Orbit Challenge 002

2025-05-25 00:03

Status: #adult

Tags: programming Golang bgce

Credits

First of all, This quiz was made possible because of some of our dedicated member of this community and their server nickname are `j1407b🪐` `unicorn🦄` `Tai Lung` `Master Shifu`

All quiz answer

1.

```
s := []int{}
for i := 0; i < 5; i++ {
 s = append(s, i)
 fmt.Printf("len=%d cap=%d\n", len(s), cap(s))
 }
```

What are capacity values printed?

1. `1 2 4 8 16`,
2. `1 2 3 4 5`,
3. `1 2 4 6 8`,
4. `1 2 4 4 4`,
5. None of the above

The Correct option is Option 5. It's None of them

```
len=1 cap=1
len=2 cap=2
len=3 cap=4
len=4 cap=4
len=5 cap=8
```

The capacity increase only when the length is equal to it and you are adding another element.

2.

```
func slcmaker() (int, int) {
 slc := make([]int, 17)
 slc1 := append(slc, 1)
 return cap(slc), cap(slc1)
 }
```

So what does this function `slcmaker()` return?

1. 17, 34
2. 17, 17
3. 17, 36
4. 34, 34

The correct answer is Option 3. It's 17, 36.
Although in theory below 1024, If you wanna add another element to a slice with same length and capacity it will increase 100% hence 2x. But this theory is it estimate or get a approximate cap value for the slice. In truth in GO, slice increase a bit differently. it's not always 100% below 1024 and not always 25% after 1024. It has it's own rules. to get a a better idea you can look at the Go source code. But it does increases through memory block or byte number.
Look at this table:

> This is how the capacity of a slice increases when you try to add a new element to slice with the same length and capacity

Cap before	Cap after adding an element	Percentage of growth
1	2	+100.00%
2	4	+100.00%
3	6	+100.00%
4	8	+100.00%
5	10	+100.00%
6	12	+100.00%
7	14	+100.00%
8	16	+100.00%
9	18	+100.00%
10	20	+100.00%
11	22	+100.00%
12	24	+100.00%
13	26	+100.00%
14	28	+100.00%
15	30	+100.00%
16	32	+100.00%
17	36	+111.76%
18	36	+100.00%
19	40	+110.53%
20	40	+100.00%
21	44	+109.52%
22	44	+100.00%
23	48	+108.70%
24	48	+100.00%
25	52	+108.00%
26	52	+100.00%
27	56	+107.41%
28	56	+100.00%
29	60	+106.90%
30	60	+100.00%
31	64	+106.45%
32	64	+100.00%
33	72	+118.18%
34	72	+111.76%
35	72	+105.71%
36	72	+100.00%
37	80	+116.22%
38	80	+110.53%
39	80	+105.13%
40	80	+100.00%
41	88	+114.63%
42	88	+109.52%

Cap before	Cap after adding an element	Percentage of growth
43	88	+104.65%
44	88	+100.00%
45	96	+113.33%
46	96	+108.70%
47	96	+104.26%
48	96	+100.00%
49	112	+128.57%
50	112	+124.00%
51	112	+119.61%
52	112	+115.38%
53	112	+111.32%
54	112	+107.41%
55	112	+103.64%
56	112	+100.00%
57	128	+124.56%
58	128	+120.69%
59	128	+116.95%
60	128	+113.33%
61	128	+109.84%
62	128	+106.45%
63	128	+103.17%
64	128	+100.00%
65	144	+121.54%
66	144	+118.18%
67	144	+114.93%
68	144	+111.76%
69	144	+108.70%
70	144	+105.71%
71	144	+102.82%
72	144	+100.00%
73	160	+119.18%
74	160	+116.22%
75	160	+113.33%
76	160	+110.53%
77	160	+107.79%
78	160	+105.13%
79	160	+102.53%
80	160	+100.00%
81	176	+117.28%
82	176	+114.63%
83	176	+112.05%
84	176	+109.52%
85	176	+107.06%
86	176	+104.65%
87	176	+102.30%
88	176	+100.00%
89	192	+115.73%
90	192	+113.33%
91	192	+110.99%
92	192	+108.70%
93	192	+106.45%
94	192	+104.26%
95	192	+102.11%
96	192	+100.00%
97	224	+130.93%
98	224	+128.57%
99	224	+126.26%
100	224	+124.00%

```go
func main() {
    var x uint8 = 255
    x++
}
```

What's the value of `x`?

1. Compile error
2. Runtime panic
3. Prints 0
4. Shifts right

The correct answer is option 1
We get Compile-time Error because In Go, you can't use a negative number with shift operators like << or >>. So x << -1 is not allowed.

4.

```go
func main() {
    var a int8 = -1
    var b uint8 = 1
    c := a + int8(b)
    _ = c
}
```

What is the value of c?

1. 0
2. -1
3. 1
4. Compile error

Answer: `Option 1` → 0

Explanation:
a = -1 (as int8)
b = 1 (as uint8)
int8(b) converts b from uint8 to int8, which is still 1 (since it's in range).
Now we have: -1 + 1 = 0

5. What is the size of struct{}?

A) 0
B) 1
C) Depends on the compiler
D) Undefined

Answer: A >>> Empty structs struct{} consume 0 bytes as they hold no data, but are still addressable for bookkeeping.

6.

```
func sum(nums ...int) int { /* ... */ }
sum(1, 2, 3)
```

A) []int
B) [3]int
C) int
D) interface{}

Answer: A >>> Variadic parameters like nums are converted to a slice, so inside the function, nums has type []int (a slice, not an array).

7.

```
func main() {
    for i := 0; i < 3; i++ {
        defer func() { fmt.Print(i) }()
    }
}
```

What will be the output?

A) 012
B) 210
C) 333
D) 111

Answer: B >>> The normal print order is: 0 1 2, but since defer stacks the evaluations in reverse, the outputs appear in reverse order: 2 1 0. That's why your output is: 210

8.

```
var s []int
s = append(s, nil...)
```

What happens?

A) s == nil remains true
B) s == nil becomes false
C) Runtime panic
D) Compile error

Answer: D >>> nil cannot be spread (...) as []int, so Compile error occurs. Invalid operation for slice append.

9.

```
s := []int{1, 2, 3}
t := s[:2]
t = append(t, 99)
```

What will be the value of `s`?

1. [1, 2, 99]
2. [1, 2, 3]
3. [1, 99, 3]
4. Compilation error

Answer: Option 1 → [1, 2, 99]
It's because `s` and `t` are pointing the same underlying array. so `s` gets also gets modified with `t`

10.

```
func main() {
 x := 10
   {
      x := x + 1
      fmt.Println(x)
    }
 }
```

What is true about the above code snippet?

1. x was shadowed perfectly in the new block and will print it's new value
2. Go doesn't allow shadowing a variable using itself so this would give a compiler error
3. It goes against Go's code clarity and Go compiler will give you a warning
4. x won't be shadowed instead it will just be reassigned using it's previous value

Answer: Option 3. yes it might seem like it gets shadowed perfectly. But in Go, although it does allow shadowing a variable using itself, it not what Go code clarity prefers. That's why in your code editor you will see a warning 😴

11.

```
func mutate(arr *[3]int) {
 arr[0] = 999
}


 func main() {
 a := [3]int{1, 2, 3}
 b := a
 mutate(&b)
}
```

What is the value of `a` and `b`?

```
1) [1 2 3] [1 2 3]
2) [999 2 3] [1 2 3]
3) [1 2 3] [999 2 3]
4) [999 2 3] [999 2 3]
```

Answer: Option 3 → `[1 2 3] [999 2 3]`

🧠 **Explanation:**
In Go, arrays are **value types**, so `b := a` creates a **copy** of `a`.
`mutate(&b)` only modifies `b`, leaving `a` unchanged.

12.

```
a := 42
ans := 63
b := &a
c := &b
**c = 21
d, ans := 12, a
_ = d
```

What is the value of `ans`?

1. 42
2. 21
3. 63
4. 12
5. Error

Correct Answer: Option 2 → 21

🧠 **Explanation:**
`c` is a pointer to `b`, which is a pointer to `a`. So `**c = 21` updates `a` to `21`.
Then `ans := 63` is shadowed by `ans := a` in `d, ans := 12, a`, so `ans` becomes `21`.
The original `63` is replaced due to partial declaration with `:=` where `ans` get ressigned.

13.

```
type Foo struct {
    bar int
}

func (f *Foo) change() {
    f.bar = 100
}

func main() {
    var f *Foo
    f.change()
}
```

What will happen if you try to print the value of `f`?

1. {100}
2. 100
3. compiler error

4. runtime error

Correct Answer: Option 4 → runtime error

🧠 **Explanation:**
`f` is a **nil pointer** (`var f *Foo`), and calling `f.change()` tries to access `f.bar`.
This causes a **runtime panic** (`nil pointer dereference`) because there's no actual `Foo` instance allocated.

14.

```
s := []*int{new(int), new(int)}
for _, v := range s {
    *v = 1
}
fmt.Println(*s[0])
```

What will be the output?

A) 0
B) 1
C) Compile error
D) Undefined

Answer: B >>> The loop modifies the heap-allocated integers through pointers, so `*s[0]` reflects the assigned value 1. The slice stores pointers, not copies.

15.

```
type T struct{}
 a := make([]T, 10)
 fmt.Println(len(a), cap(a))
```

Output?

A) 0 0
B) 10 10
C) 10 0
D) Compile error

Answer: B >>> Zero-sized types like struct{} still count toward length/capacity, so make([]T, 10) creates a slice with 10 10. Memory optimization happens internally.

16.

```
type T1 int
type T2 int
var x T1 = 1
var y T2 = 1
fmt.Println(x == y)
```

What will be the output?

A) true
B) false
C) Compile error
D) Runtime panic

Answer: C >>> Different named types (T1 and T2) are distinct in Go and cannot be directly compared, causing Compile error. Type conversion is required.

17.

```go
type T struct { x int }
a := T{1}
b := T{1}
fmt.Println(a == b)
```

What will be the output?

A) true
B) false
C) Compile error
D) Runtime panic

Answer: A >>> Structs with identical field values are comparable in Go, so the == operator compares field values.

18. Where are environment variables stored in a process?

A) Code segment
B) Heap
C) Data segment
D) Kernel space

Answer : C >>> Environment variables are stored in the data segment, specifically in the higher memory addresses near the stack.

19. What is the size of EAX?

A) 8-bit
B) 16-bit
C) 32-bit
D) 64-bit

Answer : C >>> EAX is a 32-bit register (Extended AX). The "E" stands for extended in x86 architecture.

20. Which register is conventionally used for function return values in x86-64?

A) RAX
B) RBX
C) RSP
D) RBP

Answer : A >>> RAX is the standard register for return values in x86-64 calling conventions.

21. Which field is not typically stored in the PCB?

A) Process ID (PID)
B) CPU register values
C) Heap-allocated variables
D) Process state

Answer: C >>> The PCB stores process metadata (PID, registers, state) but not the actual heap/stack data, so Heap-allocated variables are excluded.

22. Which register is not general-purpose in x86?

A) AX
B) SP
C) IR
D) CX

Answer: C >>> IR (Instruction Register) is an internal CPU register not accessible to programmers, unlike general-purpose registers (AX, SP, CX).

23. Which resource is shared by threads but not by processes?

A) Heap memory
B) Program counter (PC)
C) Stack memory
D) File descriptors

Answer : A >>>> Threads within the same process share the heap memory (for dynamic allocation), but each has its own stack (for local variables) and program counter (execution state). File descriptors may or may not be shared depending on design. Thus, Heap memory is uniquely shared among threads.

24. `var pi float32 = 22/7`
    What is the value of `pi` here?

A) 3.14
B) 3.14159
C) 3.142857
D) 3

Answer: Option D → `3`
🧠 **Explanation:**
`22/7` is an **integer division**, so it evaluates to `3` before being converted to `float32`.
To get a precise result like `3.142857`, you'd need at least one operand to be a float: `22.0/7` or `22/7.0`.

25.

```
func main() {
    a := 7
    b := 1
    result := a & b
    _ = result
}
```

what is the `result`?
A) 0
B) 2

C) 1
D) Compile error

Answer: Option C → 1

Explantion:
The & operator compares bits of a and b. 7 is 0111 in binary 1 is 0001 in binary Only the last bit matches, so the result is 0001, which is 1.

26.

```
type Foo struct {
    Name string
}

func memory() {
    f := Foo{Name: "GoLang"}
    _ = f
}
```

Where is the string "GoLang" allocated?

A) Stack only
B) Heap
C) Code Segment
D) Data Segment

Correct Answer: D
String literals like "GoLang" are stored in the data segment, which holds static, read-only data that's part of the compiled binary. Even if f is on the stack, the actual string "GoLang" lives in the data segment.

27. `cool := [...]int{3, 1, 5, 10, 100}` if we append another element in `cool` what is gonna happen?

A) the new value of cool will be [3, 1, 5, 10, 100, 10],
B) the new length will be 6 and capacity will be 6,
C) the new length will be 6 and capacity will be 10,
D) Error

Answer: Option D → Error

It will throw an error cause cool is not a slice but an array. in Go, we cannot append element to an array

28.

```
func arrquiz()bool {
  var arr2 [4] int
  arr2_cpy := arr2
  arr2_cpy[0] = 25
  return arr2_cpy[0] == arr2[0]
}
```

What will the function return?

1. true
2. false
3. nil
4. Invalid syntax

Answer: Option 2 → `false`

It will return false cause array copy the whole array and point to different address unlike slice which in this same case would've point to the same underlying array. so that's for array after changing the first value of `arr2_cpy` it will be different from `arr2` so `false`

29.

```go
type Container struct {
    Data []int
}

func add(c Container) {
    c.Data = append(c.Data, 100)
}

func main() {
    c := Container{Data: []int{1, 2}}
    add(c)
}
```

What will it print?

A) [1 2 100]
B) [1 2]
C) []
D) Depends on memory layout

Answer: Option B → [1, 2]
The add function gets a copy of the Container, not the original. So when it appends 100, it changes the copy—not the original. That's why c.Data in main stays [1 2].

30.

```go
func MySlice() []int {
    s := make([]int, 1_000_000)
    return s
}
```

Where is the slice backing array allocated?

A) Always on stack
B) Always on heap
C) Depends on whether s escapes
D) On heap because of slice size

Correct Answer: C

The backing array is allocated on the stack if it doesn't escape the function, but if the slice `s` is returned and used outside, it escapes to the heap to stay valid after the function ends. So allocation depends on whether `s` escapes.