Data Types and Control Flows in Go

Ambush Journey Program

Ambush

1. Basic data types in Go
2. Arrays and Slices
3. Control Flows
4. Memory, Pointers and References
5. Custom Types and Structs

Basic data types in Go

Data Types

Go has the following basic **data types**:

- `string`
- `int, int8, int16, uint, …`
- `float32, float64`
- `bool`

Data Types

Go has the following basic data types:

- `string`
- `int, int8, int16, uint, …`
- `float32, float64`
- `bool`

And also the following **operators**: `==, !=, <, >, <=, >=, &&, ||, !`

Data Types

Besides the default data types, there are a few **type aliases**:

- `byte` is an alias for `uint8` and can be used for characters
- `rune` is an alias for `int32` and can be used for representing Unicode code points

Arrays and Slices

Arrays and Slices

For dealing with sequential data, Go provides two structures:

- Arrays (fixed length)
- Slices (dynamic length)

Arrays and Slices

For dealing with sequential data, Go provides two structures:

- Arrays (fixed length)
  - `var arr [10]int`
- Slices (dynamic length)

Arrays and Slices

For dealing with sequential data, Go provides two structures:

- Arrays (fixed length)
- Slices (dynamic length)
  - `var slc []int`

Arrays and Slices

- Every slice contains an array underneath
- Since slices reference an array, modifying them will also modify the underlying array
- Given an array `arr` of length L, a slice can be made from it using the following syntax:

```
arr[x:y]
```

- Where `x` and `y` represent an interval between 0 and L

Arrays and Slices

- Slices have **length** and **capacity**
  - Length represents the number of items in the *slice* itself
  - Capacity represents the number of items in the *underlying array*
  - They can be obtained using the `len(s)` and `cap(s)` functions respectively

Arrays and Slices

- Data can be appended to a slice using the built-in function `append`

```go
var foo []int
foo = append(foo, 1)
foo = append(foo, 2, 3, 4)
fmt.Println(foo)
```

Arrays and Slices

- Data can be appended to a slice using the built-in function `append`
- When `append` is used, it returns a new slice
    - If the underlying array of the original slice is too small to fit the new elements, it creates a **new array** and returns a slice that references it

Arrays and Slices

- If a new slice needs to be created with a specific length or capacity, the `make` function can be used

```
b := make([]int, 2, 5) // len(b)=2, cap(b)=5
```

Practice

1. Write a function "printSlice" that takes a slice as a parameter and prints it, along with its length and capacity. You can use the `Printf` method from the `fmt` package

Control Flows

Control Flow

Go has the following basic **control flow** structures:

- if-else
- for
- switch-case

Control Flow

Go has the following basic **control flow** structures:

```
if (op == "add") {
    return num1 + num2
}
```

```
for i := 0; i < 10; i++ {
    fmt.Println(i)
}
```

```
switch (op) {
case "add":
    return num1 + num2
case "multiply":
    return num1 * num2
case "subtract":
    return num1 - num2
default:
    return num1 + num2
}
```

Control Flow

What about **while**?

- It's a **for** loop!

Control Flow

What about **while**?

- It's a **for** loop!

```
func find_first_even(arr []int) int {
    i := 0
    found := false
    for i < len(arr) && !found {
        if (arr[i] % 2 == 0) {
            found = true
        } else {
            i++
        }
    }
```

Control Flow

The **for** loop can also be used as a **for range** loop

- (similar to `for .. in` from Javascript)

```go
for index, value := range collection {
    fmt.Printf("Index: %d, Value: %v\n", index, value)
}
```

Control Flow

A few extra notes:

- **if** allows for variable initialization just like **for** loops

```
if t := time.Now(); t.Hour() < 12 {
    fmt.Println("Good morning")
} else {
    fmt.Println("Good afternoon")
}
```

Control Flow

A few extra notes:

- **if** allows for variable initialization just like **for** loops

```go
if user := getUser(); user != nil {
    fmt.Println("User found:", user.Name)
} else {
    fmt.Println("No user found")
}
```

Control Flow

A few extra notes:

- **if** allows for variable initialization just like **for** loops

```
if v := math.Pow(x, n); v < lim {
    return v
} else {
    fmt.Printf("%g >= %g\n", v, lim)
}
```

Control Flow

A few extra notes:

- **if** allows for variable initialization just like **for** loops
- Multiple if blocks can be made with **switch** blocks

```go
t := time.Now()
switch {
case t.Hour() < 12:
    fmt.Println("Good morning")
case t.Hour() < 17:
    fmt.Println("Good afternoon")
default:
    fmt.Println("Good evening")
}
```

Control Flow

A few extra notes:

- **if** allows for variable initialization just like **for** loops
- Multiple if blocks can be made with **switch** blocks
- **switch-case** blocks don't fall through by default, and thus there is no `break` statement for it*

Control Flow

A few extra notes:

- **if** allows for variable initialization just like **for** loops
- Multiple if blocks can be made with **switch** blocks
- **switch-case** blocks don't fall through by default, and thus there is no `break` statement for it*
    - But they can fall through with the `fallthrough` statement

Control Flow

```go
switch day {
  case "Monday":
    fmt.Println("It's Monday!")
  case "Saturday":
    fallthrough
  case "Sunday":
    fmt.Println("It's Weekend! ")
  default:
    fmt.Println("It's another working day! ")
}
```

Control Flow

A few extra notes:

- **if** allows for variable initialization just like **for** loops
- Multiple if blocks can be made with **switch** blocks
- **switch-case** blocks don't fall through by default, and thus there is no `break` statement for it*
- Conditions don't need to be wrapped in parentheses

Control Flow

A few extra notes:

- **if** allows for variable initialization just like **for** loops
- Multiple if blocks can be made with **switch** blocks
- **switch-case** blocks don't fall through by default, and thus there is no `break` statement for it*
- Conditions don't need to be wrapped in parentheses
- Go has no ternary operator

Control Flow - Defer

Go has a special type of control flow mechanism called **defer**

- It basically stacks a function to be executed after the surrounding function finishes.

```go
func main() {
    defer fmt.Println("world")

    fmt.Println("hello")
}
```

Control Flow - Defer

Go has a special type of control flow mechanism called **defer**

- It basically stacks a function to be executed after the surrounding function finishes
- More than one function can be stacked for deferring
  - Stacked functions will execute in a last-in-first-out order

Control Flow – Defer

```go
func main() {
    fmt.Println("countdown")

    for i := 0; i < 10; i++ {
        defer fmt.Println(i)
    }

    fmt.Println("done")
}
```

```
countdown
done
9
8
7
6
5
4
3
2
1
0
```

Control Flow – Defer

Go has a special type of control flow mechanism called **defer**

- It basically stacks a function to be executed after the surrounding function finishes
- More than one function can be stacked for deferring
  - Stacked functions will execute in a last-in-first-out order
- This is normally used for cleanup purposes (such as closing file handlers, closing network connections, etc)

Practice

2. Write a function "calc" that takes two numbers and a string. The string represents an operation to be made: one of "add", "subtract", "multiply", "divide". Make it return the result of that operation between the two numbers. Have it work for integers first, then for floating point numbers. Make sure to handle division by zero!

Exercise

Let's apply what we have learned to a real world scenario

Temperature Data Analysis

Description:

Imagine you are developing software for a weather station. This station records temperatures throughout the day. Write a program in Go that processes a list of temperatures and provides the highest, lowest, and average temperature.

Steps:

Initialize a slice of integers representing temperatures at different times of the day.

Iterate through the slice to find the maximum and minimum temperatures.

Calculate the average temperature of the day.

Print out the maximum, minimum, and average temperatures.

Memory, Pointers and References

Memory and Pointers

Pointers are a special type of variable that contain a **reference** to the space of *memory* that holds its value

- But what exactly does that mean?

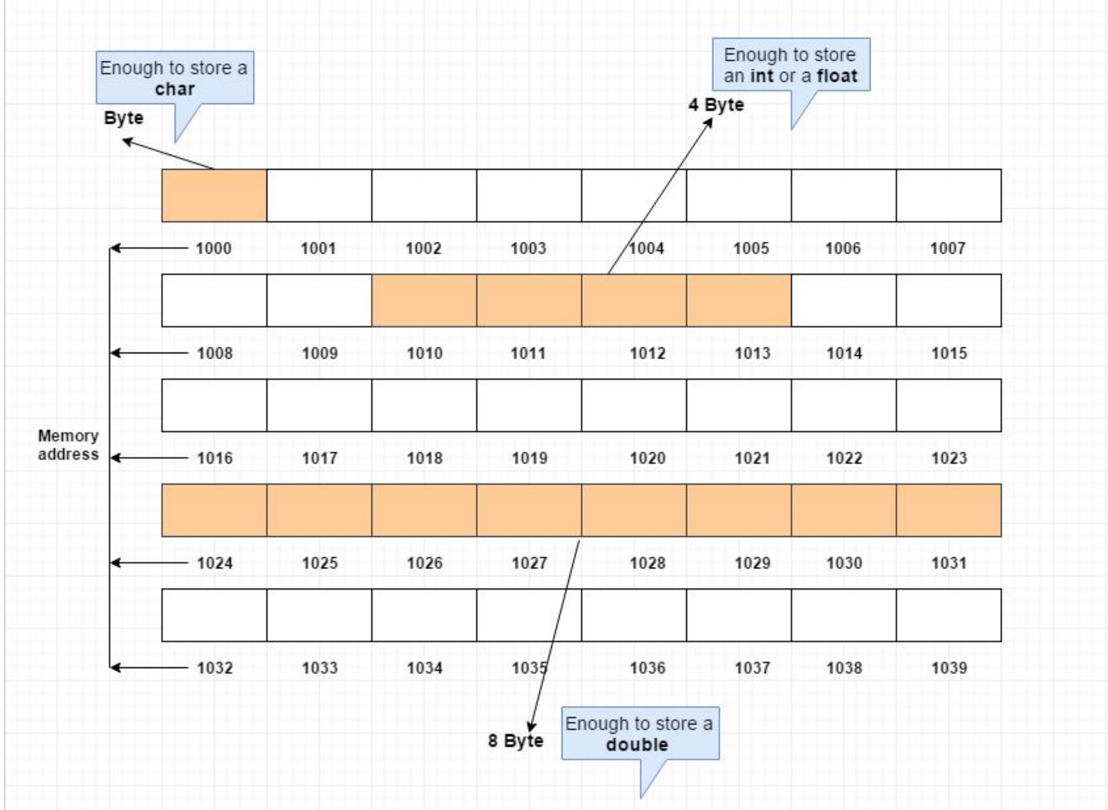
Memory and Pointers

Pointers are a special type of variable that contain a **reference** to the space of *memory* that holds its value

- But what exactly does that mean?

A computer memory is divided into hexadecimal **addresses**

How a computer memory works



Source: $\log_2{}^2$ (Computer memory address basics)

*addresses represented using decimal numbers for simplicity

How a computer memory works

- In the previous picture, each block is a **byte**
    - A byte consists of **8 bits**
- When a program needs to store a value, it **allocates** a specific number of bytes in the memory
    - For a character, 1 byte (8 bits) is enough*
    - For integers, the greater number of bytes used, more values can be represented

How a computer memory works

- For integers, the greater number of bytes used, more values can be represented
    - How many values can be represented by a 8 bits integer (`int8`)? What are those values?
    - And for a `int32`?

How a computer memory works

- For a string, we store the number of blocks matching the string's length
  - A string is a sequence of **characters** (1 byte*)
  - A 16 character long string needs 16 bytes

*Unicode characters may require 2 bytes

How a computer memory works

- For an array, something similar to a string is made
    - An 16-bit integer array of length 5 requires 10 bytes (2 bytes per integer * 5 values)

How a computer memory works

- When you declare a variable, you are basically telling the program to *find a place in memory* to **store** a value of that type
  - To find a **place in memory** = To find an **address**
  - For normal variables, the compiler will know the address its stored, *abstracting* this from us

How a computer memory works

- But what if you want to know the address the variable is stored in?

How a computer memory works

- But what if you want to know the address the variable is stored in?
  - When you pass a parameter into a function, you are basically making a copy of it that can be accessed inside the function
  - This is not a problem for small **objects** (integers, small arrays, etc)
  - But what if your object has a very complex structure (an array with 1000 items for instance)?

Pointers and References

```go
func foo(slc []int) {
    fmt.Printf("Address of slice in function: %p\n", &slc)
}


func main() {
    slc := []int{1,2,3,4,5,6,7,8,9,10,11,12,13,14,15}
    fmt.Printf("Address of slice in main: %p\n", &slc)
    foo(slc)
```

```
Address of slice in main: 0x1400000c018
Address of slice in function: 0x1400000c030
```

`&slc` returns the memory address where the data in `slc` is stored

Pointers and References

- But what if your object has a very complex structure (an array with 1000 items for instance)?
    - By passing the object to the function directly, the same amount of memory would be allocated twice!
    - In order to prevent this from happening, using less memory, we can use **pointers**!

Pointers and References

- When we declare a pointer, we are basically declaring a variable that stores the **memory address** for a value
  - **Memory address = Reference**
  - In the previous example, `&slc` contained the memory address of the `slc` variable
    - `&slc` is a **reference** to `slc`

Pointers and References

```go
num := 5
fmt.Printf("Value of num: %d\n", num)
fmt.Printf("Address of num: %p\n", &num)

words := "Hello world"
fmt.Printf("Value of words: %s\n", words)
fmt.Printf("Address of words: %p\n", &words)

slc := []int{1,2,3,4,5,6,7,8,9,10}
fmt.Printf("Value of slc: %v\n", slc)
fmt.Printf("Address of slc: %p\n", &slc)
```

```
Value of num: 5
Address of num: 0x1400000e0a0
Value of words: Hello world
Address of words: 0x14000010070
Value of slc: [1 2 3 4 5 6 7 8 9 10]
Address of slc: 0x1400000c018
```

Pointers and References

- When we declare a pointer, we are basically declaring a variable that stores the **memory address** for a value
  - In order to declare a variable that is a pointer, the asterisk character (*) can be used before the **data type**

Pointers and References

```go
num := 5
var pointer *int
pointer = &num

fmt.Printf("Value of num: %d\n", num)
fmt.Printf("Address of num: %p\n", &num)

fmt.Printf("Value of pointer: %p\n", pointer)
```

```
Value of num: 5
Address of num: 0x1400000e0a0
Value of pointer: 0x1400000e0a0
```

Pointers and References

```go
num := 5
var pointer *int
pointer = &num

fmt.Printf("Value of num: %d\n", num)
fmt.Printf("Address of num: %p\n", &num)

fmt.Printf("Value of pointer: %p\n", pointer)
```

```
Value of num: 5
Address of num: 0x1400000e0a0
Value of pointer: 0x1400000e0a0
```

`pointer` is of type `*int` (a pointer to an integer)

Accessing the value of a pointer

- When we declare a pointer, we are basically declaring a variable that stores the **memory address** for a value
    - In order to declare a variable that is a pointer, the asterisk character (*) can be used before the **data type**
    - In order to access the value present in a pointer, we can also use an asterisk before the variable name

Accessing the value of a pointer

```go
    num := 5
    var pointer *int
    pointer = &num

    fmt.Printf("Value of num: %d\n", num)
    fmt.Printf("Address of num: %p\n", &num)

    fmt.Printf("Value of pointer: %p\n", pointer)
    fmt.Printf("Value contained in the memory address from pointer: %d\n", *pointer)
```

```
Value of num: 5
Address of num: 0x14000110018
Value of pointer: 0x14000110018
Value contained in the memory address from pointer: 5
```

Practice

1. Write a function "sum_all" that takes as a parameter a slice of integers and returns the sum of all its elements. You can use a `for` loop for this.
2. After the function has been implemented, change it to take a pointer to a slice of integers as a parameter

Custom Types and Structs

Structs

Complex types can be created using structs, and accessed using dots

```go
type Vertex struct {
    X int
    Y int
}
```

```go
v := Vertex{1, 2}
v.X = 4
fmt.Println(v.X)
```

Structs

Structs can also be accessed and manipulated using pointers

```go
type Vertex struct {
	X int
	Y int
}
```

```go
v := Vertex{1, 2}
p := &v
p.X = 4
fmt.Println(v.X)
```

Structs

Structs can also be accessed and manipulated using pointers

- Go allows us to use the direct pointer variable in order to access and modify the properties of a struct, instead of having to dereference it using asterisk
  - We can write `p.X = 4` instead of `(*p).X = 4`

Structs – Anonymous fields

If we declare a field for a struct without a name, it becomes an **anonymous** field.

- We can invoke that field by using its type as the field name

```go
type A struct {
    int
}

func main() {
    a := A{5}
    fmt.Println(a.int)
}
```

Structs – Promoted fields

Similar to anonymous fields, we can have a struct inherit the fields of another one and "extending" it

- The inherited fields become **promoted** fields

```go
type A struct {
    foo int
}

type B struct {
    A
}

func main() {
    b := B{A{5}}
    fmt.Println(b.foo)
}
```

Exercise

Let's apply what we have learned to a real world scenario

Exercise 2: Simple Bank System

Description:

In this exercise, you'll create a simple banking system. Users can deposit and withdraw money, and you'll need to keep track of each user's balance. Ensure that users cannot withdraw more money than they have.

Steps:

Define a struct Account with at least two fields: Name (string) and Balance (float64).

Create a slice of Account representing multiple users.

Write functions to Deposit and Withdraw money from an account.

Ensure the Withdraw function checks for sufficient funds.

Iterate over the slice of accounts and perform a series of transactions.

Print out the final balance of each account.

Data Types and Control Flows in Go

Ambush Journey Program

Ambush