Table of Contents

> **Please go to effective-go-zh-en@Github or effective-go-zh-en@Gitbook which are more complete.**

Reference from Offical Effective Go

Reference from qqbuby's Effective Go

Fork me on GitHub

Overview

Go is a new language. Although it borrows ideas from existing languages, it has unusual properties that make an effective Go program different in character from programs written in its relatives. A straightforword tranlation of a C++ or Java program into Go is unlikely to produce a satisfactory result-Java programs are written in Java, not Go. In other words, to write Go well, it's important to understand its properties and idioms. It's also important to know the established conventions for programming in Go, such as naming, formatting, program construction, and so on, so that programs you write will be easy for other Go programmers to understand.

This document gives tips for writing clear, idiomatic Go code.

Formatting

Formatting issues are the most contentious but the leat consequential. People can adapt to different formatting styles but it's better if they don't have to, and less time is devoted to the topic if every one adheres to the same style. With Go we take an unusual approach and let the machine take care of most formatting issues. The **gofmt** program (also avaiable as **go fmt**, which operates at the package level rather than source file level) reads a Go program and emits the source in a standard style of indentation and vertical alignment, retaining and if necessary reformatting comments.

All Go code in the standard packages has been formatted with *gofmt*.

Some formatting details remain. Very briefly:

- Indentation

  We use tabs for indentation and *gofmt* emits them by default. Use spaces only if you must.

- Line lenght

  Go has no line limit. If a line feels two long, wrap it and indent with and extra tab.

- Parentheses

  Go needs fewer parentheses than C and Java: control structures (**if**, **for**, **switch**) do not have parentheses in their syntax.

Commentary

Go provides C-style / / block comments and C++-style // line comments. Line comments are the norm; block comments appear mostly as package comments, but are useful within an expression or to disable large swaths of code.

The program—and web server—**godoc** processes Go source files to extract documentation about the contents of the package. Comments that appear before top-level declarations, with no intervening newlines, are extracted along with the declaration to serve as explanatory text for the item. The nature and style of these comments determines the quality of the documentation **godoc** produces.

Every package should have a *package comment*, a block comment preceding the package clause. For multi-file packages, the package comment only needs to be present in one file, and any one will do.

Comments do not need extra formatting such as banners of stars. The generated output may not even be presented in a fixed-width font, so don't depend on spacing for alignment —**godoc**, like **gofmt**, takes care of that. The comments are uninterpreted plain text, so HTML and ohter annotations such **_this_** will reproduce *verbatim* and should not be used.

Every exported (capitalized) name in a program should have a doc comment.

Go's declaration syntax allows grouping of declarations. A single doc comment can introduce a group of related constants or variables.

```
// Error codes returned by failures to parse an expression.
var (
    ErrInternal      = errors.New("regexp: internal error")
    ErrUnmatchedLpar = errors.New("regexp: unmatched '('")
    ErrUnmatchedRpar = errors.New("regexp: unmatched ')'")
    ...
)
```

Names

Names are as important in Go as in any other language. They even have semantic effect: *the visibility of a name outside a package is determined by whether its first character is upper case*.

Package names

When a package is imported, the package name becomes an accessor for the contents. After

```
import "bytes"
```

the import package can talk about `bytes.Buffer` .

Packages name should be good: short, concise, evocative. By convention, packages are given lower case, single-word name; there should be no need for underscores or mixedCaps.

Another convention is that the package name is the base name of its source direcotry; the package in *src/encoding/base64* is imported as "encoding/base64" but has name base64, not encoding_base64 and not encodingBase64.

The importer of a package will use the name to refer to its contents, so exported names in the package can use that fact to avoid stutter.

Getters

Go doesn't provide automatic support for getters and setters. There's nothing wrong with providing getters and setters yourself, and it's often appropriate to do so, but it's neither idomatic nor necessary to put *Get* into the getter's name. If you have a field called *owner* (lower case, unexported), the getter method should be called *Owner* (upper case, exported), not *GetOwner*. The use of upper-case names for export provides the hook to discriminate the field from the method. A setter function, if needed, will likely be called *SetOwner*. Both names read well in pratice:

```
owner := obj.Owner()
if owner != user {
    obj.SetOwner(user)
}
```

Interface names

By convention, one-method interfaces are named by the method name plus an *-er* suffix or similar modification to construct an agent noun: Reader, Writer, Formatter, CloseNotifier etc.

To avoid confusion, don't give your method one of those names unless it has the same signature and meaning. Conversely, if your type implements a method with the same meaning as a method on a well-known type, give it the same name and signature; call your string-converter method *String* not *ToString*.

MixedCaps

Finally, the convention in Go is to use MixedCaps or mixedCaps rather than underscores to write multiword names.

Semicolons

Like C, Go's formal grammar uses semicolons to terminate statements, but unkie in C, those semicolons do not appear in the source. Instead the lexer uses a simple rule to insert semicolons automatically as it scans, so the input text is mostly free of them. The rule is this. If the last token before a newline is an identifier (which includes words like `int` and `float63`), a basic literal such as a number or string constant, or one of the tokens

```
break continue fallthrough return ++ -- ) }
```

the lexer always inserts a semicolon after the token. This could be summarized as, "if the newline comes after a token that could end a statement, instert a semicolon".

A semicolon can also be ommited immediately before a closing brace, so a statement such as

```
go func() { for { dst <- <-src }}()
```

needs no semicolons. Idiomatic Go programs have semicolons only in places such as for loop clauses, to separate the intializer, condition, and continuation elements. They are also necessary to separate mulitple statements on a line, should you write code that way.

One consequence of the smicolon insertion rules is that you cannot put the opening brace of a control structure (if, for, switch, or select) on the next line. If you do, a semicolon will be inserted before the brace, which could cause unwanted effects. Write them like this

```
if i < f() {
    g()
}
```

not like

```
if i < f()  // wrong!
{           // wrong!
    g()
}
```

Control structures

The control structures of Go are related to those of C but differ in important ways. There is no *do* or *while* loop, only a slightly generalized *for*; *switch* is more flexible; *if* and *switch* accept an optional initialization statement like that of *for*; *break* and *continue* statements take an optional label to identify what to break or continue; and there are new control structures including a type switch and a multiway communications multiplexer, *select*. The syntax is also slightly different: *there are no parentheses and the bodies must always be brace-delimited*.

If

In Go a simple if looks like this:

```
if x > 0 {
    return y
}
```

Mandatory braces encourage writing simple *if* statements on multiple lines. It's good style to do so anyway, especially when the body contains a control statement such as a *return* or *break*.

Since *if* and *switch* accept an intialization statement, it's common to see one used to set up a local variable.

```
if err := file.Chmod(0664); err != nil {
    log.Print(err)
    return err
}
```

In the Go libraries, when an *if* statement doesn't flow into the next statement—that is, the body ends in *break*, *continue*, *goto*, or *return*—the unnecessary *else* is omitted.

```
f, err := os.Open(name)
if err != nil {
    return err
}
codeUsing(f)
```

Redeclaration and reassignment

In a **:=** declaration a variable *v* may appear even if it has already been declared, provided:

- this delcaration is in the same scope as the existing declaration of *v* (if *v* is already declared in an outer scope, the declaration will create a new variable §),
- the corresponding value in the intialization is assignable to *v*, and
- there is at least one other variable in the declaration that is being declared anew.

§ It's worth noting here that in Go the scope of function parameters and return values is the same as the function body, even though they appear lexically outside the braces that enclose the body.

For

The Go *for* loop is similar to—but not the same as—C's. It unifies *for* and *while* and there is no *do-while*.

There are three forms, only one of which has semicolons.

```
// Like a C for
for init; condition; post { }

// Like a C while
for condition { }

// Like a C for(;;)
for { }
```

Short declarations make it easy to declare the index variable right in the loop.

```
sum := 0
for i := 0; i < 10; i++ {
    sum += i
}
```

If you're looping over an *array*, *slice*, *string*, or *map*, or reading from a *channel*, a **range** clause can manage the loop.

```
for key, value := range oldMap {
    newMap[key] value
}
```

If you only need the first item in the range (the key or index), drop the second:

```
for key := range m {
    if key.expired() {
        delete(m, key)
    }
}
```

If you only need the second item in the range (the value), use the *blank indeifier*, an underscore, to discard the first:

```
sum := 0
for _, value := range array {
    sum += value
}
```

Finally, Go has no comma operator and ++ and -- are statements not expressions. Thus if you want to run multiple variables in a *for* you should use parallel assignment (although that precludes ++ and --).

```
// Reverse a
for i, j  := 0, len(a)-1; i < j; i, j = i+1, j-1 {
    a[i], a[j] = a[j], a[i]
}
```

Switch

Go's *switch* is more general than C's. The expressoins need not be constants or even integers, the cases are evaluated top to bottom until a match is found, and if the *switch* has no expression it switchs on *true*. It's therefore possiable—and idomatic—to write and *if-else-if-else* chain as a *switch*.

There is no automatic fall through, but cases can be presented in comma-separated lists.

```
func shouldEscape(c byte) bool {
    switch c {
        case ' ', '?', '&', '=', '#', '+', '%':
            return true
    }
    return false
}
```

Although they are not nearly as common in Go as some other C-like languages, *break* statements can be used to terminate a *switch* early.

Of course, the *continue* statement also accepts an optional label but it applies only to loops.

```
// Compare returns an integer comparing the two byte slices,
// lexicographically.
// The result will be 0 if a == b, -1 if a < b, and +1 if a > b
func Compare(a, b []byte) int {
    for i := 0; i < len(a) && i < len(b); i++ {
        switch {
        case a[i] > b[i]:
            return 1
        case a[i] < b[i]:
            return -1
        }
    }
    switch {
    case len(a) > len(b):
        return 1
    case len(a) < len(b):
        return -1
    }
    return 0
}
```

Type switch

A *switch* can also be used to discover the dynamic type of and interface variable. Such a *type switch* uses the syntax of a *type assertion* with the keyworkd `type` inside the parenthese. If the switch declares a variable in the expressoin, the variable will have the corresponding type in each caluse. It's also idiomatic to reuse the name in such case, in effect declaring a new variable with the same name but a different type in each case.

```
var t interface{}
t = functionOfSomeType()
switch t := t.(type) {
default:
    fmt.Printf("unexpected type %T\n", t)     // %T prints  whatever type t has
case bool:
    fmt.Printf("boolean %t\n", t)             // t has type     bool
case int:
    fmt.Printf("integer %d\n", t)             // t has type     int
case *bool:
    fmt.Printf("pointer to boolean %t\n", *t) // t has type     *bool
case *int:
    fmt.Printf("pointer to integer %d\n", *t) // t has type     *int
}
```

Functions

Multiple return values

One of Go's unusual features is that functions and methods can return multiple values. This form can be used to improve on a couple of clumsy idioms in C programs: in-band error returns such as -1 for EOF and modifying an argument passed by address.

In C, a write error is signaled by a negative count with the error code secreted away in a volatile location. In Go, `Write` can return a count *and* an error: "Yes, you wrote some bytes but not all of them because you filled the device". The signature of the `Write` method on files form package `os` is:

```
func (file *File) Write(b []byte) (n int, err error)
```

and as the documentation says, it returns the number of bytes written and a non-nil `error` when `n != len(b)` .

Named result parameters

The return or result "parameters" of a Go function can be given names and used as regular variables, just like the incoming parameters. When named, they are initialized to the zero values of their types when the funciton begins; if the function executes a *return* statement with no arguments, the current values of the result parameters are used as the returned values.

Because named results are initialized and tied to an unadorned return, they can simplify as well as clarify.

```
func ReadFull(r Reader, buf []byte) (n int, err error) {
    for len(buf) > 0 && err == nil {
        var nr int
        nr, err = r.Read(buf)
        n += nr
        buf = buf[nr:]
    }
    return
}
```

Defer

Go's *defer* statement schedules a function call (the *deferred* function) to be run immediately before the function executing the *defer* returns. It's an unusual but effective way to deal with situations such as resources that must be released regardless of which path a function takes to return.

```
// Contents returns the file's contents as a string
func Contents(filename string) (string, error) {
    f, err := os.Open(filename)
    if err := nil {
        return "", err
    }
    defer f.Close()     // f.Close will run when we're finished.
    //to do sth...
    return "content", nil
}
```

The arguments to the deferred function (which include the receiver if the function is a method) are evaluated when the *defer* executes, not the the *call* executes.

```
for i := 0; i < 5; i++ {
    defer fmt.Printf("%d ", i)
}
```

Deferred functions are executed in LIFO order, so this code cause 4 3 2 1 0 to be printed when function returns. A more plausible example is a simple way to trace function execution through the program. We could write a couple of simple tracing routines like this:

```
func trace(s string)    { fmt.Println("entering:", s) }
func untrace(s string)  { fmt.Println("leaving:", s) }

// Use them like this:
func a() {
    trace("a")
    defer untrace("a")
    // do something....
}
```

Data

Allocation

Go has two allocation primitives, the built-in functions `new` and `make`. They do different things and apply to different types.

Allocation with new

new, a built-in function that allocates memory, but unlike its namesakes in some other languages it does not *intialize* the memory, it only *zeros* it. That is, `new(T)` allocates zeroed storage for a new item of type `T` and return its address, a value of type `*T`. In Go terminology, it returns a pointer to a newly allocated zero value of type `T`.

Since the memory returned by `new` is zeroed, it's helpful to arrange when designing your data structures that the zero of each type can be used without further initialization. This means a user of the data structure can create one with `new` and get right to work.

```
type SyncedBuffer struct {
    lock    sync.Mutex      // the zero value for a sync.Mutex is defined to be an unl
ocked mutex.
    buffer  sync.Buffer     // the zero value for Buffer is an empty buffer ready to u
se.
}
```

Values of type `SyncedBuffer` are also ready to use immediately upon allocation or just declaration. In the next snippet, both `p` and `v` will work correctly without further arrangement.

```
p := new(SyncedBuffer)      // type *SyncedBuffer
var v SyncedBuffer          // type  SyncedBuffer
```

Constructors and compsite literals

Sometimes the zero value isn't good enough and an initializing constructor is necessary, as in this example derived form package `os`.

```
func NewFile(fd int, name string) *File {
    if fd < 0 {
        return nil
    }
    f := new(File)
    f.fd = fd
    f.name = name
    f.dirinfo = nil
    f.nepipe = 0
    return f
}
```

There's a lot of boiler plate in there. We can simplify it using a *composite literal*, which is an expression that creates a new instance each time it is evaluated.

```
func NewFile(fd int, name string) *File {
    if fd < 0 {
        return nil
    }
    f := File{fd, name, nil, 0}
    return &f
}
```

In fact, taking the address of a composite literal allocates a fresh instance each time it is evaluated, so we can combine these last two lines.

```
return &File{fd, name, nil, 0}
```

The fields of a composite literal are laid out in order and must all be present. However, by labeling the elements explicitly as *field:value* pairs, the intializers can appear in any order, with the missing ones left as their respective zero values. Thus we could say

```
return &File{fd: fd, name: name}
```

As a limiting case, if a composite literal contains no fields at all, it create a zero value for the type. The expression `new(File)` and `&File{}` are equivalent.

Composite literals can also be created for *arrays*, *slices*, and *maps*, with the field labels being indices or map keys as appropriate.

```
a := [...]string   {Enone: "no error", Eio: "Eio", Einval: "invalid argument"}
s := []string      {Enone: "no error", Eio: "Eio", Einval: "invalid argument"}
m := map[int]string{Enone: "no error", Eio: "Eio", Einval: "invalid argument"}
```

Allocation with make

The built-in function `make(T, args)` serves a purpos different from `new(T)`. It creates *slices*, *maps*, and *channels* only, and it returns an *intialized* (not *zeroed*) value of type `T` (not `*T`). The reason for the distinction is that these three types represent, under the covers, references to data structures that must be initialized before use. A slice, for example, is a three-item descriptor containing a pointer to the data (inside an array), the lenght, and the capacity, and until those items are intialized, the slice is `nil`. For slices, maps, and channels, `make` intializes the internal data structure and prepares the value to use.

```
var p *[]int = new([]int)       // allocates slice structure; *p == nil; rarely useful
var v  []int = make([]int, 100) // the slice v now refers to a new array of 100 ints

// Unnecessarily complex:
var p *[]int = new([]int)
*p = make([]int, 100, 100)

// Idiomatic:
v := make([]int, 100)
```

Remember that `make` applies only to maps, slices and channels and does not return a pointer. To obtain an explicit pointer allocate with `new` or take the address of a variable explicitly.

Arrays

Arrays are useful when planning the detailed layout of memory and sometimes can help avoid allocation, but primarily they are a building block for slices.

There are major differences between the ways arrays work in Go and C. In Go,

- Arrays are values. Assigning one array to another copies all the elements.
- In particluar, if you pass an array to a function, it will receive a *copy* of the array, not a pointer to it.
- The size of an array is part of its type. The type `[10]int` and `[20]int` are distinct.

If you want C-like behavior and efficiency, you can pass a pointer to the array, but even this style isn't idomatic Go. Use slcies instead.

Slices

Slices wrap arrays to give a more general, powerful, and convenient interface to sequences of data. Except for items with explicit dimension such as transformation matrices, most array programming in Go is done with slcies rather than simple arrays.

Slices hold references to an underlying array, and if you assign one slice to another, both refer to the same array. If a function takes a slice argument, changes it makes to the elements of the slice will be visible to the caller, analogous to passing a pointer to the underlying array. A `Read` function can therefore accept a slice argument rather than a pointer and a count; the length within the slice sets an upper limit of how much data to read.

```
func (f *File) Read(buf []byte) (n int, err error)
```

The method returns the number of bytes read and an error value, if any. To read into the first 32 bytes of a larger buffer `buf`, *slice* (here used as a verb) the buffer.

```
n, err := f.Read(buf[0:32])
```

The length of a slice may be changed as long as it still fits within the limits of the underlying array; just assign it to a slice of itself. The *capacity* of a slice, accessible by the built-in function `cap`, reports the maximum length the slice may assume.

Two-dimensional slices

Go's arrays and slices are one-dimensinal. To create the equivalent of a 2D array or slice, it is necessary to define an array-of-arrays or slice-of-slices, like this:

```
type Transform [3][3]float64  // A 3x3 array, really an array of arrays.
type LinesOfText [][]byte     // A slice of byte slices.
```

Because slices are variable-length, it is possible to have each inner slice be a different length.

```
text := LinesOfText{
    []byte("Now is the time"),
    []byte("for all good gophers"),
    []byte("to bring some fun to the party."),
}
```

Sometimes it's necessary to allocate a 2D slice, a situation that can arise when processing scan lines of pixels, for instance. There are two ways to achieve this. One is to allocate each slice independently; the other is to allocate a single array and point the individual slices into it.

Maps

Maps are a convenient and powerful built-in data structure that associate values of one type (the *key*) with values of another type (the *element* or *value*). The key can be of any type for which the equality operator is defined, such as integers, floating point and complex numbers, strings, pointers, interfaces (as long as the dynamic type supports equality), structs and arrays. Slices cannot be used as map keys, because equality is not defined on them. Like slices, maps hold references to an underlying data structure.

Maps can be constructed using the usual composite literal syntax with colon-separated key-value pairs.

```
var timeZone = map[string]int{
    "UTC":  0*60*60,
    "EST": -5*60*60,
    "CST": -6*60*60,
    "MST": -7*60*60,
    "PST": -8*60*60,
}
```

Assigining and fetching map values looks syntactically just like doing the same for arrays and slices except that the index doesn't need to be an integer.

```
offset := timeZone["EST"]
```

An attempt to fetch a map value with a key that is not present in the map will return zero value for the type of the entries in the map.

Sometimes you need to distinguish a missing entry from a zero value with a form of multiple assignment.

```
var seconds int
var ok bool
seconds, ok = timeZone[tz]
```

For obvious reasons this is called the "comma ok" idiom. In this example, if `tz` is present, `seconds` will be set appropriately and `ok` will be true; if not, `seconds` will be set to zero and `ok` will be false.

To test for presence in the map without worrying about the actual value, you can use the *blank identifier (_)* in place of the usual variable for the value.

```
_, present := timeZone[tz]
```

To delete a map entry, use the `delete` built-in function, whose arguments are the map and the key to be deleted. If's safe to do this even if the key is already absent from the map.

```
delete(timeZone, "PDT")     // Now on Standard Time
```

Printing

Formatted printing in Go uses a style similar to C's `printf` family but is richer and more general. The functions live in the `fmt` package and have capitalized names: `fmt.Printf`, `fmt.Fprintf`, `fmt.Sprintf` and so on. The string functions (Sprintf etc.) return a string rather than filling in a provided buffer.

You don't need to provide a format string. For each of `Printf`, `Fprintf`, and `Sprintf` there is another pair of functions, for instance `Print` and `Println`. These functions do not take a format string but instead generate a default format for each argument. The `Println` versions also insert a blank between arguments and append a newline to the output while the `Print` versions add blanks only if the operand on neither side is a string.

```
fmt.Printf("Hello %d\n", 23)
fmt.Fprintf(os.Stdout, "Hello ", 23, "\n")
fmt.Println("Hello", 23)
fmt.Println(fmt.Sprint("Hello ", 23))
```

The formatted print function `fmt.Fprint` and friends take as a first argument any object that implements the `io.Writer` interface; the variables `os.Stdout` and `os.Stderr` are familiar instances.

Here things start to diverge from C. First, the numeric formats such as `%d` do not take flags for signedness or size; instead, the printing routines use the type of the arguement to decide these properties.

```
var x uint64 = 1<<64 - 1
fmt.Printf("%d %x; %d %x\n", x, x, int64(x), int64(x))
```

prints

```
18446744073709551615 ffffffffffffffff; -1 -1
```

If you just want the default conversion, such as decimal for integers, you can use the catchall format `%v` (for "value"); the result is exactly what `Print` and `Println` would produce. Moreover, that format can print *any* value, even arrays, slices, structs, and maps.

When printing a struct, the modified format `%+v` annotates the fields of the structure with their names, and for any value the alternate format `%#v` prints the value in full Go syntax.

```
type T struct {
    a int
    b float64
    c string
}
t := &T{ 7, -2.35, "abc\tdef" }
fmt.Printf("%v\n", t)
fmt.Printf("%+v\n", t)
fmt.Printf("%#v\n", t)
```

prints

```
&{7 -2.35 abc   def}
&{a:7 b:-2.35 c:abc     def}
&main.T{a:7, b:-2.35, c:"abc\tdef"}
```

(Note the ampersands.) That quoted string format is also available through `%q` when applied to a value of type `string` or `[]byte`. Also, `%x` works on strings, byte arrays and byte slices as well as on integers, generating a long hexadecimal string, and with a space in the format (% x) it puts space between the bytes.

Another handy format is `%T`, which prints the *type* of a value.

```
fmt.Printf("%T\n", timeZone)
```

prints

```
map[string] int
```

If you want to control the default format for a custom type, all that's required is to define a method with the signature `String() string` on the type.

```
func (t *T) String() string {
    return fmt.Sprintf("%d/%g/%q", t.a, t.b, t.c)
}
fmt.Printf("%v\n", t)
```

to print in the format

```
7/-2.35/"abc\tdef"
```

(If you need to print *values* of type `T` as well as pointers to `T`, the receiver for `String` must be of value type; used a pointer because that's more efficient and idiomatic for struct types.)

Our `String` method is able to call `Sprintf` because the print routines are fully reentrant and can be wrapped this way. There is one important detail to understand about this approach, however: don't construct a `String` method by calling `Sprintf` in a way that will recur into your `String` method indefinitely. This can happen if the `Sprintf` call attempts to print the receiver directly as a string, which in turn will invoke the method again. It's a common and easy mistake to make, as this example shows.

```
type MyString string

func (m MyString) String() string {
    return fmt.Sprintf("MyString=%s", m)    // Error: will recur forever.
}
```

It's also easy to fix: convert the argument to the basic string type, which does not have the method.

```
type MyString string
func (m MyString) String() string {
    fmt.Sprintf("MyString=%s", string(m))   // OK: note conversion.
}
```

Another printing technique is to pass a print routine's arguments directly to another such routine. The signature of `Printf` uses the type `...interface{}` for its final argument to specify that an arbitrary number of parameters (of arbitrary type) can appear after the format.

```
func Printf(format string, v ...interface{}) (n int, err error) {
```

Within the function `Printf`, `v` acts like a variable of type `[]interface{}` but if it is passed to another variadic function, it acts like a regular list of arguments. Here is the implementation of the function `log.Println` we used above. It passes its arguments directly to `fmt.Sprintln` for the actual formatting.

```
// Println prints to the standard logger in the manner of fmt.Println.
func Println(v ...interface{}) {
    std.Output(2, fmt.Sprintln(v...))   // Output takes parameters (int, string)
}
```

We write **...** after `v` in the nested call to `Sprintln` to tell the compiler to treat `v` as a list of arguments; otherwise it would just past `v` as a single slice argument.

By the way, a **...** parameter can be of a specific type, for instance `...int` for a min function:

```
// Min function that chooses the least of a list of integers
func Min(a ...int) int {
    min := int(^uint(0) >> 1)   // largest int
    for _, i : range a {
        if i < min {
            min = i
        }
    }
    return min
}
```

Append

The signature of bulti-in funciton `append` as blows:

```
func append(slice []T, elements ...T) []T
```

where `T` is a placeholder for any give type. You cann't actually write a funciton in Go where the type `T` is determined by the caller. That's why `append` is built in: it needs support from the compiler.

What `append` does is append the elements to the end of the slice and return the result. The result needs to be returned because the underlying array may change.

```
x := []int{1, 2, 3}
x = append(x, 4, 5, 6)
```

So `append` works a little like `Printf`, collecting an arbitrary number of arguments.

Intialization

Although it doesn't llok superficially very different from initialization in C or C++, intialization in Go is more powerful. Complex structures can be built during intialziation and the ordering issues among initialized objects, even among different packages, are handled correctly.

Constants

Constants in Go are just that-constant. They are created at compile time, even when defined as locals in fucntions, and can only be numbers, characters (runes), strings or booleans. Because of the compile-time restriction, the expressions that define them must be constant expressions, evalutable by the compiler. For instance, `1<<3` is a constant expression, while `math.Sin(math.Pi/4)` is not because the function call to `math.Sin` needs to happen at run time.

In Go, enumerated constants are created using the **iota** enumerator. Since **iota** can be part of an expression and expressions can be implicitly repeated, it is easy to build intricate sets of values.

```
type ByteSize float64

const (
    _           = iota // ignore first value by assigning to blank identifier
    KB ByteSize = 1 << (10 * iota)
    MB
    GB
    TB
    PB
    EB
    ZB
    YB
)
```

The ability to attach a method such `String` to any user-defined type makes it possible for arbitrary values to format themselves automatically for printing.

```
func (b ByteSize) String() string {
    switch {
    case b >= YB:
    return fmt.Sprintf("%.2fYB", b/YB)
    case b >= ZB:
        return fmt.Sprintf("%.2fZB", b/ZB)
    case b >= EB:
        return fmt.Sprintf("%.2fEB", b/EB)
    case b >= PB:
        return fmt.Sprintf("%.2fPB", b/PB)
    case b >= TB:
        return fmt.Sprintf("%.2fTB", b/TB)
    case b >= GB:
        return fmt.Sprintf("%.2fGB", b/GB)
    case b >= MB:
        return fmt.Sprintf("%.2fMB", b/MB)
    case b >= KB:
        return fmt.Sprintf("%.2fKB", b/KB)
    }
    return fmt.Sprintf("%.2fB", b)
}
```

Variables

Variables can be intialized just like constants but the initializer can be a general expression computed at run time.

```
var (
    home   = os.Getenv("HOME")
    user   = os.Getenv("USER")
    gopath = os.Getenv("GOPATH")
)
```

The init function

Finally, each source file can define its own niladic `init` function to set up whatever state is required. (Actually each file can have multiple `init` function.) And finally means finally: `init` is called after all the variable declarations in the package have evaluated their initializers, and those are evaluated only after all the imported packages have been intialized.

Besides initializations that cannot be expressed as declarations, a common use of `init' functions is to verify or repair correctness of the program state before real execution begins.

```
func init() {
    if user == "" {
        log.Fatal("$USER not set")
    }
    if home == "" {
        home = "/home/" + user
    }
    if gopath == "" {
        gopath = home + "/go"
    }
    // gopath may be overridden by --gopath flag on command line.
    flag.StringVar(&gopath, "gopath", gopath, "override     default GOPATH")
}
```

Methods

Pointers vs. Values

Methods can be defined for any named type (except a pointer or an interface).

```
type ByteSlice []byte
func (p *ByteSlice) Write(data []byte) (n int, err error) {
    slice := *p
    //to do sth....
    *p = slice
    return len(data), nil
}
```

The rule about pointers vs. values for receivers is that value methods can be invoked on pointers and values, but pointer methods can only be invoked on pointers.

This rule arises because pointer methods can modify the receiver; invoking them on a value would cause the method to receive a copy of the value, so any modifications would be discarded. The language therefore disallows this mistake. There is a handy exception, though. When value is addressable, the language takes care of the common case of invoking a pointer method on a value by inserting the address operator automatically. In our example, the variable `b` is addressable, so we can call its `Write` method with just `b.Write`. The compiler will rewrite that to `(&b).Write` for us.

Interface and other types

Interface

Interfaces in Go provide a way to specify the behavior of an object: if something do *this*, then it can used *here*. Interfaces with only one or two methods are common in Go code, and are usually given a name derived from the method, sunch as `io.Writer` for something that implements `Write`. A type can implement multiple interfaces.

Conversions

It's an idiom in Go program to convert the type of an expression to access a different set of methods.

```
type Sequence []int

// Method for printing - sorts the elements before printing
func (s Sequence) String() string {
    sort.IntSlice(s).Sort()
    return fmt.Sprint([]int(s))
}
```

Interface conversions and type assertions

Type switches are a form of conversion: they take an interface and, foreach case in the switch, in a sence convert it to the type of that case.

```
type Stringer interface {
    String() string
}

var value interface{} // Value provided by caller
switch str := value.(type) {
    case string:
        return str
    case Stringer:
        return str.String()
}
```

The first case finds a concrete value; the second converts the interface into another interface.

A **type assertion** takes an interface value and extracts from it a value of the specified explicit type. The syntax borrows from the clause opening a type switch, but with an explicit type rather than the `type` keyword:

```
value.(typeName)
```

and the result is a new value with the static type `typeName`. That type must either be the concrete type held by the interface, or a second interface type that the value can be converted to. To extract the string we knwow is in the value, we could write:

```
str := value.(string)
```

But if it turns out that the value does not contain a string, the program will crash with a run-time error. To guard against that, use the "comma, ok" idiom to test, safely, whether the value is a string:

```
str, ok = value.(string)
if ok {
    fmt.Printf("string value is %q\n", str)
} else {
    fmt.Printf("value is not a string\n")
}
```

If the type assertion fails, `str` will still exist and be of type string, but it will have the zero value, an empty string.

As illustration of the capability, here's an *if-else* statement that's equvalent to the *type switch* that opened this section.

```
if str, ok := value.(string); ok {
    return str
} else if str, ok := value.(Stringer); ok {
    return str.String()
}
```

Generality

If a type exists only to implement an interface and has no exported methods beyond that interface, there is no need to export the type itself. Exporting just the interfaace makes it clear that it's the behavior that matters, not the implementation, and that other implementations with different properties can mirror the behavior of the original type. It also avoids the need to repeat the documentation on every instance of a common method.

In such cases, the constructor should return an interface value rather than an implementing type. As an example, in the hash libraries both `crc32.NewIEEE` and `adler32.New` return the interface type `hash.Hash32`. Substituting the CRC-32 algorithm for Adler-32 in a Go program requires only changing the constructor call; the rest of the code is unaffeted by the change of algorithm. (Depedency Inversion)

Interface and methods

Since almost anything can have methods attached, almost anything can satisfy an interface, all because interface are just sets of methods, which can be defined for (almost) any type (except pointers and interfaces).

The blank identifier

The blank identifier can be assigned or declared with any value of any type, with the value discarded harmlessly. It's a bit like writing to the Unix */dev/null* file: it represents a write-only value to be used as a place-holder where a variable is needed but the actual value is irrelevant.

The blank identifier in multiple assignment

The use of blank identifier in a *for range* loop is a special case of a general situaltion: multiple assignment.

If an assignment requires multiple values on the left side, but one of the values will not be used by the program, a blank identifier on the left-hand-side of the assignment avoids the need to create a dummy variable and make it clear that the value is to be discarded. For instance, when calling a function that returns a value and an error, but only the error is important, use the blank identifier to discard the irrelevant value.

```
if _, err := os.Stat(pat); os.IsNotExist(err) {
    fmt.Printf("%s does not exist\n", path)
}
```

Occasionally you'll see code that discard the error value in order to ignore the error; this is terrible pratice.

Unused imports and variables

It is an error to import a package or to declare a varible without using it. Unused imports bloat the program and slow compilation, while a variable that is intialized but not used is at least a wasted computation and perhaps indicative of a larger bug. When a program is under active development, however, unsued imports and variables often arise and it can be annoying to delete them just to have the compilation proceed, only to have them be needed again later.

To silence compaints about the unused imports, use a blank identifier to refer to a symbol from the imported package. Similarly, assigning the unused variable `fd` to the blank identifier will silence the unused variable error.

```
package main

import (
    "fmt"
    "io"
    "log"
    "os"
)

var _ = fmt.Printf // For debugging; delete when done.
var _ io.Reader    // For debugging; delete when done.

func main() {
    fd, err := os.Open("test.go")
    if err != nil {
        log.Fatal(err)
    }
    // TODO: use fd.
    _ = fd
}
```

By convention, the global declarations to silence import erros should come right after the imports and be commented, both to make them easy to find and as a reminder to clean things up later.

Import for side effect

Sometimes it is used to import a package only for side effects, without an explicit use. For example, during its `init` function, the `net/http/pprof` package registers HTTP handlers that provide debugging information. It has an exported API, but most clients need only the handler registration and access the data through a web page. To import the package only for its side effects, rename the package to the blank identifier:

```
import _ "net/http/pprof"
```

This form of import makes clear that the package is being imported for its side effects, because there no other possible use of the package: in this file, it doesn't have a name. (If it did, and we didn't use that name, the compiler would reject the program.)

Interface checks

A type need not declare explicitly that it implements an interface, instead, a type implements the interface just by implementing the interface's methods. In practice, most interface conversions are static and therefore checked at compile time. For examle, pass an `*os.File` to a function expecting an `io.Reader` will compile unless `*os.File` implements the `io.Reader` interface.

Some interface checks do happen at rum-time, though. One instance is in the `encoding/json` package, which defines a `Marshaler` interface. When the JSON receives a value that implements that inferface, the encoder invokes the value's marshaling method to convert it to JSON instead of doing the standard conversion. The encoder checks this property at run time with a *type assertion* like:

```
m, ok := val.(json.Marshaler)
```

or

```
_, ok := val.(json.Marshaler)
```

It it's necessary only to ask whether a type implements an interface, without actually using the interface itself, perhaps as part of an error check.

One place this situation arises is when it is necessary to guarantee within the package implementing the type that it actually satisfies the interface.

```
var _ json.Marshaler = (*RawMessage)(nil)
```

In this declaration, the assignment involving a conversion of a `*RawMessage` to a `Marshaler` requires that `*RawMessage` implements `Marshaler`, and that property will be checked at compile time.

The appearance of the blank identifier in this construct indicates that the declaration exists only for type checking, no to create a variable. Don't do this for every type that satisfies an interface, though. By Convention, such declarations are only used when there are no static conversions already present in the code, which is a rare event.

Embedding

Go does not provide the typical, type-driven notion of subclassing, but it does have the ablility to "borrow" pieces of an implementation by *embedding* types within a struct or interface.

Interface embedding is very simple.

```
package io

type Reader interface {
    Read(p []byte) (n int, err error)
}

type Writer interface {
    Write(p []byte) (n int, err error)
}

// ReadWriter is the interface that combines the Reader and Writer interfaces.
type ReadWriter interface {
    Reader
    Writer
}
```

A `ReadWriter` can do what a `Reader` *and* what a `Writer` does; it is a union of the embedded interfaces (which must be disjoint sets of methods). Only interfaces can be embeded within interfaces.

The same basic idea applies to structs, but with more far-reaching implications.

```
package bufio

// ReadWriter stores pointer to a Reader and a Writer but does not give them field nam
es.
// It implements io.ReadWriter
type ReadWriter struct {
    *Reader     // *buffio.Reader
    *Writer     // *buffio.Writer
}
```

The embedded elements are pointers to structs and of course must be intialized to point to valid structs before they can be used. The methods of embeded types come along for free.

The `ReadWriter` struct could be written as

```
type ReadWriter struct {
    reader *Reader
    writer *Writer
}
```

but then to promote the methods of the fields and to satisfy the `io` interfaces, we would also need to provide forwarding methods, like this:

```
func (rw *ReadWriter) Read(p []byte) (n int, err error) {
    return rw.reader.Read(p)
}
```

There's an important way in whch embedding differs from subclassing. When we embed a type, the methods of that type become methods of the outer type, but when they are invoked the receiver of the method is the inner type, not the outer one.

Embedding can also be a simple convenience.

```
type Job struct {
    Command string
    *log.Logger
}
```

If we need to refer to an embedded field directly, the type name of the field, ignoring the package qualifier, serves as a field name, as it did in the `Read` method of the `ReadWriter` struct.

Embedding types introduce the problem of name conflicts.

- First, a field or method `X` hides any other item `X` in a more deeply nested part of the type. If `log.Logger` contained a field or method called `Command`, the `Command` field of the `Job` would dominate it.
- Second, if the same name appears at the same nesting level, it is usually an error; it would be erroneous to embed `log.Logger` if the `Job` struct contained another field or method called `Logger`. However, if the duplicate name is never mentioned in the program outside the type definition, it is OK. This qualification provides some protection against changes made to types embedded from outside; there is no problem if a field is added that conflicts with another field in another subtype if neither field is ever used.

Concurrency

Share by communicating

Concurrent programming in many environments is made difficult by the subtleties required to implement correct access to shard variables. Go encourages a different approach in which shared values are passed around on channels and, in fact, never actively shared by separate threads of execution. Only one `goroutine` has access to the value at any given time. Data races cannot occur, by design. To encourage this way of thinking we have reduced it to a slogan:

Do not communicate by sharing memory; instead, share memory by communicating.

One way to think about this model is to consider a typical single-threaded program running on the CPU. It has no need for synchronization primitives. Now run another such instance; it too needs no synchronization. Now let those two communicate; if the communication is the synchronizer, there's still no need for other synchroniztion. Unix pipelines, for example, fit this model perfectly. Although Go's approach to concurrency originates in Hoare's Communicating Sequential Processes (CSP), it can also be as a type-safe generalization of Unix pipes.

Goroutines

They'are called *goroutines* because the existing terms—threads, coroutines, processes, and so on—convey inaccurate connotations. A goroutine has a simple model: it is a function executing concurrently with other goroutines in the same address space. It is lightweight, costing little more than the allocation of stack space. And the stacks start small, so they are cheap, and grow by allocating (and freeing) heap storage as required.

Goroutines are multiplexed onto multiple OS threads so if one should block, such as while waiting for I/O, others continue to run. Their design hides many of the complexities of thread creation and manangement.

Prefix a function or method call with the **go** keyword to run the call in a new goroutine. When the call completes, the goroutine exits, silently. (The effect is similar to the Unix shell's **&** notation for running a command in the background.)

```
go list.sort()  // run list.sort concurrently; don't wait for it
```

A function literal can be handy in a goroutine invocation.

```
func Annouce(message string, delay time.Duration) {
    go func() {
        time.Sleep(delay)
        fmt.Println(message)
    }()     // Note the parentheses - must call the function.
}
```

In Go, function literals are **closure**s: the implementation makes sure the variables referred to by the function survive as long as they are active.

Channels

Like maps, channels are allocated with `make`, and the resulting value acts as a reference to an underlying data structure. If an optional integer parameter is provided, it sets the buffer size for the channel. The default is zero, for an unbuffered or synchronous channel.

```
ci := make(chan int)            // unbuffered channel of integers
cj := make(chan int, 0)         // unbuffered channel of integers
cs := make(chan *os.File, 100)  // buffered channel of pointers to Files
```

Unbuffered channels combine communication—the exchange fo a value—with synchronization—guaranteeing that two calculations (goroutines) are in a known state.

```
// A channel can allow the launching goroutine to wait for the sort to complete.

c := make(chan int)     // Allocate a channel
// Start the sort in a goroutine; when it completes, signal on the channel.
go func() {
    list.Sort()
    c <- 1  // Send a signal; value does not matter.
}()
doSomethingForAWhile()
<-c     // Wati for sort to finish; discard sent value.
```

Receivers always block until there is data to receive. If the channel is unbuffered, the sender blocks until the receiver has received the value. If the channel has a buffer, the sender blocks only until the value has been copied to the buffer; if the buffer is full, this means waiting until some receiver has retrieved a value.

A buffered channel can be used like a semaphore, for instance to limit throughput.

In this example, incoming requests are passed to `handle`, which sends a value into the channel, processes the request, and then receives a value from the channel to ready the "semaphore" for the next consumer. The capacity of the channel buffer limits the number of simultaneous calls the `process`.

```
var sem = make(chan int, MaxOutstanding)

func handle(r *Request) {
    sem <- 1    // Wait for active queue to drain.
    process(r)  // May take a long time.
    <-sem       // Done; enable next request to run.
}

func Serve(queue chan *Request) {
    for {
        req := <-queue
        go handle(req)  // Don't wait for handle to finish.
    }
}
```

This design has a problem, though: Serve creates a new goroutine for every incomming request, even though only `MaxOutstanding` of them can run at any moment. As a result, the program can consume unlimited resources if the requests come in too fast. We can address that deficiency by changing `Server` to gate the creation of the goroutines.

```
func Serve(queue chan *Request) {
    for req := range queue {
        sem <- 1
        go func() {
            process(req) // Buggy; see explanation below.
            <-sem
        }()
    }
}
```

The bug is that in a Go *for loop*, ***the loop variable is reused for each iteration***, so the `req` variable is shared across all goroutines.

Here's one way to do that, passing the value of `req` as an argument to the **closure** in the goroutine:

```
func Serve(queue chan *Request) {
    for req := range queue {
        sem <- 1
        go func(req *Request) {
            process(req)
            <-sem
        }(req)
    }
}
```

Another solution is just to create a new variable with the same name, as the belows:

```
func Serve(queue chan *Request) {
    for req := range queue {
        req := req // Create new instance of req for the goroutine.
        sem <- 1
        go func() {
            process(req)
            <-sem
        }()
    }
}
```

It may seem odd to write

```
req := req // :=, redeclaration and reassignment
```

but it's legal and idiomatic in Go to do this. You get a fresh version of the variable with the same name, deliberately shadowing the loop variable locally but unique to each goroutine.

Channels of channels

One of the most important properties of Go is that a channel is a first-class value that can be allocated and passed around like any other. A common use of this property is to implement safe, parallel demultiplexing.

If a type includes a channel on which to reply, each client can provide its own path for the answer. Here's a schematic definition of type Request.

```
type Request struct {
    args        []int
    f           func([]int) int
    resultChan  chan int
}
```

The client provides a function and its arguments, as well as a channel inside the request object on which to receive the answer.

```
func sum(a []int) (s int) {
    for _, v := range a {
        s += v
    }
    return
}

request := &Request{[]int{3, 4, 5}, sum, make(chan int)}
// Send request
clientRequests <- request
// Wait for response.
fmt.Printf("answer: %d\n", <-request.resultChan)
```

On the server side, the handler function is the only thing that changes.

```
func handle(queue chan *Request) {
    for req := range queue {
        req.resultChan <- req.f(req.args)
    }
}
```

There's clearly a lot more to do make it realistic, but this code is a framework for a rate-limited, parallel, non-blocking RPC system, and there's not a mutex in sight.

Parallelization

If a calculation can be broken into separate pieces that can execute independently, it can be to parallelized across multiple CPU cores, with a channel to signal when each piece completes.

Let's say we have an expensive operation to perform on a vector of itmes, and that the value of the operation on each item is independent, as in this idealized example:

```
type Vector []float64

// Apply the operation to v[i], v[i+1] ... up to [vn-1].
func (v *Vector) DoSome(i, n int, u Vector, c chan int) {
    for ; i < n; i++ {
        v[i] += u.Op(v[i])
    }
    c <- 1      // signal that this piece is done
}

const numCPU = 4    // number of CPU cores

func (v Vector) DoAll(u Vector) {
    c := make(chan int, numCPU)     // Buffering optional but sensible.
    for i := 0; i < numCPU; i++ {
        go v.DoSome(i*len(v)/numCPU, (i+1)*len(v)/numCPU, u, c)
    }
    // Drain the channel.
    for i := 0; i < numCPU; i++ {
        <-c     // wait for one task to complete
    }
    // All done.
}
```

Rather than create a constant value for numCPU, the function `runtime.NumCPU` can return the number of hardware CPU cores in the machine.

```
var numCPU = runtime.NumCPU()
```

There is also a function `runtime.GOMAXPROCS`, which reports (or sets) the user-specified number of cores that a Go program can have running simultaneously. It defaults to the value of `runtime.NumCPU` but can be overridden by setting the similarly named shell environment variable or by call the function with a positive number. Calling it with zero just qureis the value. Therefore if we want to honor the user's resource request, we should write

```
var numCPU = runtime.GOMAXPROCS(o)
```

Be sure not to confuse the ideas of **concurrency**—structuring a program as independentyly executing components—and **parallelism**—executing calculations in parallel for efficiency on multiple CPUs.

Although the concurrency features of Go can make some problems easy to structure as parallel computations, Go is a concurrent language, not a parallel one, and not all parallelization problems fit Go's model.

A leaky buffer

To be continued....

Errors

Libraries routines must often return some sort of error indication to the caller. As mentioned earlier, Go's multivalue return makes it easy to return a detailed error description alongside the normal return value. It is good style to use this feature to provide detailed error information.

By convention, errors have type `error`, a simple built-in interface.

```
type error interface {
    Error() string
}
```

A library writer is free to implement this interface with a richer model under the convers, making it possible not only to see the error but also to provide some context.

```
package os

// PathError records an error and the operation and
// file path that caused it.
type PathError struct {
    Op string    // "open", "unlink", etc.
    Path string  // The associated file.
    Err error    // Returned by the system call.
}

func (e *PathError) Error() string {
    return e.Op + " " + e.Path + ": " + e.Err.Error()
}
```

PathError's Error generates a string like this:

```
open /etc/passwx: no such file or directory
```

It is much more informative than the plain "no such file or directory".

When feasible, error strings should identify their origin, such as by having a prefix naming the operation or package that generated the error. For example, in package *package*, the string representation for a decoding error due to an unkown format is "image: unkown format".

Callers that care about the precise error details can use a *type switch* or a *type assertion* to look for specific errors and extract details.

```
for try := 0; try < 2; try++ {
    file, err = os.Create(filename)
    if err == nil {
        return
    }
    if e, ok := err.(*os.PathError); ok && e.Err == syscall.ENOSPC {
        deleteTempFiles()  // Recover some space.
        continue
    }
    return
}
```

Panic

The usual way to report an error to a caller is to return an `error` as an extra return value. But sometimes the program simply cannot continue with a error that is unrecoverable.

There is a built-in function `panic` that takes a single argument of arbitrary type—often a string—to printed as the program dies to create a run-time error that will stop the program. It's also a way to indicate that something impossible has happened, such as exiting an infinite loop.

```
// A toy implementation of cube root using Newton's method.
func CubeRoot(x float64) float64 {
    z := x/3   // Arbitrary initial value
    for i := 0; i < 1e6; i++ {
        prevz := z
        z -= (z*z*z-x) / (3*z*z)
        if veryClose(z, prevz) {
            return z
        }
    }
    // A million iterations has not converged; something is     wrong.
    panic(fmt.Sprintf("CubeRoot(%g) did not converge", x))
}
```

This is only an example but real library functions should avoid `panic`. If the program can be masked or worked around, it's always better to let things continue to run rather than taking down the whole program.

Recover

When `panic` is called, including implicitly for runtime errors such indexing a slice out of bounds or failing a type assertion, it immediately stops execution of the current function and begins unwinding the stack of the goroutine, running any deferred functions along the way. If that unwinding reaches the top of the goroutine's stack, the program dies. However, it is possible to use the built-in function `recover` to regain control of the goroutine and resume normal execution.

A call to `recover` stops the unwinding and returns the argument passed to `panic`. Because the only code that runs while unwinding is inside deferred functions, `recover` is only useful inside deferred functions.

On application of `recover` is to shut down a failing goroutine inside a server without killing the other executing goroutines.

```
func server(workChan <-chan *Work) {
    for work := range workChan {
        go safelyDo(work)
    }
}

func safelyDo(work *Work) {
    defer func() {
        if err := recover(); err != nil {
            log.Println("work failed:", err)
        }
    }()
    do(work)
}
```

In this example, if `do(work)` panics, the result will be logged and the goroutine will exit cleanly without disturbing the others.

With the recovery pattern in place, the `do` function (and anything it calls) can get out of any bad situation cleanly by calling `panic`. We can use that idea to simplify error handling in complex software.

By the way, the re-panic idiom could changes the panic value if an actual error occurs.