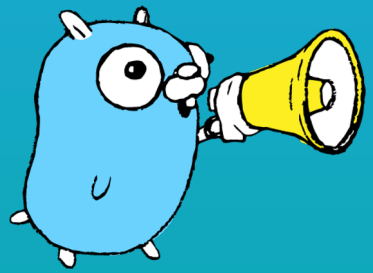
Fundamentals of Programming – Term 1/2020

Basic Data Types

Asst.Prof.Orathai Sangpetch, Dr.Akkarit Sangpetch

KMITL / CMKL University

Computer Innovation Engineering

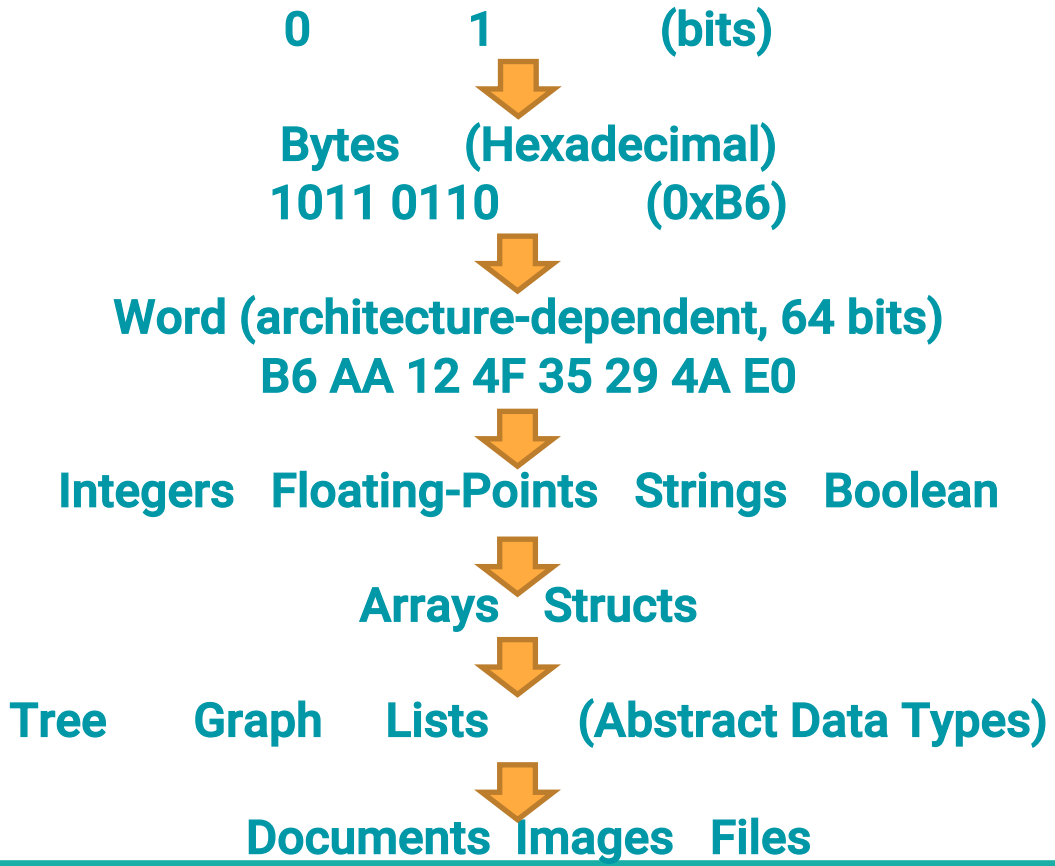


Today’s (glorious) blather.

Go Data Types

- **Data Types** are used to organize data – from matching hardware features to convenient abstractions / data structures
- **4 Categories of types**
  - **Basic Types**: numbers, strings, booleans
  - **Aggregate Types**: arrays, structs
  - **Reference Types**: pointers, slices, maps, functions, channels
  - **Interface Types**: abstract types exposing methods / behaviors

Data for Computer



Numeric types for signed / unsigned integer arithmetic

```
int8        int16   rune/int32  int64
uint8/byte  uint16  uint32      uint64

int       platform-dependent integer
uint      platform-dependent unsigned integer
uintptr   unsigned int that can hold a pointer value
```

Unsigned Integer	Signed Integer	Bits
uint8	int8	8
uint16	int16	16
uint32	int32	32
uint64	int64	64

Multiple types representing integers

Precedence

- 5 levels of precedence

```
* / % << >> & &^
+ |
^
== != < <= > >=
&&
||
```

- Integer arithmetic operators

```
+ , -, *, /
Applied to integer, floating point, complex numbers
The remainder operator % applies only to integers
```

Comparison and Binary Operators

- Type of a comparison expression is a boolean
  - == equal to
  - != not equal to
  - < less than
  - <= less than or equal to
  - > greater than
  - >= greater than or equal to
- Bitwise binary operators
  - & bitwise AND
  - | bitwise OR
  - ^ bitwise XOR
  - &^ bit clear (AND NOT)
  - << left shift
    - x<<n is equivalent to multiplication by 2^n
  - >> right shift
    - x>>n is equivalent to the floor of division of 2^n

Quick Exercise

```
var x uint8 = 1<<1 | 1<<5
var y uint8 = 1<<1 | 1<<2

fmt.Printf("%08b\n", x) // "00100010", the set {1, 5}
fmt.Printf("%08b\n", y) // "00000110", the set {1, 2}
fmt.Printf("%08b\n", x&y) // "00000010", the intersection {1}
fmt.Printf("%08b\n", x|y) // "00100110", the union {1, 2, 5}
fmt.Printf("%08b\n", x^y) // "00100100", the symmetric difference {2, 5}
fmt.Printf("%08b\n", x&^y) // "00100000", the difference {5}
```

Signed or unsigned int

```
medals := []string{"gold", "silver", "bronze"}
for i := len(medals)-1 ;i >= 0; i-- {
    fmt.Println(medals[i]) // "bronze", "silver", "gold"
}
```

- The built-in len function returns a signed int
- What'll happen if len returned an unsigned number?

Type Conversion

- Explicit conversion is required to convert a value of one type to another
  Change in value/lose precision when converting a big integer to a smaller one or from integer to floating-point

```
f := 3.141 // a float64
i := int(f)
fmt.Println(f, i) // "3.141 3"
f = 1.99
fmt.Println(int(f)) // "1"
```

- Binary operators for arithmetic and logic (except shifts) must have operands of the same type

```
var apples int32 = 1
var oranges int16 = 2
var compote int = apples + oranges // compile error

One possible fix
var compote = int(apples) + int(oranges)
```

Floating-Point Numbers

- 2 sizes
  - float32
    - About 6 decimal digits of precision
  - float64
    - About 15 decimal digits of precision
    - Preferred for most purposes

```
var f float32 = 16777216 // 1 << 24
fmt.Println(f == f+1) // "true"!
```

- Limits of floating-point values can be found in the `math` package
- NaN (not a number) e.g. 0/0 or sqrt(-1)

  Comparison with NaN always yields false

```
var z float64
fmt.Println(z, z,1/z, 1/z, z/z) // "0 0+Inf Inf  NaN"
```

Complex Numbers

- Complex number is created from its real and imaginary components e.g. 1 + 2i
  - complex64
    - Component is float32
  - complex128
    - Component is float64

```
var x complex 128 = complex(1, 2)
x := 1 + 2i
```

Booleans

- Type `bool` or Boolean with 2 possible values
  - true
  - false
- Logical negation !
  - `!true == false`
- Combine Boolean values
  - `&&` or `||`
- “short circuit” behavior
  - If the answer is already determined by the value of the left operand, the right operand is not evaluated

    ```
    s != "" && s[0] == ‘x’ //safe to write
    ```
- No implicit conversion from a Boolean value to a numeric value like 0 or 1

Strings

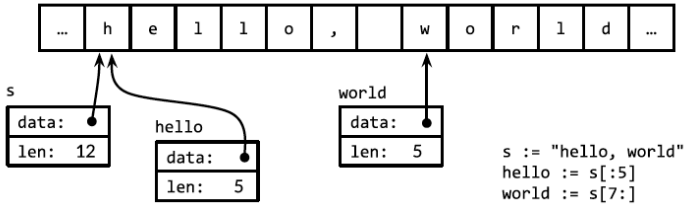
Immutable Sequence of bytes

```
s := “hello, world”
fmt.Println(s)       // “12”
fmt.Println(s[0], s[7]) // ‘h’ and ‘w’

c := s[len(s)]       // panic: index out of range

fmt.Println(s[0:5]) // “hello”
fmt.Println(s[:5])  // “hello”
fmt.Println(s[7:])  // “world”
fmt.Println(s[:])   // “hello, world”

s := “left”          // create a new string s
t := s               // “left” - no memory allocation
s += “,right”        // s is a new string “left,right”
s[0] = ‘L’           // compile error: cannot assign s[0]
```



len(s) return number of bytes in a string

index operation s[i] returns the i^{th} byte of string 0 <= i < len(s)

substring s[i:j] yields a new string from byte i to byte j-1

Operator + creates a new string by concatenating two strings

Immutable string means a string value can never be changed

String Literals

A string value are written as a string literal, enclosed in double quotes

```
“hello, \n”

\b          backspace
\n          new line
\r          carriage return
\t          tab

\xhh        hex escape e.g. “\x41” == “A”
\ooo        octal escape e.g. “\101” == “A”


Multi-line raw string literal
`hello this is
a test of time`
```

Go source files are encoded in UTF-8

Arbitrary byte values can be inserted using *escape sequences*, begin with \

A *raw string literal* is written using backquotes (no escape sequence – useful for documentation)

Unicode & UTF-8

Unicode collects all of the characters in the world's writing systems

```
UTF-8 encoding
0xxxxxxx                            runes 0-127 (ASCII)
110xxxxx 10xxxxxx                   128-2047 (<128 unused)
1110xxxx 10xxxxxx 10xxxxxx          2048-65535 (<2048 unused)
11110xxx 10xxxxxx 10xxxxxx 10xxxxxx 65536-0x10ffff

Unicode escape
\u        16-bit value \u4e16
\U        32-bit value \U00004e16

import “unicode/utf8”
s := “Hello, 世界”
fmt.Println(len(s))
fmt.Println(utf8.RuneCountInstring(s))
```

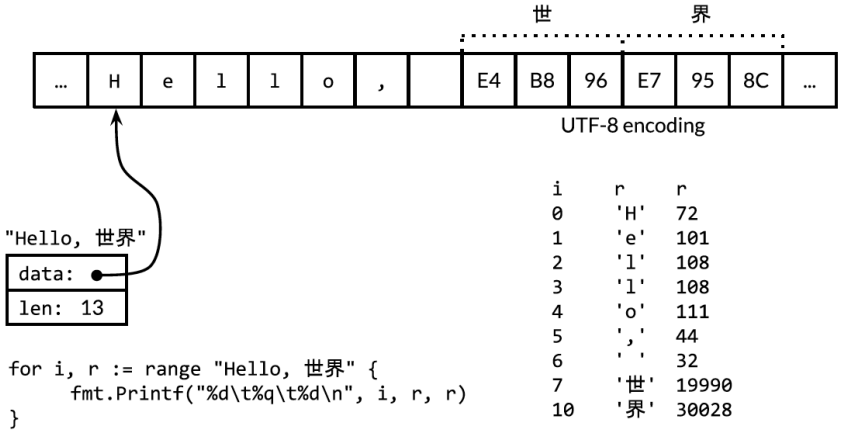
US-ASCII uses 7 bits to represent 128 characters.

Each character is assigned one standard number called a *Unicode code point* or a *rune*, represented by an int32 (UTF-32).

UTF-8 is a variable-length encoding of Unicode code points as bytes + allow byte operations on Unicode strings without decoding

Range Decoding

- Go range loop automatically decodes a Unicode string



- rune[ ] conversion returns the sequence of Unicode code points that the string encodes

```
// "program" in Japanese katakana
s := "プログラム"
fmt.Printf("% x\n", s) // "e3 83 97 e3 83 ad e3 82 b0 e3 83 a9 e3 83 a0"
r := []rune(s)
fmt.Printf("%x\n", r)  // "[30d7 30ed 30b0 30e9 30e0]"
```

String and Byte Slices

bytes.Buffer type provides efficient manipulation of strings data

```
String <-> byte slice conversion
s := “abc”
b := []byte(s)
s2 := string(b)


bytes.Buffer
import “bytes”

var b bytes.Buffer       // an empty ready-to-use buffer
b.Write([]byte(“Hello ”))     // “Hello ”
fmt.Fprintf(&b, “world!”)     // “Hello world!”
b.WriteTo(os.Stdout)          // print b to stdout
```

A string contains an array of *immutable* bytes.

A byte slice contains *mutable* elements that can be modified.

A string can be converted to byte slices and back.

A bytes.Buffer is a variable-sized buffer of bytes that can be read or write.

String and Number Conversions

```
Int <-> String
i, err := strconv.Atoi(“-42”)
s := strconv.Itoa(-42)

String -> Other types
b, err := strconv.ParseBool("true")
f, err := strconv.ParseFloat("3.1415", 64)
i, err := strconv.ParseInt("-42", 10, 64)
u, err := strconv.ParseUint("42", 10, 64)

Other types -> String
s := fmt.Sprintf("x=%b", x) // "x=1111011"
```

strconv can be used for numeric conversion

fmt can be used to create a string from formatted values

Constants

Constants are expression whose value is known and evaluated during compile-time

```
const pi = 3.14159

const (
  e = 2.7182818
  myE // 2.7182818
  pi = 3.14159265
  myPi // 3.14159265
)

const IPv4Len = 4
// parseIPv4 parses an IPv4 address (d.d.d.d).
func parseIPv4(s string) IP {
  var p [IPv4Len]byte
  // ...
}
```

Constant values are evaluated during compile-time and can appear in types.

Constant Generator

`iota` begins at zero and increments by one for each item in a sequence

```
type Weekday int
const (
  Sunday Weekday = iota         // 0
  Monday                        // 1
  Tuesday                       // 2
  Wednesday                     // 3
  Thursday                      // 4
  Friday                        // 5
  Saturday                      // 6
)

type Flags uint
const (
  FlagUp Flags = 1 << iota      // 0001
  FlagBroadcast                 // 0010
  _                             // skip (0100)
  FlagPointToPoint              // 1000
)
```

iota can be used with expressions to define named constants, also known as *enumerations* or *enums* in other languages

unwanted values can be skipped using _

Untyped Constants

Constants may remain uncommitted to a type until utilized

```
const Pi = 3.14159265358979323846264338327950288419716939937510582097494459

var x float32 = math.Pi
var y float64 = math.Pi
var z complex128 = math.Pi

var f float64 = 212
fmt.Println((f 32) * 5 / 9) // "100"; (f 32) * 5 is a float64
fmt.Println(5 / 9 * (f 32)) // "0"; 5/9 is an untyped integer=0
fmt.Println(5.0 / 9.0 * (f 32)) // "100"; 5.0/9.0 is an untyped float

i := 0 // untyped integer; implicit int(0)
r := '\000' // untyped rune; implicit rune('\000')
f := 0.0 // untyped floating point; implicit float64(0.0)
c := 0i // untyped complex; implicit complex128(0i)

const (
  deadbeef = 0xdeadbeef // untyped int with value 3735928559
  a = uint32(deadbeef) // uint32 with value 3735928559
  b = float32(deadbeef) // float32 with value 3735928576 (rounded up)
  c = float64(deadbeef) // float64 with value 3735928559 (exact)
  d = int32(deadbeef) // compile error: constant overflows int32
  e = float64(1e309) // compile error: constant overflows float64
  f = uint(1) // compile error: constant underflows uint
)
```

Untyped constants retain their higher precision until later

6 flavors of untyped constants: untyped boolean, untyped integer, untyped rune, untyped floating-point, untyped complex and untyped string.