Frequently Asked Questions (FAQ)

Table of Contents

Origins

What is the purpose of the project?

At the time of Go's inception, only a decade ago, the programming world was different from today. Production software was usually written in C++ or Java, GitHub did not exist, most computers were not yet multiprocessors, and other than Visual Studio and Eclipse there were few IDEs or other high-level tools available at all, let alone for free on the Internet.

Meanwhile, we had become frustrated by the undue complexity required to use the languages we worked with to develop server software. Computers had become enormously quicker since languages such as C, C++ and Java were first developed but the act of programming had not itself advanced nearly as much. Also, it was clear that multiprocessors were becoming universal but most languages offered little help to program them efficiently and safely.

We decided to take a step back and think about what major issues were going to dominate software engineering in the years ahead as technology developed, and how a new language might help address them. For instance, the rise of multicore CPUs argued that a language should provide first-class support for some sort of concurrency or parallelism. And to make resource management tractable in a large concurrent program, garbage collection, or at least some sort of safe automatic memory management was required.

These considerations led to a series of discussions from which Go arose, first as a set of ideas and desiderata, then as a language. An overarching goal was that Go do more to help the working programmer by enabling tooling, automating mundane tasks such as code formatting, and removing obstacles to working on large code bases.

A much more expansive description of the goals of Go and how they are met, or at least approached, is available in the article, Go at Google: Language Design in the Service of Software Engineering.

What is the history of the project?

Robert Griesemer, Rob Pike and Ken Thompson started sketching the goals for a new language on the white board on September 21, 2007. Within a few days the goals had settled into a plan to do something and a fair idea of what it would be. Design continued part-time in parallel with unrelated work. By January 2008, Ken had started work on a compiler with which to explore ideas; it generated C code as its output. By mid-year the language had become a full-time project and had settled enough to attempt a production compiler. In May 2008, Ian Taylor independently started on a GCC front end for Go using the draft specification. Russ Cox joined in late 2008 and helped move the language and libraries from prototype to reality.

Go became a public open source project on November 10, 2009. Countless people from the community have contributed ideas, discussions, and code.

There are now millions of Go programmers—gophers—around the world, and there are more every day. Go's success has far exceeded our expectations.

What's the origin of the gopher mascot?

The mascot and logo were designed by Renée French, who also designed Glenda, the Plan 9 bunny. A blog post about the gopher explains how it was derived from one she used for a WFMU T-shirt design some years ago. The logo and mascot are covered by the Creative Commons Attribution 3.0 license.

The gopher has a model sheet illustrating his characteristics and how to represent them correctly. The model sheet was first shown in a talk by Renée at Gophercon in 2016. He has unique features; he's the *Go gopher*, not just any old gopher.

Is the language called Go or Golang?

The language is called Go. The "golang" moniker arose because the web site was originally *golang.org*. (There was no *.dev* domain then.) Many use the golang name, though, and it is handy as a label. For instance, the Twitter tag for the language is "#golang". The language's name is just plain Go, regardless.

A side note: Although the official logo has two capital letters, the language name is written Go, not GO.

Why did you create a new language?

Go was born out of frustration with existing languages and environments for the work we were doing at Google. Programming had become too difficult and the choice of languages was partly to blame. One had to choose either efficient compilation, efficient execution, or ease of programming; all three were not available in the same mainstream language. Programmers who could were choosing ease over safety and efficiency by moving to dynamically typed languages such as Python and JavaScript rather than C++ or, to a lesser extent, Java.

We were not alone in our concerns. After many years with a pretty quiet landscape for programming languages, Go was among the first of several new languages—Rust, Elixir, Swift, and more—that have made programming language development an active, almost mainstream field again.

Go addressed these issues by attempting to combine the ease of programming of an interpreted, dynamically typed language with the efficiency and safety of a statically typed, compiled language. It also aimed to be modern, with support for networked and multicore computing. Finally, working with Go is intended to be *fast*: it should take at most a few seconds to build a large executable on a single computer. To meet these goals required addressing a number of linguistic issues: an expressive but lightweight type system; concurrency and garbage collection; rigid dependency specification; and so on. These cannot be addressed well by libraries or tools; a new language was called for.

The article Go at Google discusses the background and motivation behind the design of the Go language, as well as providing more detail about many of the answers presented in this FAQ.

What are Go's ancestors?

Go is mostly in the C family (basic syntax), with significant input from the Pascal/Modula /Oberon family (declarations, packages), plus some ideas from languages inspired by Tony Hoare's CSP, such as Newsqueak and Limbo (concurrency). However, it is a new language across the board. In every respect the language was designed by thinking about what programmers do and how to make programming, at least the kind of programming we do,

more effective, which means more fun.

What are the guiding principles in the design?

When Go was designed, Java and C++ were the most commonly used languages for writing servers, at least at Google. We felt that these languages required too much bookkeeping and repetition. Some programmers reacted by moving towards more dynamic, fluid languages like Python, at the cost of efficiency and type safety. We felt it should be possible to have the efficiency, the safety, and the fluidity in a single language.

Go attempts to reduce the amount of typing in both senses of the word. Throughout its design, we have tried to reduce clutter and complexity. There are no forward declarations and no header files; everything is declared exactly once. Initialization is expressive, automatic, and easy to use. Syntax is clean and light on keywords. Repetition (`foo.Foo* myFoo = new(foo.Foo)`) is reduced by simple type derivation using the `:=` declare-and-initialize construct. And perhaps most radically, there is no type hierarchy: types just *are*, they don't have to announce their relationships. These simplifications allow Go to be expressive yet comprehensible without sacrificing, well, sophistication.

Another important principle is to keep the concepts orthogonal. Methods can be implemented for any type; structures represent data while interfaces represent abstraction; and so on. Orthogonality makes it easier to understand what happens when things combine.

Usage

Is Google using Go internally?

Yes. Go is used widely in production inside Google. One easy example is the server behind golang.org. It's just the `godoc` document server running in a production configuration on Google App Engine.

A more significant instance is Google's download server, `dl.google.com`, which delivers Chrome binaries and other large installables such as `apt-get` packages.

Go is not the only language used at Google, far from it, but it is a key language for a number of areas including site reliability engineering (SRE) and large-scale data processing.

What other companies use Go?

Go usage is growing worldwide, especially but by no means exclusively in the cloud computing space. A couple of major cloud infrastructure projects written in Go are Docker and Kubernetes, but there are many more.

It's not just cloud, though. The Go Wiki includes a page, updated regularly, that lists some of the many companies using Go.

The Wiki also has a page with links to success stories about companies and projects that are using the language.

Do Go programs link with C/C++ programs?

It is possible to use C and Go together in the same address space, but it is not a natural fit and can require special interface software. Also, linking C with Go code gives up the memory safety and stack management properties that Go provides. Sometimes it's absolutely necessary to use C libraries to solve a problem, but doing so always introduces an element of risk not present with pure Go code, so do so with care.

If you do need to use C with Go, how to proceed depends on the Go compiler implementation. There are three Go compiler implementations supported by the Go team. These are `gc`, the default compiler, `gccgo`, which uses the GCC back end, and a somewhat less mature `gollvm`, which uses the LLVM infrastructure.

`Gc` uses a different calling convention and linker from C and therefore cannot be called directly from C programs, or vice versa. The `cgo` program provides the mechanism for a "foreign function interface" to allow safe calling of C libraries from Go code. SWIG extends this capability to C++ libraries.

You can also use `cgo` and SWIG with `Gccgo` and `gollvm`. Since they use a traditional API, it's also possible, with great care, to link code from these compilers directly with GCC/LLVM-compiled C or C++ programs. However, doing so safely requires an understanding of the calling conventions for all languages concerned, as well as concern for stack limits when calling C or C++ from Go.

What IDEs does Go support?

The Go project does not include a custom IDE, but the language and libraries have been designed to make it easy to analyze source code. As a consequence, most well-known editors and IDEs support Go well, either directly or through a plugin.

The list of well-known IDEs and editors that have good Go support available includes Emacs, Vim, VSCode, Atom, Eclipse, Sublime, IntelliJ (through a custom variant called Goland), and many more. Chances are your favorite environment is a productive one for programming in Go.

Does Go support Google's protocol buffers?

A separate open source project provides the necessary compiler plugin and library. It is available at github.com/golang/protobuf/.

Can I translate the Go home page into another language?

Absolutely. We encourage developers to make Go Language sites in their own languages. However, if you choose to add the Google logo or branding to your site (it does not appear on golang.org), you will need to abide by the guidelines at www.google.com/permissions/guidelines.html

Design

Does Go have a runtime?

Go does have an extensive library, called the *runtime*, that is part of every Go program. The runtime library implements garbage collection, concurrency, stack management, and other critical features of the Go language. Although it is more central to the language, Go's runtime is analogous to `libc`, the C library.

It is important to understand, however, that Go's runtime does not include a virtual machine, such as is provided by the Java runtime. Go programs are compiled ahead of time to native machine code (or JavaScript or WebAssembly, for some variant implementations). Thus, although the term is often used to describe the virtual environment in which a program runs, in Go the word “runtime” is just the name given to the library providing critical language services.

What's up with Unicode identifiers?

When designing Go, we wanted to make sure that it was not overly ASCII-centric, which meant extending the space of identifiers from the confines of 7-bit ASCII. Go's rule—identifier characters must be letters or digits as defined by Unicode—is simple to understand and to implement but has restrictions. Combining characters are excluded by design, for instance, and that excludes some languages such as Devanagari.

This rule has one other unfortunate consequence. Since an exported identifier must begin with an upper-case letter, identifiers created from characters in some languages can, by definition, not be exported. For now the only solution is to use something like X日本語, which is clearly unsatisfactory.

Since the earliest version of the language, there has been considerable thought into how best to expand the identifier space to accommodate programmers using other native languages. Exactly what to do remains an active topic of discussion, and a future version of the language may be more liberal in its definition of an identifier. For instance, it might adopt some of the ideas from the Unicode organization's recommendations for identifiers. Whatever happens, it must be done compatibly while preserving (or perhaps expanding) the way letter case determines visibility of identifiers, which remains one of our favorite features of Go.

For the time being, we have a simple rule that can be expanded later without breaking

programs, one that avoids bugs that would surely arise from a rule that admits ambiguous identifiers.

Why does Go not have feature X?

Every language contains novel features and omits someone's favorite feature. Go was designed with an eye on felicity of programming, speed of compilation, orthogonality of concepts, and the need to support features such as concurrency and garbage collection. Your favorite feature may be missing because it doesn't fit, because it affects compilation speed or clarity of design, or because it would make the fundamental system model too difficult.

If it bothers you that Go is missing feature *X*, please forgive us and investigate the features that Go does have. You might find that they compensate in interesting ways for the lack of *X*.

When did Go get generic types?

The Go 1.18 release added type parameters to the language. This permits a form of polymorphic or generic programming. See the language spec and the proposal for details.

Why was Go initially released without generic types?

Go was intended as a language for writing server programs that would be easy to maintain over time. (See this article for more background.) The design concentrated on things like scalability, readability, and concurrency. Polymorphic programming did not seem essential to the language's goals at the time, and so was initially left out for simplicity.

Generics are convenient but they come at a cost in complexity in the type system and run-time. It took a while to develop a design that we believe gives value proportionate to the complexity.

Why does Go not have exceptions?

We believe that coupling exceptions to a control structure, as in the `try-catch-finally` idiom, results in convoluted code. It also tends to encourage programmers to label too many ordinary errors, such as failing to open a file, as exceptional.

Go takes a different approach. For plain error handling, Go's multi-value returns make it easy to report an error without overloading the return value. A canonical error type, coupled with Go's other features, makes error handling pleasant but quite different from that in other languages.

Go also has a couple of built-in functions to signal and recover from truly exceptional conditions. The recovery mechanism is executed only as part of a function's state being

torn down after an error, which is sufficient to handle catastrophe but requires no extra control structures and, when used well, can result in clean error-handling code.

See the Defer, Panic, and Recover article for details. Also, the Errors are values blog post describes one approach to handling errors cleanly in Go by demonstrating that, since errors are just values, the full power of Go can be deployed in error handling.

Why does Go not have assertions?

Go doesn't provide assertions. They are undeniably convenient, but our experience has been that programmers use them as a crutch to avoid thinking about proper error handling and reporting. Proper error handling means that servers continue to operate instead of crashing after a non-fatal error. Proper error reporting means that errors are direct and to the point, saving the programmer from interpreting a large crash trace. Precise errors are particularly important when the programmer seeing the errors is not familiar with the code.

We understand that this is a point of contention. There are many things in the Go language and libraries that differ from modern practices, simply because we feel it's sometimes worth trying a different approach.

Why build concurrency on the ideas of CSP?

Concurrency and multi-threaded programming have over time developed a reputation for difficulty. We believe this is due partly to complex designs such as pthreads and partly to overemphasis on low-level details such as mutexes, condition variables, and memory barriers. Higher-level interfaces enable much simpler code, even if there are still mutexes and such under the covers.

One of the most successful models for providing high-level linguistic support for concurrency comes from Hoare's Communicating Sequential Processes, or CSP. Occam and Erlang are two well known languages that stem from CSP. Go's concurrency primitives derive from a different part of the family tree whose main contribution is the powerful notion of channels as first class objects. Experience with several earlier languages has shown that the CSP model fits well into a procedural language framework.

Why goroutines instead of threads?

Goroutines are part of making concurrency easy to use. The idea, which has been around for a while, is to multiplex independently executing functions—coroutines—onto a set of threads. When a coroutine blocks, such as by calling a blocking system call, the run-time automatically moves other coroutines on the same operating system thread to a different, runnable thread so they won't be blocked. The programmer sees none of this, which is the point. The result, which we call goroutines, can be very cheap: they have little overhead beyond the memory for the stack, which is just a few kilobytes.

To make the stacks small, Go's run-time uses resizable, bounded stacks. A newly minted goroutine is given a few kilobytes, which is almost always enough. When it isn't, the run-time grows (and shrinks) the memory for storing the stack automatically, allowing many goroutines to live in a modest amount of memory. The CPU overhead averages about three cheap instructions per function call. It is practical to create hundreds of thousands of goroutines in the same address space. If goroutines were just threads, system resources would run out at a much smaller number.

Why are map operations not defined to be atomic?

After long discussion it was decided that the typical use of maps did not require safe access from multiple goroutines, and in those cases where it did, the map was probably part of some larger data structure or computation that was already synchronized. Therefore requiring that all map operations grab a mutex would slow down most programs and add safety to few. This was not an easy decision, however, since it means uncontrolled map access can crash the program.

The language does not preclude atomic map updates. When required, such as when hosting an untrusted program, the implementation could interlock map access.

Map access is unsafe only when updates are occurring. As long as all goroutines are only reading—looking up elements in the map, including iterating through it using a `for range` loop—and not changing the map by assigning to elements or doing deletions, it is safe for them to access the map concurrently without synchronization.

As an aid to correct map use, some implementations of the language contain a special check that automatically reports at run time when a map is modified unsafely by concurrent execution.

Will you accept my language change?

People often suggest improvements to the language—the mailing list contains a rich history of such discussions—but very few of these changes have been accepted.

Although Go is an open source project, the language and libraries are protected by a compatibility promise that prevents changes that break existing programs, at least at the source code level (programs may need to be recompiled occasionally to stay current). If your proposal violates the Go 1 specification we cannot even entertain the idea, regardless of its merit. A future major release of Go may be incompatible with Go 1, but discussions on that topic have only just begun and one thing is certain: there will be very few such incompatibilities introduced in the process. Moreover, the compatibility promise encourages us to provide an automatic path forward for old programs to adapt should that situation arise.

Even if your proposal is compatible with the Go 1 spec, it might not be in the spirit of Go's

design goals. The article *Go at Google: Language Design in the Service of Software Engineering* explains Go's origins and the motivation behind its design.

Types

Is Go an object-oriented language?

Yes and no. Although Go has types and methods and allows an object-oriented style of programming, there is no type hierarchy. The concept of “interface” in Go provides a different approach that we believe is easy to use and in some ways more general. There are also ways to embed types in other types to provide something analogous—but not identical—to subclassing. Moreover, methods in Go are more general than in C++ or Java: they can be defined for any sort of data, even built-in types such as plain, “unboxed” integers. They are not restricted to structs (classes).

Also, the lack of a type hierarchy makes “objects” in Go feel much more lightweight than in languages such as C++ or Java.

How do I get dynamic dispatch of methods?

The only way to have dynamically dispatched methods is through an interface. Methods on a struct or any other concrete type are always resolved statically.

Why is there no type inheritance?

Object-oriented programming, at least in the best-known languages, involves too much discussion of the relationships between types, relationships that often could be derived automatically. Go takes a different approach.

Rather than requiring the programmer to declare ahead of time that two types are related, in Go a type automatically satisfies any interface that specifies a subset of its methods. Besides reducing the bookkeeping, this approach has real advantages. Types can satisfy many interfaces at once, without the complexities of traditional multiple inheritance. Interfaces can be very lightweight—an interface with one or even zero methods can express a useful concept. Interfaces can be added after the fact if a new idea comes along or for testing—without annotating the original types. Because there are no explicit relationships between types and interfaces, there is no type hierarchy to manage or discuss.

It's possible to use these ideas to construct something analogous to type-safe Unix pipes. For instance, see how `fmt.Fprintf` enables formatted printing to any output, not just a file, or how the `bufio` package can be completely separate from file I/O, or how the `image` packages generate compressed image files. All these ideas stem from a single interface (`io.Writer`) representing a single method (`Write`). And that's only scratching the surface. Go's interfaces have a profound influence on how programs are structured.

It takes some getting used to but this implicit style of type dependency is one of the most productive things about Go.

Why is `len` a function and not a method?

We debated this issue but decided implementing `len` and friends as functions was fine in practice and didn't complicate questions about the interface (in the Go type sense) of basic types.

Why does Go not support overloading of methods and operators?

Method dispatch is simplified if it doesn't need to do type matching as well. Experience with other languages told us that having a variety of methods with the same name but different signatures was occasionally useful but that it could also be confusing and fragile in practice. Matching only by name and requiring consistency in the types was a major simplifying decision in Go's type system.

Regarding operator overloading, it seems more a convenience than an absolute requirement. Again, things are simpler without it.

Why doesn't Go have "implements" declarations?

A Go type satisfies an interface by implementing the methods of that interface, nothing more. This property allows interfaces to be defined and used without needing to modify existing code. It enables a kind of structural typing that promotes separation of concerns and improves code re-use, and makes it easier to build on patterns that emerge as the code develops. The semantics of interfaces is one of the main reasons for Go's nimble, lightweight feel.

See the question on type inheritance for more detail.

How can I guarantee my type satisfies an interface?

You can ask the compiler to check that the type `T` implements the interface `I` by attempting an assignment using the zero value for `T` or pointer to `T`, as appropriate:

```
type T struct{}
var _ I = T{}       // Verify that T implements I.
var _ I = (*T)(nil) // Verify that *T implements I.
```

If `T` (or `*T`, accordingly) doesn't implement `I`, the mistake will be caught at compile time.

If you wish the users of an interface to explicitly declare that they implement it, you can add a method with a descriptive name to the interface's method set. For example:

```
type Fooer interface {
    Foo()
```

```
    ImplementsFooer()
}
```

A type must then implement the `ImplementsFooer` method to be a `Fooer`, clearly documenting the fact and announcing it in go doc's output.

```
type Bar struct{}
func (b Bar) ImplementsFooer() {}
func (b Bar) Foo() {}
```

Most code doesn't make use of such constraints, since they limit the utility of the interface idea. Sometimes, though, they're necessary to resolve ambiguities among similar interfaces.

Why doesn't type T satisfy the Equal interface?

Consider this simple interface to represent an object that can compare itself with another value:

```
type Equaler interface {
    Equal(Equaler) bool
}
```

and this type, T:

```
type T int
func (t T) Equal(u T) bool { return t == u } // does not satisfy Equaler
```

Unlike the analogous situation in some polymorphic type systems, `T` does not implement `Equaler`. The argument type of `T.Equal` is `T`, not literally the required type `Equaler`.

In Go, the type system does not promote the argument of `Equal`; that is the programmer's responsibility, as illustrated by the type `T2`, which does implement `Equaler`:

```
type T2 int
func (t T2) Equal(u Equaler) bool { return t == u.(T2) }  // satisfies Equaler
```

Even this isn't like other type systems, though, because in Go *any* type that satisfies `Equaler` could be passed as the argument to `T2.Equal`, and at run time we must check that the argument is of type `T2`. Some languages arrange to make that guarantee at compile time.

A related example goes the other way:

```
type Opener interface {
   Open() Reader
}

func (t T3) Open() *os.File
```

In Go, `T3` does not satisfy `Opener`, although it might in another language.

While it is true that Go's type system does less for the programmer in such cases, the lack of subtyping makes the rules about interface satisfaction very easy to state: are the function's names and signatures exactly those of the interface? Go's rule is also easy to implement efficiently. We feel these benefits offset the lack of automatic type promotion. Should Go one day adopt some form of polymorphic typing, we expect there would be a way to express the idea of these examples and also have them be statically checked.

Can I convert a []T to an []interface{}?

Not directly. It is disallowed by the language specification because the two types do not have the same representation in memory. It is necessary to copy the elements individually to the destination slice. This example converts a slice of `int` to a slice of `interface{}`:

```
t := []int{1, 2, 3, 4}
s := make([]interface{}, len(t))
for i, v := range t {
    s[i] = v
}
```

Can I convert []T1 to []T2 if T1 and T2 have the same underlying type?

This last line of this code sample does not compile.

```
type T1 int
type T2 int
var t1 T1
var x = T2(t1) // OK
var st1 []T1
var sx = ([]T2)(st1) // NOT OK
```

In Go, types are closely tied to methods, in that every named type has a (possibly empty) method set. The general rule is that you can change the name of the type being converted (and thus possibly change its method set) but you can't change the name (and method set) of elements of a composite type. Go requires you to be explicit about type conversions.

Why is my nil error value not equal to nil?

Under the covers, interfaces are implemented as two elements, a type `T` and a value `V`. `V` is a concrete value such as an `int`, `struct` or pointer, never an interface itself, and has type `T`. For instance, if we store the `int` value 3 in an interface, the resulting interface value has, schematically, (`T=int`, `V=3`). The value `V` is also known as the interface's *dynamic* value, since a given interface variable might hold different values `V` (and corresponding types `T`) during the execution of the program.

An interface value is `nil` only if the `V` and `T` are both unset, (`T=nil`, `V` is not set), In

particular, a `nil` interface will always hold a `nil` type. If we store a `nil` pointer of type `*int` inside an interface value, the inner type will be `*int` regardless of the value of the pointer: (`T=*int`, `V=nil`). Such an interface value will therefore be non-`nil` *even when the pointer value `V` inside is* `nil`.

This situation can be confusing, and arises when a `nil` value is stored inside an interface value such as an `error` return:

```
func returnsError() error {
        var p *MyError = nil
        if bad() {
                p = ErrBad
        }
        return p // Will always return a non-nil error.
}
```

If all goes well, the function returns a `nil` `p`, so the return value is an `error` interface value holding (`T=*MyError`, `V=nil`). This means that if the caller compares the returned error to `nil`, it will always look as if there was an error even if nothing bad happened. To return a proper `nil` `error` to the caller, the function must return an explicit `nil`:

```
func returnsError() error {
        if bad() {
                return ErrBad
        }
        return nil
}
```

It's a good idea for functions that return errors always to use the `error` type in their signature (as we did above) rather than a concrete type such as `*MyError`, to help guarantee the error is created correctly. As an example, `os.Open` returns an `error` even though, if not `nil`, it's always of concrete type `*os.PathError`.

Similar situations to those described here can arise whenever interfaces are used. Just keep in mind that if any concrete value has been stored in the interface, the interface will not be `nil`. For more information, see The Laws of Reflection.

Why are there no untagged unions, as in C?

Untagged unions would violate Go's memory safety guarantees.

Why does Go not have variant types?

Variant types, also known as algebraic types, provide a way to specify that a value might take one of a set of other types, but only those types. A common example in systems programming would specify that an error is, say, a network error, a security error or an application error and allow the caller to discriminate the source of the problem by examining

the type of the error. Another example is a syntax tree in which each node can be a different type: declaration, statement, assignment and so on.

We considered adding variant types to Go, but after discussion decided to leave them out because they overlap in confusing ways with interfaces. What would happen if the elements of a variant type were themselves interfaces?

Also, some of what variant types address is already covered by the language. The error example is easy to express using an interface value to hold the error and a type switch to discriminate cases. The syntax tree example is also doable, although not as elegantly.

Why does Go not have covariant result types?

Covariant result types would mean that an interface like

```
type Copyable interface {
        Copy() interface{}
}
```

would be satisfied by the method

```
func (v Value) Copy() Value
```

because `Value` implements the empty interface. In Go method types must match exactly, so `Value` does not implement `Copyable`. Go separates the notion of what a type does—its methods—from the type's implementation. If two methods return different types, they are not doing the same thing. Programmers who want covariant result types are often trying to express a type hierarchy through interfaces. In Go it's more natural to have a clean separation between interface and implementation.

Values

Why does Go not provide implicit numeric conversions?

The convenience of automatic conversion between numeric types in C is outweighed by the confusion it causes. When is an expression unsigned? How big is the value? Does it overflow? Is the result portable, independent of the machine on which it executes? It also complicates the compiler; “the usual arithmetic conversions” are not easy to implement and inconsistent across architectures. For reasons of portability, we decided to make things clear and straightforward at the cost of some explicit conversions in the code. The definition of constants in Go—arbitrary precision values free of signedness and size annotations—ameliorates matters considerably, though.

A related detail is that, unlike in C, `int` and `int64` are distinct types even if `int` is a 64-bit type. The `int` type is generic; if you care about how many bits an integer holds, Go encourages you to be explicit.

How do constants work in Go?

Although Go is strict about conversion between variables of different numeric types, constants in the language are much more flexible. Literal constants such as `23`, `3.14159` and `math.Pi` occupy a sort of ideal number space, with arbitrary precision and no overflow or underflow. For instance, the value of `math.Pi` is specified to 63 places in the source code, and constant expressions involving the value keep precision beyond what a `float64` could hold. Only when the constant or constant expression is assigned to a variable—a memory location in the program—does it become a "computer" number with the usual floating-point properties and precision.

Also, because they are just numbers, not typed values, constants in Go can be used more freely than variables, thereby softening some of the awkwardness around the strict conversion rules. One can write expressions such as

```
sqrt2 := math.Sqrt(2)
```

without complaint from the compiler because the ideal number `2` can be converted safely and accurately to a `float64` for the call to `math.Sqrt`.

A blog post titled Constants explores this topic in more detail.

Why are maps built in?

The same reason strings are: they are such a powerful and important data structure that providing one excellent implementation with syntactic support makes programming more pleasant. We believe that Go's implementation of maps is strong enough that it will serve for the vast majority of uses. If a specific application can benefit from a custom implementation, it's possible to write one but it will not be as convenient syntactically; this seems a reasonable tradeoff.

Why don't maps allow slices as keys?

Map lookup requires an equality operator, which slices do not implement. They don't implement equality because equality is not well defined on such types; there are multiple considerations involving shallow vs. deep comparison, pointer vs. value comparison, how to deal with recursive types, and so on. We may revisit this issue—and implementing equality for slices will not invalidate any existing programs—but without a clear idea of what equality of slices should mean, it was simpler to leave it out for now.

In Go 1, unlike prior releases, equality is defined for structs and arrays, so such types can be used as map keys. Slices still do not have a definition of equality, though.

Why are maps, slices, and channels references while arrays are values?

There's a lot of history on that topic. Early on, maps and channels were syntactically pointers and it was impossible to declare or use a non-pointer instance. Also, we struggled with how arrays should work. Eventually we decided that the strict separation of pointers and values made the language harder to use. Changing these types to act as references to the associated, shared data structures resolved these issues. This change added some regrettable complexity to the language but had a large effect on usability: Go became a more productive, comfortable language when it was introduced.

Writing Code

How are libraries documented?

There is a program, `godoc`, written in Go, that extracts package documentation from the source code and serves it as a web page with links to declarations, files, and so on. An instance is running at golang.org/pkg/. In fact, `godoc` implements the full site at golang.org/.

A `godoc` instance may be configured to provide rich, interactive static analyses of symbols in the programs it displays; details are listed here.

For access to documentation from the command line, the go tool has a doc subcommand that provides a textual interface to the same information.

Is there a Go programming style guide?

There is no explicit style guide, although there is certainly a recognizable "Go style".

Go has established conventions to guide decisions around naming, layout, and file organization. The document Effective Go contains some advice on these topics. More directly, the program `gofmt` is a pretty-printer whose purpose is to enforce layout rules; it replaces the usual compendium of do's and don'ts that allows interpretation. All the Go code in the repository, and the vast majority in the open source world, has been run through `gofmt`.

The document titled Go Code Review Comments is a collection of very short essays about details of Go idiom that are often missed by programmers. It is a handy reference for people doing code reviews for Go projects.

How do I submit patches to the Go libraries?

The library sources are in the `src` directory of the repository. If you want to make a significant change, please discuss on the mailing list before embarking.

See the document Contributing to the Go project for more information about how to proceed.

Why does "go get" use HTTPS when cloning a repository?

Companies often permit outgoing traffic only on the standard TCP ports 80 (HTTP) and 443 (HTTPS), blocking outgoing traffic on other ports, including TCP port 9418 (git) and TCP port 22 (SSH). When using HTTPS instead of HTTP, `git` enforces certificate validation by default, providing protection against man-in-the-middle, eavesdropping and tampering attacks. The `go get` command therefore uses HTTPS for safety.

`Git` can be configured to authenticate over HTTPS or to use SSH in place of HTTPS. To authenticate over HTTPS, you can add a line to the `$HOME/.netrc` file that git consults:

```
machine github.com login USERNAME password APIKEY
```

For GitHub accounts, the password can be a personal access token.

`Git` can also be configured to use SSH in place of HTTPS for URLs matching a given prefix. For example, to use SSH for all GitHub access, add these lines to your `~/.gitconfig`:

```
[url "ssh://git@github.com/"]
        insteadOf = https://github.com/
```

How should I manage package versions using "go get"?

The Go toolchain has a built-in system for managing versioned sets of related packages, known as *modules*. Modules were introduced in Go 1.11 and have been ready for production use since 1.14.

To create a project using modules, run `go mod init`. This command creates a `go.mod` file that tracks dependency versions.

```
go mod init example/project
```

To add, upgrade, or downgrade a dependency, run `go get`:

```
go get golang.org/x/text@v0.3.5
```

See Tutorial: Create a module for more information on getting started.

See Developing modules for guides on managing dependencies with modules.

Packages within modules should maintain backward compatibility as they evolve, following the import compatibility rule:

> If an old package and a new package have the same import path,
> the new package must be backwards compatible with the old package.

The Go 1 compatibility guidelines are a good reference here: don't remove exported names, encourage tagged composite literals, and so on. If different functionality is required, add a

new name instead of changing an old one.

Modules codify this with semantic versioning and semantic import versioning. If a break in compatibility is required, release a module at a new major version. Modules at major version 2 and higher require a major version suffix as part of their path (like `/v2`). This preserves the import compatibility rule: packages in different major versions of a module have distinct paths.

Pointers and Allocation

When are function parameters passed by value?

As in all languages in the C family, everything in Go is passed by value. That is, a function always gets a copy of the thing being passed, as if there were an assignment statement assigning the value to the parameter. For instance, passing an `int` value to a function makes a copy of the `int`, and passing a pointer value makes a copy of the pointer, but not the data it points to. (See a later section for a discussion of how this affects method receivers.)

Map and slice values behave like pointers: they are descriptors that contain pointers to the underlying map or slice data. Copying a map or slice value doesn't copy the data it points to. Copying an interface value makes a copy of the thing stored in the interface value. If the interface value holds a struct, copying the interface value makes a copy of the struct. If the interface value holds a pointer, copying the interface value makes a copy of the pointer, but again not the data it points to.

Note that this discussion is about the semantics of the operations. Actual implementations may apply optimizations to avoid copying as long as the optimizations do not change the semantics.

When should I use a pointer to an interface?

Almost never. Pointers to interface values arise only in rare, tricky situations involving disguising an interface value's type for delayed evaluation.

It is a common mistake to pass a pointer to an interface value to a function expecting an interface. The compiler will complain about this error but the situation can still be confusing, because sometimes a pointer is necessary to satisfy an interface. The insight is that although a pointer to a concrete type can satisfy an interface, with one exception *a pointer to an interface can never satisfy an interface*.

Consider the variable declaration,

```
var w io.Writer
```

The printing function `fmt.Fprintf` takes as its first argument a value that satisfies `io.Writer`—something that implements the canonical `Write` method. Thus we can write

```
fmt.Fprintf(w, "hello, world\n")
```

If however we pass the address of `w`, the program will not compile.

```
fmt.Fprintf(&w, "hello, world\n") // Compile-time error.
```

The one exception is that any value, even a pointer to an interface, can be assigned to a variable of empty interface type (`interface{}`). Even so, it's almost certainly a mistake if the value is a pointer to an interface; the result can be confusing.

Should I define methods on values or pointers?

```
func (s *MyStruct) pointerMethod() { } // method on pointer
func (s MyStruct)  valueMethod()   { } // method on value
```

For programmers unaccustomed to pointers, the distinction between these two examples can be confusing, but the situation is actually very simple. When defining a method on a type, the receiver (`s` in the above examples) behaves exactly as if it were an argument to the method. Whether to define the receiver as a value or as a pointer is the same question, then, as whether a function argument should be a value or a pointer. There are several considerations.

First, and most important, does the method need to modify the receiver? If it does, the receiver *must* be a pointer. (Slices and maps act as references, so their story is a little more subtle, but for instance to change the length of a slice in a method the receiver must still be a pointer.) In the examples above, if `pointerMethod` modifies the fields of `s`, the caller will see those changes, but `valueMethod` is called with a copy of the caller's argument (that's the definition of passing a value), so changes it makes will be invisible to the caller.

By the way, in Java method receivers are always pointers, although their pointer nature is somewhat disguised (and there is a proposal to add value receivers to the language). It is the value receivers in Go that are unusual.

Second is the consideration of efficiency. If the receiver is large, a big `struct` for instance, it will be much cheaper to use a pointer receiver.

Next is consistency. If some of the methods of the type must have pointer receivers, the rest should too, so the method set is consistent regardless of how the type is used. See the section on method sets for details.

For types such as basic types, slices, and small `structs`, a value receiver is very cheap so unless the semantics of the method requires a pointer, a value receiver is efficient and clear.

What's the difference between new and make?

In short: `new` allocates memory, while `make` initializes the slice, map, and channel types.

See the relevant section of Effective Go for more details.

What is the size of an `int` on a 64 bit machine?

The sizes of `int` and `uint` are implementation-specific but the same as each other on a given platform. For portability, code that relies on a particular size of value should use an explicitly sized type, like `int64`. On 32-bit machines the compilers use 32-bit integers by default, while on 64-bit machines integers have 64 bits. (Historically, this was not always true.)

On the other hand, floating-point scalars and complex types are always sized (there are no `float` or `complex` basic types), because programmers should be aware of precision when using floating-point numbers. The default type used for an (untyped) floating-point constant is `float64`. Thus `foo := 3.0` declares a variable `foo` of type `float64`. For a `float32` variable initialized by an (untyped) constant, the variable type must be specified explicitly in the variable declaration:

```
var foo float32 = 3.0
```

Alternatively, the constant must be given a type with a conversion as in `foo := float32(3.0)`.

How do I know whether a variable is allocated on the heap or the stack?

From a correctness standpoint, you don't need to know. Each variable in Go exists as long as there are references to it. The storage location chosen by the implementation is irrelevant to the semantics of the language.

The storage location does have an effect on writing efficient programs. When possible, the Go compilers will allocate variables that are local to a function in that function's stack frame. However, if the compiler cannot prove that the variable is not referenced after the function returns, then the compiler must allocate the variable on the garbage-collected heap to avoid dangling pointer errors. Also, if a local variable is very large, it might make more sense to store it on the heap rather than the stack.

In the current compilers, if a variable has its address taken, that variable is a candidate for allocation on the heap. However, a basic *escape analysis* recognizes some cases when such variables will not live past the return from the function and can reside on the stack.

Why does my Go process use so much virtual memory?

The Go memory allocator reserves a large region of virtual memory as an arena for

allocations. This virtual memory is local to the specific Go process; the reservation does not deprive other processes of memory.

To find the amount of actual memory allocated to a Go process, use the Unix `top` command and consult the `RES` (Linux) or `RSIZE` (macOS) columns.

Concurrency

What operations are atomic? What about mutexes?

A description of the atomicity of operations in Go can be found in the Go Memory Model document.

Low-level synchronization and atomic primitives are available in the sync and sync/atomic packages. These packages are good for simple tasks such as incrementing reference counts or guaranteeing small-scale mutual exclusion.

For higher-level operations, such as coordination among concurrent servers, higher-level techniques can lead to nicer programs, and Go supports this approach through its goroutines and channels. For instance, you can structure your program so that only one goroutine at a time is ever responsible for a particular piece of data. That approach is summarized by the original Go proverb,

Do not communicate by sharing memory. Instead, share memory by communicating.

See the Share Memory By Communicating code walk and its associated article for a detailed discussion of this concept.

Large concurrent programs are likely to borrow from both these toolkits.

Why doesn't my program run faster with more CPUs?

Whether a program runs faster with more CPUs depends on the problem it is solving. The Go language provides concurrency primitives, such as goroutines and channels, but concurrency only enables parallelism when the underlying problem is intrinsically parallel. Problems that are intrinsically sequential cannot be sped up by adding more CPUs, while those that can be broken into pieces that can execute in parallel can be sped up, sometimes dramatically.

Sometimes adding more CPUs can slow a program down. In practical terms, programs that spend more time synchronizing or communicating than doing useful computation may experience performance degradation when using multiple OS threads. This is because passing data between threads involves switching contexts, which has significant cost, and that cost can increase with more CPUs. For instance, the prime sieve example from the Go specification has no significant parallelism although it launches many goroutines; increasing

the number of threads (CPUs) is more likely to slow it down than to speed it up.

For more detail on this topic see the talk entitled Concurrency is not Parallelism.

How can I control the number of CPUs?

The number of CPUs available simultaneously to executing goroutines is controlled by the `GOMAXPROCS` shell environment variable, whose default value is the number of CPU cores available. Programs with the potential for parallel execution should therefore achieve it by default on a multiple-CPU machine. To change the number of parallel CPUs to use, set the environment variable or use the similarly-named function of the runtime package to configure the run-time support to utilize a different number of threads. Setting it to 1 eliminates the possibility of true parallelism, forcing independent goroutines to take turns executing.

The runtime can allocate more threads than the value of `GOMAXPROCS` to service multiple outstanding I/O requests. `GOMAXPROCS` only affects how many goroutines can actually execute at once; arbitrarily more may be blocked in system calls.

Go's goroutine scheduler is not as good as it needs to be, although it has improved over time. In the future, it may better optimize its use of OS threads. For now, if there are performance issues, setting `GOMAXPROCS` on a per-application basis may help.

Why is there no goroutine ID?

Goroutines do not have names; they are just anonymous workers. They expose no unique identifier, name, or data structure to the programmer. Some people are surprised by this, expecting the `go` statement to return some item that can be used to access and control the goroutine later.

The fundamental reason goroutines are anonymous is so that the full Go language is available when programming concurrent code. By contrast, the usage patterns that develop when threads and goroutines are named can restrict what a library using them can do.

Here is an illustration of the difficulties. Once one names a goroutine and constructs a model around it, it becomes special, and one is tempted to associate all computation with that goroutine, ignoring the possibility of using multiple, possibly shared goroutines for the processing. If the `net/http` package associated per-request state with a goroutine, clients would be unable to use more goroutines when serving a request.

Moreover, experience with libraries such as those for graphics systems that require all processing to occur on the "main thread" has shown how awkward and limiting the approach can be when deployed in a concurrent language. The very existence of a special thread or goroutine forces the programmer to distort the program to avoid crashes and other problems caused by inadvertently operating on the wrong thread.

For those cases where a particular goroutine is truly special, the language provides features such as channels that can be used in flexible ways to interact with it.

Functions and Methods

Why do T and *T have different method sets?

As the Go specification says, the method set of a type `T` consists of all methods with receiver type `T`, while that of the corresponding pointer type `*T` consists of all methods with receiver `*T` or `T`. That means the method set of `*T` includes that of `T`, but not the reverse.

This distinction arises because if an interface value contains a pointer `*T`, a method call can obtain a value by dereferencing the pointer, but if an interface value contains a value `T`, there is no safe way for a method call to obtain a pointer. (Doing so would allow a method to modify the contents of the value inside the interface, which is not permitted by the language specification.)

Even in cases where the compiler could take the address of a value to pass to the method, if the method modifies the value the changes will be lost in the caller. As an example, if the `Write` method of `bytes.Buffer` used a value receiver rather than a pointer, this code:

```
var buf bytes.Buffer
io.Copy(buf, os.Stdin)
```

would copy standard input into a *copy* of `buf`, not into `buf` itself. This is almost never the desired behavior.

What happens with closures running as goroutines?

Some confusion may arise when using closures with concurrency. Consider the following program:

```
func main() {
    done := make(chan bool)

    values := []string{"a", "b", "c"}
    for _, v := range values {
        go func() {
            fmt.Println(v)
            done <- true
        }()
    }

    // wait for all goroutines to complete before exiting
    for _ = range values {
        <-done
    }
}
```

One might mistakenly expect to see `a, b, c` as the output. What you'll probably see instead is `c, c, c`. This is because each iteration of the loop uses the same instance of the variable `v`, so each closure shares that single variable. When the closure runs, it prints the value of `v` at the time `fmt.Println` is executed, but `v` may have been modified since the goroutine was launched. To help detect this and other problems before they happen, run `go vet`.

To bind the current value of `v` to each closure as it is launched, one must modify the inner loop to create a new variable each iteration. One way is to pass the variable as an argument to the closure:

```
    for _, v := range values {
        go func(u string) {
            fmt.Println(u)
            done <- true
        }(v)
    }
```

In this example, the value of `v` is passed as an argument to the anonymous function. That value is then accessible inside the function as the variable `u`.

Even easier is just to create a new variable, using a declaration style that may seem odd but works fine in Go:

```
    for _, v := range values {
        v := v // create a new 'v'.
        go func() {
            fmt.Println(v)
            done <- true
        }()
    }
```

This behavior of the language, not defining a new variable for each iteration, may have been a mistake in retrospect. It may be addressed in a later version but, for compatibility, cannot change in Go version 1.

Control flow

Why does Go not have the `?:` operator?

There is no ternary testing operation in Go. You may use the following to achieve the same result:

```
if expr {
    n = trueVal
} else {
    n = falseVal
}
```

The reason `?:` is absent from Go is that the language's designers had seen the operation used too often to create impenetrably complex expressions. The `if-else` form, although longer, is unquestionably clearer. A language needs only one conditional control flow construct.

Type Parameters

Why does Go have type parameters?

Type parameters permit what is known as generic programming, in which functions and data structures are defined in terms of types that are specified later, when those functions and data structures are used. For example, they make it possible to write a function that returns the minimum of two values of any ordered type, without having to write a separate version for each possible type. For a more in-depth explanation with examples see the blog post Why Generics?.

How are generics implemented in Go?

The compiler can choose whether to compile each instantiation separately or whether to compile reasonably similar instantiations as a single implementation. The single implementation approach is similar to a function with an interface parameter. Different compilers will make different choices for different cases. The standard Go 1.18 compiler ordinarily emits a single instantiation for every type argument with the same shape, where the shape is determined by properties of the type such as the size and the location of pointers that it contains. Future releases will experiment with the tradeoff between compile time, run-time efficiency, and code size.

How do generics in Go compare to generics in other languages?

The basic functionality in all languages is similar: it is possible to write types and functions using types that are specified later. That said, there are some differences.

Java

> In Java, the compiler checks generic types at compile time but removes the types at run time. This is known as type erasure. For example, a Java type known as `List<Integer>` at compile time will become the non-generic type `List` at run time. This means, for example, that when using the Java form of type reflection it is impossible to distinguish a value of type `List<Integer>` from a value of type `List<Float>`. In Go the reflection information for a generic type includes the full compile-time type information.
>
> Java uses type wildcards such as `List<? extends Number>` or `List<? super Number>` to implement generic covariance and contravariance. Go does not have

these concepts, which makes generic types in Go much simpler.

C++

Traditionally C++ templates do not enforce any constraints on type arguments, although C++20 supports optional constraints via concepts. In Go constraints are mandatory for all type parameters. C++20 concepts are expressed as small code fragments that must compile with the type arguments. Go constraints are interface types that define the set of all permitted type arguments.

C++ supports template metaprogramming; Go does not. In practice, all C++ compilers compile each template at the point where it is instantiated; as noted above, Go can and does use different approaches for different instantiations.

Rust

The Rust version of constraints is known as trait bounds. In Rust the association between a trait bound and a type must be defined explicitly, either in the crate that defines the trait bound or the crate that defines the type. In Go type arguments implicitly satisfy constraints, just as Go types implicitly implement interface types. The Rust standard library defines standard traits for operations such as comparison or addition; the Go standard library does not, as these can be expressed in user code via interface types.

Python

Python is not a statically typed language, so one can reasonably say that all Python functions are always generic by default: they can always be called with values of any type, and any type errors are detected at run time.

Why does Go use square brackets for type parameter lists?

Java and C++ use angle brackets for type parameter lists, as in Java `List<Integer>` and C++ `std::vector<int>`. However, that option was not available for Go, because it leads to a syntactic problem: when parsing code within a function, such as `v := F<T>`, at the point of seeing the `<` it's ambiguous whether we are seeing an instantiation or an expression using the `<` operator. This is very difficult to resolve without type information.

For example, consider a statement like

```
a, b = w < x, y > (z)
```

Without type information, it is impossible to decide whether the right hand side of the assigment is a pair of expressions (`w < x` and `y > z`), or whether it is a generic function instantiation and call that returns two result values (`(w<x, y>)(z)`).

It is a key design decision of Go that parsing be possible without type information, which seems impossible when using angle brackets for generics.

Go is not unique or original in using square brackets; there are other languages such as Scala that also use square brackets for generic code.

Why does Go not support methods with type parameters?

Go permits a generic type to have methods, but, other than the receiver, the arguments to those methods cannot use parameterized types. The methods of a type determines the interfaces that the type implements, but it is not clear how this would work with parameterized arguments for methods of generic types. It would require either instantiating functions at run time or instantiating every generic function for every possible type argument. Neither approach seems feasible. For more details, including an example, see the proposal. Instead of methods with type parameters, use top-level functions with type parameters, or add the type parameters to the receiver type.

Why can't I use a more specific type for the receiver of a parameterized type?

The method declarations of a generic type are written with a receiver that includes the type parameter names. Some people think that a specific type can be used, producing a method that only works for certain type arguments:

```
type S[T any] struct { f T }

func (s S[string]) Add(t string) string {
        return s.f + t
}
```

This fails with a compiler error like `operator + not defined on s.f (variable of type string constrained by any)`, even though the `+` operator does of course work on the predeclared type `string`.

This is because the use of `string` in the declaration of the method `Add` is simply introducing a name for the type parameter, and the name is `string`. This is a valid, if strange, thing to do. The field `s.f` has type `string`, not the usual predeclared type `string`, but rather the type parameter of `S`, which in this method is named `string`. Since the constraint of the type parameter is `any`, the `+` operator is not permitted.

Why can't the compiler infer the type argument in my program?

There are many cases where a programmer can easily see what the type argument for a generic type or function must be, but the language does not permit the compiler to infer it. Type inference is intentionally limited to ensure that there is never any confusion as to which type is inferred. Experience with other languages suggests that unexpected type inference can lead to considerable confusion when reading and debugging a program. It is always possible to specify the explicit type argument to be used in the call. In the future new forms of inference may be supported, as long as the rules remain simple and clear.

Packages and Testing

How do I create a multifile package?

Put all the source files for the package in a directory by themselves. Source files can refer to items from different files at will; there is no need for forward declarations or a header file.

Other than being split into multiple files, the package will compile and test just like a single-file package.

How do I write a unit test?

Create a new file ending in `_test.go` in the same directory as your package sources. Inside that file, `import "testing"` and write functions of the form

```
func TestFoo(t *testing.T) {
    ...
}
```

Run `go test` in that directory. That script finds the `Test` functions, builds a test binary, and runs it.

See the How to Write Go Code document, the `testing` package and the `go test` subcommand for more details.

Where is my favorite helper function for testing?

Go's standard `testing` package makes it easy to write unit tests, but it lacks features provided in other language's testing frameworks such as assertion functions. An earlier section of this document explained why Go doesn't have assertions, and the same arguments apply to the use of `assert` in tests. Proper error handling means letting other tests run after one has failed, so that the person debugging the failure gets a complete picture of what is wrong. It is more useful for a test to report that `isPrime` gives the wrong answer for 2, 3, 5, and 7 (or for 2, 4, 8, and 16) than to report that `isPrime` gives the wrong answer for 2 and therefore no more tests were run. The programmer who triggers the test failure may not be familiar with the code that fails. Time invested writing a good error message now pays off later when the test breaks.

A related point is that testing frameworks tend to develop into mini-languages of their own, with conditionals and controls and printing mechanisms, but Go already has all those capabilities; why recreate them? We'd rather write tests in Go; it's one fewer language to learn and the approach keeps the tests straightforward and easy to understand.

If the amount of extra code required to write good errors seems repetitive and overwhelming, the test might work better if table-driven, iterating over a list of inputs and

outputs defined in a data structure (Go has excellent support for data structure literals). The work to write a good test and good error messages will then be amortized over many test cases. The standard Go library is full of illustrative examples, such as in the formatting tests for the `fmt` package.

Why isn't *X* in the standard library?

The standard library's purpose is to support the runtime, connect to the operating system, and provide key functionality that many Go programs require, such as formatted I/O and networking. It also contains elements important for web programming, including cryptography and support for standards like HTTP, JSON, and XML.

There is no clear criterion that defines what is included because for a long time, this was the *only* Go library. There are criteria that define what gets added today, however.

New additions to the standard library are rare and the bar for inclusion is high. Code included in the standard library bears a large ongoing maintenance cost (often borne by those other than the original author), is subject to the Go 1 compatibility promise (blocking fixes to any flaws in the API), and is subject to the Go release schedule, preventing bug fixes from being available to users quickly.

Most new code should live outside of the standard library and be accessible via the `go` tool's `go get` command. Such code can have its own maintainers, release cycle, and compatibility guarantees. Users can find packages and read their documentation at godoc.org.

Although there are pieces in the standard library that don't really belong, such as `log/syslog`, we continue to maintain everything in the library because of the Go 1 compatibility promise. But we encourage most new code to live elsewhere.

Implementation

What compiler technology is used to build the compilers?

There are several production compilers for Go, and a number of others in development for various platforms.

The default compiler, `gc`, is included with the Go distribution as part of the support for the `go` command. `Gc` was originally written in C because of the difficulties of bootstrapping—you'd need a Go compiler to set up a Go environment. But things have advanced and since the Go 1.5 release the compiler has been a Go program. The compiler was converted from C to Go using automatic translation tools, as described in this design document and talk. Thus the compiler is now "self-hosting", which means we needed to face the bootstrapping problem. The solution is to have a working Go installation already in place,

just as one normally has with a working C installation. The story of how to bring up a new Go environment from source is described here and here.

`Gc` is written in Go with a recursive descent parser and uses a custom loader, also written in Go but based on the Plan 9 loader, to generate ELF/Mach-O/PE binaries.

At the beginning of the project we considered using LLVM for `gc` but decided it was too large and slow to meet our performance goals. More important in retrospect, starting with LLVM would have made it harder to introduce some of the ABI and related changes, such as stack management, that Go requires but are not part of the standard C setup. A new LLVM implementation is starting to come together now, however.

The `Gccgo` compiler is a front end written in C++ with a recursive descent parser coupled to the standard GCC back end.

Go turned out to be a fine language in which to implement a Go compiler, although that was not its original goal. Not being self-hosting from the beginning allowed Go's design to concentrate on its original use case, which was networked servers. Had we decided Go should compile itself early on, we might have ended up with a language targeted more for compiler construction, which is a worthy goal but not the one we had initially.

Although `gc` does not use them (yet?), a native lexer and parser are available in the `go` package and there is also a native type checker.

How is the run-time support implemented?

Again due to bootstrapping issues, the run-time code was originally written mostly in C (with a tiny bit of assembler) but it has since been translated to Go (except for some assembler bits). `Gccgo`'s run-time support uses `glibc`. The `gccgo` compiler implements goroutines using a technique called segmented stacks, supported by recent modifications to the gold linker. `Gollvm` similarly is built on the corresponding LLVM infrastructure.

Why is my trivial program such a large binary?

The linker in the `gc` toolchain creates statically-linked binaries by default. All Go binaries therefore include the Go runtime, along with the run-time type information necessary to support dynamic type checks, reflection, and even panic-time stack traces.

A simple C "hello, world" program compiled and linked statically using gcc on Linux is around 750 kB, including an implementation of `printf`. An equivalent Go program using `fmt.Printf` weighs a couple of megabytes, but that includes more powerful run-time support and type and debugging information.

A Go program compiled with `gc` can be linked with the `-ldflags=-w` flag to disable DWARF generation, removing debugging information from the binary but with no other loss

of functionality. This can reduce the binary size substantially.

Can I stop these complaints about my unused variable/import?

The presence of an unused variable may indicate a bug, while unused imports just slow down compilation, an effect that can become substantial as a program accumulates code and programmers over time. For these reasons, Go refuses to compile programs with unused variables or imports, trading short-term convenience for long-term build speed and program clarity.

Still, when developing code, it's common to create these situations temporarily and it can be annoying to have to edit them out before the program will compile.

Some have asked for a compiler option to turn those checks off or at least reduce them to warnings. Such an option has not been added, though, because compiler options should not affect the semantics of the language and because the Go compiler does not report warnings, only errors that prevent compilation.

There are two reasons for having no warnings. First, if it's worth complaining about, it's worth fixing in the code. (And if it's not worth fixing, it's not worth mentioning.) Second, having the compiler generate warnings encourages the implementation to warn about weak cases that can make compilation noisy, masking real errors that *should* be fixed.

It's easy to address the situation, though. Use the blank identifier to let unused things persist while you're developing.

```
import "unused"

// This declaration marks the import as used by referencing an
// item from the package.
var _ = unused.Item  // TODO: Delete before committing!

func main() {
    debugData := debug.Profile()
    _ = debugData // Used only during debugging.
    ....
}
```

Nowadays, most Go programmers use a tool, goimports, which automatically rewrites a Go source file to have the correct imports, eliminating the unused imports issue in practice. This program is easily connected to most editors to run automatically when a Go source file is written.

Why does my virus-scanning software think my Go distribution or compiled binary is infected?

This is a common occurrence, especially on Windows machines, and is almost always a

false positive. Commercial virus scanning programs are often confused by the structure of Go binaries, which they don't see as often as those compiled from other languages.

If you've just installed the Go distribution and the system reports it is infected, that's certainly a mistake. To be really thorough, you can verify the download by comparing the checksum with those on the downloads page.

In any case, if you believe the report is in error, please report a bug to the supplier of your virus scanner. Maybe in time virus scanners can learn to understand Go programs.

Performance

Why does Go perform badly on benchmark X?

One of Go's design goals is to approach the performance of C for comparable programs, yet on some benchmarks it does quite poorly, including several in golang.org/x/exp/shootout. The slowest depend on libraries for which versions of comparable performance are not available in Go. For instance, pidigits.go depends on a multi-precision math package, and the C versions, unlike Go's, use GMP (which is written in optimized assembler). Benchmarks that depend on regular expressions (regex-dna.go, for instance) are essentially comparing Go's native regexp package to mature, highly optimized regular expression libraries like PCRE.

Benchmark games are won by extensive tuning and the Go versions of most of the benchmarks need attention. If you measure comparable C and Go programs (reverse-complement.go is one example), you'll see the two languages are much closer in raw performance than this suite would indicate.

Still, there is room for improvement. The compilers are good but could be better, many libraries need major performance work, and the garbage collector isn't fast enough yet. (Even if it were, taking care not to generate unnecessary garbage can have a huge effect.)

In any case, Go can often be very competitive. There has been significant improvement in the performance of many programs as the language and tools have developed. See the blog post about profiling Go programs for an informative example.

Changes from C

Why is the syntax so different from C?

Other than declaration syntax, the differences are not major and stem from two desires. First, the syntax should feel light, without too many mandatory keywords, repetition, or arcana. Second, the language has been designed to be easy to analyze and can be parsed without a symbol table. This makes it much easier to build tools such as debuggers,

dependency analyzers, automated documentation extractors, IDE plug-ins, and so on. C and its descendants are notoriously difficult in this regard.

Why are declarations backwards?

They're only backwards if you're used to C. In C, the notion is that a variable is declared like an expression denoting its type, which is a nice idea, but the type and expression grammars don't mix very well and the results can be confusing; consider function pointers. Go mostly separates expression and type syntax and that simplifies things (using prefix `*` for pointers is an exception that proves the rule). In C, the declaration

```
int* a, b;
```

declares `a` to be a pointer but not `b`; in Go

```
var a, b *int
```

declares both to be pointers. This is clearer and more regular. Also, the `:=` short declaration form argues that a full variable declaration should present the same order as `:=` so

```
var a uint64 = 1
```

has the same effect as

```
a := uint64(1)
```

Parsing is also simplified by having a distinct grammar for types that is not just the expression grammar; keywords such as `func` and `chan` keep things clear.

See the article about Go's Declaration Syntax for more details.

Why is there no pointer arithmetic?

Safety. Without pointer arithmetic it's possible to create a language that can never derive an illegal address that succeeds incorrectly. Compiler and hardware technology have advanced to the point where a loop using array indices can be as efficient as a loop using pointer arithmetic. Also, the lack of pointer arithmetic can simplify the implementation of the garbage collector.

Why are `++` and `--` statements and not expressions? And why postfix, not prefix?

Without pointer arithmetic, the convenience value of pre- and postfix increment operators drops. By removing them from the expression hierarchy altogether, expression syntax is simplified and the messy issues around order of evaluation of `++` and `--` (consider `f(i++)` and `p[i] = q[++i]`) are eliminated as well. The simplification is significant. As for postfix vs. prefix, either would work fine but the postfix version is more traditional; insistence on

prefix arose with the STL, a library for a language whose name contains, ironically, a postfix increment.

Why are there braces but no semicolons? And why can't I put the opening brace on the next line?

Go uses brace brackets for statement grouping, a syntax familiar to programmers who have worked with any language in the C family. Semicolons, however, are for parsers, not for people, and we wanted to eliminate them as much as possible. To achieve this goal, Go borrows a trick from BCPL: the semicolons that separate statements are in the formal grammar but are injected automatically, without lookahead, by the lexer at the end of any line that could be the end of a statement. This works very well in practice but has the effect that it forces a brace style. For instance, the opening brace of a function cannot appear on a line by itself.

Some have argued that the lexer should do lookahead to permit the brace to live on the next line. We disagree. Since Go code is meant to be formatted automatically by `gofmt`, *some* style must be chosen. That style may differ from what you've used in C or Java, but Go is a different language and `gofmt`'s style is as good as any other. More important—much more important—the advantages of a single, programmatically mandated format for all Go programs greatly outweigh any perceived disadvantages of the particular style. Note too that Go's style means that an interactive implementation of Go can use the standard syntax one line at a time without special rules.

Why do garbage collection? Won't it be too expensive?

One of the biggest sources of bookkeeping in systems programs is managing the lifetimes of allocated objects. In languages such as C in which it is done manually, it can consume a significant amount of programmer time and is often the cause of pernicious bugs. Even in languages like C++ or Rust that provide mechanisms to assist, those mechanisms can have a significant effect on the design of the software, often adding programming overhead of its own. We felt it was critical to eliminate such programmer overheads, and advances in garbage collection technology in the last few years gave us confidence that it could be implemented cheaply enough, and with low enough latency, that it could be a viable approach for networked systems.

Much of the difficulty of concurrent programming has its roots in the object lifetime problem: as objects get passed among threads it becomes cumbersome to guarantee they become freed safely. Automatic garbage collection makes concurrent code far easier to write. Of course, implementing garbage collection in a concurrent environment is itself a challenge, but meeting it once rather than in every program helps everyone.

Finally, concurrency aside, garbage collection makes interfaces simpler because they don't need to specify how memory is managed across them.

This is not to say that the recent work in languages like Rust that bring new ideas to the problem of managing resources is misguided; we encourage this work and are excited to see how it evolves. But Go takes a more traditional approach by addressing object lifetimes through garbage collection, and garbage collection alone.

The current implementation is a mark-and-sweep collector. If the machine is a multiprocessor, the collector runs on a separate CPU core in parallel with the main program. Major work on the collector in recent years has reduced pause times often to the sub-millisecond range, even for large heaps, all but eliminating one of the major objections to garbage collection in networked servers. Work continues to refine the algorithm, reduce overhead and latency further, and to explore new approaches. The 2018 ISMM keynote by Rick Hudson of the Go team describes the progress so far and suggests some future approaches.

On the topic of performance, keep in mind that Go gives the programmer considerable control over memory layout and allocation, much more than is typical in garbage-collected languages. A careful programmer can reduce the garbage collection overhead dramatically by using the language well; see the article about profiling Go programs for a worked example, including a demonstration of Go's profiling tools.