1. **Какой самый эффективный способ конкатенации строк?**

Самый эффективный способ конкатенации строк – использование структуры strings.Builder, которая минимизирует копирование памяти. Использовать string в случае, когда необходимо постоянно складывать строки неэффективно, т.к. требуется выделение памяти на каждую конкатенацию ввиду иммутабельности строки.

Пример:

```
package main

import (
   "fmt"
   "strings"
)

func main() {
   var sb strings.Builder

   for i := 0; i < 1000; i++ {
      sb.WriteString("a")
   }

   fmt.Println(sb.String())
}
```

2. **Что такое интерфейсы, как они применяются в Go?**

Интерфейсы представляют собой тип данных, которые позволяют определять абстракцию поведения других типов. Интерфейсы определяют набор абстрактных методов без реализации, функционал которых необходимо реализовать в типах, которые имплементируют данный интерфейс.

Интерфейс позволяет использовать полиморфизм, когда при общем интерфейсе имеется разная реализация. Программа может использовать абстракцию в виде интерфейса без необходимости переопределять каждый раз метод для различных объектов.

Интерфейсы в Golang содержат список(можно пустой) методов без их реализации. Чтобы тип данных соответствовал интерфейсу, необходимо реализовать все методы данного интерфейса.

Также, пустой интерфейс `interface{}` в Golang считается реализованным у любого типа и может использоваться в качестве универсального типа. Тип *any* является синонимом пустого интерфейса.

3. **Чем отличаются RWMutex от Mutex?**

При использовании Mutex в случае блокировки ресурса только одна горутина имеет доступ к нему: на чтение и на запись. При использовании же RWMutex можно читать неограниченным количеством потоков, но блокировать для одной горутины на запись. Метод RLock() дает возможность параллельного чтения. При этом метод Lock() не будет доступен, пока не будет отпущен мьютекс через RUnlock(). В случае Lock()/Unlock(), доступ к ним будет по очереди.

4. **Чем отличаются буферизированные и не буферизированные каналы?**

Буферизированные каналы при инициализации требуют размер буфера:

```
ch := make(chan int, size)
```

Где size – размер. При превышении размеров буфера (в случае отправки) канал блокируется и ждет освобождения буфера. Аналогично при опустошении буфера – канал блокируется до появления в нем данных; в таких ситуациях часто может возникнуть дедлок.

Небуферизованные каналы блокируются до получения данных. Данные можно отправлять только в пустой канал, в отличие от буферизованного, который имеет предельный размер. После отправки данных горутина блокируется до тех пор, пока данные из канала не будут получены.

Также, при закрытии буферизованного канала, с него можно читать до опустошения буфера, но нельзя писать в него.

5. **Какой размер у структуры struct{}{}?**

0 байт

6. **Есть ли в Go перегрузка методов или операторов?**

Нет

7. **В какой последовательности будут выведены элементы map[int]int?**

Пример:
m[0]=1
m[1]=124
m[2]=281

В случайной

8. **В чем разница make и new**

new возвращает указатель на тип, когда make возвращает сам экземпляр типа. Также make можно использовать только для slice, map и chan.

make необходим для их инициализации. К примеру, он выделяет память для массива, который лежит в структуре слайса, задается длина и вместимость. В случае использования new для создания слайса – будет возвращено zero-value для ссылочного типа равное **nil.** Массив внутри слайса не будет создан, будет возвращен пустой указатель. Аналогично с мапой и с каналом.

9. **Сколько существует способов задать переменную типа slice или map?**

Для slice, где T - тип:

Определить через var:

```
var mySlice []T // неинициализированная
```

Инициализировать значениями:

```
var mySlice []T = {val1,val2}
mySlice := []T{val1,val2}
```

Из массива или слайса:

```
mySlice := arr[нижняя_граница:верхняя_граница]
mySlice := notmySlice[нижняя_граница:верхняя_граница]
```

Через make:

```
var mySlice = make([]T,длина,вместимость)
mySlice := make([]T,длина,вместимость)
```

Для map, T,V – типы:

Через var:

```
var myMap[T]V // неинициализированная
```

Инициализировать значениями (аналогично как слайс через var можно еще):

```
myMap := map[T]V{key: val}
```

Через make(аналогично через var можно):

```
mySlie := make(map[T]V,вместимость)
```

10. Что выведет данная программа и почему?

```
func update(p *int) {
  b := 2
  p = &b
}

func main() {
  var (
    a = 1
    p = &a
  )
  fmt.Println(*p)
  update(p)
  fmt.Println(*p)
}
```

Первый print выведет 1, т.к. указатель на a = 1;

Второй print выведет 1, т.к. в golang в функциях идет передача по значению. В данном случае мы передаем значение указателя, которое локально в функции перезаписывается, не меняя указатель в main.

11. Что выведет данная программа и почему?

```
func main() {
  wg := sync.WaitGroup{}
  for i := 0; i < 5; i++ {
    wg.Add(1)
    go func(wg sync.WaitGroup, i int) {
      fmt.Println(i)
      wg.Done()
    }(wg, i)
  }
  wg.Wait()
  fmt.Println("exit")
}
```

Выдаст значения от 0 до 4 в случайном порядке, а также ошибку дедлок (сломается на wg.Wait()), т.к. необходимо передавать указатель на wg, иначе каждая горутина имеет свою локальную копию wg.

12. Что выведет данная программа и почему?

```
func main() {
  n := 0
  if true {
    n := 1
    n++
  }
  fmt.Println(n)
}
```

Выведет 0, потому что в блоке if создается своя локальная переменная n

13. Что выведет данная программа и почему?

```
func someAction(v []int8, b int8) {
  v[0] = 100
  v = append(v, b)
}

func main() {
  var a = []int8{1, 2, 3, 4, 5}
  someAction(a, 6)
  fmt.Println(a)
}
```

Выведет 100 2 3 4 5.

Слайс – ссылочный тип. В функцию мы передаем копию слайса, который содержит в себе указатель на массив. Поэтому через обращение по индексу можно поменять значения во внутреннем массиве. После применения append слайс в случае нехватки capacity еще и реаллоцируется. Поэтому переменная v содержит уже указатель на новый внутренний массив.

Кроме того, len и cap передаются по значению, а не по ссылке. Поэтому, даже если бы cap было достаточно для того, чтобы не выделять новую память, при выводе слайса «a» выводилось бы [100 2 3 4 5], так как значение len остается неизменным и менялось только у слайса-копии «v» в функции.

14. Что выведет данная программа и почему?

```
func main() {
  slice := []string{"a", "a"}

  func(slice []string) {
    slice = append(slice, "a")
    slice[0] = "b"
    slice[1] = "b"
```

```
    fmt.Print(slice)
  }(slice)
  fmt.Print(slice)
}
```

Выведет b b a , после чего выведет a a.

В анонимную функцию передается копия слайса, которая содержит указатель на тот же внутренний массив. Но так как изначальная capacity слайса = 2, после вызова функции append выделяется новая память под внутренний массив и значения со старого массива копируются в новый. Поэтому при попытке поменять значения элементов через индекс в исходном слайсе в main функции изменений не происходит, так как слайс в анонимной функции ссылается уже на другой внутренний массив. Кроме того, у копии слайса меняется и значение len, так как оно передается по значению. Исходный же слайс в main функции содержит значение len =2 и выводит соответственно буквы a a.

В случае, если бы в main функции изначально был слайс с cap=4

slice := make([]string, 0, 4)

slice = append(slice, "a", "a")

Тогда при вызове slice = append(slice, "a") в анонимной функции указатель на внутренний массив бы не менялся, т.к. capacity > len(slice)+len(data). Но в конце программы вывод был бы «b b», так как len=2 у исходного слайса не меняется в функции.

В функции слайс перезаписывается при вызове метода append, так как он копирует значения слайса (в конкретно данном случае реаллоцируется) и записывается в локальную переменную анонимной функции. Если бы slice[0] =”b” стояли до append, то в слайсе из main значения бы поменялись тоже.