Go

Take-Aways

- mechanical sympathy
  - your language must work well with your hardware
- readable code is your #1 priority
  - don’t write clever code
  - write code as simple as possible
- type
  - understanding type is the foundation of go programming

Type

- type provides two pieces of info:
  - **size**
    - the size, or amount of memory, we’re looking to read/write
  - **representation**
    - the representation of that memory: int, string, float, etc

Type

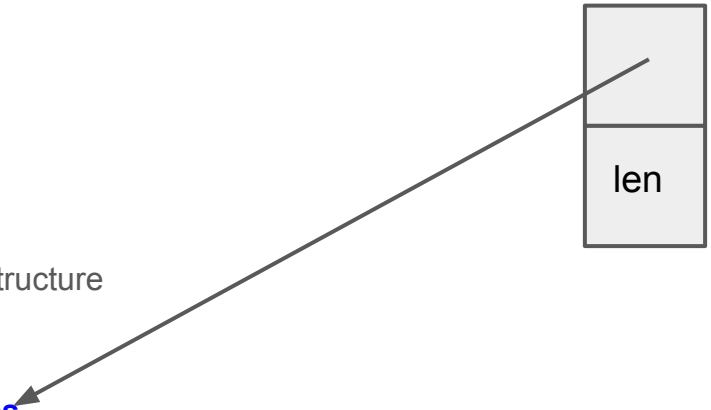
- float64
  - gives you two pieces of info:
    - **size**
    - **representation**

Type

- int
  - architect specific
  - “word” size
    - word is 8 bytes on 64 bit architecture
    - word is 4 bytes on 32 bit architecture
  - use “int” unless you have hardware reasons to use “int8” or “int16”

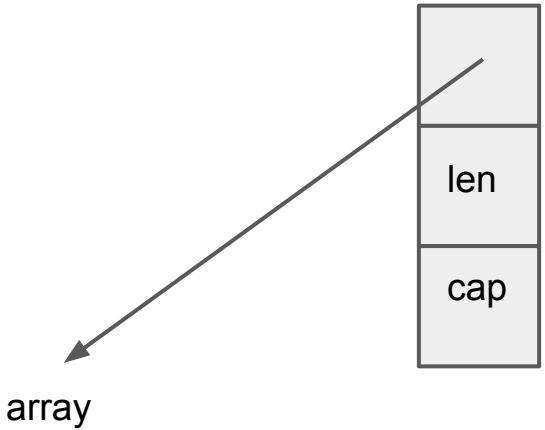
Type

- string
  - built in type
  - also a reference type
  - string is a two word data structure
    - either 8 or 16 bytes
    - first **word**
      - **slice of bytes**
    - second **word**
      - **len of those bytes**
  - all reference types have a pointer
    - when we copy a string we are making a copy of the two words:
      - **word 1: the pointer to the underlying array of bytes**
      - **word 2: the length**



Slice

- three word data structure
  - pointer to underlying array
  - length
  - capacity
- don’t
  - have slice of pointers
- do
  - have a slice of values
    - have slices of core data
  - core data needs to be in one place
    - as contiguous as possible
- three index slice
  - slice2 := slice1[2:4:4]



stack trace

Variable Initialization

- short variable declaration operator
  - :=
  - declare and initialize at the same time
- if you are setting to **zero value**
  - use
    - var aa int
  - don’t use
    - aa := 0

Constants

- constants of a:
  - kind
    - don’t need a type
      - constants without a type are called “kind”

        // Untyped Constants.

        const ui = 12345 // kind: integer

        const uf = 3.141592 // kind: floating-point
    - allows implicit conversion
  - type
    - a constant of a type

      const ti int = 12345 // type: int64

      const tf float64 = 3.141592 // type: float64
    - can’t be implicitly converted
      - only explicit conversion
    - when we’re working with variables of a type
      - we have to have like types
      - this is static typing

Mechanical Sympathy

- keep things in memory as contiguous as possible
  - linked lists are bad
    - not necessarily contiguous
  - arrays are our friends
    - they give us contiguous blocks of memory
    - however they’re fixed in size
      - *hello slices!*
        - slices are built on top of arrays
        - slices grow in size
    - hardware can feed caches easily with arrays
  - stacks
    - today in go stacks are contiguous

Sharing A Value vs Reference

- sharing copies is better than sharing references
  - keeps it on the stack, not the heap
- using pointers is not more performant than sharing data
  - in most cases
    - if sharing large amounts of data, might not be true
  - mostly, however, share copies and not pointers (references)
    - sharing copies of values, as opposed to sharing pointers, allows keeping data on the stack
      - keeps it out of the heap
      - the heap is for what is shared
      - keep the data as contiguous as possible at all times

OOP

- **encapsulation**
  - fields
  - methods
- **polymorphism**
  - polymorphism
    - a function that can take types implementing the interface
      - many different types, all of which implement the interface, can be passed to the polymorphic function
  - interface types
    - only declare behavior
    - there is not state; no fields
    - it is a type
      - so we can create values of a type interface
  - name with “er” if they have one method
    - interfaces do things
    - for example
      - interface notifier
        - runs the notify method
- **extend / override an existing type** - inheritance