Core Go Language Notes

UNDERSTANDING > KNOWLEDGE

DANCING CLOUD SERVICES, LLC

Basics

- Packages provide namespaces and encapsulation.
- Only elements starting with *capital letters* are visible outside the package that declares them

```
// double-forward-slash for comment to end of line, /* … */ comments a range
package main // program entry must be in main package
import "fmt" // single import, usually use multiple form below

// program entry is main(), no arguments, no return type
func main() {
   fmt.Println("Hello Go World!")
}
```

Modules and packages

- Packages are used to group related code and provide encapsulation boundaries
- A single directory should map to a single package and package and directory names should match
- Elements are encapsulated by a package unless their names have a capital first letter
  - This is Go's mechanism for encapsulation
- Multiple packages can exist in subdirectories in a single module
- If code exists in the root directory of a module it should be in only one of two possible packages
  - main – which is used to contain the entry point for a program, defined by a function also called `main`
  - or a package whose name matches the name of the module
- The root directory of a module is identified by a special file called `go.mod` which describes:
  - The full name of the module (which should end with the name of this directory)
  - The minimum version of Go that is adequate for this module
  - Any other modules upon which this one depends using "require" directives
  - Additional information, perhaps including how to find other modules
- A module's name is made up of a URL-like prefix and a path

A project with a single module and two packages

- Given this directory layout:

```
.
└── projmain
    ├── go.mod
    ├── main.go
    └── support
        └── extras.go
```

- The go.mod file declares the full name of the module and minimum Go version:

```
module dancingcloudservices.com/projmain
go 1.21
```

- The code has two packages:
  - One is necessarily called support, in the directory support
  - The other could be called projmain, or simply main
- The file extras.go declares itself in the package support and contains this code:

```
package support
var SupportMessage = "My supporting data"
```

- Notice the variable is exported due to the capital first letter

A project with a single module and two packages

- In this example, the file **`main.go`** declares itself in the package **`main`** and contains the **`main`** function which is the program's entry point
- The program imports the **`support`** package from the subdirectory (but notice the full name is used)
- It also imports a package called **`fmt`**, used for basic printing and formatting features

```
package main
import (
    "dancingcloudservices.com/projmain/support"
    "fmt"
)
func main() {
    fmt.Println(support.SupportMessage)
}
```

- When run, the code produce the output "`My supporting data`"

A project with two local modules

- In this project, we have two modules:
  - `dancingcloudservices.com/support`
  - `dancingcloudservices.com/projmain`
- The directory tree now looks like this:

```
.
├── projmain
│   ├── go.mod
│   └── main.go
└── support
    ├── go.mod
    └── messages
        └── extras.go
```

- The file `extras.go` is essentially identical to the previous version, in a package called `messages`
- The go.mod file in the support directory contains the following:

```
module dancingcloudservices.com/support
go 1.21
```

Literals

- Literal types are:
  - integer: `99`
  - floating point: `3.14`, `3e6`
  - string: `"Hello"`
  - Unicode (aka "rune"): `'a'`
  - boolean: `true`
  - complex: `3i`
- Literal types are very high precision, and can be assigned directly to compatible targets with conversion automatically applied
- Underscores can separate digits: `1_000`
- Binary, octal, and hexadecimal numeric literals are supported: `0b0101010`, `012`, and `0xCAFE`

- Arrays with literal initialization use a curly brace form

```
fib1 := [...]int{1, 1, 2, 3, 5, 7} // has 6 elements
fib2 := [10]int{1, 1, 2, 3, 5, 7}   // has 10 elements, last 4 are zero
```

Constants

- Literals form constants, constants can also be declared as such, with explicit type if desired:

```
const NINETY_NINE = 99.0
const PI float64 = 3.14
```

Constants:

- have a much higher precision than any of the built in variable types
- calculations done with them are handled by the compiler, at high precision
- can be assigned to variables of any sensible target type, in which case type conversion is applied (during compilation)
- some range checking is applied on assignment to a variable

Constant initialization with iota

- A series of distinct integer type constants can be initialized using the constant generator **iota**

```
type Month int8 // Not essential, could simply be int
const (
    January  Month = iota + 1
    February Month = iota + 1
    March    Month = iota + 1
    // etc
)
fmt.Println(March) // prints 3 (type is <package>.Month)
```

- **iota** starts with the value zero, is reinitialized with each use of the const keyword, and increments with each declaration in that const block, so in this example, the month values start at 1
- This can be used in a variety of ways, for example, defining mutually independent values of a set as bits:

```
const (
   READABLE = 1 << iota   // 1
   WRITEABLE = 1 << iota  // 2
   EXECUTABLE = 1 << iota // 4
)
```

Built in types

Go types provide hardware-level control over storage format

- `bool` (true, false)
- `string` (double quotes)
- `int` (usually machine word width)
- `int8  int16  int32  int64`
- `uint` (unsigned, machine word again)
- `uint8 uint16 uint32 uint64 uintptr`
- `float32 float64`
- `complex64 complex128`
- `byte` (alias for `uint8`)
- `rune` (alias of `uint32`, unicode)

- `struct`
- array
- slice
- `function`
- `map`
- `interface`
- `channel`
- pointer

- uint8 and byte can be freely assigned to each other

Documentation of built-in features

- Documentation for many things that are considered built-in is in a "pseudo-package" called builtin
  - It's not really a package, but API documentation is organized by package
  - https://pkg.go.dev/builtin
- This "package" includes several type declarations, for example:

  ```
  type byte = uint8
  ```
- Also present are a few utility functions:

  ```
  max(a, b)
  min(a, b)
  new(t) // allocates a new data item of the given type, returns a pointer
  make(t, ???...) // allocates a slice, map, or channel
  ```
- The package also contains legacy `print` and `println` functions that should usually be avoided, except perhaps for debugging in situations where it's desirable to avoid importing the fmt package or similar.
- Some additional utilities that are built-ins, and more details, are described later.

Variable Declarations

Regular: `var x int = 99`
Multiple: `var x, y int = 99, 100`
Short: `x := 99` Type is inferred from constant type
`x, y := getXVal(), getYVal()`

- Code with unused variables will not compile
- Variables in nested scopes can shadow the same name in an outer scope (likely poor style!)
- Variables declared using short form in an inner scope will create a new variable, shadowing the existing one. Take care to avoid this error!

```
var a int = 99
{
     a := 100 // new variable shadowing variable a above
     fmt.Printf("a is %d\n", a) // prints 100
}
fmt.Printf("a is %d\n", a) // prints 99
```

Basic string concepts and operations

- Go's primary text type is called string, all lower case
  - Under the hood it's a sequence of bytes that usually represent UTF-8
  - So, for example, a string with 3 "characters" might have 9 bytes
- Literal strings are enclosed in double quotes
  - These are composed of the UTF-8 characters from the source file
- Values of string type are immutable, but in general variables of string type can be reassigned to contain a different string object
- Copying a string is an automatic consequence of simple assignment
  - Unless the variable being assigned is specifically a pointer to a string
- Obtain the length using the built-in function `len` (which also works on several other structures):
  ```
  fmt.Println(len(str))
  ```
- Raw strings can be declared using backticks
- Variables of string type contain the structure, known as a "slice" that *describes* a string
  - That slice structure describes the start and end point of this string in an array of bytes
  - The target array is not intended to be accessed directly

Substring operations

- A substring can be taken from a string using the slice construction (more later)

```
str := "Hello"
ss := str[1:4] // first included index : first excluded index
fmt.Println(str, ss) // output: Hello ell
```

- Strings are "slices" of ***bytes***. Be careful with multi-byte characters:

```
msg := "日本語"
msgPart := msg[0:3]
fmt.Println("msg: ", msg)
fmt.Println("msgPart: ", msgPart)
fmt.Println("len(msgPart): ", len(msgPart))
fmt.Println("wrong offset gives:", msg[2:])
```

- Prints this output:
  msg: 日本語
  msgPart: 日
  len(msgPart): 3
  wrong offset gives: ?本語

The meaning of `nil`

- The special value `nil` represents the "default state"
- It can be used to initialize any of:
  - pointer
  - `chan`
  - `func`
  - `interface`
  - `map`
  - slice
- Note: `nil` is not used with struct types
- Simple types like `int` have a suitable zero value, but cannot be initialized from `nil`
- The result of assigning `nil` to a pointer type is a zero pointer value
- The result for all other types is an actual object, initialized to a zero-like state
  - These objects can be at different addresses (and not at the zero address)

Assignments ***copy*** values

- Assignment copies the value of one expression into storage described on the left side of the assignment
- Unless a variable is a pointer type, this copies the value(s) in the right side expression
  - including in the case where the right side is an aggregate data type, such as a struct or array
- Function arguments are handled the same way
- This also happens when iterating over values in a loop
  - Take care that modifying the value provided during iteration will not (except in cases where pointers are in use) modify the value in the data set being iterated

Multiple Assignments

- Go does not provide tuples, but can make multiple simultaneous assignments (as with initialized variable declarations)
- Functions can return multiple values (and this is commonly used to represent an error status and a return value separately).

```
func getVals() (int, int) {
   return 20, 40
}
func main() {
fmt.Println("Hello Go World!")
// call with e.g.:
t, u := getVals()
```

- If multiple values are on the right side of an assignment, the left side must provide the expected number of targets. However, it's possible to ignore one or more, which might be useful where a function returns several values, but not all values are needed:

```
var p int
p, _ = 10, 20
q, _ := 10, 20
```

Type Conversions

- With the exception of constants, which are handled by the compiler, Go types *do not* automatically convert types, instead conversion must be explicit:

```
x := 99 // int by default
var flX float64 = x // FAILS
var flX float64 = float64(x) // explicit conversion is OK
fmt.Printf("value %v, type %T\n", flX, flX) // value 99, type float64
```

- This applies even if a type is defined to be another:

```
type month int8
var jan month = 1
fmt.Printf("jan is %v of type %T\n", jan, jan)
var janN int8 = jan // NO: "Cannot use 'jan' (type month) as the type int8"
```

- Note that conversion of integer types to **`string`** treat the integer as a *UTF-8 character*, and do NOT create a textual representation of the number:

```
var letterA = 65
fmt.Println(string(letterA)) // prints: A
```

Type assertions

- If a variable is of an interface type the actual struct is of some concrete type, and it's possible that the actual struct might satisfy other interfaces
- The "type assertion" mechanism can create a new expression of another type, either a concrete struct, or an interface
- The syntax looks like this:

```
var being any = Person{"Fred"}
p1, ok := being.(Person) // ok is false if being is not a Person
```

- The assigned variable (**`p1`** above) will have a copy of the value of being. If the assertion failed, it will have a *zero-like* struct of the specified type
- If the assignment above is made to only a single variable (as **`p1`** above) then the conversion causes a panic if the target is not of the specified type

```
var being any = Person{"Fred"}
p1 := being.(Person) // panic if being is not a Person
```

- Additional type testing and conversion is supported using the "type switch" construction

Formatted printing - 1

- Package **`fmt`** provides for printing and formatting:

```
import "fmt"
type month int8

var jan month = 1
fmt.Printf("jan is %v of type %T\n", jan, jan)
```

- Results in: **`jan is 1 of type main.month`**

Formatted printing - 2

- `Printf` (and related functions) allow many conversions (verbs), some common ones are:

- `%v` - default conversion of a value
- `%#v` - present a value in "source code" form
- `%T` - the type of a value
- `%d` - decimal number
- `%f`, `%g`, `%e` - floating point in short, longer, and exponent presentations
- `%t` - boolean
- `%c` - rune (character)
- `%s` - string
- `%p` - a pointer value (in hexadecimal)
- `%%` - a percent sign

- Width and precision may be specified, e.g. `%8.3f`

Operators

Precedence	Operator
highest	`*  /  %  <<  >>  &  &^`
	`+ -	^`
	`==  !=  <  <=  >  >=`
	`&&`
lowest	`||`

- Operators of the same precedence associate left to right
- `%` is remainder, not 'mod'
- `&^` is "and not", which clears those bits which are set in the right hand operand
- `&&` and `||` are short-circuiting
- Note there is no conditional (a.k.a. "ternary") operator (`?  :` in C, C++, Java and more)

Operator considerations

- Assignment does ***not*** have value; the following (which would be valid in C, C++, Java and more) does ***not*** work:

```
var x, y int = 10, 0
y = (x += 10) // FAILS)
```

- Increment / decrement operations are statements, ***not*** expressions (they do not produce a value):

```
var x, y int = 10, 0
x++ // Success
y = x++ // FAILS
```

- Arithmetic does not check for overflow:

```
var x int8 = 120
fmt.Println(x)
x += 30 // overflows range of a byte (-128 to +127)
fmt.Println(x) // prints -106
```

`for` loops

- Go has only a `for` loop, but multiple forms are supported

  ```
  for <init> ; <test> ; <update> { … }
  ```

- `<init>` uses the short form declaration

  ```
  for x := 0; x < 3; x++ {
     fmt.Printf("x is %v\n", x)
  }
  ```

- Variables declared here are scoped to the loop and can shadow variables already in scope
- `<init>` phase might use multiple assignment:

  ```
  for a, b := getTwoValues(); a < b; a += 5 { …
  ```

`for` loop variations

- Infinite loop:

```
for {
    fmt.Print(".")
}
```

- While-type loop:

```
a := 3
for a > 0 {
    fmt.Printf("a is %v\n", a)
    a--
}
```

`for` loop variations

- **`for`** with **`range`** iterates over iterables, including characters in a string:

```
name := "Fred"
for pos, ch := range name {
   fmt.Printf("at position %d is %c\n", pos, ch)
}
```

- NOTES:
  - `ch` will be a ***copy*** of the value stored in `name`; changing it will not change the value in `name`
  - When iterating over a string, whole runes (possibly multi-byte characters) will be read
  - and, in the case of a string, indexes will be the byte index, so will be non-contiguous where multi-byte characters are concerned

Conditions with `if`

- A simple if/else structure takes a **`bool`** expression

```
value := rand.Float64()
if value > 0.5 {
    fmt.Printf("Big enough: %f\n", value)
} else {
    fmt.Println("Bah, too small!")
}
```

- The test must be a **`bool`** expression
- Curly brace blocks are required
- The **`else`** part is optional
- No "elif" is required or provided, since blocking is governed by curlies, not by indentation

Initialization statements with `if`

- An **`if`** statement may have an additional "initialization" statement prior to the test:

```
if fmt.Println("setting up") ; value > 0.5 { ...
```

- The statement typically initializes variables. Such variables have a scope associated with the **`if`** statement, even though they're declared before the curly braces:

```
if x, y := getVal(), getOtherVal(); x == y {
    fmt.Printf("x = %d, y = %d, x equals y!\n", x, y)
}
// x and y are out of scope here
```

- The scope of **`x`** and **`y`** in the above ends with the closing curly brace
- Only *one statement* is allowed in this form, and note the separating semicolon, however, multiple-variable assignment, as shown above, constitutes a single statement

Conditions with **`switch`**

- Go provides a **`switch`** statement similar to C, C++, Java and others.
- Where constants are used as the targets of case, they must be distinct
- The **`case`** values can be ***expressions*** rather than constants, in which case each is evaluated in turn, and the first match is executed, after which the **`switch`** completes
- Omitting the **`switch`** expression implies a switch target of **`true`** and requires **`bool`** expressions in the **`case`** parts
  - This combination creates a brief multi-way if/else-if type structure
- Search for a match stops at the first success
- No **`break`** statement is required to avoid fallthrough
  - Fallthrough can be requested explicitly using the keyword `fallthrough` except on the last case
  - **`break`** can be used if a case has multiple statements and it's desired to skip out of the rest of them
- An optional "initialization" statement can precede the expression

Code examples with **`switch`**

- A simple scenario:

```
num := 2
switch num {
case 1:
    fmt.Println("It's one!")
case 2:
    fmt.Println("two is the number")
    fallthrough
default:
    fmt.Println("It's not one...") // two will also print this
}
fmt.Println(num) // num is still in scope
```

Code examples with **`switch`**

- A **`switch`** with a ***local*** variable declaration & initialization:

```
func getSmallInt() int {
   return rand.Intn(3) // returns 0, 1, or 2
}
// …
num := -1000
switch num := getSmallInt(); num { // new num declared and initialized here
case 1:
   fmt.Println("It's one!")
case 2:
   fmt.Println("two is the number") // note, no "fallthrough" here now
default:
   fmt.Println("It's not one, nor two...")
}
fmt.Println(num) // prints -1000, new num has ceased to exist
```

Code examples with **`switch`**

- The expressions in **`case`** do not have to be constants, this calls **`getSmallInt`** up to three times and can match at any of the targets

```
func getSmallInt() int {
    rv := rand.Intn(3)
    fmt.Println("returning:", rv)
    return rv
}

switch num := getSmallInt(); num {
case getSmallInt():
    fmt.Println("switch got:", num)
case getSmallInt():
    fmt.Println("switch found:", num)
default:
    fmt.Println("switch at the third try!")
}
```

Type switches

- Go maintains ***runtime*** information about type
  - **`switch`** statements provide a means of identifying type, and of providing a reference to a more specific type
- A switch on type can be used only if the variable has a non-specific type
  - This implies it's an "interface" type

```
var q interface{} = "99.0"
switch v := q.(type) { // v := … captures the value into a typed variable
case int:      fmt.Println("It's an int ", v) // v has int type here
case float64: fmt.Println("float64 ", v)     // v has float64 type here
case string:  fmt.Println("string ", v)      // v has string type here
default:      fmt.Println("something else")
}
```

- Note that the use of **`interface{}`** indicates a type about which nothing is known, this can also be represented by the type **`any`**

Simple function declaration

- The core form of a named function declaration looks like this:
  `func` *<name>* **`(`**`[`*<formal parameter list>*`]`**`)`** `[`*<return type specification(s)>*`]` **`{`**
  `<function body>`
  `}`
- For example:
```
func simple(s1, s2 string, num int) int {
   return len(s1) + len(s2) + num
}
```
- Formal parameters are enumerated in parentheses, comma separated, with types specified after the formal parameter name
- Consecutive formal parameters of the same type can share a single type specification
- The parentheses are required even if there are zero formal parameters
- The return type is specified after the closing parenthesis surrounding the formal parameter list
  - If a function returns nothing, no return type is specified (Go does not use "void" in this context)
- The body of the function is enclosed in curly braces
- A **`return`** statement, usually with a compatible expression, is required for each path through the function, unless that path issues a panic

Parameter and return passing

- Parameters to a function are passed ***by value***, that is, ***the function gets a copy***
- Because of this, **if a function is to mutate caller data, that data must be passed as a pointer**
  - Note that if what is copied ***contains*** a pointer (e.g. a slice) that aspect will be mutable in the function
- Return values will be copied on use, whether by assignment to a variable, or passing into another function invocation.
- Passing data by copying allows allocated memory to be reclaimed by means of the stack frame collapse, avoiding the need for garbage collection and reachability analysis
- If values are passed out of functions using their addresses, Go uses "escape analysis" to determine that the data must be created on the heap, and it becomes subject to garbage collection
  - It's not safe to make assumptions about the relative performance of copying and pointer + garbage collection approaches
  - Focus on readability first, fix documentable performance problems after they're measured

Returning multiple values from a function

- A function can return multiple values, e.g.:

```
func multiple(s string, num int) (int, string) {
    rvs := fmt.Sprintf("The text was %s and the number %d", s, num)
    return len(rvs), rvs
}
```

- Declare the return types using parentheses to enclose the types
- Follow the **`return`** keyword with a comma separated list of values to be returned
  - ***Do not*** surround the list of values in parentheses
- Assign the results of such a function to multiple variables

```
a, b := multiple("Hello", 3)
```

- Use an underscore as the variable name if a value is not needed

```
_, b := multiple("Hello", 3)
```

- Return values can be named, which declares a local variable of that name

```
func multiple(s string, num int) (x int, s1 string) {
```

Returning multiple values from a function

- Return values can be named, which declares a local variable of that name

```
func multiple(s string, num int) (x int, s1 string) { …
```

- If named return values have been declared, the **`return`** statement can be shortened:

```
func multiple(s string, num int) (x int, s1 string) {
    return // returns x and s1 implicitly
}
```

- A multi-return function can be an argument to a function that accepts multiple suitable arguments

```
func intstring(x int, s string) {
    fmt.Printf("instring() received %d, %s\n", x, s)
}
intstring(multiple("Hello", 3))
```

Declaring variadic functions

- A function can declare a variable length argument list in the last position of the formal parameter list

```
func variSum(nums ...int) int {
    rv := 0
    for _, v := range nums {
        rv += v
    }
    return rv
}
```

- The variadic formal parameter arrives as a "slice", which is similar in concept to an array, and can be iterated directly by a for loop

Calling variadic functions

- A variadic formal parameter can be satisfied by any number (including zero) of arguments of the right type in the invocation. Separate the individual arguments with commas

```
fmt.Println("Sum 1..4 is ", variSum(1, 2, 3, 4))
```

- The variadic parameter can also be supplied using a slice followed by ellipsis

```
nums := []int{1, 2, 3, 4} // slice of int values
fmt.Println("Sum 1..4 is ", variSum(nums...))
```

- Note that this spreading syntax requires a slice; *an array is not acceptable*
  - However, a slice can be built from an array directly using the suffix `[:]`, which creates a slice of the entire array

Function values

- Functions are first class values in Go, they can be stored in variables, passed as arguments, and be returned from function invocation
- Variables that have function type behave much like a reference or pointer to the function, in particular, assigning such variables does not duplicate the code
- Variables of function type cannot be compared, except against the value **`nil`**
- Invoking a **`nil`** function raises a panic
- A function can be assigned to a variable by using the name without the parentheses:

```
func add(a, b int) int {
   return a + b
}
// ...
f1 := add
```

- The function referred to by a variable can be invoked simply by using the variable name with an actual parameter list in parentheses:

```
fmt.Printf("add 1, 2 gives %d\n", f1(1, 2)) // calls "add"
```

Anonymous functions

- A function declaration can omit the name, this creates an anonymous function
- An anonymous function is an expression of function type
- The expression value must be used, for example it could be assigned to a variable:

```
func main() {
    add := func (a, b int) int { // OK!
        return a + b
    }
}
```

- Functions defined inside a block ***must be anonymous***, this fails:

```
func main() {
    // Not allowed, named functions must be at package level!
    func add(a, b int) int {
        return a + b
    }
}
```

Closures

- Anonymous functions create "closures", which capture the variables that are in their lexical scope
- In go, closures are mutable, as demonstrated here:

```
func getAdder(a int) (func(int) int, func(int)) {
    add := func(b int) int {
        fmt.Println("adding:", a, "and", b)
        return a + b
    }
    change := func(c int) { a = c }
    return add, change // two functions, one reads a, the other mutates it
}
func main() {
    adder, changer := getAdder(3)
    fmt.Println("before, adder(3) is:", adder(3)) // 3 + 3 -> 6
    changer(4)
    fmt.Println("after, adder(3) is:", adder(3))  // 3 + 4 -> 7
}
```

Declaring a recursive closure

- To allow recursion of an anonymous function, that function must be stored in a variable to allow it to reference itself. This works:

```
var length func(string) int
length = func(s string) int {
   if s == "" { return 0 }
   return 1 + length(s[1:])
}
fmt.Println("length is:", length("Hello")) // prints length is: 5
```

- But this does not compile, the call to length(s[1:]) is not resolved

```
length := func(s string) int {
   if s == "" { return 0 }
      return 1 + length(s[1:]) // Can't find length
   }
}
```

Closure warning–older Go versions

- Take care that a value captured by closure is not altered by some other operation unexpectedly
- In particular, in older Go versions (e.g. 1.18) a for / range iteration creates a ***single variable*** for the index and value that is reused in each iteration of the loop:

```
names := []string{"Fred", "Jim", "Sheila"}
for i, n := range names {
    name := n
    fmt.Println("index", i, "name", n)
    fmt.Printf("n is at %p\n", &n) // always the same
    fmt.Printf("name is at %p\n", &name) // different each time
}
```

- Notice that the loop variable called **`n`** is overwritten each time around this loop
- The variable declared in the loop, called **`name`**, however is reallocated each time round

Closure warning

- In this example, a function is built and appended to a slice, then all are executed afterwards:

- This code prints "`Name is: Sheila`" three times as shown
- If the line **`n := n`** is uncommented, the three names are printed
- This is because a ***new, unique, variable*** called **`n`** is created in local scope (inside the loop) and initialized to the current value of the loop variable **`n`** each time round
- It would be better to have a variable with a different name, but this code is valid as shown

Reporting catastrophic failure with `panic`

- The normal way to indicate that a function has failed is by returning an error code (often alongside a `nil` or similar "nothing to see" value in place of the regular value in successful completion
- Sometimes the nature of failure indicates that a catastrophic, unrecoverable, situation has arisen
  - This is the situation if the code finds itself in a state that by design should never happen–such a situation indicates a bug either in the design or implementation, and attempting recovery is generally pointless, since it's impossible to know how much damage has been caused already.
- For fatal situations, Go uses the notion of a panic, which is very similar conceptually to exceptions in other languages
- Calling `panic(xxx)` typically causes the function to exit immediately
  - The value `xxx` describes the problem and should usually be an error type: `errors.New("it broke")`
  - Any deferred functions are executed on the way out
  - No return value is provided to the caller
  - The panic "appears" in the caller, and will propagate up the call hierarchy unless intercepted, repeating this series of behaviors at each step up the call stack
- A panic might be issued from a library or utility function call, which will cause the code to behave the same way as an explicit call to `panic()`

Deferring function invocation

- When paired operations are used, such as open/close, or lock/unlock, safe programming generally requires that the code guarantee to perform the second operation if the first has executed
  - Commonly, this should be enforced even in cases of catastrophe represented by a panic
- Go addresses this with the ability to specify a function that must be executed as an enclosing function exits, no matter how that exit occurs
- This requirement is indicated using the **`defer`** construct
- The keyword **`defer`** is followed by a function *invocation* (not simply a function reference)
- The arguments to that function invocation are evaluated at the time the **`defer`** statement executes, but the function itself is ***not*** executed until the enclosing function returns
- If multiple **`defer`** statements are used in the same function all the deferred functions will be executed, in reverse order of their appearance, when the enclosing function returns

Execution of deferred function calls

- This example illustrates deferred function calls:

```
func show(s string) {
    fmt.Println("Running show function, s is:", s)
}
func main() {
    fmt.Println("Starting")
    defer show(getName())
    defer show("Also showing Albert!")
    fmt.Println("defer completed")
    switch r := rand.Float32(); { // note semicolon, this is switch on true
        case r < 0.3: fmt.Println("Continuing")
        case r < 0.7: fmt.Println("returning"); return
        default: panic(errors.New("Death to big numbers!"))
    }
    fmt.Println("Out of the switch, exiting normally.")
}
```

Execution of deferred function calls

- The preceding example generates this output if the random number is between 0.3 and 0.7:

```
Starting
getting name!
defer completed
returning
Running show function, s is: Also showing Albert!
Running show function, s is: Freddy
```

- Notice that the `getName` function is called during execution of the first `defer`, but the function `show(string)` that it refers to, is not executed until the function exits
- Notice also that the second deferred call to `show` executes first, and the first executes last
- Notice that the `main` function exits directly from the middle of the `switch`, without executing the final `Println`

Scope of action of deferred function calls

- The point at which the deferred behavior executes is not controllable by means of blocks; it's always at the exit from the function
- A function that has triggered a panic cannot continue, the function will exit entirely, even after recovery
- If recovery is necessary, migrate the code region that might fail out to a new function and call that from within the function that must continue
- If **`defer`** calls are made in a loop, none will be executed until the function *as a whole* ends
    - If the loop processes IO operations that should be closed, it's possible to run out of connections because of this
- Again, moving the body of the loop into a function of its own, called from within the loop, will address the problem

Using `defer` and `recover` to recover from `panic`

- Conditions that cause a panic should usually be treated as fatal, there's usually no way to determine the limits of damage
  - Sometimes, however, some kind of controlled shutdown might be preferable to an uncontrolled crash
  - Take great care that the controlled shutdown does not depend on state that could possibly be corrupted
- A panic can be intercepted using a call to the built-in function `recover()`, embedded in a deferred function
- If a call of the form `panic(x)` caused the invocation of the deferred function, then calling `recover()` will return the `x` object, and "cancel" the panic
- If `recover()` is called when no panic had occurred, `recover()` returns `nil`
- If multiple deferred functions call `recover()`, only the first to execute will see any panic that had arisen
  - The first deferred function to execute will be the last one registered

Example with **`defer`** and **`recover()`**

- This example defers twice to the same function, which calls `recover()` and prints a "recovering" message if a panic is being handled, or another message if not. Only the deferred call `maybe("Second")` will see the panic, which is cleared before the invocation of the deferred `maybe("First")`

```
func maybe(msg string) {
   fmt.Printf("in deferred maybe(%v)\n", msg)
   if p := recover(); p != nil {
      fmt.Printf("recovering from panic(%v)\n", p)
   } else { fmt.Println("no panic reported") }
   fmt.Printf("deferred maybe(%v) completing\n", msg)
}
func main() {
   defer maybe("First")
   defer maybe("Second")
   if rand.Float32() > 0.5 { panic(errors.New("Scream and shout!")) }
   fmt.Println("Exiting normally")
}
```

Array declaration and initializations

- Arrays of any type are contiguous memory blocks
- Size is fixed at creation, and is part of the type's identity
  - Arrays of different sizes cannot be assigned nor compared
- Declaration and initialization:

```
var ia [3]int // declares an uninitialized variable
var ia = [3]int{1,2,3} // initialized [3]int
var ia = [...]int{1,2,3} // also initialized [3]int
```

- **Caution**, leaving out the size creates a "slice", not an array

```
var si = []int{1,2,3} // this is not an array!
```

- An array can have only the first few elements initialized if preferred

```
var ia = [6]int{1,2,3} // has 3 trailing zero values
```

More array initializations

- Array literals must have either a (redundant) comma, or a closing curly brace, on the last line

```
valid1 := [10]int{0, 1}
valid2 := [10]int{0, 1,}
valid3 := [10]int{
    0, 1,
}
invalid := [10]int{
    0,
    1 // FAILS! Must have a comma or the closing curly on this line!
}
```

Even more array initializations

- Arrays can be initialized with explicit subscripts, which do not have to be in order

  ```
  var s = [4]string{3: "Hearts", 1: "Diamonds", 0: "Spades", 2: "Clubs"}
  ```

  which configures `s[0]` as `"Spades"`, and so on
- Indexes can be auto-incrementing after an explicit index

  ```
  var s = [4]string{3: "Hearts", 0: "Spades", "Diamonds", "Clubs"}
  ```

  which has `"Diamonds"` at index 1, and `"Clubs"` at index 2

Accessing array elements

- Go accesses array elements (and also elements of *slices*, not yet introduced) for read or write using numeric expressions as subscripts provided in square brackets

```
var ia [3]int // create array of three ints, initialized to zero
ia[0] = 99 // assign 99 to the zeroth element
fmt.Println("ia[0] is", ia[0], "and ia[1] is", ia[1]) // print elements
```

- The size of an array, in addition to the element type, is part of the type.
  - Assignment is only possible if the target array variable has the same size and base type.
  - Comparing arrays is supported, provided the size and base type is identical.
- This works:

```
ia := [...]int{0, 1, 2, 3, 4}
var ia2 [5]int
ia2 = ia
```

- But this fails:

```
ia := [...]int{0, 1, 2, 3, 4}
var ia2 [6]int// different sizes
ia2 = ia // Compilation fails!
```

Iterating over array elements

- Arrays are readily iterated using a `for` loop and `range` which give pairs of values, being the index and the value at that index

```
for idx, val := range ia {
    fmt.Println("at subscript:", idx, "is value:", val)
}
```

- It's important to realize that the `val` variable provides a ***copy*** of the array element, so cannot be used to mutate the array element. This example prints `[5]int{0, 1, 2, 3, 4}` twice:

```
ia := [...]int{0, 1, 2, 3, 4}
fmt.Printf("%#v\n", ia)
for _, val := range ia { // ignore the index here
    val += 10// val is a copy of an array element, not the element
}
fmt.Printf("%#v\n", ia) // array contents, and output, are unchanged
```

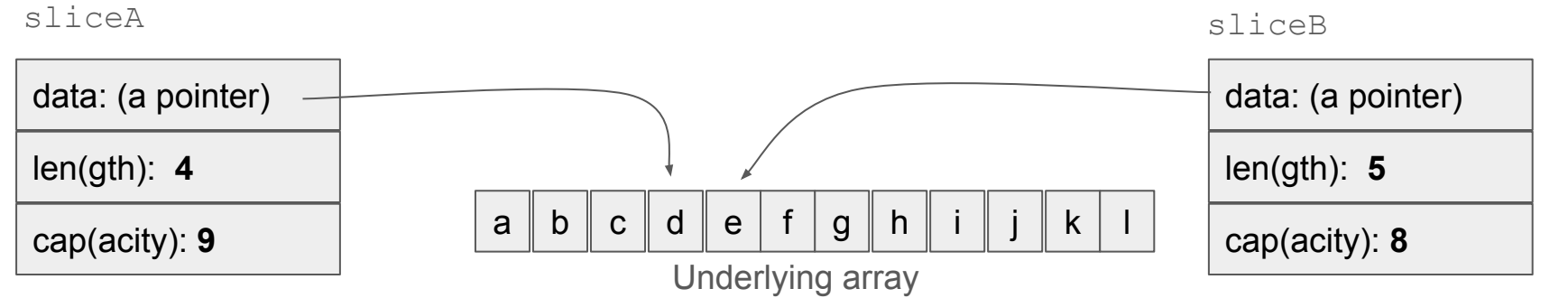
- To fix this problem, store the loop's index counter as **`idx`** and use it to mutate the original element

```
    ia[idx] = val + 10
```

Slices

- A slice is very similar in concept and element access syntax to an array, but number of elements represented by the slice is dynamic, and is not significant to the type
- It's more common to use slices than to use arrays directly
- A slice is actually a small structure that accesses an underlying array for data storage.
- A slice also describes:
  - a starting offset into that underlying storage
  - a count of consecutive elements of the storage that are "valid" from the perspective of the slice
  - The notion of "capacity"--how much underlying array space is available from the starting offset to the end of that underlying array
- A slice can also be looked on as if it were an array-like structure that can grow–but the starting offset and size of the underlying array places some constraint on this
- Another perspective is to look at a slice as a sub-array
- Multiple slices can refer to the same underlying array
  - Mutations made through any one such slice change the underlying array, and generally change other slices
- The backing array is not normally accessed (nor readily accessible) directly through the slice

Examining slices



This slice represents **d, e, f, g**
The capacity is the space from d
to the end of the underlying array

This slice represents **e, f, g, h, i**
The capacity is the space from e
to the end of the underlying array

Note executing:
`sliceA[2] = 'X'`
changes the `'f'` in the array and results in `sliceB` containing **e, X, g, h, i**

Creating / initializing slices

- A slice can be created using a syntax similar to that for arrays, but without a size specification:
  ```
  sl1 := []int{1, 2, 3}
  ```
  The above allocates an underlying array with three elements
- A slice can be created with an explicit length (0 here) and capacity (10 here) using the **`make`** function:
  ```
  sl2 := make([]int, 0, 10)
  ```
  The above allocates an underlying array with ten elements
  Note that "uninitialized" elements are always initialized to a nil value (zero for numeric types)
- A slice can be created as a "sub-array" backed with an explicitly provided array
  ```
  numbers := [...]int{10, 11, 12, 13, 14, 15, 16, 17} // array
  someNumbers := numbers[2:6] // 2 is first index, 6 is first-excluded index
  fmt.Printf("%v\n", someNumbers) // output: [12 13 14 15]
  ```
  - The capacity of **`someNumbers`** above is 6, being the remaining space in the underlying array from offset 2

Creating / initializing / comparing slices

- A slice can be taken from another slice:

```
numbers := [...]int{10, 11, 12, 13, 14, 15, 16, 17}
someNumbers := numbers[2:6] // take a slice
fewNumbers := someNumbers[1:] // take a slice from the slice
fmt.Printf("%v, cap is %d\n", fewNumbers, cap(fewNumbers))
```

  - In this case, the output is **`[13 14 15], cap is 5`**
  - Note the data ends where the *first slice* data ended, *not* at the end of the underlying array
  - The capacity is determined from the underlying array

- It's possible to take a slice from a slice that's longer than the original slice, provided the underlying array supports it. Changing the above like this:

```
fewNumbers := someNumbers[1:6]
```

  results in the output [13 14 15 16 17], cap is 5, i.e. it includes data beyond the "end" of the original slice, but that was present in the underlying array
- Attempting to included data that does not exist in the underlying array will panic
- Unlike arrays, ***slices cannot be compared with ==*** (except against nil)
  - The library provides **bytes.Equal**, but only for byte slices

Extending, and appending to, slices

- The built-in function **`append`** writes elements to slice storage beyond the current length of this slice

```
someMore := append(some, item1, item2)
```

- Note that append returns a slice structure, it doesn't modify the original structure.
  - This new slice value must be assigned to be useful (and to silence the compiler's complaints)
- If the existing backing array is long enough to hold the data, it is ***updated***:

```
numbers := [...]int{10, 11, 12, 13, 14, 15, 16, 17}
someNumbers := numbers[2:5] // 12, 13, 14
someNumbers = append(someNumbers, 99, 98)
fmt.Printf("numbers is %v\n", numbers)
fmt.Printf("someNumbers is %v\n", someNumbers)
```

- Output:

```
numbers is [10 11 12 13 14 99 98 17]
someNumbers is [12 13 14 99 98]
```

- **Note** that if the underlying storage is shared with other slices, or the array is accessible directly, this might be undesirable

Extending, and appending to, slices

- To avoid unexpected updates to an accessible underlying array, or to other slices sharing that array, make a new slice, and copy the contents of the existing slice explicitly using `append`, then `append` the new elements

```
numbers := [...]int{10, 11, 12, 13, 14, 15, 16, 17}
someNumbers := numbers[2:5]          // 12, 13, 14
someMoreSpace := make([]int, 0, 20) // empty, but space for 20 elements
someNumbers = append(someMoreSpace, someNumbers...)
someNumbers = append(someNumbers, 99, 98)
fmt.Printf("numbers is %v\n", numbers)
fmt.Printf("someNumbers is %v\n", someNumbers)
```

- This version produces the following output–***note that the original array is unchanged***:

```
numbers is [10 11 12 13 14 15 16 17]
someNumbers is [12 13 14 99 98]
```

Sorting slices

- The sort package provides several means for sorting
- A slice of strings can be sorted directly:

```
nameList := []string{"Fred", "Jim", "Alice", "Jules", "Bob", "Michael"}
sort.Strings(nameList) // sort strings in ascending order
```

- Or a slice of an arbitrary type can be sorted by providing a "less" function that compares two elements at the specified indexes:

```
sort.Slice(nameList, func(i, j int) bool { return nameList[j] < nameList[i] })
```

- Other features are provided by the package too
- Note that the elements of the slice are sorted in place, that is, the original slice is mutated

Maps

- Go has a built in map type. Maps have a literal form (note the trailing comma is needed)

```
releaseYear := map[string]int{
    "Fearless":       2008,
    "Try This":       2003,
    "?":              1984,
}
```

- Or create an empty map explicitly with **`make`**:

```
releaseYear := make(map[string]int)
```

- Access an element by its key:

```
year, found := releaseYear["Fearless"]
```

  - Reading a non-existent key returns the zero value for the map's value type, and false for the found value
  - The found value can be omitted entirely if preferred (unlike normal two-return functions, which would require assignment to _ to ignore the value)

- Insert /replace an element:

```
releaseYear["Fearless"] = 2008
```

- Remove an entry:

```
delete(releaseYear, "Fearless")
```

Maps

- Map keys must be `comparable` (an interface)
- Map values are mutable, based on their type, and mutating a non-existent entry creates it with the "zero value" for the type:

```
delete(releaseYear, "Delayed")
// create new entry with value and immediately increment it to 1
releaseYear["Delayed"]++
```

- Map elements are not addressable; taking address is prohibited
  - They can move during a rehash operation
- Maps can be iterated using **`range`**, which yields the key and associated value.
- During iteration the order is neither guaranteed, nor stable:

```
for k, v := range releaseYear {
    fmt.Printf("In %d the album %s was released\n", v, k)
}
```

Using map keys as a set

- Go does not have a 'set' type, but the keys of a map are unique which is sometimes used to provide a set-like functionality.
- A value is typically stored against a bool "true" to readily identify it as present

Struct types

- A **`struct`** is itself an anonymous type describing an aggregate of data
- Structs are almost always used in declaring a new named type by using the **`struct`** as the target of a type declaration

```
type Person struct {
    name string
    address string
    credit int
}
```

- The sequence (including order) of field name-type pairs determines whether a structure is potentially assignable or convertible to another structure

```
type Person struct {
    name    string
    address string
    credit  int
}
type P struct {
    name    string
    address string
    credit  int
}
```

```
alan := Person{"Alan", "Here", 1000}
fmt.Printf("alan: %v, type: %T\n", alan, alan)
//var alan2 P = alan // this FAILS
var alan2 P = P(alan) // this succeds
fmt.Printf("alan2: %v, type: %T\n", alan2, alan2)
```

Prints:

```
alan: {Alan Here 1000}, type: main.Person
alan2: {Alan Here 1000}, type: main.P
```

Allocating structs

- There are a number of ways to allocate **`struct`** types in memory:
- By declaring an uninitialized variable, which creates a structure with all the fields set to the type-specific nil value

```
var person Person
```

- With an explicit assignment from a short-form struct literal. In this form, all fields must be provided, and are matched with the target based on their position. It's the responsibility of the programmer to ensure that values of like type are placed in the semantically correct position since the compiler cannot check.

```
alan := Person{"Alan", "Here", 1000}
```

- With an explicit assignment from a long-form struct literal. In this form, the order of the fields does not matter, since the fields are named, and it's permitted to omit some fields. Omitted fields will be initialized to the **`nil`** value for the target type. *All* specified fields must have a name element in this form.

```
alan := Person{address: "Here", name: "Alan"} // credit = 0
```

Allocating structs

- As assignment into a pointer-type variable, using a literal preceded with **`&`** (note you cannot precede literals of general types with **`&`**, e.g. **`&99`**, or **`&"Str"`**, are prohibited)

```
px := &Person{name: "Pointer", address: "On Heap"} // px has type *Person
```

- **`struct`** types are assigned by *copying*, which can involve creating new storage

```
var alan3 = alan2 // alan3 is new storage with copies of alan2's fields
```

- Structs can be allocated by **`new`**, which returns a pointer to the newly allocated object

```
pp := new(Person)
pp.Name = "Alfie"
pp.Address = "Round the corner"
fmt.Printf("pp is %v, %p, %T\n", pp, pp, pp)
```

- Note that the dot is used to dereference a pointer, just as it's used to select an item in a struct

Declaring methods

- Go allows an infix notation for function invocation, if the function is declared for it
  - This is called a "method" and makes code look like "regular object-oriented" code in this form:
    `likesBanana := somePerson.likes("Banana")`
- The method is can be declared so that the receiver (**`somePerson`** in the above) has either the **`struct`** type or a pointer to that type (these are distinct declarations, however)
- This example is invoked on a **`Person`**, not a pointer to a **`Person`**:

```
type Person struct {
    Name, Address string
}
func (p Person) String() string {
    return "Name: " + p.Name + " address: " + p.Address
}
```

- A method named **`String()`** provided on a **`struct`** type (not a pointer) is used automatically to convert to text, e.g. when **`fmt.Printf("%v", somePerson)`** is used

More about methods

- Argument passing in Go is ***by value***. If a **`struct`** is specified as the receiver type, the called code will see a ***copy*** of that **`struct`**
  - In this case, ***mutation*** to the original struct's values is ***not possible***
- To provide for mutation of a caller's object, the method (or simple function) ***must take a pointer parameter***, not a simple type
- If a method is declared with a receiver parameter that's a pointer type, but the invocation is made on an instance, the compiler takes the address of a struct type automatically

```
type Thing struct{ value int }
func (t *Thing) show2() { fmt.Printf("t is %#v at %p\n", *t, t) }
t := Thing{123}
(&t).show2() // manual pointer extraction (parens are required)
t.show2()    // OR: address taken automatically!
```

- The receiver parameter takes a programmer chosen name (simply **`p`** in this example)
  - No special name, such as this or self is used
- A method must be declared in the same package as the receiver type
- Methods and fields share the same namespace, and must not conflict

Embedding structs

- A struct declaration can include instances of other structs in two ways:
  - By declaring a named member of that type
  - By "embedding". In this case, the member has no name
- The basic syntax looks like this:

```
type Color struct {
    Hue, Saturation, Lightness  int
}
type Car struct {
    Color // embedded Color as part of Car
    Make string
}
```

- Given an embedded struct, and provided no name collision occurs, members of the embedded type might be accessible directly from the enclosing type variable

```
fmt.Println("Hue of car is", c.Hue)
```

Embedding structs

- If more than one occurrence of any given type are required, any after the first must be explicitly named:

```
type Car struct {
    Color
    CarpetColor Color
    Make        string
}
```

- Also, if there are naming collisions, the affected element(s) of the embedded struct must be called out explicitly using the type of the embedded struct as a field name:

```
type Car struct {
    Color
    carpetColor Color
    Make        string
    Hue         int
}
```

- requires:

```
fmt.Println("Hue of car is", c.Hue)
fmt.Println("Hue of car is", c.Color.Hue)
```

Embedding structs

- Embedded structs creates the impression of "inheritance" as is familiar from other object oriented languages. This effect is strengthened further because methods associated with the embedded type can operate directly on instances of the enclosing type:

```
func (c Color) isDark() bool {
    return c.Saturation > 50 && c.Lightness < 50
}
```

- Allows:

```
var c Car = Car{Color{33, 85, 15}, "Ford"}
fmt.Println("Car %v is ", "dark? %t\n", c.isDark())
```

Limitations of embedded structs

- The illusion of "is a" provided by the embedded structs feature has a couple of limitations
- Initialization must be performed "by hand":

```
var c Car = Car{Color{33, 85, 15}, "Ford"}
```

- Variables of the embedded type cannot be used to refer to instances that contain that type. Given the original example:

```
type Color struct { Hue, Saturation, Lightness  int }
type Car struct {
   Color // embedded Color as part of Car
   Make string
}
```

- This is ***not*** permitted:

```
var pink ~~Color~~ = Car{Color{0, 25, 95}, "Ford"} // FAILS Car isn't Color
```

- The use of interfaces can address this somewhat

Interfaces

- An interface, similar to other languages, describes the syntax of interactions without being attached to an implementation
  - This abstraction allows substitution of different implementations into a particular use case, such as a function, and satisfies the coloquial "is-a" relationship
- Go interfaces differ from most other interface systems in that a type does not declare that it conforms to a given interface, rather a type satisfies an interface simply by having the methods required by that interface
  - This means that a type can satisfy interfaces its designer didn't even know existed
  - This carries a slight risk that a type might satisfy an interface *syntactically*, while actually being *semantically incompatible*. A compiler cannot check semantics. In practice this is unlikely to be a real problem.

Interface example

- This example shows that the interface Photographer declares the ability to take a photograph of a subject, and is implicitly satisfied by two otherwise unrelated types:

```
type Photographer interface { TakePhoto(subject string) string }
type MomWithCamera struct {
   Name string
   Fee  int
}
type SpySatellite struct {
   Altitude   int
   Stationery bool
}
func (m MomWithCamera) TakePhoto(subject string) string {
   fmt.Printf("Smile! Click! My name is %s, please tell your friends!\n", m.Name)
   return fmt.Sprintf("Lovely photo of %s with great smiles!", subject)
}
func (s SpySatellite) TakePhoto(subject string) string {
   fmt.Println("Bleep bleep, sending data to secret government agency")
   return fmt.Sprintf("Image of %s from altitude of %d", subject, s.Altitude)
}
```

Interface example

- The code above permits this to work:

```
photogs := []Photographer{
    MomWithCamera{"Alice", 1000},
    SpySatellite{10_000, false},
}
subjects := []string{"my kids", "enemy spy"}
for _, p := range photogs {
   for _, s := range subjects {
      fmt.Println("IMAGE:", p.TakePhoto(s))
   }
}
```

Interfaces under the hood

- A variable of interface type has two parts:
  - The dynamic type description
  - A ***copy*** of the original value
- Because of this mechanism, interfaces have runtime costs in terms of performance and memory
- As with mutator methods/functions, which require a pointer to a target object that is to be mutated, an interface can only mutate a struct's values if the interface maps onto a pointer to that struct

- The type **`any`** is defined as **`interface{}`**, which can store anything, since no methods are required
- The interface comparable describes all the types that can be used with == and !=

Channel programming model

- The programming model with channels is relatively simple:
- A channel is created and passed to each of two goroutines
  - The channel can be shared with more than two, but the basic model has one goroutine pushing data to the other
- One goroutine, the "producer" incrementally creates a data structure
  - During preparation, the goroutine owns this data entirely, and can therefore mutate it safely
- Once the structure's values are stable, the data is written to a channel
- At most one goroutine receives a copy of the data from the channel
- Because the data is copied across the channel, shared mutable state is largely avoided
  - ***Note that this is NOT the case if the data copied includes a pointer!***
- The channel addresses synchronizing (in time) the two goroutines
  - data cannot be read until after it is written, making the reader wait
  - if there is no room in the channel's buffer, the writing goroutine is made to wait until space is available
- By default, a channel has no buffer space, but it can be created with with a number of slots, allowing a degree of temporal decoupling between sender and receiver

Example with goroutines and channels

```
func producer(out chan int, stop chan any)
{
   // create this many numbers
   count := rand.Intn(50)
   for x := count; x > 0; x-- {
      data := rand.Intn(65535)
      out <- data // send the data
      time.Sleep(time.Duration(
                  rand.Intn(500)+500) *
                    time.Millisecond)
   }
   // negative value flags end of data
   out <- -1
   // flag producer is done
   stop <- nil
   fmt.Println("producer stopping")
}
```

```
func consumer(in chan int, stop chan any) {
   done := false
   for !done {
      data := <-in
      // look for negative indicating end
      if data >= 0 {
         fmt.Println("Received:", data)
      }
      if data < 0 {
         done = true
      }
   }
   // flag consumer done
   stop <- nil
   fmt.Println("Consumer stopping")
}
```

Example with goroutines and channels

```
func main() {
    // channel for data transfer
    comms := make(chan int)
    // channel to indicate when all finished
    control := make(chan any)
    // kick off the processes
    go producer(comms, control)
    go consumer(comms, control)
    // wait for two (empty) messages to indicate completion
    for x := 2; x > 0; x-- {
        // blocks until a message arrives, abandons the data
        <- control
    }
    fmt.Println("All done, main stopping")
}
```