Effective Go[1]

go1.0.1

Introduction

Go is a new language. Although it borrows ideas from existing languages, it has unusual properties that make effective Go programs different in character from programs written in its relatives. A straightforward translation of a C++ or Java program into Go is unlikely to produce a satisfactory result—Java programs are written in Java, not Go. On the other hand, thinking about the problem from a Go perspective could produce a successful but quite different program. 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. It augments the language specification, the *Tour of Go*, and *How to Write Go Code*, all of which you should read first.

Examples

The Go package sources are intended to serve not only as the core library but also as examples of how to use the language. If you have a question about how to approach a problem or how something might be implemented, they can provide answers, ideas and background.

[1] `http://golang.org/doc/effective_go.html`

Formatting

Formatting issues are the most contentious but the least 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 everyone adheres to the same style. The problem is how to approach this Utopia without a long prescriptive style guide.

With Go we take an unusual approach and let the machine take care of most formatting issues. The `gofmt` program (also available 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. If you want to know how to handle some new layout situation, run `gofmt`; if the answer doesn't seem right, rearrange your program (or file a bug about `gofmt`), don't work around it.

As an example, there's no need to spend time lining up the comments on the fields of a structure. `Gofmt` will do that for you. Given the declaration

```
type T struct {
    name string // name of the object
    value int // its value
}
```

`gofmt` will line up the columns:

```
type T struct {
    name    string // name of the object
    value   int    // its value
}
```

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 length Go has no line length limit. Don't worry about overflowing a punched card. If a line feels too long, wrap it and indent with an extra tab.

Parentheses Go needs fewer parentheses: control structures (`if`, `for`, `switch`) do not have parentheses in their syntax. Also, the operator precedence hierarchy is shorter and clearer, so

```
x<<8 + y<<16
```

means what the spacing implies.

Commentary

Go provides C-style `/* */` block comments and C++-style `//` line comments. Line comments are the norm; block comments appear mostly as package comments and are also useful 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. The package comment should introduce the package and provide information relevant to the package as a whole. It will appear first on the `godoc` page and should set up the detailed documentation that follows.

```
/*
    Package regexp implements a simple library for
    regular expressions.

    The syntax of the regular expressions accepted is:

    regexp:
        concatenation { '|' concatenation }
    concatenation:
        { closure }
    closure:
        term [ '*' | '+' | '?' ]
    term:
        '^'
        '$'
        '.'
        character
        '[' [ '^' ] character-ranges ']'
        '(' regexp ')'
*/
package regexp
```

If the package is simple, the package comment can be brief.

```
// Package path implements utility routines for
// manipulating slash-separated filename paths.
```

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 other annotations such as `_this_` will reproduce verbatim and should not be used. Depending on the context, `godoc` might not even reformat comments, so make sure they look good straight up: use correct spelling, punctuation, and sentence structure, fold long lines, and so on.

Inside a package, any comment immediately preceding a top-level declaration serves as a *doc comment* for that declaration. Every exported (capitalized) name in a program should have a doc comment.

Doc comments work best as complete sentences, which allow a wide variety of automated presentations. The first sentence should be a one-sentence summary that starts with the name being declared.

```
// Compile parses a regular expression and returns, if successful, a Regexp
// object that can be used to match against text.
func Compile(str string) (regexp *Regexp, err error) {
```

Go's declaration syntax allows grouping of declarations. A single doc comment can introduce a group of related constants or variables. Since the whole declaration is presented, such a comment can often be perfunctory.

```
// 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 ')'")
    ...
)
```

Even for private names, grouping can also indicate relationships between items, such as the fact that a set of variables is protected by a mutex.

```
var (
    countLock   sync.Mutex
    inputCount  uint32
    outputCount uint32
    errorCount  uint32
)
```

Names

Names are as important in Go as in any other language. In some cases they even have semantic effect: for instance, the visibility of a name outside a package is determined by whether its first character is upper case. It's therefore worth spending a little time talking about naming conventions in Go programs.

Package names

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

```
import "bytes"
```

the importing package can talk about `bytes.Buffer`. It's helpful if everyone using the package can use the same name to refer to its contents, which implies that the package name should be good: short, concise, evocative. By convention, packages are given lower case, single-word names; there should be no need for underscores or mixedCaps. Err on the side of brevity, since everyone using your package will be typing that name. And don't worry about collisions a priori. The package name is only the default name for imports; it need not be unique across all source code, and in the rare case of a collision the importing package can choose a different name to use locally. In any case, confusion is rare because the file name in the import determines just which package is being used.

Another convention is that the package name is the base name of its source directory; the package in `src/pkg/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 (the `import .` notation is intended mostly for tests and other unusual situations and should be avoided unless necessary), so exported names in the package can use that fact to avoid stutter. For instance, the buffered reader type in the `bufio` package is called `Reader`, not `BufReader`, because users see it as `bufio.Reader`, which is a clear, concise name. Moreover, because imported entities are always addressed with their package name, `bufio.Reader` does not conflict with `io.Reader`. Similarly, the function to make new instances of `ring.Ring`—which is the definition of a constructor in Go—would normally be called `NewRing`, but since `Ring` is the only type exported by the package, and since the package is called `ring`, it's called just `New`, which clients of the package see as `ring.New`. Use the package structure to help you choose good names.

Another short example is `once.Do`; `once.Do(setup)` reads well and would not be improved by writing `once.DoOrWaitUntilDone(setup)`. Long names don't automatically make things more readable. If the name represents something intricate or subtle, it's usually better to write a helpful doc comment than to attempt to put all the information into the name.

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 idiomatic 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 practice:

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

Interface names

By convention, one-method interfaces are named by the method name plus the *-er* suffix: `Reader`, `Writer`, `Formatter` etc.

There are a number of such names and it's productive to honor them and the function names they capture. `Read`, `Write`, `Close`, `Flush`, `String` and so on have canonical signatures and meanings. 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; unlike 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 `float64`), 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, insert a semicolon".

A semicolon can also be omitted 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 initializer, condition, and continuation elements. They are also necessary to separate multiple statements on a line, should you write code that way.

One caveat. You should never 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 this

```
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`; 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 initialization 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, you'll find that 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)
```

This is an example of a common situation where code must guard against a sequence of error conditions. The code reads well if the successful flow of control runs down the page, eliminating error cases as they arise. Since error cases tend to end in `return` statements, the resulting code needs no `else` statements.

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

Redeclaration

An aside: The last example in the previous section demonstrates a detail of how the `:=` short declaration form works. The declaration that calls `os.Open` reads,

```
f, err := os.Open(name)
```

This statement declares two variables, `f` and `err`. A few lines later, the call to `f.Stat` reads

```
d, err := f.Stat()
```

which looks as if it declares `d` and `err`. Notice, though, that `err` appears in both statements. This duplication is legal: `err` is declared by the first statement, but only re-assigned in the second. This means that the call to `f.Stat` uses the existing `err` variable declared above, and just gives it a new value.

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

- this declaration 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 initialization is assignable to `v`, and
- there is at least one other variable in the declaration that is being declared anew.

This unusual property is pure pragmatism, making it easy to use a single `err` value, for example, in a long `if-else` chain. You'll see it used often.

`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 expired(key) {
        delete(m, key)
    }
}
```

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

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

For strings, the `range` does more work for you, breaking out individual Unicode characters by parsing the UTF-8. Erroneous encodings consume one byte and produce the replacement rune U+FFFD. The loop

```
for pos, char := range "дъб" {
    fmt.Printf("character %c starts at byte position %d\n", char, pos)
}
```

prints

```
character д starts at byte position 0
character ъ starts at byte position 2
character б starts at byte position 4
```

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.

```
// 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 expressions 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 switches on `true`. It's therefore possible—and idiomatic—to write an `if-else-if-else` chain as a `switch`.

```
func unhex(c byte) byte {
    switch {
    case '0' <= c && c <= '9':
        return c - '0'
    case 'a' <= c && c <= 'f':
        return c - 'a' + 10
    case 'A' <= c && c <= 'F':
        return c - 'A' + 10
    }
    return 0
}
```

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
}
```

Here's a comparison routine for byte arrays that uses two `switch` statements:

```
// Compare returns an integer comparing the two byte arrays,
// 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
}
```

A switch can also be used to discover the dynamic type of an interface variable. Such a type switch uses the syntax of a type assertion with the keyword `type` inside the parentheses. If the switch declares a variable in the expression, the variable will have the corresponding type in each clause.

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

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.

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 `File.Write` in 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)`. This is a common style; see the section on error handling for more examples.

A similar approach obviates the need to pass a pointer to a return value to simulate a reference parameter. Here's a simple-minded function to grab a number from a position in a byte array, returning the number and the next position.

```
func nextInt(b []byte, i int) (int, int) {
    for ; i < len(b) && !isDigit(b[i]); i++ {
    }
    x := 0
    for ; i < len(b) && isDigit(b[i]); i++ {
        x = x*10 + int(b[i])-'0'
    }
    return x, i
}
```

You could use it to scan the numbers in an input array `a` like this:

```
    for i := 0; i < len(a); {
        x, i = nextInt(a, i)
        fmt.Println(x)
    }
```

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 for their types when the function begins; if the function executes a `return` statement with no arguments, the current values of the result parameters are used as the returned values.

The names are not mandatory but they can make code shorter and clearer: they're documentation. If we name the results of `nextInt` it becomes obvious which returned `int` is which.

```
func nextInt(b []byte, pos int) (value, nextPos int) {
```

Because named results are initialized and tied to an unadorned return, they can simplify as well as clarify. Here's a version of `io.ReadFull` that uses them well:

```
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. The canonical examples are unlocking a mutex or closing a file.

```
// 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.

    var result []byte
    buf := make([]byte, 100)
    for {
        n, err := f.Read(buf[0:])
        result = append(result, buf[0:n]...) // append is discussed later.
        if err != nil {
            if err == io.EOF {
                break
            }
            return "", err  // f will be closed if we return here.
        }
    }
    return string(result), nil // f will be closed if we return here.
}
```

Deferring a call to a function such as `Close` has two advantages. First, it guarantees that you will never forget to close the file, a mistake that's easy to make if you later edit the function to add a new return path. Second, it means that the close sits near the open, which is much clearer than placing it at the end of the function.

The arguments to the deferred function (which include the receiver if the function is a method) are evaluated when the defer executes, not when the call executes. Besides avoiding worries about variables changing values as the function executes, this means that a single deferred call site can defer multiple function executions. Here's a silly example.

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

Deferred functions are executed in LIFO order, so this code will cause `4 3 2 1 0` to be printed when the 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....
}
```

We can do better by exploiting the fact that arguments to deferred functions are evaluated when the `defer` executes. The tracing routine can set up the argument to the untracing routine. This example:

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

func un(s string) {
    fmt.Println("leaving:", s)
}

func a() {
    defer un(trace("a"))
    fmt.Println("in a")
}

func b() {
    defer un(trace("b"))
    fmt.Println("in b")
    a()
}

func main() {
    b()
}
```

prints

```
entering: b
in b
entering: a
in a
leaving: a
leaving: b
```

For programmers accustomed to block-level resource management from other languages, `defer` may seem peculiar, but its most interesting and powerful applications come precisely from the fact that it's not block-based but function-based. In the section on `panic` and `recover` we'll see another example of its possibilities.

Data

Allocation with `new`

Go has two allocation primitives, the built-in functions `new` and `make`. They do different things and apply to different types, which can be confusing, but the rules are simple. Let's talk about `new` first. It's a built-in function that allocates memory, but unlike its namesakes in some other languages it does not initialize the memory, it only zeros it. That is, `new(T)` allocates zeroed storage for a new item of type `T` and returns 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 value 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. For example, the documentation for `bytes.Buffer` states that "the zero value for `Buffer` is an empty buffer ready to use." Similarly, `sync.Mutex` does not have an explicit constructor or `Init` method. Instead, the zero value for a `sync.Mutex` is defined to be an unlocked mutex.

The zero-value-is-useful property works transitively. Consider this type declaration.

```
type SyncedBuffer struct {
    lock    sync.Mutex
    buffer  bytes.Buffer
}
```

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 composite literals

Sometimes the zero value isn't good enough and an initializing constructor is necessary, as in this example derived from 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
}
```

Note that, unlike in C, it's perfectly OK to return the address of a local variable; the storage associated with the variable survives after the function returns. 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 initializers 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 creates a zero value for the type. The expressions `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. In these examples, the initializations work regardless of the values of `Enone`, `Eio`, and `Einval`, as long as they are distinct.

```
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`

Back to allocation. The built-in function `make(T, `*args*`)` serves a purpose different from `new(T)`. It creates slices, maps, and channels only, and it returns an initialized (not zeroed) value of type `T` (not `*T`). The reason for the distinction is that these three types are, 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 length, and the capacity, and until those items are initialized, the slice is `nil`. For slices, maps, and channels, `make` initializes the internal data structure and prepares the value for use. For instance,

```
make([]int, 10, 100)
```

allocates an array of 100 ints and then creates a slice structure with length 10 and a capacity of 100 pointing at the first 10 elements of the array. (When making a slice, the capacity can be omitted; see the section on slices for more information.) In contrast, `new([]int)` returns a pointer to a newly allocated, zeroed slice structure, that is, a pointer to a `nil` slice value.

These examples illustrate the difference between `new` and `make`.

```
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`.

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, the subject of the next section. To lay the foundation for that topic, here are a few words about arrays.

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 particular, 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 types `[10]int` and `[20]int` are distinct.

The value property can be useful but also expensive; if you want C-like behavior and efficiency, you can pass a pointer to the array.

```
func Sum(a *[3]float64) (sum float64) {
    for _, v := range *a {
        sum += v
    }
    return
}

array := [...]float64{7.0, 8.5, 9.1}
x := Sum(&array)  // Note the explicit address-of operator
```

But even this style isn't idiomatic Go. Slices are.

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 slices rather than simple arrays.

Slices are reference types, which means that if you assign one slice to another, both refer to the same underlying array. For instance, 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. Here is the signature of the `Read` method of the `File` type in package `os`:

```
func (file *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 `b`, slice (here used as a verb) the buffer.

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

Such slicing is common and efficient. In fact, leaving efficiency aside for the moment, the following snippet would also read the first 32 bytes of the buffer.

```
    var n int
    var err error
    for i := 0; i < 32; i++ {
        nbytes, e := f.Read(buf[i:i+1])  // Read one byte.
        if nbytes == 0 || e != nil {
            err = e
            break
        }
        n += nbytes
    }
```

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. Here is a function to append data to a slice. If the data exceeds the capacity, the slice is reallocated. The resulting slice is returned. The function uses the fact that `len` and `cap` are legal when applied to the `nil` slice, and return 0.

```
func Append(slice, data[]byte) []byte {
    l := len(slice)
    if l + len(data) > cap(slice) {  // reallocate
        // Allocate double what's needed, for future growth.
        newSlice := make([]byte, (l+len(data))*2)
        // The copy function is predeclared and works for any slice type.
        copy(newSlice, slice)
        slice = newSlice
    }
    slice = slice[0:l+len(data)]
    for i, c := range data {
        slice[l+i] = c
    }
    return slice
}
```

We must return the slice afterwards because, although `Append` can modify the elements of `slice`, the slice itself (the run-time data structure holding the pointer, length, and capacity) is passed by value.

The idea of appending to a slice is so useful it's captured by the `append` built-in function. To understand that function's design, though, we need a little more information, so we'll return to it later.

Maps

Maps are a convenient and powerful built-in data structure to associate values of different types. 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 are a reference type. If you pass a map to a function that changes the contents of the map, the changes will be visible in the caller.

Maps can be constructed using the usual composite literal syntax with colon-separated key-value pairs, so it's easy to build them during initialization.

```
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,
}
```

Assigning and fetching map values looks syntactically just like doing the same for arrays 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 the zero value for the type of the entries in the map. For instance, if the map contains integers, looking up a non-existent key will return `0`. A set can be implemented as a map with value type `bool`. Set the map entry to `true` to put the value in the set, and then test it by simple indexing.

```
attended := map[string] bool {
    "Ann": true,
    "Joe": true,
    ...
}

if attended[person] { // will be false if person is not in the map
    fmt.Println(person, "was at the meeting")
}
```

Sometimes you need to distinguish a missing entry from a zero value. Is there an entry for `"UTC"` or is that zero value because it's not in the map at all? You can discriminate 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. Here's a function that puts it together with a nice error report:

```
func offset(tz string) int {
    if seconds, ok := timeZone[tz]; ok {
        return seconds
    }
    log.Println("unknown time zone:", tz)
    return 0
}
```

To test for presence in the map without worrying about the actual value, you can use the blank identifier (_). The blank identifier can be assigned or declared with any value of any type, with the value discarded harmlessly. For testing just presence in a map, 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. It's safe to do this 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. In this example each line produces the same output.

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

As mentioned in the *Tour*, `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 argument 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, structs, and maps. Here is a print statement for the time zone map defined in the previous section.

```
fmt.Printf("%v\n", timeZone)  // or just fmt.Println(timeZone)
```

which gives output

```
map[CST:-21600 PST:-28800 EST:-18000 UTC:0 MST:-25200]
```

For maps the keys may be output in any order, of course. 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)
fmt.Printf("%#v\n", timeZone)
```

prints

```
&{7 -2.35 abc   def}
&{a:7 b:-2.35 c:abc     def}
&main.T{a:7, b:-2.35, c:"abc\tdef"}
map[string] int{"CST":-21600, "PST":-28800, "EST":-18000, "UTC":0, "MST":-25200}
```

(Note the ampersands.) That quoted string format is also available through `%q` when applied to a value of type `string` or `[]byte`; the alternate format `%#q` will use backquotes instead if possible. Also, `%x` works on strings and arrays of bytes as well as on integers, generating a long hexadecimal string, and with a space in the format (`% x`) it puts spaces 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. For our simple type `T`, that might look like this.

```
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; this example used a pointer because that's more efficient and idiomatic for struct types. See the section below on pointers vs. value receivers for more information.)

Our `String` method is able to call `Sprintf` because the print routines are fully reentrant and can be used recursively. We can even go one step further and 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 pass `v` as a single slice argument.

There's even more to printing than we've covered here. See the `godoc` documentation for package `fmt` for the details.

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

Now we have the missing piece we needed to explain the design of the `append` built-in function. The signature of `append` is different from our custom `Append` function above. Schematically, it's like this:

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

where `T` is a placeholder for any given type. You can't actually write a function 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, as with our hand-written `Append`, the underlying array may change. This simple example

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

prints `[1 2 3 4 5 6]`. So `append` works a little like `Printf`, collecting an arbitrary number of arguments.

But what if we wanted to do what our `Append` does and append a slice to a slice? Easy: use `...` at the call site, just as we did in the call to `Output` above. This snippet produces identical output to the one above.

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

Without that `...`, it wouldn't compile because the types would be wrong; `y` is not of type `int`.

Initialization

Although it doesn't look superficially very different from initialization in C or C++, initialization in Go is more powerful. Complex structures can be built during initialization and the ordering issues between initialized objects in 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 functions, and can only be numbers, strings or booleans. Because of the compile-time restriction, the expressions that define them must be constant expressions, evaluatable 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 as `String` to a type makes it possible for such values to format themselves automatically for printing, even as part of a general type.

```
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)
}
```

The expression `YB` prints as `1.00YB`, while `ByteSize(1e13)` prints as `9.09TB`.

Note that it's fine to call `Sprintf` and friends in the implementation of `String` methods, but beware of recurring into the `String` method through the nested `Sprintf` call using a string format (`%s`, `%q`, `%v`, `%x` or `%X`). The `ByteSize` implementation of `String` is safe because it calls `Sprintf` with `%f`.

Variables

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

```
var (
    HOME = os.Getenv("HOME")
    USER = os.Getenv("USER")
    GOROOT = os.Getenv("GOROOT")
)
```

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` functions.) 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 initialized.

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 = "/usr/" + USER
    }
    if GOROOT == "" {
        GOROOT = HOME + "/go"
    }
    // GOROOT may be overridden by --goroot flag on command line.
    flag.StringVar(&GOROOT, "goroot", GOROOT, "Go root directory")
}
```

Methods

Pointers vs. values

Methods can be defined for any named type that is not a pointer or an interface; the receiver does not have to be a struct.

In the discussion of slices above, we wrote an `Append` function. We can define it as a method on slices instead. To do this, we first declare a named type to which we can bind the method, and then make the receiver for the method a value of that type.

```
type ByteSlice []byte

func (slice ByteSlice) Append(data []byte) []byte {
    // Body exactly the same as above
}
```

This still requires the method to return the updated slice. We can eliminate that clumsiness by redefining the method to take a pointer to a `ByteSlice` as its receiver, so the method can overwrite the caller's slice.

```
func (p *ByteSlice) Append(data []byte) {
    slice := *p
    // Body as above, without the return.
    *p = slice
}
```

In fact, we can do even better. If we modify our function so it looks like a standard `Write` method, like this,

```
func (p *ByteSlice) Write(data []byte) (n int, err error) {
    slice := *p
    // Again as above.
    *p = slice
    return len(data), nil
}
```

then the type `*ByteSlice` satisfies the standard interface `io.Writer`, which is handy. For instance, we can print into one.

```
    var b ByteSlice
    fmt.Fprintf(&b, "This hour has %d days\n", 7)
```

We pass the address of a `ByteSlice` because only `*ByteSlice` satisfies `io.Writer`. 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 is because pointer methods can modify the receiver; invoking them on a copy of the value would cause those modifications to be discarded.

By the way, the idea of using `Write` on a slice of bytes is implemented by `bytes.Buffer`.

Interfaces and other types

Interfaces

Interfaces in Go provide a way to specify the behavior of an object: if something can do *this*, then it can be used *here*. We've seen a couple of simple examples already; custom printers can be implemented by a `String` method while `Fprintf` can generate output to anything with a `Write` method. Interfaces with only one or two methods are common in Go code, and are usually given a name derived from the method, such as `io.Writer` for something that implements `Write`.

A type can implement multiple interfaces. For instance, a collection can be sorted by the routines in package `sort` if it implements `sort.Interface`, which contains `Len()`, `Less(i, j int) bool`, and `Swap(i, j int)`, and it could also have a custom formatter. In this contrived example `Sequence` satisfies both.

```
type Sequence []int

// Methods required by sort.Interface.
func (s Sequence) Len() int {
    return len(s)
}
func (s Sequence) Less(i, j int) bool {
    return s[i] < s[j]
}
func (s Sequence) Swap(i, j int) {
    s[i], s[j] = s[j], s[i]
}

// Method for printing - sorts the elements before printing
func (s Sequence) String() string {
    sort.Sort(s)
    str := "["
    for i, elem := range s {
        if i > 0 {
            str += " "
        }
        str += fmt.Sprint(elem)
    }
    return str + "]"
}
```

Conversions

The `String` method of `Sequence` is recreating the work that `Sprint` already does for slices. We can share the effort if we convert the `Sequence` to a plain `[]int` before calling `Sprint`.

```
func (s Sequence) String() string {
    sort.Sort(s)
    return fmt.Sprint([]int(s))
}
```

The conversion causes `s` to be treated as an ordinary slice and therefore receive the default formatting. Without the conversion, `Sprint` would find the `String` method of `Sequence` and recur indefinitely. Because the two types (`Sequence` and `[]int`) are the same if we ignore the type name, it's legal to convert between them. The conversion doesn't create a new value, it just temporarily acts as though the existing value has a new type. (There are other legal conversions, such as from integer to floating point, that do create a new value.)

It's an idiom in Go programs to convert the type of an expression to access a different set of methods. As an example, we could use the existing type `sort.IntSlice` to reduce the entire example to this:

```
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))
}
```

Now, instead of having `Sequence` implement multiple interfaces (sorting and printing), we're using the ability of a data item to be converted to multiple types (`Sequence`, `sort.IntSlice` and `[]int`), each of which does some part of the job. That's more unusual in practice but can be effective.

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 interface 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 the 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 unaffected by the change of algorithm.

A similar approach allows the streaming cipher algorithms in the various `crypto` packages to be separated from the block ciphers they chain together. The `Block` interface in the `crypto/cipher` package specifies the behavior of a block cipher, which provides encryption of a single block of data. Then, by analogy with the `bufio` package, cipher packages that implement this interface can be used to construct streaming ciphers, represented by the `Stream` interface, without knowing the details of the block encryption.

The `crypto/cipher` interfaces look like this:

```
type Block interface {
    BlockSize() int
    Encrypt(src, dst []byte)
    Decrypt(src, dst []byte)
}

type Stream interface {
    XORKeyStream(dst, src []byte)
}
```

Here's the definition of the counter mode (CTR) stream, which turns a block cipher into a streaming cipher; notice that the block cipher's details are abstracted away:

```
// NewCTR returns a Stream that encrypts/decrypts using the given Block in
// counter mode. The length of iv must be the same as the Block's block size.
func NewCTR(block Block, iv []byte) Stream
```

`NewCTR` applies not just to one specific encryption algorithm and data source but to any implementation of the `Block` interface and any `Stream`. Because they return interface values, replacing CTR encryption with other encryption modes is a localized change. The constructor calls must be edited, but because the surrounding code must treat the result only as a `Stream`, it won't notice the difference.

Interfaces and methods

Since almost anything can have methods attached, almost anything can satisfy an interface. One illustrative example is in the `http` package, which defines the `Handler` interface. Any object that implements `Handler` can serve HTTP requests.

```
type Handler interface {
    ServeHTTP(ResponseWriter, *Request)
}
```

`ResponseWriter` is itself an interface that provides access to the methods needed to return the response to the client. Those methods include the standard `Write` method, so an `http.ResponseWriter` can be used wherever an `io.Writer` can be used. `Request` is a struct containing a parsed representation of the request from the client.

For brevity, let's ignore POSTs and assume HTTP requests are always GETs; that simplification does not affect the way the handlers are set up. Here's a trivial but complete implementation of a handler to count the number of times the page is visited.

```
// Simple counter server.
type Counter struct {
    n int
}

func (ctr *Counter) ServeHTTP(w http.ResponseWriter, req *http.Request) {
    ctr.n++
    fmt.Fprintf(w, "counter = %d\n", ctr.n)
}
```

(Keeping with our theme, note how `Fprintf` can print to an `http.ResponseWriter`.) For reference, here's how to attach such a server to a node on the URL tree.

```
import "net/http"
...
ctr := new(Counter)
http.Handle("/counter", ctr)
```

But why make `Counter` a struct? An integer is all that's needed. (The receiver needs to be a pointer so the increment is visible to the caller.)

```
// Simpler counter server.
type Counter int

func (ctr *Counter) ServeHTTP(w http.ResponseWriter, req *http.Request) {
    *ctr++
    fmt.Fprintf(w, "counter = %d\n", *ctr)
}
```

What if your program has some internal state that needs to be notified that a page has been visited? Tie a channel to the web page.

```
// A channel that sends a notification on each visit.
// (Probably want the channel to be buffered.)
type Chan chan *http.Request

func (ch Chan) ServeHTTP(w http.ResponseWriter, req *http.Request) {
    ch <- req
    fmt.Fprint(w, "notification sent")
}
```

Finally, let's say we wanted to present on `/args` the arguments used when invoking the server binary. It's easy to write a function to print the arguments.

```
func ArgServer() {
    for _, s := range os.Args {
        fmt.Println(s)
    }
}
```

How do we turn that into an HTTP server? We could make `ArgServer` a method of some type whose value we ignore, but there's a cleaner way. Since we can define a method for any type except pointers and interfaces, we can write a method for a function. The `http` package contains this code:

```
// The HandlerFunc type is an adapter to allow the use of
// ordinary functions as HTTP handlers.  If f is a function
// with the appropriate signature, HandlerFunc(f) is a
// Handler object that calls f.
type HandlerFunc func(ResponseWriter, *Request)

// ServeHTTP calls f(c, req).
func (f HandlerFunc) ServeHTTP(w ResponseWriter, req *Request) {
    f(w, req)
}
```

`HandlerFunc` is a type with a method, `ServeHTTP`, so values of that type can serve HTTP requests. Look at the implementation of the method: the receiver is a function, `f`, and the method calls `f`. That may seem odd but it's not that different from, say, the receiver being a channel and the method sending on the channel.

To make `ArgServer` into an HTTP server, we first modify it to have the right signature.

```
// Argument server.
func ArgServer(w http.ResponseWriter, req *http.Request) {
    for _, s := range os.Args {
        fmt.Fprintln(w, s)
    }
}
```

`ArgServer` now has same signature as `HandlerFunc`, so it can be converted to that type to access its methods, just as we converted `Sequence` to `IntSlice` to access `IntSlice.Sort`. The code to set it up is concise:

```
http.Handle("/args", http.HandlerFunc(ArgServer))
```

When someone visits the page `/args`, the handler installed at that page has value `ArgServer` and type `HandlerFunc`. The HTTP server will invoke the method `ServeHTTP` of that type, with `ArgServer` as the receiver, which will in turn call `ArgServer` (via the invocation `f(c, req)` inside `HandlerFunc.ServeHTTP`). The arguments will then be displayed.

In this section we have made an HTTP server from a struct, an integer, a channel, and a function, all because interfaces are just sets of methods, which can be defined for (almost) any type.

Embedding

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

Interface embedding is very simple. We've mentioned the `io.Reader` and `io.Writer` interfaces before; here are their definitions.

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

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

The `io` package also exports several other interfaces that specify objects that can implement several such methods. For instance, there is `io.ReadWriter`, an interface containing both `Read` and `Write`. We could specify `io.ReadWriter` by listing the two methods explicitly, but it's easier and more evocative to embed the two interfaces to form the new one, like this:

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

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

The same basic idea applies to structs, but with more far-reaching implications. The `bufio` package has two struct types, `bufio.Reader` and `bufio.Writer`, each of which of course implements the analogous interfaces from package `io`. And `bufio` also implements a buffered reader/writer, which it does by combining a reader and a writer into one struct using embedding: it lists the types within the struct but does not give them field names.

```
// ReadWriter stores pointers to a Reader and a Writer.
// It implements io.ReadWriter.
type ReadWriter struct {
    *Reader  // *bufio.Reader
    *Writer  // *bufio.Writer
}
```

The embedded elements are pointers to structs and of course must be initialized to point to valid structs before they can be used. 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)
}
```

By embedding the structs directly, we avoid this bookkeeping. The methods of embedded types come along for free, which means that `bufio.ReadWriter` not only has the methods of `bufio.Reader` and `bufio.Writer`, it also satisfies all three interfaces: `io.Reader`, `io.Writer`, and `io.ReadWriter`.

There's an important way in which 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. In our example, when the `Read` method of a `bufio.ReadWriter` is invoked, it has exactly the same effect as the forwarding method written out above; the receiver is the `reader` field of the `ReadWriter`, not the `ReadWriter` itself.

Embedding can also be a simple convenience. This example shows an embedded field alongside a regular, named field.

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

The `Job` type now has the `Log`, `Logf` and other methods of `*log.Logger`. We could have given the `Logger` a field name, of course, but it's not necessary to do so. And now, once initialized, we can log to the `Job`:

```
job.Log("starting now...")
```

The `Logger` is a regular field of the struct and we can initialize it in the usual way with a constructor,

```
func NewJob(command string, logger *log.Logger) *Job {
    return &Job{command, logger}
}
```

or with a composite literal,

```
job := &Job{command, log.New(os.Stderr, "Job: ", log.Ldate)}
```

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. If we needed to access the `*log.Logger` of a `Job` variable `job`, we would write `job.Logger`. This would be useful if we wanted to refine the methods of `Logger`.

```
func (job *Job) Logf(format string, args ...interface{}) {
    job.Logger.Logf("%q: %s", job.Command, fmt.Sprintf(format, args...))
}
```

Embedding types introduces the problem of name conflicts but the rules to resolve them are simple. 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 `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 is a large topic and there is space only for some Go-specific highlights here.

Concurrent programming in many environments is made difficult by the subtleties required to implement correct access to shared 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.

This approach can be taken too far. Reference counts may be best done by putting a mutex around an integer variable, for instance. But as a high-level approach, using channels to control access makes it easier to write clear, correct programs.

One way to think about this model is to consider a typical single-threaded program running on one 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 synchronization. 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 seen as a type-safe generalization of Unix pipes.

Goroutines

They're 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 management.

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 Announce(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 closures: the implementation makes sure the variables referred to by the function survive as long as they are active.

These examples aren't too practical because the functions have no way of signaling completion. For that, we need channels.

Channels

Like maps, channels are a reference type and are allocated with `make`. 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
```

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

There are lots of nice idioms using channels. Here's one to get us started. In the previous section we launched a sort in the background. 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   // Wait 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. The capacity of the channel buffer limits the number of simultaneous calls to `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.
    }
}
```

Here's the same idea implemented by starting a fixed number of `handle` goroutines all reading from the request channel. The number of goroutines limits the number of simultaneous calls to `process`. This `Serve` function also accepts a channel on which it will be told to exit; after launching the goroutines it blocks receiving from that channel.

```
func handle(queue chan *Request) {
    for r := range queue {
```

```
        process(r)
    }
}

func Serve(clientRequests chan *Request, quit chan bool) {
    // Start handlers
    for i := 0; i < MaxOutstanding; i++ {
        go handle(clientRequests)
    }
    <-quit  // Wait to be told to exit.
}
```

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.

In the example in the previous section, `handle` was an idealized handler for a request but we didn't define the type it was handling. If that 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 to 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

Another application of these ideas is to parallelize a calculation across multiple CPU cores. If the calculation can be broken into separate pieces that can execute independently, it can be parallelized, with a channel to signal when each piece completes.

Let's say we have an expensive operation to perform on a vector of items, 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 v[n-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
}
```

We launch the pieces independently in a loop, one per CPU. They can complete in any order but it doesn't matter; we just count the completion signals by draining the channel after launching all the goroutines.

```
const NCPU = 4  // number of CPU cores

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

The current implementation of the Go runtime will not parallelize this code by default. It dedicates only a single core to user-level processing. An arbitrary number of goroutines can be blocked in system calls, but by default only one can be executing user-level code at any time. It should be smarter and one day it will be smarter, but until it is if you want CPU parallelism you must tell the run-time how many goroutines you want executing code simultaneously. There are two related ways to do this. Either run your job with environment variable `GOMAXPROCS` set to the number of cores to use or import the `runtime` package and call `runtime.GOMAXPROCS(NCPU)`. A helpful value might be `runtime.NumCPU()`, which reports the number of logical CPUs on the local machine. Again, this requirement is expected to be retired as the scheduling and run-time improve.

A leaky buffer

The tools of concurrent programming can even make non-concurrent ideas easier to express. Here's an example abstracted from an RPC package. The client goroutine loops receiving data from some source, perhaps a network. To avoid allocating and freeing buffers, it keeps a free list, and uses a buffered channel to represent it. If the channel is empty, a new buffer gets allocated. Once the message buffer is ready, it's sent to the server on `serverChan`.

```
var freeList = make(chan *Buffer, 100)
var serverChan = make(chan *Buffer)

func client() {
    for {
        var b *Buffer
        // Grab a buffer if available; allocate if not.
        select {
        case b = <-freeList:
            // Got one; nothing more to do.
        default:
            // None free, so allocate a new one.
            b = new(Buffer)
        }
        load(b)              // Read next message from the net.
        serverChan <- b      // Send to server.
    }
}
```

The server loop receives each message from the client, processes it, and returns the buffer to the free list.

```
func server() {
    for {
        b := <-serverChan    // Wait for work.
        process(b)
        // Reuse buffer if there's room.
        select {
        case freeList <- b:
            // Buffer on free list; nothing more to do.
        default:
            // Free list full, just carry on.
        }
    }
}
```

The client attempts to retrieve a buffer from `freeList`; if none is available, it allocates a fresh one. The server's send to `freeList` puts `b` back on the free list unless the list is full, in which case the buffer is dropped on the floor to be reclaimed by the garbage collector. (The `default` clauses in the `select` statements execute when no other case is ready, meaning that the `selects` never block.) This implementation builds a leaky bucket free list in just a few lines, relying on the buffered channel and the garbage collector for bookkeeping.

Errors

Library 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. 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 covers, making it possible not only to see the error but also to provide some context. For example, `os.Open` returns an `os.PathError`.

```
// 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
```

Such an error, which includes the problematic file name, the operation, and the operating system error it triggered, is useful even if printed far from the call that caused it; 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 package that generated the error. For example, in package `image`, the string representation for a decoding error due to an unknown format is "image: unknown 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 `PathErrors` this might include examining the internal `Err` field for recoverable failures.

```
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
}
```

The second `if` statement here is idiomatic Go. The type assertion `err.(*os.PathError)` is checked with the "comma ok" idiom (mentioned earlier in the context of examining maps). If the type assertion fails, `ok` will be false, and `e` will be `nil`. If it succeeds, `ok` will be true, which means the error was of type `*os.PathError`, and then so is `e`, which we can examine for more information about the error.

panic

The usual way to report an error to a caller is to return an `error` as an extra return value. The canonical `Read` method is a well-known instance; it returns a byte count and an `error`. But what if the error is unrecoverable? Sometimes the program simply cannot continue.

For this purpose, there is a built-in function `panic` that in effect creates a run-time error that will stop the program (but see the next section). The function takes a single argument of arbitrary type—often a string—to be printed as the program dies. It's also a way to indicate that something impossible has happened, such as exiting an infinite loop. In fact, the compiler recognizes a `panic` at the end of a function and suppresses the usual check for a `return` statement.

```go
// 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 problem can be masked or worked around, it's always better to let things continue to run rather than taking down the whole program. One possible counterexample is during initialization: if the library truly cannot set itself up, it might be reasonable to panic, so to speak.

```go
var user = os.Getenv("USER")

func init() {
    if user == "" {
        panic("no value for $USER")
    }
}
```

`recover`

When `panic` is called, including implicitly for run-time errors such as indexing an array 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.

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

```go
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. There's no need to do anything else in the deferred closure; calling `recover` handles the condition completely.

Because `recover` always returns `nil` unless called directly from a deferred function, deferred code can call library routines that themselves use `panic` and `recover` without failing. As an example, the deferred function in `safelyDo` might call a logging function before calling `recover`, and that logging code would run unaffected by the panicking state.

With our 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. Let's look at an idealized excerpt from the `regexp` package, which reports parsing errors by calling `panic` with a local error type. Here's the definition of `Error`, an `error` method, and the `Compile` function.

```
// Error is the type of a parse error; it satisfies the error interface.
type Error string
func (e Error) Error() string {
    return string(e)
}

// error is a method of *Regexp that reports parsing errors by
// panicking with an Error.
func (regexp *Regexp) error(err string) {
    panic(Error(err))
}

// Compile returns a parsed representation of the regular expression.
func Compile(str string) (regexp *Regexp, err error) {
    regexp = new(Regexp)
    // doParse will panic if there is a parse error.
    defer func() {
        if e := recover(); e != nil {
            regexp = nil    // Clear return value.
            err = e.(Error) // Will re-panic if not a parse error.
        }
    }()
    return regexp.doParse(str), nil
}
```

If `doParse` panics, the recovery block will set the return value to `nil`—deferred functions can modify named return values. It then will then check, in the assignment to `err`, that the problem was a parse error by asserting that it has the local type `Error`. If it does not, the type assertion will fail, causing a run-time error that continues the stack unwinding as though nothing had interrupted it. This check means that if something unexpected happens, such as an array index out of bounds, the code will fail even though we are using `panic` and `recover` to handle user-triggered errors.

With error handling in place, the `error` method makes it easy to report parse errors without worrying about unwinding the parse stack by hand.

Useful though this pattern is, it should be used only within a package. `Parse` turns its internal `panic` calls into `error` values; it does not expose `panics` to its client. That is a good rule to follow.

By the way, this re-panic idiom changes the panic value if an actual error occurs. However, both the original and new failures will be presented in the crash report, so the root cause of the problem will still be visible. Thus this simple re-panic approach is usually sufficient—it's a crash after all—but if you want to display only the original value, you can write a little more code to filter unexpected problems and re-panic with the original error. That's left as an exercise for the reader.

A web server

Let's finish with a complete Go program, a web server. This one is actually a kind of web re-server. Google provides a service at `http://chart.apis.google.com` that does automatic formatting of data into charts and graphs. It's hard to use interactively, though, because you need to put the data into the URL as a query. The program here provides a nicer interface to one form of data: given a short piece of text, it calls on the chart server to produce a QR code, a matrix of boxes that encode the text. That image can be grabbed with your cell phone's camera and interpreted as, for instance, a URL, saving you typing the URL into the phone's tiny keyboard.

Here's the complete program. An explanation follows.

```
package main

import (
    "flag"
    "log"
    "net/http"
    "text/template"
)

var addr = flag.String("addr", ":1718", "http service address")  // Q=17, R=18

var templ = template.Must(template.New("qr").Parse(templateStr))

func main() {
    flag.Parse()
    http.Handle("/", http.HandlerFunc(QR))
    err := http.ListenAndServe(*addr, nil)
    if err != nil {
        log.Fatal("ListenAndServe:", err)
    }
}

func QR(w http.ResponseWriter, req *http.Request) {
    templ.Execute(w, req.FormValue("s"))
}

const templateStr = `
<html>
<head>
<title>QR Link Generator</title>
</head>
<body>
{{if .}}
<img src=
"http://chart.apis.google.com/chart?chs=300x300&cht=qr&choe=UTF-8&chl={{urlquery .}}"/>
<br>
{{html .}}
<br>
<br>
```

```
{{end}}
<form action="/" name=f method="GET"><input maxLength=1024 size=70
name=s value="" title="Text to QR Encode"><input type=submit
value="Show QR" name=qr>
</form>
</body>
</html>
`
```

The pieces up to `main` should be easy to follow. The one flag sets a default HTTP port for our server. The template variable `templ` is where the fun happens. It builds an HTML template that will be executed by the server to display the page; more about that in a moment.

The `main` function parses the flags and, using the mechanism we talked about above, binds the function `QR` to the root path for the server. Then `http.ListenAndServe` is called to start the server; it blocks while the server runs.

`QR` just receives the request, which contains form data, and executes the template on the data in the form value named `s`.

The template package is powerful; this program just touches on its capabilities. In essence, it rewrites a piece of text on the fly by substituting elements derived from data items passed to `templ.Execute`, in this case the form value. Within the template text (`templateStr`), double-brace-delimited pieces denote template actions. The piece from `{{if .}}` to `{{end}}` executes only if the value of the current data item, called `.` (dot), is non-empty. That is, when the string is empty, this piece of the template is suppressed.

The snippet `{{urlquery .}}` says to process the data with the function `urlquery`, which sanitizes the query string for safe display on the web page.

The rest of the template string is just the HTML to show when the page loads. If this is too quick an explanation, see the documentation for the template package for a more thorough discussion.

And there you have it: a useful web server in a few lines of code plus some data-driven HTML text. Go is powerful enough to make a lot happen in a few lines.