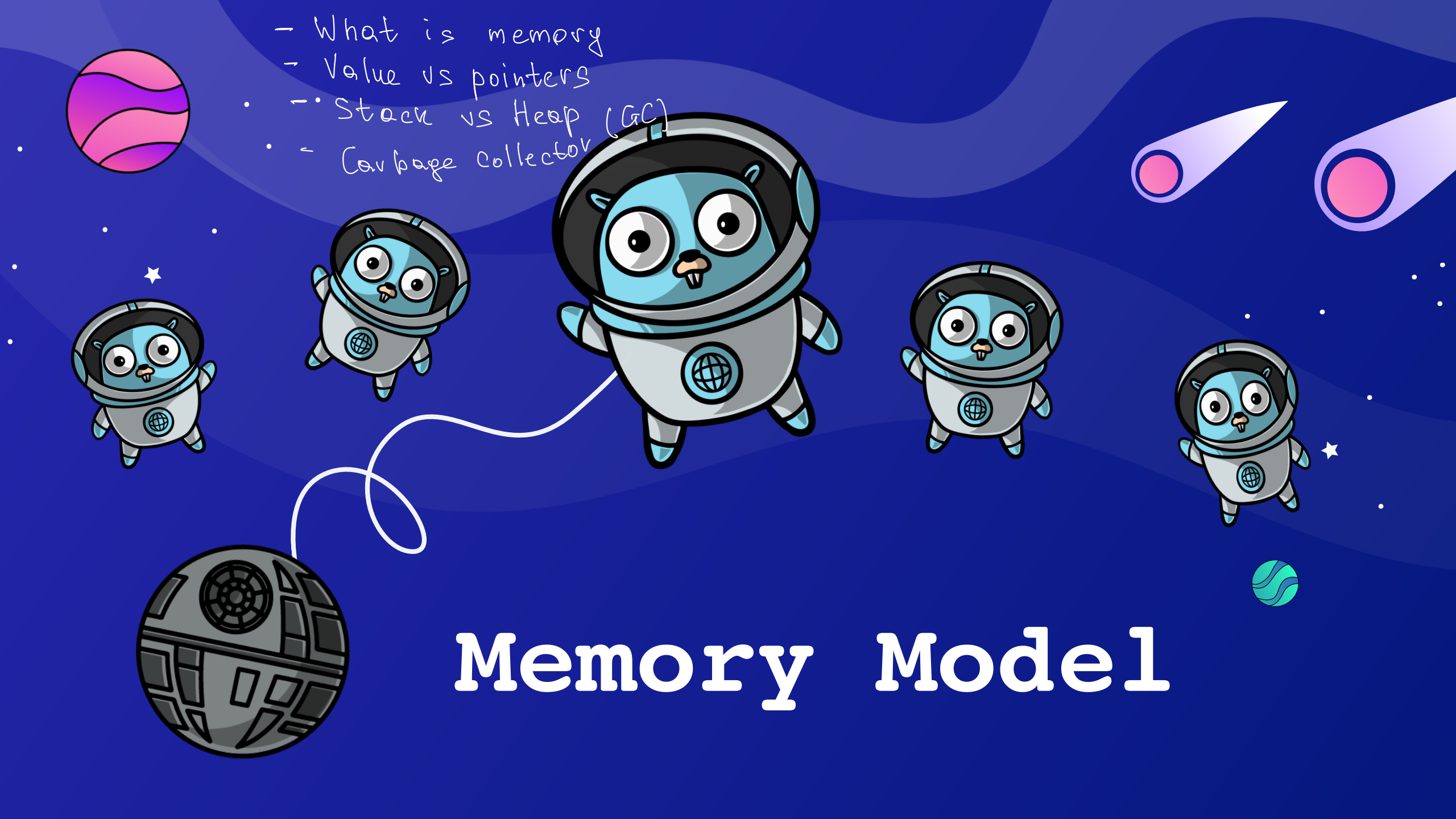

- What is memory
- Value vs pointers
- Stack vs Heap (GC)
- Garbage collector
Memory Model

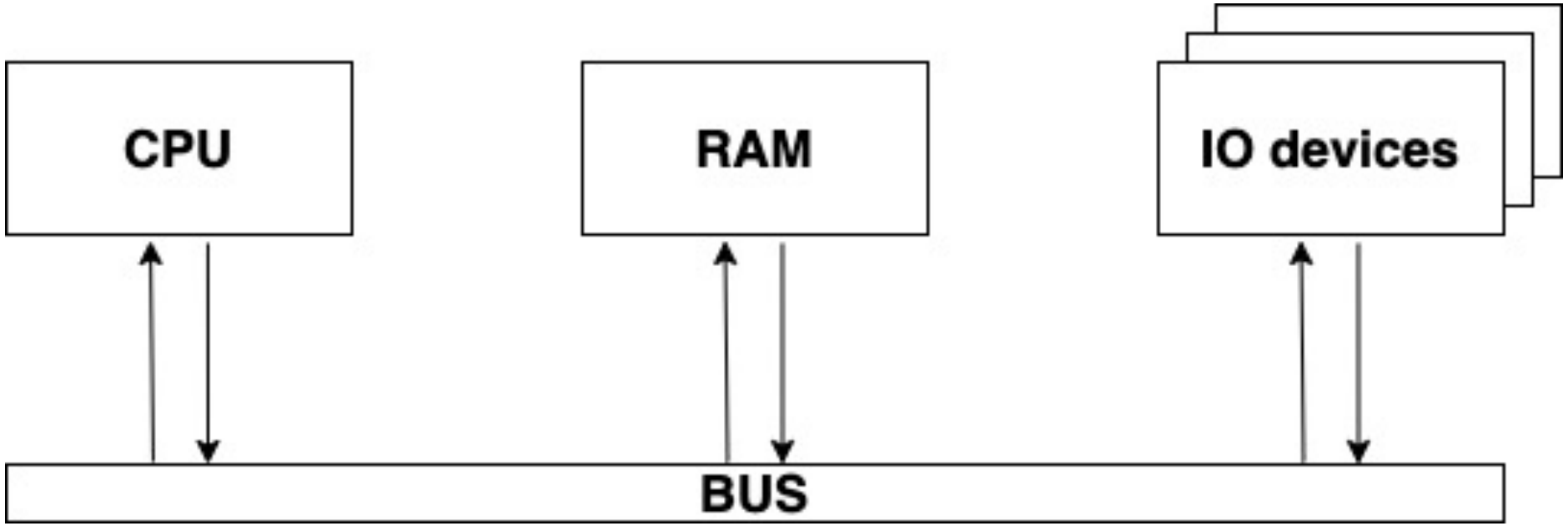
What is memory

Processor registers
+ Different levels of cache

Random access memory

Persistent memory

Simplified memory model



Memory type

CPU:

- Processor registers
- L1, L2, L3 registers

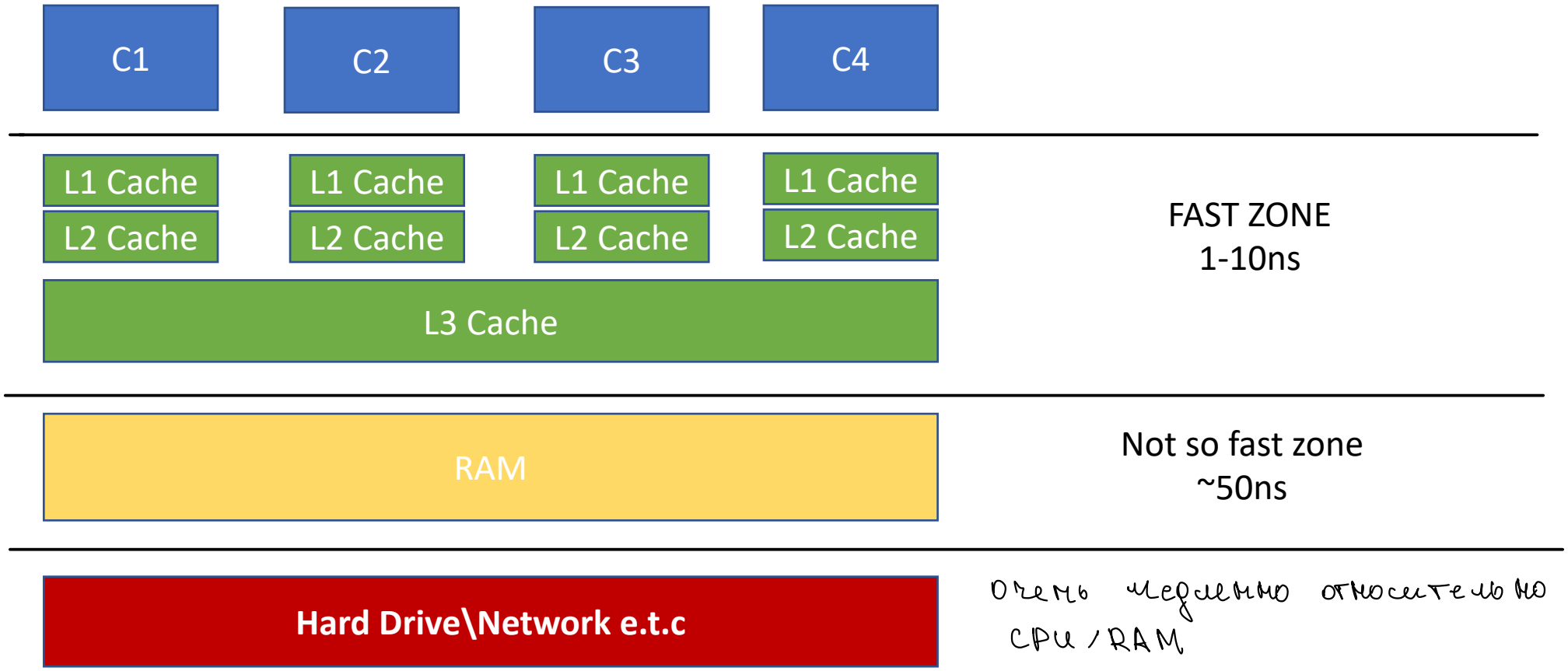
RAM:

- Memory solded or inserted into mother bord (not persistent)

IO devices:

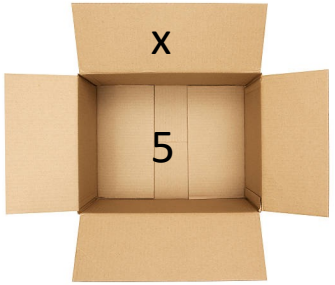
- External drives
- CD/DVD
- Flash cards

Physical memory

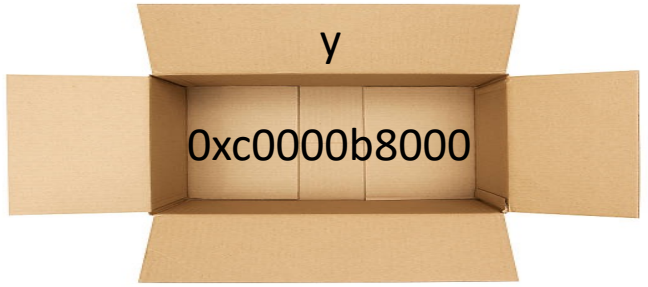


Variables

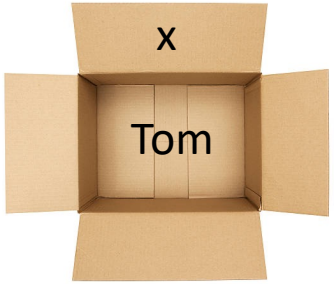
```go
package main

import "fmt"

func main() {
    var x int = 5

    var y *int = &x

    fmt.Println(x, y)
}

```



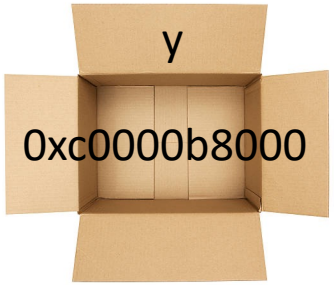
https://goplay.space/#Vzd2VwWJRpe



Variables



- Where is Tom?
- He is here!



Where is Tom?
He is at 0xc0000b8000!

What can we do with pointers?

```go
package main

import "fmt"

func main() {

    var x int = 5

    var y *int = &x

    *y = 10

    fmt.Println(x, y)
}

```

&x - get an address of the variable(reference)

***y –** get actuall value stored at pointer address (dereference)

(in some cases compiler can do ***y**) implicitly, when it can be confirmed as safe

https://goplay.space/#LnoMkBtckoN

How does it works?

```go
package main

import "fmt"

func main() {

    var x int = 5

    var y *int = &x

    *y = 10

    fmt.Println(x, y)
}

```

./main.go:15:13: inlining call to fmt.Println

./main.go:9:6: moved to heap: x

./main.go:15:13: x escapes to heap

./main.go:15:13: []interface {}{...} does not escape

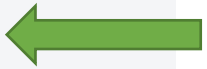
go build -gcflags="-m" main.go

What is “stack”

- A **Stack** is a region in memory created to store temporary variables ***bound to a function***.
- It’s **self cleaning**.
- It’s can **expands** and **shrinks** accordingly.

Stack

```go
package main

import (
    "fmt"
)

func main() {
    var x int = 5
    var y int = 7

    var s int = sum(x, y)

    fmt.Println(s)
}

func sum(d1, d2 int) int {
    r := d1 + d2

    return r
}
```



Stack frame

https://goplay.space/#FSbyfn08oRf

Stack

```go
package main

import (
    "fmt"
)

func main() {
    var x int = 5
    var y int = 7

    var s int = sum(x, y)

    fmt.Println(s)
}

func sum(d1, d2 int) int {
    r := d1 + d2

    return r
}
```

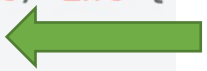
STACK

func main()
x = 5
y = 7
s = 0

https://goplay.space/#FSbyfn08oRf

Stack

```
1  package main
2
3  import (
4      "fmt"
5  )
6
7  func main() {
8      var x int = 5
9      var y int = 7
10
11     var s int = sum(x, y)
12
13     fmt.Println(s)
14 }
15
16 func sum(d1, d2 int) int {
17     r := d1 + d2
18
19     return r
20 }
21
```

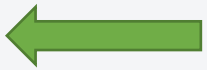


STACK
main() x = 5 y = 7 s = 0
sum() d1 = 5 d2 = 7 r = 12

https://goplay.space/#FSbyfn08oRf

Stack

```go
package main

import (
    "fmt"
)

func main() {
    var x int = 5
    var y int = 7

    var s int = sum(x, y)

    fmt.Println(s)
}

func sum(d1, d2 int) int {
    r := d1 + d2

    return r
}
```

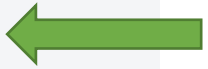


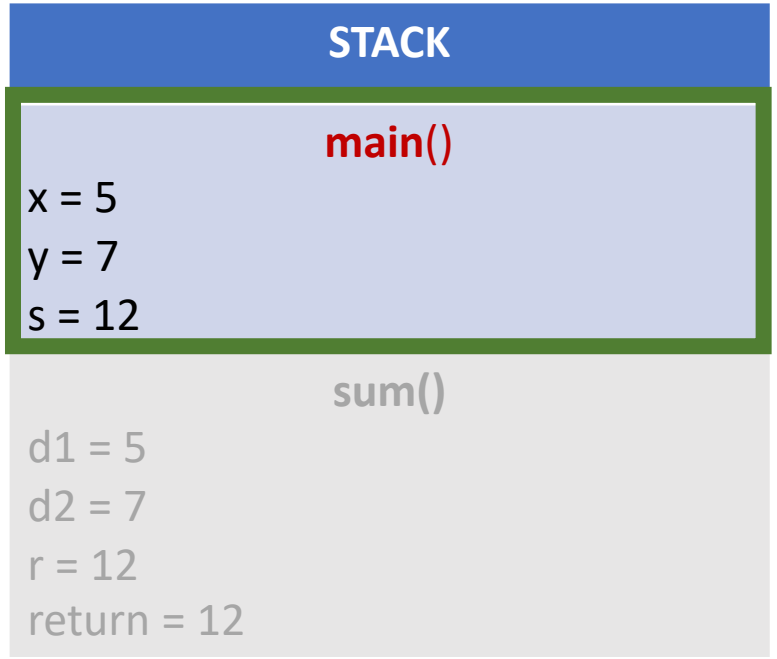
STACK
main() x = 5 y = 7 s = 0
sum() d1 = 5 d2 = 7 r = 12 return = 12

https://goplay.space/#FSbyfn08oRf

Stack

```
1  package main
2
3  import (
4      "fmt"
5  )
6
7  func main() {
8      var x int = 5
9      var y int = 7
10
11     var s int = sum(x, y)
12
13     fmt.Println(s)
14 }
15
16 func sum(d1, d2 int) int {
17     r := d1 + d2
18
19     return r
20 }
21
```

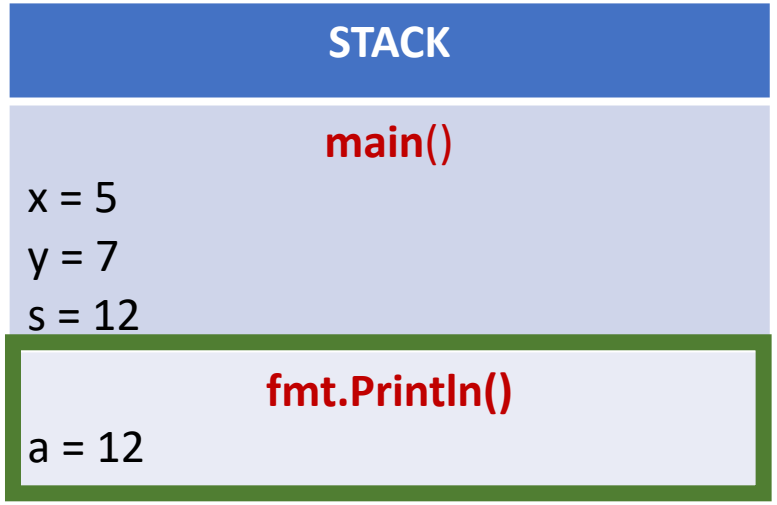




https://goplay.space/#FSbyfn08oRf

Stack

```go
package main

import (
    "fmt"
)

func main() {
    var x int = 5
    var y int = 7

    var s int = sum(x, y)

    fmt.Println(s)
}

func sum(d1, d2 int) int {
    r := d1 + d2

    return r
}
```



https://goplay.space/#FSbyfn08oRf

Stack or heap?

```go
package main

import (
    "fmt"
)

func main() {
    var x int = 5
    var y int = 7

    var s int = sum(x, y)

    fmt.Println(s)
}

func sum(d1, d2 int) int {
    r := d1 + d2

    return r
}
```

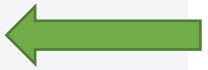
go build -gcflags '-m'

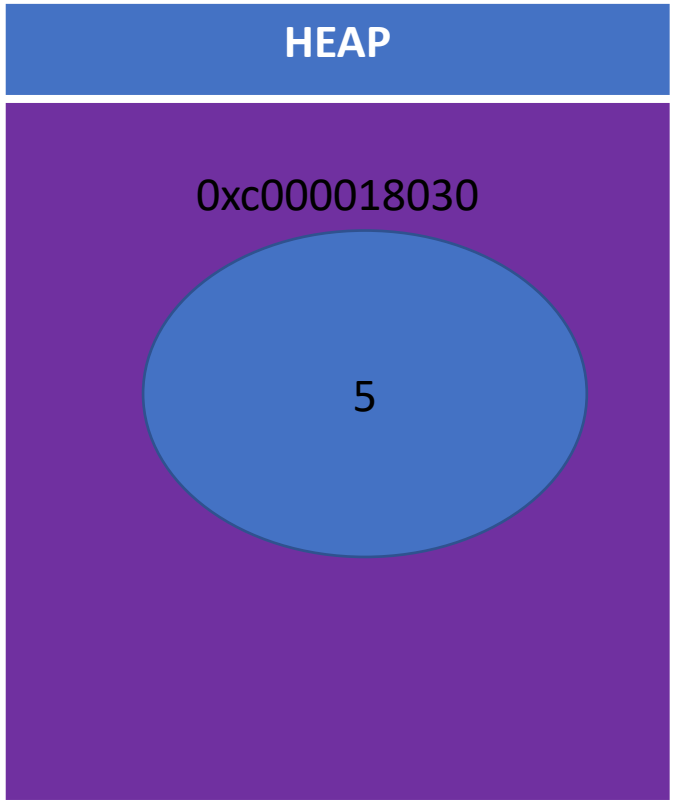
```
# command-line-arguments
.\mm.go:16:6: can inline sum
.\mm.go:11:17: inlining call to sum
.\mm.go:13:13: inlining call to fmt.Println
.\mm.go:13:13: s escapes to heap
.\mm.go:13:13: []interface {} literal does not escape
<autogenerated>:1: .this does not escape
<autogenerated>:1: .this does not escape
```

https://goplay.space/#FSbyfn08oRf

Stack and heap

```go
package main

import "fmt"

func main() {

    var x int = 5

    var y *int = &x

    var z *int = &x

    *y = 10

    fmt.Println(x, y, z)
}

```

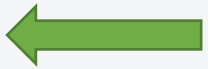


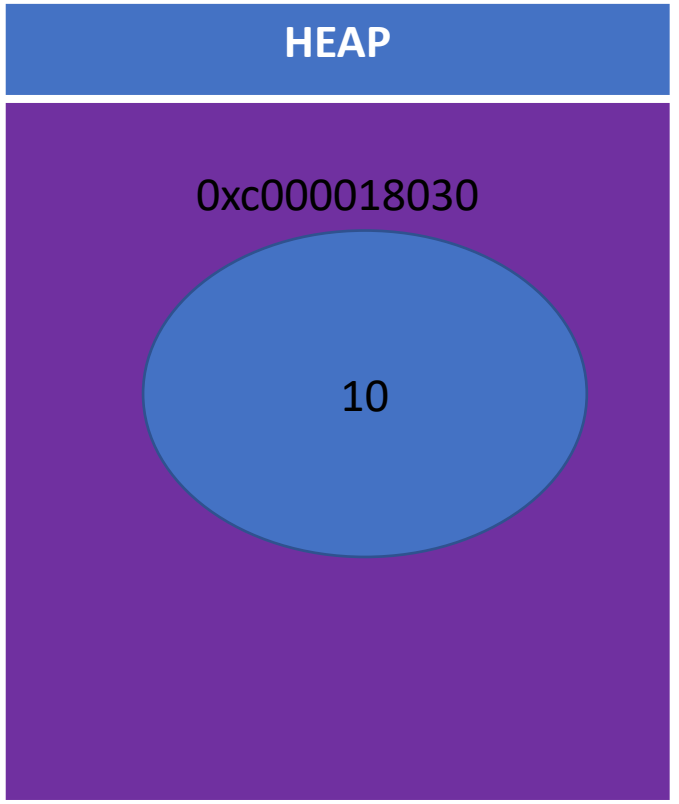


https://goplay.space/#FSbyfn08oRf

Stack and heap

```go
package main

import "fmt"

func main() {

    var x int = 5

    var y *int = &x

    var z *int = &x

    *y = 10

    fmt.Println(x, y, z)
}

```





https://goplay.space/#FSbyfn08oRf

Stack and heap

```go
package main

import "fmt"

func main() {

    var x int = 5

    var y *int = &x

    var z *int = &x

    *y = 10

    fmt.Println(x, y, z)
}

```

go build -gcflags '-m'

```
# command-line-arguments
.\mm.go:15:13: inlining call to fmt.Println
.\mm.go:7:6: moved to heap: x
.\mm.go:15:13: x escapes to heap
.\mm.go:15:13: []interface {} literal does not escape
<autogenerated>:1: .this does not escape
<autogenerated>:1: .this does not escape
```

https://goplay.space/#FSbyfn08oRf

When it escapes to heap?

https://goplay.space/#FSbyfn08oRf

What is escapes to heap ?
Escape analysis

- When possible, the Go compiler will allocate variables that are local to a function in that function's **stack frame**.
- Go looks for variables that **outlive** the current stack frame and therefore need to be heap-allocated
- If a variable has its address taken, that variable is a candidate for allocation on the heap. However, a basic escape analysis recognizes some cases when such variables will not live past the return from the function and can reside on the stack.

Escape analysis

```go
package main

type Agent struct {
    Name string
}

//go:noinline
func EscapeFunction(name string) *Agent {
    agent := &Agent{
        Name: name,
    }
    return agent
}

//go:noinline
func NoEscapeFunction(name string) Agent {
    agent := new(Agent)
    agent.Name = name
    return *agent
}

func main() {
    _ = EscapeFunction("007")

    _ = NoEscapeFunction("07")
}
```

- .\task01.go:8:21: leaking param: name
- .\task01.go:9:11: &Agent literal escapes to heap
- .\task01.go:16:23: leaking param: name
- .\task01.go:17:14: new(Agent) does not escape

Garbage collection

- Garbage collection first time starts when heap is >= 4mb
- Garbage collector might be triggered in two cases
  - There is not garbage collection for more than 2 minutes
  - Heap grows by specific percent (**100%** by default)
- You can configure how often garbage collector starts with ***GOGC** env variable*

Garbage collection

https://goplay.space/#7Cp4sXKZHk5

```go
type User struct {
    id       int
    username string
}

const (
    ReplaceOp = iota
    RemoveOp
)

func main() {

    const bufferSize = 1000000
    users := make([]*User, bufferSize)

    for i := 0; i < 100; i++ {
        op := i % 2
        fmt.Println(op)

        switch op {
        case ReplaceOp:
            for j := 0; j < bufferSize; j++ {
                users[j] = &User{id: j, username: strings.Repeat(string(j), j%1000)}
            }

        case RemoveOp:
            for j := 0; j < bufferSize; j++ {
                users[j] = nil
            }
        }

        time.Sleep(1 * time.Second)
    }
}
```

GODEBUG=gctrace=1 go run main.go

gc 1 @0.014s 0%: 0.025+0.49+0.011 ms clock, 0.41+0.87/0.80/0.30+0.19 ms cpu, ***4->4->0 MB****, 5 MB goal, 16 P*

gc 2 @0.030s 0%: 0.038+0.30+0.021 ms clock, 0.61+0.19/0.54/0.24+0.34 ms cpu, ***4->4->0 MB****, 5 MB goal, 16 P*

gc 7 @0.282s 0%: 0.047+2.1+0.023 ms clock, 0.75+0.065/7.7/18+0.37 ms cpu, ***438->440->437 MB****, 449 MB goal, 16 P*

gc 8 @0.495s 0%: 0.025+3.4+0.023 ms clock, 0.40+0.12/12/33+0.37 ms cpu, ***853->857->851 MB****, 875 MB goal, 16 P*

gc 9 @0.898s 0%: 0.022+8.7+0.052 ms clock, 0.36+0.054/33/34+0.84 ms cpu, ***1659->1673->1662 MB****, 1702 MB goal, 16 P*

gc 10 @3.744s 0%: 0.070+5.1+0.019 ms clock, 1.1+0.095/18/46+0.31 ms cpu, ***3242->3248->1204 MB****, 3325 MB goal, 16 P*

gc 11 @5.985s 0%: 0.077+2.3+0.018 ms clock, 1.2+0.095/7.8/15+0.29 ms cpu, ***2348->2355->316 MB****, 2409 MB goal, 16 P*

gc 12 @6.078s 0%: 0.062+3.0+0.063 ms clock, 0.99+0.074/11/24+1.0 ms cpu, ***616->625->620 MB****, 632 MB goal, 16 P*

Garbage collection

https://goplay.space/#7Cp4sXKZHk5

```go
type User struct {
    id       int
    username string
}

const (
    ReplaceOp = iota
    RemoveOp
)

func main() {

    const bufferSize = 1000000
    users := make([]*User, bufferSize)

    for i := 0; i < 100; i++ {
        op := i % 2
        fmt.Println(op)

        switch op {
        case ReplaceOp:
            for j := 0; j < bufferSize; j++ {
                users[j] = &User{id: j, username: strings.Repeat(string(j), j%1000)}
            }

        case RemoveOp:
            for j := 0; j < bufferSize; j++ {
                users[j] = nil
            }
        }

        time.Sleep(1 * time.Second)
    }
}
```

GOGC= 10 GODEBUG=gctrace=1

go run main.go

gc 1 @0.007s 1%: 0.017+0.62+0.050 ms clock, 0.28+0.64/0.67/0+0.81 ms cpu, ***2->2->0 MB****, 3 MB goal, 16 P*

gc 2 @0.009s 2%: 0.037+0.53+0.015 ms clock, 0.59+0.46/0.90/0.10+0.25 ms cpu, ***1->1->0 MB****, 2 MB goal, 16 P*

gc 3 @0.011s 4%: 0.084+1.4+0.027 ms clock, 1.3+1.3/1.6/0.043+0.44 ms cpu, ***1->2->0 MB****, 2 MB goal, 16 P*

……

gc 30 @0.190s 5%: 0.026+1.7+0.035 ms clock, 0.43+0.24/5.4/10+0.57 ms cpu, ***189->190->189 MB****, 190 MB goal, 16 P*

gc 31 @0.209s 5%: 0.029+2.3+0.026 ms clock, 0.46+0.15/7.7/14+0.42 ms cpu, ***207->209->208 MB****, 208 MB goal, 16 P*

gc 32 @0.231s 5%: 0.025+2.6+0.015 ms clock, 0.40+0.10/8.9/18+0.24 ms cpu, ***227->229->229 MB****, 229 MB goal, 16 P*

gc 33 @0.254s 5%: 0.083+3.5+0.043 ms clock, 1.3+0.97/11/20+0.69 ms cpu, ***250->251->251 MB****, 251 MB goal, 16 P*

gc 34 @0.282s 4%: 0.029+3.0+0.051 ms clock, 0.47+0.081/10/18+0.83 ms cpu, ***274->276->275 MB****, 276 MB goal, 16 P*