A Tour of Go

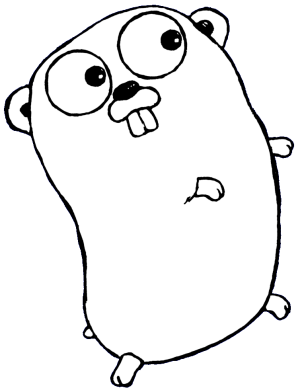
Russ Cox
rsc@golang.org
`http://golang.org/`

USENIX 2010

Go is a new, experimental, concurrent, garbage-collected programming language developed at Google over the last two years and open sourced in November 2009. It aims to combine the speed and safety of a static language like C or Java with the flexibility and agility of a dynamic language like Python or JavaScript. It is intended to serve as a convenient, lightweight, fast language, especially for writing concurrent systems software such as Web servers and distributed systems.

The tutorial today is a fast-paced tour of Go. There's no way we could cover everything about Go in one day, and we're not even going to try. Instead, we'll cover a few basics and then start writing real programs that do interesting things. More advanced aspects of the language will have to be left for you to explore on your own once we're done here. A good jumping off point would be Effective Go, linked at `http://golang.org/` and included at the end of this packet.

Today is structured as four 90-minute sessions, culminating in a room-wide file distribution system, a baby BitTorrent. Those sessions will all be hands on, with you coding for at least an hour in each. There are a series of exercises in each session, more than you'll have time to do. The first few convey the most important lessons; the rest typically cover more advanced tangents. If you don't get to them today, you might find them interesting to attack later.



Session 1. Nuts and Bolts

This session introduces the basics of Go program construction. At the end you should be able to write programs that import packages, define functions, call functions, iterate, and use simple data structures.

Installation

Everyone here should already have a laptop with Go installed. If you need to install Go, visit `http://golang.org/doc/install.html`. The short version is:

```
hg clone -r release https://go.googlecode.com/hg go
export GOROOT=$(pwd)/go
export GOARCH=386  # or amd64
export GOOS=linux  # or freebsd or darwin
export GOBIN=$HOME/bin  # somewhere writable in your $PATH
cd $GOROOT/src
./all.bash
```

If you don't have Mercurial (hg) installed, `http://swtch.com/usenix/go.tgz` is a checked out tree.

Course Materials

We're going to write a few programs today with the help of some library code. To install:

```
cd $GOROOT
curl http://swtch.com/usenix/tour.tgz | gunzip | tar xf -
cd tour
make install
```

Code

Now we're ready to write some code. The installation of the tour files installed a program called `goplay` which you can use to play with toy programs interactively. To run it, start `goplay` in a terminal window and then visit `http://localhost:3999/`. You'll see a simple hello, world program. Type Shift-Enter to compile and run it. As we examine Go's basic constructs, you might find this a nice, simple way to experiment.

Compiling binaries

Eventually you'll want to compile a real binary instead of using `goplay`. The directory `$GOROOT/tour/code` has a `Makefile` that will build the program `hello` if you write `hello.go` (for any value of `hello`). If you haven't already, write a simple ''hello, world'' program just to get started.

```
cd $GOROOT/tour/code
make hello
./hello
```

As a shorthand,

```
make hello.run
```

builds and runs `hello`.

Exercise: Loops and functions

As a simple way to play with functions and loops, implement the square root function using Newton's method. Your function should have the signature:

```
func Sqrt(x float64) float64
```

In this case, Newton's method is to approximate `Sqrt(x)` by picking a starting point z and then repeating:

$$z = z - \frac{z^2 - x}{2z}$$

To begin with, just repeat that calculation 10 times and see how close you get to the answer for various values (1, 2, 3, ...).

Next, change the loop condition to stop once the value has stopped changing. See if that's more or fewer iterations. How close are you to the actual square root? Call `math.Sqrt`. If you need to look up the documentation for that function, you can use `godoc`:

```
godoc –http=:6060 &
```
and visit `http://localhost:6060/pkg/math/`

or, from the command line, `godoc math` or `godoc math Sqrt`.

Exercise: Maps

Write a function

```
func WordCount(s string) map[string]int
```

that returns a map of the counts of each ''word'' in the string `s`. You can test it in your web browser by importing `tour/wc` in your program and calling `wc.Serve(WordCount)`. When you run the resulting binary, it will start a web server at `http://localhost:4000/` that counts words interactively. You might find `strings.Fields` helpful.

Exercise: Slices

Write a function

```
func Pic(dx, dy int) [][]uint8
```

that returns a slice of length `dy`, each element of which is a slice of `dx` 8-bit unsigned integers. Import `tour/pic` and call `pic.Serve(Pic)`. When you run the resulting binary, it will start a web server at `http://localhost:4000/` that displays the slice data interpreted as grayscale pixel values. The choice of image is up to you. Interesting functions include `x^y`, `(x+y)/2`, and `x*y`.

Advanced Exercise: Complex cube roots

Let's explore Go's built-in support for complex numbers via the `complex`, `complex64`, and `complex128` types. For cube roots, Newton's method amounts to repeating:

$$z = z - \frac{z^3 - x}{3z^2}$$

Find the cube root of 2, just to make sure the algorithm works.

Then change your algorithm to use `complex128` instead of `float64` and try to find the cube root of 1. Find all three by trying different initial values of z.

Advanced Exercise: High precision

Package `big` implements arbitrary precision integer arithmetic. Convert your square root program to work in fixed point arithmetic for some number of decimal places `P` and then compute 50 digits of the square root of 2. Double check them against the output of `godoc math Sqrt2`.

Session 2. Interfaces

This session explores the Go concept of interfaces by writing implementations of the HTTP handler and image interfaces and then combining them into an interactive Mandelbrot set viewer.

Interfaces

Many of Go's libraries are structured around *interfaces*, which are satisfied by any implementation with the right methods, even ones that didn't explicitly attempt to satisfy (or even know about) the interface. Part of the reason interfaces are widely used in Go is that they are lightweight, with no annotations or explicit type hierarchy to maintain.

Web servers

We started a few web servers indirectly in the last session. Let's look at how that worked. The package `http` implements a web server that serves incoming requests using a `Handler`, defined as

```
type Handler interface {
    ServeHTTP(conn *http.Conn, req *http.Request)
}
```

Any object with that method can serve HTTP requests. The package maintains a multiplexing handler that allows clients to register for subtrees of the file name space. If you call

```
http.Handle("/file", myFileHandler)
http.Handle("/dir/", myDirHandler)
```

then the multiplexing handler will call `myFileHandler.ServeHTTP` to handle a request for `/file`, and it will call `myDirHandler.ServeHTTP` to handle a request in the `/dir/` tree (any path beginning with `/dir/`).

The second argument to `http.ListenAndServe` is an object implementing the `Handler` interface. If you pass `nil`, the server uses the multiplexing handler that `http.Handle` manipulates.

Exercise: Hello, World 2.0

Write a web server that responds to every request by saying

`hello,` *name*

where *name* is the value of the request parameter `name`. (Call `req.FormValue("name")`; for more about the `Request`, run `godoc http Request` or visit `http://localhost:6060/pkg/http/#Request` if you still have the `godoc` web server running.) Other hints: `http.Conn` implements the `io.Writer` interface; convert a `string` to `[]byte` by using `[]byte(s)`; or use `io.WriteString`.

If you call `ListenAndServe` using the address `":4000"`, then the server will be accessible at `http://localhost:4000/`. (For the rest of the tour we'll assume that any servers you write are serving that address.)

Exercise: Methods

Implement the following types and define `ServeHTTP` methods on them. Register them to handle specific paths in your web server.

```
type String string

type Struct struct {
    Greeting string
    Punct string
    Who string
}
```

For example, you should be able to register handlers using:

```
http.Handle("/string", String("I'm a frayed knot."))
http.Handle("/struct", &Struct{"Hello", ":", "USENIX!"})
```

Images

We've been implementing the single-function `http.Handler` interface. Now a more complex example. The interface `image.Image` is defined as:

```
type Image interface {
    ColorModel() ColorModel
    Width() int
    Height() int
    // At(0, 0) returns the upper-left pixel of the grid.
    // At(Width()-1, Height()-1) returns the lower-right pixel.
    At(x, y int) Color
}
```

`Color` and `ColorModel` are themselves interfaces, but we'll ignore that by using the predefined types `image.RGBAColor` and `image.RGBAColorModel`.

Exercise: Image Viewer

Change your implementation of the function `Pic` from the last session to return an `image.Image` instead of a `[][]uint8`. You'll probably want to define your own type `Image` and then define the necessary methods on it. `ColorModel` should just return `image.RGBAColorModel`, and `At` should return `RGBAColor{p, p, p, 255}` where `p` is the pixel's grayscale value.

Call `pic.ServeImage` instead of `pic.Serve` and test your program by visiting `http://localhost:4000/`.

Exercise: PNG Encoding

Let's eliminate some more scaffolding. Instead of using `tour/pic`, write an HTTP handler that serves the image directly and register it on `/`. You could do this by implementing a `ServeHTTP` method for your `Image` type and then passing `Pic()` as the second argument to `http.Handle`.

Before the `ServeHTTP` method writes any data to `conn`, it will have to call

```
conn.SetHeader("Content-Type", "image/png")
```

to set the Content-Type line sent in the response header. To convert an `Image` into a PNG stream, take a look at the `image/png` package.

Test your program by visiting `http://localhost:4000/`.

Exercise: Mandelbrot Set Viewer

The Mandelbrot set, first studied by the mathematician Benoit Mandelbrot, is a collection of points with no firm boundary. Plotting it produces intricate pictures and serves as a surprising implementation of the `Image` interface. To test whether a particular value *c* is in the Mandelbrot set, we set *z* to zero and repeat

$$z = z^2 + c$$

for a fixed number of iterations. If we notice that *z* is at least distance 2 away from the origin (`cmath.Abs(z) > 2`), we stop the loop early and declare `c` not in the set. Otherwise, if we get to the end of the loop, `c` is in the set.

Implement a web server that serves `/mandelbrot` by generating a PNG image, much like in the last example. This image, however, should not be a two-dimensional array. It should compute the color of a particular pixel only during the call to the `At` method (on demand). The underlying image type should be a struct that records the data needed to implement the `Image`. (Notice that no matter how large an image is requested, the data structure satisfying `Image` and being passed to the PNG encoder is a constant size!)

The HTTP request parameter `p` will be a string of the form *dx dy c0 dc n*. (You can use `fmt.Sscan` to parse it.) The parameters are the pixel width *dx* and height *dy* (both `int`) of the desired image, the complex (`complex128`) value *c0* corresponding to pixel (0,0), the complex range *dc* covered by the image, and the maximum number of iterations *n*. The value of *c* to use at (*x*,*y*) is given by:

```
fx := float64(x) / float64(dx)
fy := float64(y) / float64(dy)
c := c0 + cmplx(real(dc)*fx, imag(dc)*fy)
```

To get started, make the pixel black if `c` is in the Mandelbrot set and white otherwise. You can test your code by visiting `http://localhost:4000/mandelbrot?p=512+512+-2-1i+3%2b2i+100`. If you'd rather not type that, you can use `http://swtch.com/go/mb`.

Once you get a Mandelbrot set image on your screen at that URL, import `tour/fractal` and register `fractal.MainPage` as the handler for `/`. Then visit `http://localhost:4000/`. The page handler generates an interactive web page that uses calls to your Mandelbrot handler to generate image tiles. You can pan in the viewer by dragging with the mouse and zoom by using the scroll wheel or by typing + and −.

Exercise: Colored Mandelbrot

The Mandelbrot set is usually not plotted in monochrome. More striking pictures can be generated by assigning each pixel a color that depends on the number of iterations it took before `c` got too big. The `tour/fractal` package implements some interesting color schemes. Use one of them instead of black and white, or write your own (you might look at the `$GOROOT/src/pkg/tour/fractal/color.go` source code for inspiration).

Advanced Exercise: Julia Set Viewer

The Mandelbrot calculation starts with *z* always set to zero but a different value of *c* for each pixel. Another equally intricate set, first proposed by the mathematician Gaston Julia, can be generated by making *c* a constant and using a different value of *z* for each pixel. Implement a handler `/julia` that takes the parameter string *dx dy z0 dz c n*. Test it by typing the space bar when in the Mandelbrot viewer.

Pressing the space bar in the fractal viewer will switch to a Julia set using the center of the Mandelbrot image as the value `c`. Pressing space bar again will switch back to the Mandelbrot set.

Advanced Exercise: Newton Attractors

If you type `n` the fractal viewer will start loading images from the URL `/newton`. Implement a handler for `/newton` that synthesizes images by running cube root iteration from the last session. Color each pixel red, green, or blue depending on which cube root it eventually converges to, and adjust the brightness of the pixel by how many iterations it took to converge.

Session 3. Concurrency with Channels

This session explores the Go approach to concurrency: goroutines and channels.

Share memory by communicating

Concurrent programs written in most standard languages (for example, C++ or Java) communicate between threads by sharing memory. It is as if all the threads are writing on the same sheet of paper, using marks on some sections of the paper to coordinate who should read from or write to other sections. Go encourages a different approach, in which the concurrent processes—called goroutines—share memory by communicating. That is, they send each other explicit messages, and those messages often include pointers. The convention is that sending a pointer to another goroutine gives ownership to that goroutine until it gets sent back. It is as if the goroutines are writing on many different pieces of paper and coordinating by passing the papers amongst themselves.

The topic of writing concurrent programs in this way could fill (has filled, actually) many books. We'll just look at a few simple cases here.

Goroutines

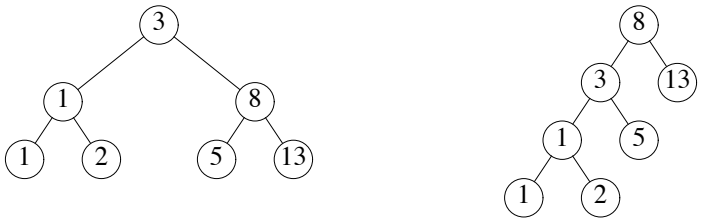
To create a new goroutine running `f(x, y, z)`, execute the statement `go f(x, y, z)`. It's that simple.

Channels

Having another goroutine running isn't that useful unless there is some way to share information with it. This is the purpose of channels. Channels are typed message buffers with send and receive operations. The send operation blocks if the buffer is full. The receive operation blocks if the buffer is empty. (You can also create a channel without a buffer, but we won't discuss that further; see ''Effective Go'' for more.)

Exercise: Equivalent Binary Trees

There can be many different binary trees with the same sequence of values stored at the leaves. For example, here are two binary trees storing the sequence 1, 1, 2, 3, 5, 8, 13.



A function to check whether two binary trees store the same sequence is quite complex in most languages. We'll use Go's concurrency and channels to write a simple solution.

Import the package `tour/tree`, which defines the type

```
type Tree struct {
        Left *Tree
        Value int
        Right *Tree
}
```

Write a function

```
func Walk(t *tree.Tree, ch chan int)
```

that walks the tree `t` sending all values from the tree into `ch`.

The function `tree.New(k)` constructs a randomly-structured binary tree holding the values `k`, `2k`, `3k`, `10k`. Test `Walk` by creating a new channel `ch` and then kicking off the walker:

```
go Walk(tree.New(1), ch)
```

Then read and print 10 values from the channel. It should be the numbers 1, 2, 3, ..., 10.

Now write a function

```
func Same(t1, t2 *tree.Tree) bool
```

that uses `Walk` to determine whether `t1` and `t2` store the same values. `Same(tree.New(1), tree.New(1))` should return true, and `Same(tree.New(1), tree.New(2))` should return false.

Web Channel

Channels are a useful way to connect independently-executing parties. Go channels connect goroutines within a single program, but a common pattern for connecting different programs is to have a goroutine serve as a proxy, reading from a channel and then sending the messages over a network. The package `tour/ajax` implements such a proxy: it delivers messages sent on `ajax.Chan` to the JavaScript running inside a web browser. The supplied browser code reacts differently to messages of different types. If you define

```
type Log string
```

and send a message of that type—like `Log("hello")`—on the channel, the browser code will display the message in a section of the page reserved for debugging messages.

Exercise: Web Logging

Import `tour/ajax` and register `ajax.LogPage` as the handler for `/`. Kick off a goroutine that sends an infinite sequence (say, 1, 2, 3, 4, 5, ...; you'll need `fmt.Sprintf`) of `Log` messages on `ajax.Chan`, sleeping for a second after each send (see `time.Sleep`). Load `http://localhost:4000/` and you should see the log messages. The implementation of `tour/ajax` broadcasts, so if you open a second page at that URL you'll see the same messages in both windows.

Exercise: Chat Room

We can take advantage of the broadcast to write a simple chat server. Again we have some prepackaged JavaScript to interact with the web server that you'll write. You need to implement handlers for three pages: `/join` handles the addition of a member in the chat room, `/say` handles the speaking of a message, and `/exit` handles the exiting of a member of the chat. All three take a form parameter `id` giving the user name of the member doing the action. The `/say` handler takes an extra parameter `msg` that is the message text.

The suggested implementation is to create a new goroutine managing the chat room. All three HTTP handlers turn into structures sent via a channel to the chat manager. For example, the handler for `/join` might define

```
type Join struct {
        Who string
}
```

and then send `chatRoom <- Join{req.FormValue("id")}`. The chat manager goroutine can use a type switch to determine what kind of message was received:

```
for {
        switch m := (<-chatRoom).(type) {
        case Join:
                ajax.Chan <- Chat{m.Who, m.Who + " is here."}
        ...
        }
}
```

As illustrated in the last snippet, sending a message of type `Chat`, defined as

```
type Chat struct {
        Who string
        Msg string
}
```

to `ajax.Chan` will make the browser display the chat message.

Import `tour/chat` and register `chat.MainPage` as the handler for `/`. Open two or more browser windows on `http://localhost:4000/`. They will prompt for a user name and then join the chat.

Advanced Exercise: User Lists

The manager goroutine didn't do much in the last exercise. Let's change that. Make a `map[string]bool` whose keys are the current chat room participants (we'll ignore the `bool` value). When a user joins (or exits), add that user's name to the map (or remove it) and send a message with a list of all the users, formatted like:

```
[+newUser, oldUser1, oldUser2]
[-userWhoLeft, remainingUser1, remainingUser2]
```

If you define

```
type Status string
```

and send the update on `ajax.Chan` as type `Status`, it will be displayed differently from chat messages. (By the way, to delete the key `k` from a map of the type above, use `m[k] = false, false`.)

The goroutine is now serving a purpose: it serializes the access to the user map. We could have used a lock instead, but framing the program as a collection of communicating entities is often easier to understand than locking disciplines.

Advanced Exercise: Chat Bot

Some IRC chat rooms have bots that can be trained to repeat specific lines. This can be useful for saving frequent responses. Change the handling of the `/say` message to send all messages to a new global variable

```
var parrotChan = make(chan interface{}, 100)
```

and start a goroutine to read and process them. If the parrot goroutine sees a message of the form

```
[user] parrot: faq = http://golang.org/doc/go_faq.html
```

it should react by sending a message acknowledging the training:

```
[parrot] user: faq = http://golang.org/doc/go_faq.html
```

If the parrot sees a message of the form

```
[user] parrot: tell jim about faq
```

it should react by sending a message:

```
[parrot] jim: http://golang.org/doc/go_faq.html
```

`strings.Split` is probably useful for implementing the parrot. (Remember that the `[user]` prefixes are not part of the incoming message string.)

Advanced Exercise: Julia Set Bot

The chat messages are arbitrary HTML. In general this is a security hole and not to be recommended, but it makes it easy to send rich media in chat messages. Hook your Julia Set image generator from the last session up as another bot, to support the following interaction:

```
[user] julia: 0.285+0.01i
[julia] user: <img src="/julia?p=320+240+-2%2bi+4-4i+0.285%2b0.01i+100">
```

(When `julia` replies with the `<img>` tag, the browser will render it as an actual image fetched from your server.)

Session 4. Networking with RPCs

This session explores the native Go RPC package, which lets a Go program on one machine call a function served by another. Building on the experience from the last two sessions, we'll build a simple BitTorrent-like media distribution network.

RPC

Go provides a simple RPC package that allows programs to make calls across a network connection. The documentation is available by using `godoc rpc` or visiting `http://localhost:6060/pkg/rpc/`.

The exercises in this session will require connecting to other machines on the network and accepting connections. You may need to change your firewall settings to allow incoming connections on TCP port 4000.

The master node running on my machine will coordinate the network. It will have the address `master.swtch.com:4000` but I'll also post the IP address so that you can hard-code it in your clients and avoid any DNS problems.

All the RPCs we're doing will be tunneled over HTTP, so use `rpc.DialHTTP` and `rpc.HandleHTTP`.

Exercise: Expression Evaluation Client

The master node is running an expression evaluation service as `Master.Eval`. The argument struct should have a string `Expr` and the result will have a string `Value`. Write an RPC client that sends arithmetic expressions (perhaps from the command line; you can use `os.Args[1]` to get the first argument) and prints the result. Try to evaluate a malformed expression too, to test error handling.

Exercise: Ping Server

Implement an RPC server that provides a method `Node.Ping` that takes an empty argument struct and sends back an empty result struct. Add another method `Node.Debug` that takes an argument struct with a `Msg string` field (and an empty result struct) and prints the message to standard error.

Connect to the master node and call `Master.TestPing`. The master node will attempt to call you back and execute first a `Node.Ping` and then a `Node.Debug` RPC. If either fails, the `TestPing` will return an error.

A Distribution Network

Now we're ready to start talking to other machines. Each machine in the network is trying to construct an image from fragments being passed from machine to machine. The routing algorithm is incredibly simple: if you receive a message and it is for you, keep it. If not, ask the master for the address of a random node in the network and forward it to that node. As your program accumulates image fragments, it will send them to your web browser so you can watch the pieces come in.

Exercise: Direct Downloads

Before we go to a full blown distributed network, let's work on the web plumbing to show a simple download.

Implement a web server that imports `tour/distro` and registers `distro.MainPage` as the handler for the URL `/`. Then implement a handler for the URL `/start` that returns immediately (it need not send any data) but kicks off a goroutine to call `Master.Start` with the argument and result types

```
type StartArgs struct {
    Distributed bool
}
type StartResult struct {
    ID string
}
```

For now, leave `Distributed` set to `false`.

Implement handlers for these RPCs:

`Node.Ping`
: Same as above.

`Node.Debug`
: Same signature as above. Can keep the same behavior, or you might find it useful to send a `Log` message to `ajax.Chan`, as in the earlier sessions.

`Node.Relay`
: The argument is a struct

```
type RelayArgs struct {
    ID string  // ignore for now
    X, Y int
    Pic []byte
}
```

and the response is an empty struct.

The implementation of `Node.Relay` needs to hand the message to the web browser, but it can't send the bytes directly. It needs to send a URL instead. The helper function `distro.Post` will remember the bytes and return a URL that will serve them.

`Node.Relay` should fill in a `distro.Fragment` structure with the `X`, `Y`, and `URL` fields set appropriately and send that structure to `ajax.Chan`. When the web browser receives the structure, it will display the image fragment.

Once you've implemented the handlers for the URL `/start` and the RPCs `Node.Ping`, `Node.Debug`, and `Node.Relay`, load `http://localhost:4000/`. It should load an image block by block. Each block's data is being sent directly from the master server using the `Relay` RPC.

Exercise: Distributed Downloads

Now we can introduce other nodes. Change the implementation of `Relay` to check whether the `ID` in the arguments matches the `ID` returned by `Master.Start`. If so, proceed as before, sending the fragment to the web browser. If not, your node needs to hand the fragment to some other node. Normally there would be a fancy routing protocol here, but we're going to do something considerably simpler (and less efficient): just relay the message to some other random node in the network.

To forward the message, call `Master.Random`, which takes an empty argument struct and returns a struct with a single `Addr string` field, the address of a random node in the network. Establish a new RPC client connecting to that address (it will include the port number and can be passed directly to `rpc.DialHTTP`) and call its `Node.Relay` method to pass the message on. Then hang up the connection (`client.Close`).

Change your call to `Master.Start` to set `Distributed` to `true`.

Load `http://localhost:4000/`. This time the image fragments should appear in random order instead of sequential order.

Advanced Exercise: Snooping

Maybe it's more fun to see everyone else's pictures instead of your own. Change `Node.Relay`: keep the relaying code, but in addition to handing off the fragment to a random node, send it to `ajax.Chan` too. Now your browser should show an amalgam of all the ''chatter'' that is routing through your node.

Advanced Exercise: PNG Board

Implement your own version of `distro.Post`. You'll need to pick a URL prefix like `/png/` and register a handler and then have `distro.Post` somehow inform the handler about new images. It will probably help to have a goroutine in charge of the posting board, just as we had a goroutine in charge of the chat room.

More Information

More information about Go can be found at `http://golang.org/`. It has links to tutorial information, an IRC channel, and a mailing list.

There are many interesting topics that we didn't even have time to mention today, including writing packages, embedding, parallelization using concurrency, reflection (the `reflect` package), templates (the `template` package), and more. The document ''Effective Go,'' included as the rest of this packet, touches on many of these and is a good choice for the next thing to read.

You might also find it instructive to look at the implementation of the various `tour/*` packages, found on your system in `$GOROOT/src/pkg/tour/`; most of them hide concepts that we didn't have time to cover today but that are interesting in their own right.

Most importantantly, have fun with Go!

Effective Go

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 [1] and the tutorial [2], both of which you should read first.

Examples

The Go package sources [3] 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.

2. 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. A program, `gofmt`, 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, fix the program (or file a bug), 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 code in the libraries 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 require parentheses in their syntax. Also, the operator precedence hierarchy is shorter and clearer, so

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

means what the spacing implies.

3. 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.

```
/*
    The regexp package 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.

```
// The path package 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. Finally, the comments are uninterpreted plain text, so HTML and other annotations such as `_this_` will reproduce *verbatim* and should not be used.

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 English 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, error os.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      = os.NewError("internal error")
    ErrUnmatchedLpar = os.NewError("unmatched '('")
    ErrUnmatchedRpar = os.NewError("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
)
```

4. 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/container/vector` is imported as `"container/vector"` but has name `vector`, not `container_vector` and not `containerVector`.

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), 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`. Clients of the package see that 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.

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.

5. 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, add 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()
}
```

6. Control structures

The control structures of Go are related to those of C but different 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: parentheses are not required 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.Stderr(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, os.O_RDONLY, 0)
if err != nil {
    return err
}
codeUsing(f)
```

This is a example of a common situation where code must analyze a sequence of error possibilities. 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, os.O_RDONLY, 0)
if err != nil {
    return err
}
d, err := f.Stat()
if err != nil {
    return err
}
codeUsing(f, d)
```

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 you.

```
var m map[string]int
sum := 0
for _, value := range m {  // key is unused
    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 "日本語" {
    fmt.Printf("character %c starts at byte position %d\n", char, pos)
}
```

prints

```
character 日 starts at byte position 0
character 本 starts at byte position 3
character 語 starts at byte position 6
```

Finally, since Go has no comma operator and ++ and –– are statements not expressions, 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)
}
```

7. Functions

Multiple return values

One of Go's unusual features is that functions and methods can return multiple values. This 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 os.Error) {
    for len(buf) > 0 && err == nil {
        var nr int
        nr, err = r.Read(buf)
        n += nr
        buf = buf[nr:len(buf)]
    }
    return
}
```

8. Data

Allocation with `new()`

Go has two allocation primitives, `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 essentially the same as its namesakes in other languages: `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 that the zeroed object 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 this 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 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 zero) 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; 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]float) (sum float) {
    for _, v := range *a {
        sum += v
    }
    return
}
array := [...]float{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 os.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, this snippet would also read the first 32 bytes of the buffer.

```
var n int
var err os.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.

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, floats, strings, pointers, and interfaces (as long as the dynamic type supports equality). Structs, arrays and slices cannot be used as map keys, because equality is not defined on those types. 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`.

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.Stderr("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*, a simple underscore (_). 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, turn the multiple assignment around by placing an extra boolean on the right; if the boolean is false, the entry is deleted. It's safe to do this even if the key is already absent from the map.

```
timeZone["PDT"] = 0, false  // 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 `ln` version also inserts a blank between arguments if neither is a string and appends a newline to the output. In this example each line produces the same output.

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

As mentioned in the tutorial [2], `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 float
    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 `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"
```

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 for its final argument to specify that an arbitrary number of parameters can appear after the format.

```
func Printf(format string, v ...) (n int, errno os.Error) {
```

Within the function `Printf`, `v` is a variable that can be passed, for instance, to another print routine. Here is the implementation of the function `log.Stderr` we used above. It passes its arguments directly to `fmt.Sprintln` for the actual formatting.

```
// Stderr is a helper function for easy logging to stderr. It is analogous to Fprint(os.Stderr).
func Stderr(v ...) {
    stderr.Output(2, fmt.Sprintln(v))  // Output takes parameters (int, string)
}
```

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

9. 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`.

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 `init()` function to set up whatever state is required. The only restriction is that, although goroutines can be launched during initialization, they will not begin execution until it completes; initialization always runs as a single thread of execution. 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.Exit("$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")
}
```

10. 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 os.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`.

11. 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 float, 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.IntArray` to reduce the entire example to this:

```
type Sequence []int
// Method for printing - sorts the elements before printing
func (s Sequence) String() string {
    sort.IntArray(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.IntArray` 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 `crypto/block` package to be separated from the block ciphers they chain together. By analogy with the `bufio` package, they wrap a `Cipher` interface and return `hash.Hash`, `io.Reader`, or `io.Writer` interface values, not specific implementations.

The interface to `crypto/block` includes:

```
type Cipher interface {
    BlockSize() int
    Encrypt(src, dst []byte)
    Decrypt(src, dst []byte)
}
// NewECBDecrypter returns a reader that reads data
// from r and decrypts it using c in electronic codebook (ECB) mode.
func NewECBDecrypter(c Cipher, r io.Reader) io.Reader
// NewCBCDecrypter returns a reader that reads data
// from r and decrypts it using c in cipher block chaining (CBC) mode
// with the initialization vector iv.
func NewCBCDecrypter(c Cipher, iv []byte, r io.Reader) io.Reader
```

`NewECBDecrypter` and `NewCBCReader` apply not just to one specific encryption algorithm and data source but to any implementation of the `Cipher` interface and any `io.Reader`. Because they return `io.Reader` interface values, replacing ECB encryption with CBC encryption is a localized change. The constructor calls must be edited, but because the surrounding code must treat the result only as an `io.Reader`, 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(*Conn, *Request)
}
```

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(c *http.Conn, req *http.Request) {
    ctr.n++
    fmt.Fprintf(c, "counter = %d\n", ctr.n)
}
```

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

```
import "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(c *http.Conn, req *http.Request) {
    *ctr++
    fmt.Fprintf(c, "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(c *http.Conn, req *http.Request) {
    ch <- req
    fmt.Fprint(c, "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 i, 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(*Conn, *Request)
// ServeHTTP calls f(c, req).
func (f HandlerFunc) ServeHTTP(c *Conn, req *Request) {
    f(c, 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(c *http.Conn, req *http.Request) {
    for i, s := range os.Args {
        fmt.Fprintln(c, 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 `IntArray` to access `IntArray.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.

12. 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 os.Error)
}
type Writer interface {
    Write(p []byte) (n int, err os.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:

```
// ReadWrite is the interface that groups the basic Read and Write methods.
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 os.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, nil, "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 ...) {
    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 `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.

13. 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 in parallel 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 in parallel; don't wait for it.
```

A function literal can be handy in a goroutine invocation.

```
func Announce(message string, delay int64) {
    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 *clientRequests, 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, 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 gc (6g, etc.) 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 (default 1); or import the runtime package and call runtime.GOMAXPROCS(NCPU). 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 {
        b, ok := <-freeList  // grab a buffer if available
        if !ok {              // if not, allocate a new one
            b = new(Buffer)
        }
        load(b)              // read next message from the net
        serverChan <- b      // send to server
    }
}
```

The server loop receives messages from the client, processes them, and returns the buffer to the free list.

```
func server() {
    for {
        b := <-serverChan    // wait for work
        process(b)
        _ = freeList <- b    // reuse buffer if room
    }
}
```

The client's non-blocking receive from freeList obtains a buffer if one is available; otherwise the client allocates a fresh one. The server's non-blocking send on 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 assignment of the send operation to the blank identifier makes it non-blocking but ignores whether the operation succeeded.) This implementation builds a leaky bucket free list in just a few lines, relying on the buffered channel and the garbage collector for bookkeeping.

14. 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 `os.Error`, a simple interface.

```
type Error interface {
    String() 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.
    Error Error  // Returned by the system call.
}
func (e *PathError) String() string {
    return e.Op + " " + e.Path + ": " + e.Error.String()
}
```

`PathError`'s `String` 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".

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 `Error` field for recoverable failures.

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

15. 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"
    "http"
    "io"
    "log"
    "template"
)
var addr = flag.String("addr", ":1718", "http service address") // Q=17, R=18
var fmap = template.FormatterMap{
    "html": template.HTMLFormatter,
    "url+html": UrlHtmlFormatter,
}
var templ = template.MustParse(templateStr, fmap)
func main() {
    flag.Parse()
    http.Handle("/", http.HandlerFunc(QR))
    err := http.ListenAndServe(*addr, nil)
    if err != nil {
        log.Exit("ListenAndServe:", err)
    }
}
func QR(c *http.Conn, req *http.Request) {
    templ.Execute(req.FormValue("s"), c)
}
func UrlHtmlFormatter(w io.Writer, v interface{}, fmt string) {
    template.HTMLEscape(w, []byte(http.URLEscape(v.(string))))
}
const templateStr = `
<html>
<head>
<title>QR Link Generator</title>
</head>
<body>
{.section @}
<img src="http://chart.apis.google.com/chart?chs=300x300&cht=qr&choe=UTF-8&chl={@|url+html}"
/>
<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, inspired by json-template [4], 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`), brace-delimited pieces denote template actions. The piece from the `{.section @}` to `{.end}` executes with the value of the data item `@`, which is a shorthand for ''the current item'', which is the form value. (When the string is empty, this piece of the template is suppressed.)

The snippet `{@|url+html}` says to run the data through the formatter installed in the formatter map (`fmap`) under the name `"url+html"`. That is the function `UrlHtmlFormatter`, which sanitizes the 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 [5] for a more thorough discussion.

And there you have it: a useful webserver 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.

References

[1] Go Language Specification, http://golang.org/doc/go_spec.html
[2] Go Tutorial, http://golang.org/doc/go_tutorial.html
[3] Go Package Sources, http://golang.org/src/pkg/
[4] JSON Template, http://code.google.com/p/json-template
[5] Documentation for Go's template package, http://golang.org/pkg/template/