens * 41.8s)
...
```

3. **# adds to `CLAUDE.md`** If you prefix a message in Claude Code with a hash symbol (#), Claude will ask if you want to save this information to your `CLAUDE.md` file (Project memory).

```
> # always use single responsibility principle when creating new methods
Where should this memory be saved?
1. Project memory (Checked in at ./CLAUDE.md)
2. Project memory (Local) (Gitignored in ./CLAUDE.local.md)
3. User memory (Saved in ~/.claude/CLAUDE.md)
```

4. **`~/.claude/CLAUDE.md` - Global** You can set up a global `CLAUDE.md` file in your `~/.claude/` directory. This will be loaded anytime you use Claude Code, across any project.

5. **Use `CLAUDE.md` in subdirs (e.g., tests)** You can have `CLAUDE.md` files in subdirectories (e.g., a `/tests` directory) to provide context specific to that part of the project.

6. **Refactor `CLAUDE.md` often** `CLAUDE.md` files can grow. Periodically review and refactor them to keep them concise and relevant, as this file is loaded as a prompt with every request. More specific prompts yield better results.

7. **Use Anthropic's prompt improver** For complex `CLAUDE.md` files, consider using a prompt optimization tool to help structure and refine the content for better performance with Claude.

Slash Commands

Slash commands are custom prompts you can define.

1. **Define slash commands in `.claude/commands`** Create files in the `.claude/commands` directory (either in your project root or your global `~/.claude/` directory). Each file represents a slash command. These are essentially prompt templates. Example `issue.md` for a `/issue` command:

   ```
   Please analyze and fix the GitHub issue: $ARGUMENTS.

   Follow these steps:
   1. Use 'gh issue view' to get the issue details.
   2. Understand the problem described in the issue.
   3. Search the codebase for relevant files.
   4. Implement the necessary changes to fix the issue.
   5. Ensure the code passes linting and type checking.
   6. Create a descriptive commit message.
   7. Push and create a PR.

   Remember to use the GitHub CLI ('gh') for all Github-related tasks.
   ```

   To use it:

   ```
   > /issue 39
   ```

2. **Use args with slash commands** As seen above, the `$ARGUMENTS` variable in your command file will be replaced by whatever you type after the slash command.

UI Tips

1. **Tab to autocomplete filenames** When typing filenames or paths in the Claude Code prompt, you can use the `Tab` key for autocompletion. Being specific with file paths helps Claude.

2. **Hit `esc` early and often** If you see Claude Code going down the wrong path or taking too long, don't hesitate to press the `Escape` key to interrupt it.

3. **Ask CC to undo** After interrupting, you can ask Claude Code to undo its last set of actions.

   ```
   > Undo the previous changes.
   ```

Version Control

Using Claude Code with version control is highly recommended.

1. **Have CC use version control** Instruct Claude Code to use Git commands.

   ```
   > Checkout a new branch named 'feature-xyz' and commit these changes.
   ```

2. **Have CC commit often** Ask Claude Code to commit changes after every significant modification. This makes it easier to roll back if needed.

   ```
   > Commit the changes with the message "Implement feature X".
   ```

3. **Have CC write your commit messages** Claude Code can often write very good, descriptive commit messages.

4. **Revert more often** Don't be afraid to use `git revert` or `git reset` if Claude makes extensive unwanted changes. Sometimes it's faster to revert to a known good state and try a more specific prompt.

5. **Install GitHub CLI (`gh`)** Claude Code can use the `gh` CLI for interactions with GitHub, such as creating pull requests or viewing issues.

6. **Or use GitHub via MCP** Alternatively, you can configure and use the GitHub MCP server for interactions with GitHub.

7. **Ask CC to file PRs** Once changes are committed, you can ask Claude to create a pull request.

   ```
   > Create a pull request for the current branch.
   ```

8. **Ask CC to review PRs** You can provide Claude Code with the context of a pull request (e.g., by pasting a link or using the `gh` CLI or MCP) and ask it to perform a code review.

Managing Context

Effectively managing context is key to getting the most out of Claude Code.

1. **Be aware of upcoming auto-compact** Keep an eye on the context left indicator (often in the bottom right of the UI). This tells you how much of the conversation history can be retained before Claude starts automatically compacting (summarizing) older parts.

2. **Proactively compact at checkpoints** When you reach a natural breakpoint in your workflow (e.g., after a feature is complete, a bug is fixed, or a commit is made), and you see the context window getting full, consider manually compacting the context using the `/compact` command. This gives you more control over the summarization.

   ```
   > /compact
   Compacting conversation history...
   ```

3. **Consider `/clear` vs `/compact`** If the current conversation thread has gone too far off track or contains a lot of irrelevant information for the next task, `/clear` might be better than `/compact`. `/clear` wipes the conversation history, giving Claude a completely fresh start (though it will still have `CLAUDE.md` and file content you provide).

4. **Use scratch pads to plan work** Tell Claude to use a scratchpad file (e.g., `SCRATCHPAD.md`) to outline its plan, list files it will modify, or jot down thoughts before making changes. This helps you guide its process and makes the context more explicit.

   ```
   > Plan the refactor of game.js in SCRATCHPAD.md before making changes.
   ```

5. **Use GH issues to plan work** Alternatively, use GitHub issues to define tasks and plans. You can then refer Claude to these issues.

6. **Smaller context -> lower cost** If you are on a token-based pricing plan, actively managing and minimizing the context (by clearing, compacting, and using external memory like files and `CLAUDE.md`) will help reduce costs.

7. **Use OpenTelemetry support** For more robust cost tracking, especially in team environments, configure Claude Code's OpenTelemetry (OTel) support. This allows you to send metrics to backends like DataDog to monitor token usage and costs. You can configure this via environment variables or a managed settings JSON file (e.g., `~/Library/Application Support/ClaudeCode/managed-settings.json` on macOS). Example `managed-settings.json` snippet:

```
{
  "env": {
    "CLAUDE_CODE_ENABLE_TELEMETRY": "1",
    "OTEL_METRICS_EXPORTER": "otlp",
    "OTEL_EXPORTER_OTLP_PROTOCOL": "grpc",
    "OTEL_EXPORTER_OTLP_ENDPOINT": "http://collector.company.com:4317",
    "OTEL_EXPORTER_OTLP_HEADERS": "Authorization=Bearer your-auth-token"
  }
}
```

8. **Upgrade to Claude Max plans** If cost per token is a concern, consider upgrading to a Claude Max plan which often bundles a larger amount of usage for a flat fee, potentially making heavy usage more cost-effective.

Back to TOC

How to use Claude Code

Claude Code in 2025: comprehensive guide to features and advanced usage

Claude Code has emerged as a transformative AI coding agent, offering unprecedented capabilities that fundamentally change how both developers and non-technical users approach software creation. This article provides a complete guide to maximizing productivity with Claude Code in 2025.

Custom slash commands creation and organization

Slash commands provide powerful workflow automation through reusable prompt templates stored as Markdown files. These commands dramatically reduce repetitive tasks and standardize team workflows.

Creating and managing commands

Project-scoped commands live in `.claude/commands/` directory and are accessible to all team members who clone the repository. Access these with `/project:command_name` syntax. For example, creating a performance optimization command:

```
echo "Analyze the performance of this code and suggest three specific
optimizations:" > .claude/commands/optimize.md
```

User-scoped commands reside in `~/.claude/commands/` and work across all projects. These personal productivity enhancers are invoked with `/user:command_name`.

Organizational best practices

Implement **hierarchical structures** using subdirectories for better categorization. Commands like `.claude/commands/frontend/component.md` become `/project:frontend:component`. Maintain consistent naming conventions with descriptive, action-oriented names using hyphens for multi-word commands.

Include the `$ARGUMENTS` placeholder for dynamic inputs, enabling commands like:

```
# .claude/commands/fix-issue.md
Please analyze and fix the GitHub issue: $ARGUMENTS
1. Use `gh issue view` to get issue details
2. Search codebase for relevant files
3. Implement necessary changes
4. Write and run tests
5. Create descriptive commit and PR
```

Planning mode usage and extended thinking

Planning mode, officially called "Extended Thinking," allows Claude to spend additional time analyzing problems before responding. This feature enables **deep reasoning for complex tasks** with configurable thinking budgets from 1,024 to 128K tokens.

Natural language triggers and token allocation

Activate different thinking levels with simple phrases:

- `"think"` → 4,000 tokens
- `"think hard"` or `"megathink"` → 10,000 tokens
- `"think harder"` or `"ultrathink"` → 31,999 tokens

The thinking process displays as italic gray text, providing transparency into Claude's reasoning. This proves invaluable for complex problem-solving, architecture decisions, debugging intricate issues, and large-scale refactoring projects.

Effective usage patterns

Deploy extended thinking for **multi-constraint problems** where Claude must balance competing requirements. For architectural decisions, request: "Design a scalable microservices architecture for our e-commerce platform. Think harder about the trade-offs between consistency and availability."

Extended thinking excels when applied to debugging complex issues, refactoring strategies, and system design challenges. The feature performs optimally with English language inputs and high-level instructions that allow Claude to determine the optimal thinking approach.

MCP servers setup and benefits

The Model Context Protocol represents a paradigm shift in AI-tool integration, serving as the **"USB-C of AI applications."** This open standard enables standardized connections between AI models and external data sources, tools, and services.

Understanding the dual architecture

Claude Code uniquely functions as both MCP client and server. As a client, it connects to multiple MCP servers simultaneously, accessing external tools while maintaining isolated connections for security. As a server, it exposes built-in tools (View, Edit, LS) to other applications, enabling programmatic access to Claude's coding capabilities.

Configuration and implementation

Set up MCP servers through three methods:

CLI Wizard (beginner-friendly):

```
claude mcp add puppeteer -s project -- npx -y @modelcontextprotocol/server-
puppeteer
```

Direct JSON configuration (advanced):

```
{
  "mcpServers": {
    "postgres": {
      "command": "npx",
      "args": ["-y", "@modelcontextprotocol/server-postgres", "postgresql://
localhost/mydb"]
    },
    "github": {
      "command": "npx",
      "args": ["-y", "@modelcontextprotocol/server-github"],
      "env": {
        "GITHUB_TOKEN": "your-token"
      }
    }
  }
}
```

Transformative benefits

MCP servers enable **visual testing** through Puppeteer integration, **error monitoring** via Sentry, **direct database operations**, and seamless **enterprise system integration**. The ecosystem has grown to over 1,000+ community servers as of 2025, with adoption by major AI providers including OpenAI, Google, and Microsoft.

Sub-agents and task delegation

Claude Code implements a sophisticated sub-agent system enabling **parallel task execution** and specialized problem-solving. This architecture allows Claude to automatically spawn sub-agents working on different aspects of complex problems simultaneously.

Delegation patterns and best practices

Effective delegation follows a structured pattern:

1. **Planning Phase**: Main agent coordinates overall strategy
2. **Execution Phase**: Sub-agents handle specialized tasks in parallel
3. **Validation Phase**: Independent verification agents check outputs
4. **Integration Phase**: Results consolidate under main agent coordination

Trigger parallel execution with requests like: “Research three separate approaches to implement OAuth2. Do it in parallel using three agents.”

Optimal task types for delegation

Research and analysis benefits from parallel investigation of competing solutions and simultaneous review of multiple codebases. **Development tasks** excel with parallel component development and independent test suite creation. **Quality assurance** improves through independent security analysis and cross-verification of implementations.

Sub-agents preserve main context by handling specialized tasks, reduce context switching overhead, and enable more efficient token usage. The system catches edge cases through independent validation and provides multiple perspectives for robust solutions.

Workflow optimization tips

Foundation setup with CLAUDE.md

Create a **central configuration file** using the `/init` command. This persistent memory includes coding conventions, build commands, testing procedures, and repository etiquette. The file dynamically updates during sessions using the # key for auto-incorporation.

Example structure:

```
# Project: MyApp
## Technologies
- React, TypeScript, Node.js
## Build Commands
- `npm run build`: Build the project
## Code Style
- Use ES modules (import/export)
## Workflow
- Always run tests before committing
```

Advanced prompting techniques

Replace vague requests with **precise instructions**. Instead of “fix this,” use “add input validation to ensure user_id is an integer and write tests for edge cases.” Implement structured workflows separating research, planning, implementation, and verification phases.

IDE integration strategies

VS Code users can launch Claude Code with `Cmd+Esc` (Mac) or `Ctrl+Esc` (Windows/Linux). The extension automatically shares current selection, open tabs, and diagnostic errors. **JetBrains** integration offers similar capabilities with full diagnostic sharing and inline diff viewing.

Performance optimization

Leverage **git worktrees** for parallel Claude sessions on different features. Use headless mode (`-p` flag) for automation and scripts. Implement allowlists for trusted tools to optimize token usage. Monitor costs with automatic session summaries and set appropriate model selection (Sonnet 4 for most tasks, Opus 4 for complex decisions).

Project initialization best practices

The /init command foundation

Running `/init` in your project root generates comprehensive project analysis, creating standardized documentation and establishing consistent development environments. This automated setup identifies technology stacks, build commands, and coding conventions.

Optimal project structure

Organize projects with dedicated directories:

```
project/
├── .claude/
│   ├── commands/          # Custom slash commands
│   └── logs/             # Conversation logs
├── CLAUDE.md             # Main configuration
├── .mcp.json             # MCP server config
└── spec.md               # Project specification
```

Team collaboration setup

Check CLAUDE.md into version control for shared team knowledge.
Include .mcp.json for common tool configurations. Document team-specific commands and establish allowlist standards for consistent security practices.

Real-world examples and productivity gains

Developer success stories

Anthropic engineers report **95% of git operations** handled through Claude. Complex refactoring tasks see dramatic improvements, with one example showing a 2-year-old broken codebase fixed in just 2 days. Rakuten validated Opus 4 running **autonomously for 7 hours** on open-source refactoring projects.

Data scientists convert exploratory notebooks to production Metaflow pipelines, saving 1-2 days per model. The CodeConcise tool demonstrates adding new programming language support in minutes versus traditional weeks-long implementations.

Non-technical user breakthroughs

Users with **no coding experience** successfully build and deploy functional applications. One testimonial describes going "from downtrodden and skeptical to dancing around the room in 15 minutes" after automating frustrating work tasks. Teachers create professional development tracking applications, while business users automate manual workflows into efficient systems.

The democratization of programming enables:

- 50-page PDF policies summarized in 3 paragraphs
- Customer support automation with empathetic response generation
- Interactive data visualizations from simple prompts
- Custom productivity calculators and utility applications

Enterprise adoption patterns

Companies leverage Claude Code for applications they "wouldn't have had bandwidth for," including AI labeling tools, sales ROI calculators, and complex multi-step automated tasks. Small teams achieve **enterprise-level output** through AI multiplication of resources.

Advanced features and future capabilities

Memory and persistence

Opus 4 creates and maintains memory files for long-term context, enabling sophisticated project understanding across sessions. Background task support through GitHub Actions integration allows continuous operation beyond interactive sessions.

Visual development

Screenshot-based development enables UI creation from mockups. Claude analyzes visual inputs to generate matching code implementations, particularly effective for frontend development and data visualization tasks.

Industry integration

Claude Sonnet 4 powers **GitHub Copilot** as its base model. The platform integrates with Cursor for state-of-the-art coding capabilities. Enterprise deployments through AWS Bedrock and Google Vertex AI enable secure, scalable implementations.

Maximizing productivity with Claude Code

Success with Claude Code requires understanding its capabilities as an **orchestrated system** rather than a simple tool. Implement specification-driven development by generating detailed specs before coding. Embrace test-driven development, which Claude excels at implementing. Use iterative improvement—2-3 passes typically yield significantly better results.

Configure your environment with comprehensive CLAUDE.md files, relevant MCP servers for your tech stack, and custom slash commands for repetitive workflows. Leverage parallel Claude instances for complex tasks, with one writing code while another reviews and tests.

The evidence demonstrates Claude Code as a **transformative force** in software development, enabling both seasoned developers and complete beginners to build sophisticated applications. As the ecosystem continues expanding with community contributions and enterprise adoption, Claude Code is positioned to fundamentally reshape how software is created, making programming accessible to broader audiences while amplifying professional developer capabilities.

With productivity gains ranging from 2x to 10x and breakthrough capabilities in autonomous operation, Claude Code represents not just an evolution in AI assistance but a revolution in how we approach software creation. The key to success lies in embracing its full capabilities—from custom commands and extended thinking to MCP integration and multi-agent workflows—to achieve unprecedented development efficiency and innovation.

Back to TOC