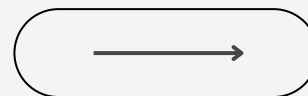


# INTRODUÇÃO À OTIMIZAÇÃO DE MEMÓRIA EM GO

28/11/2024



Auber Mardegan

# SUMÁRIO

03

GARBAGE COLLECTOR

04

ALOCAÇÃO ESTÁTICA (STACK)

05

STACK FRAME

06

ALOCAÇÃO DINÂMICA (HEAP)

09

ESCAPE ANALYSIS

10

MEMORY BENCHMARKING

11

MEMORY PROFILING

12

TRACING

# Garbage Collector

Golang utiliza uma estratégia de garbage collection de rastreamento, aonde o **GC** irá rastrear os objetos alcançáveis por uma cadeia de referências de “objetos raiz”, considerando o resto como “lixo” e desalocando.

Apesar desse mecanismo ser muito eficiente e funcionar de forma automática, não podemos esperar que o **GC** irá garantir a otimização do código, seria a mesma coisa que achar que um linter que remove os espaços em branco do código garantiria toda a qualidade de código da empresa.

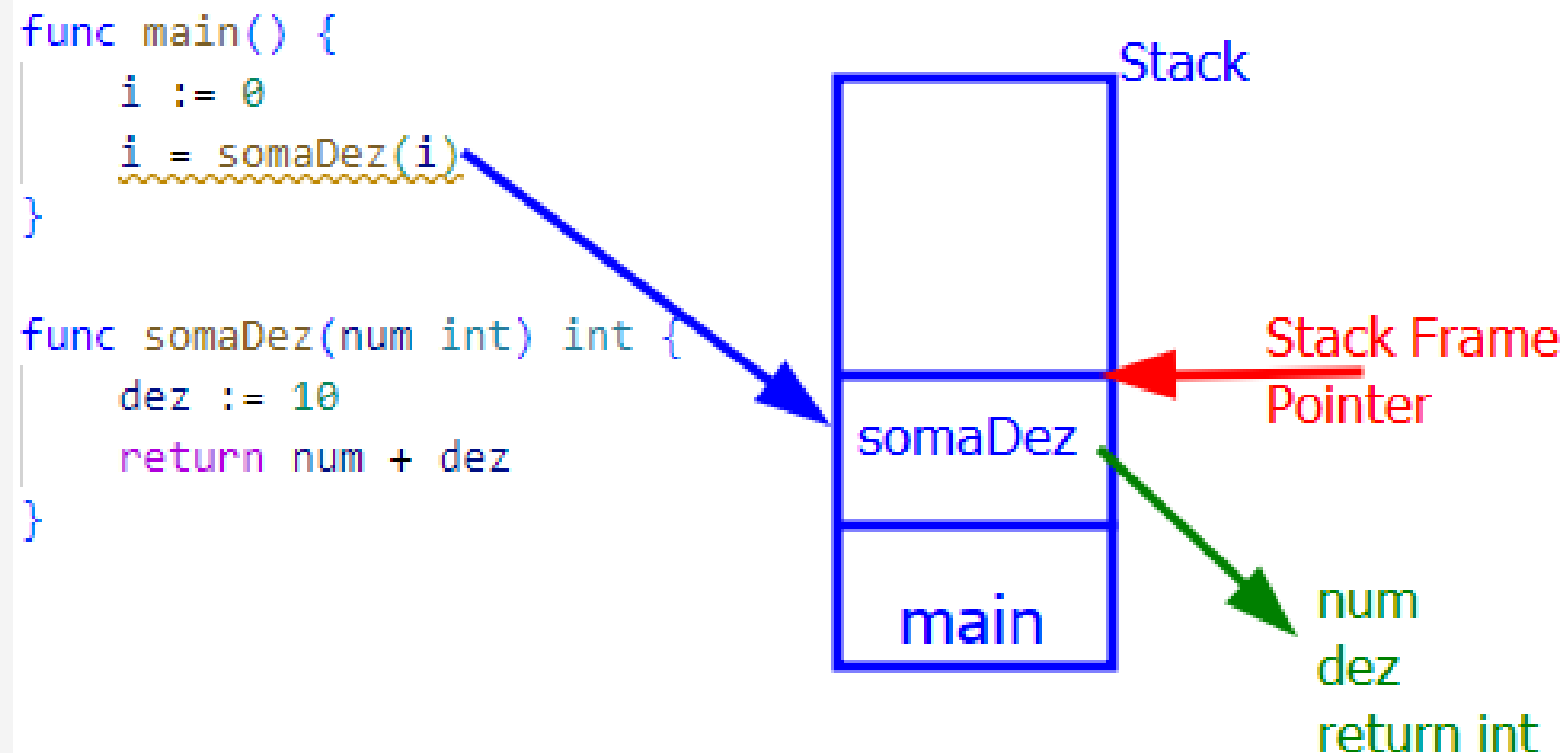
A fim de otimizar o uso de memória, precisamos recapitular o básico entre alocação estática (**Stack**) e alocação dinâmica (**Heap**).

# Alocação Estática (Stack)

- Mecanismo **LIFO** (Last in, First Out) garante que a chamada mais recente de função está sempre no topo do Stack podendo ser limpa assim que a função terminar;
- Cada função cria um **Stack Frame** com todas as informações necessárias para aquela execução. Por exemplo: variáveis locais, argumentos e o endereço de retorno;
- Possui uma limitação ao tamanho do objeto;
- Alocação usada quando o compilador identifica que o ciclo de vida da variável não ultrapassa o escopo da função.

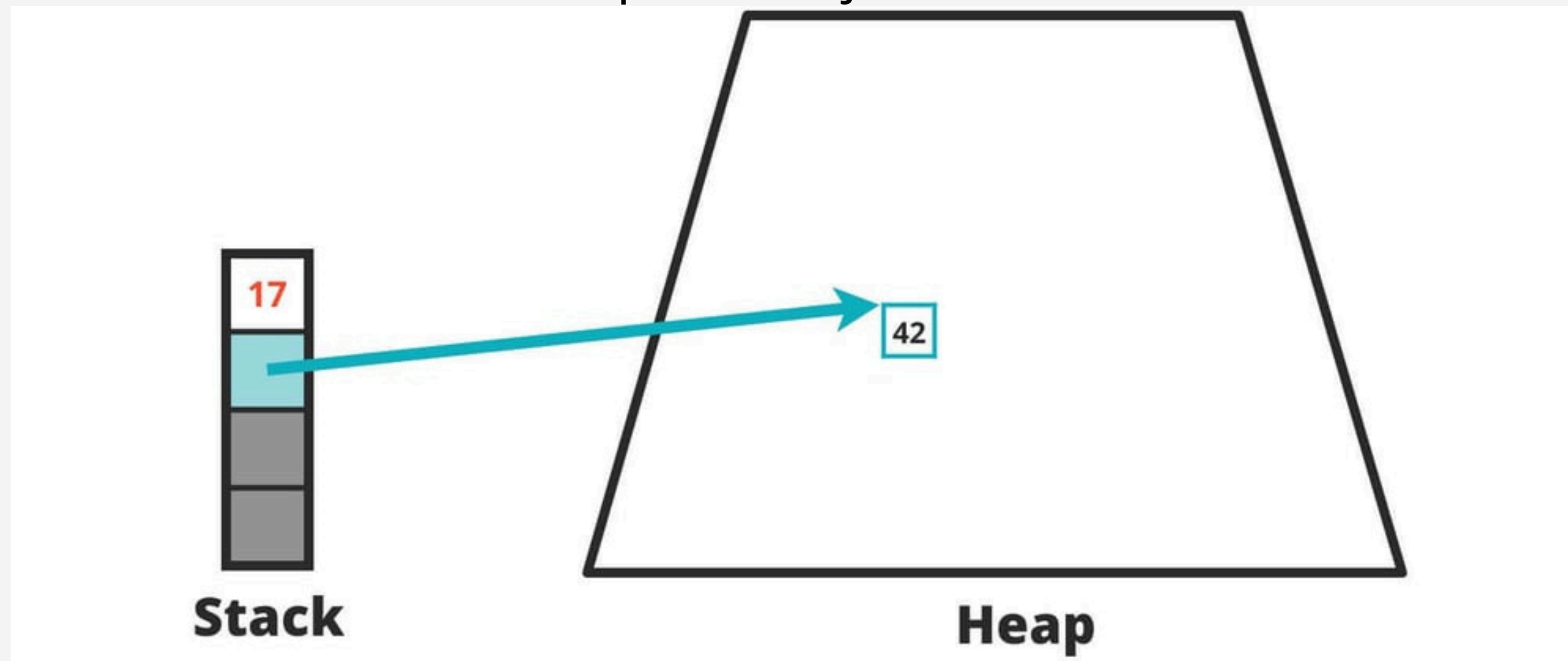
# Stack Frame

- Armazenamento local que pertence à função
- Cada stack frame está acoplado a uma goroutine, onde os objetos armazenados são usados de forma privada
- Os objetos são gerenciados pelo ciclo de vida do Frame



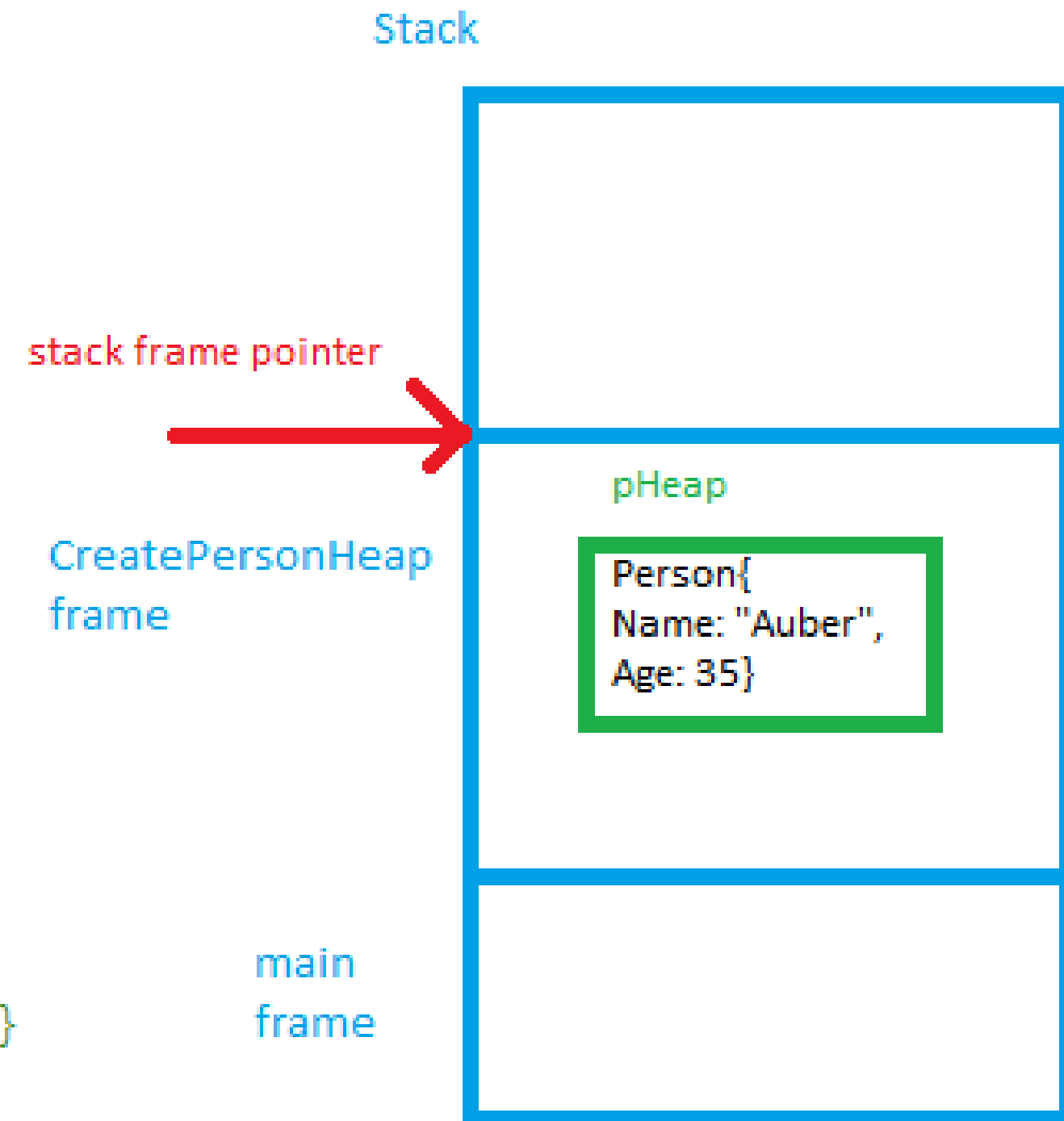
# Heap

- A alocação dinâmica é menos estruturada e mais flexível
- Um grande espaço global onde os objetos podem ser compartilhados
- Os objetos são gerenciados pelo **GC**
- Como o ciclo de vida é variável e não está preso na função, impõe um custo de **GC** para ficar conferindo o estado daqueles objetos



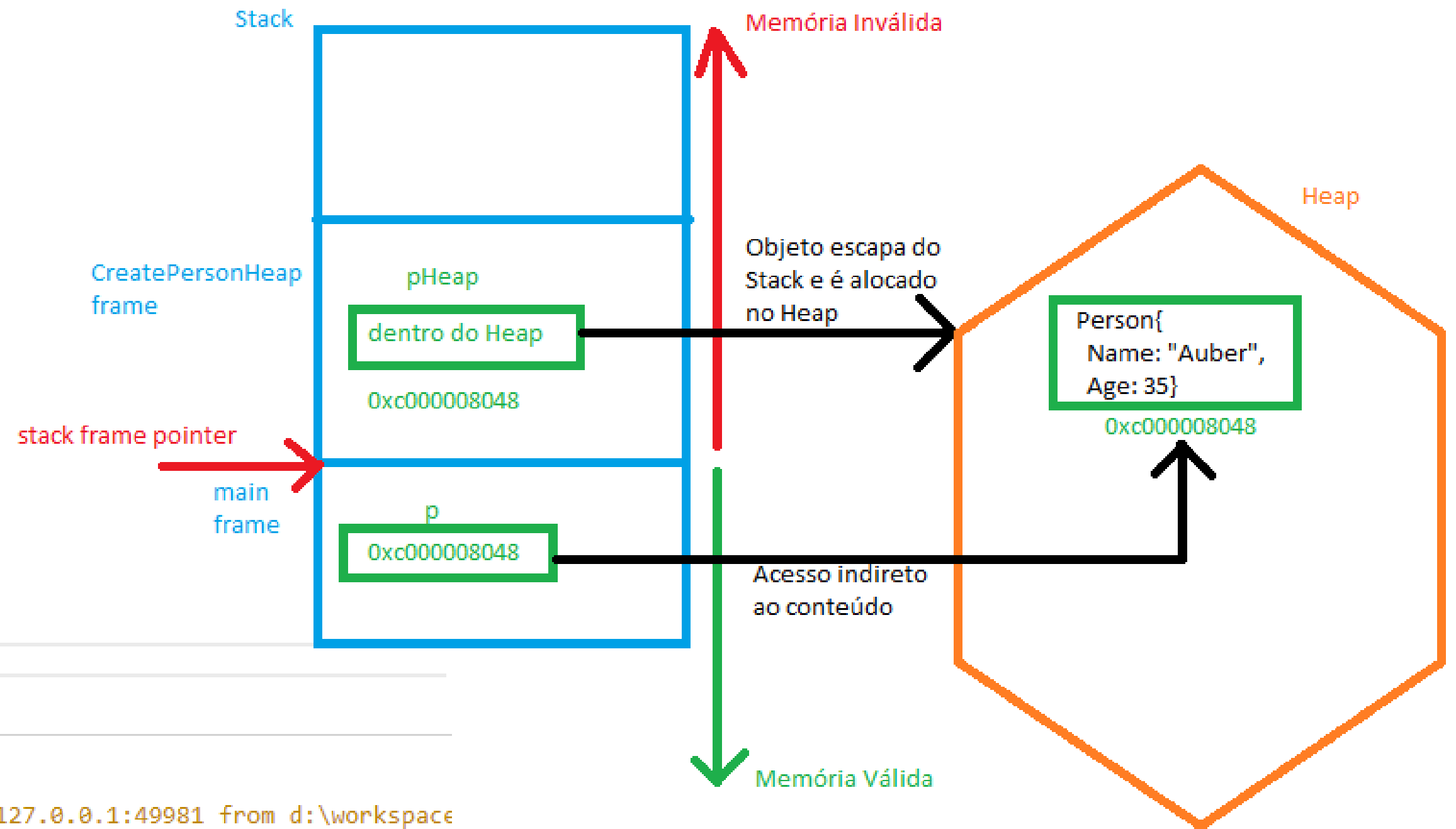
# Heap

```
1  package main
2
3  import "fmt"
4
5  func main() {
6      p := CreatePersonHeap()
7
8      fmt.Printf("%p", p)
9  }
10
11 type Person struct {
12     Name string
13     Age  int
14 }
15
16 func CreatePersonHeap() *Person {
17     pHeap := Person{Name: "Auber", Age: 35}
18     return &pHeap
19 }
20
```



# Heap

```
1 package main
2
3 import "fmt"
4
5 func main() {
6     p := CreatePersonHeap()
7
8     fmt.Printf("%p", p)
9 }
10
11 type Person struct {
12     Name string
13     Age  int
14 }
15
16 func CreatePersonHeap() *Person {
17     pHeap := Person{Name: "Auber", Age: 35}
18     return &pHeap
19 }
20
```



PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
Starting: C:\Users\Auber\go\bin\dlv.exe dap --listen=127.0.0.1:49981 from d:\workspace
DAP server listening at: 127.0.0.1:49981
Type 'dlv help' for list of commands.
0xc000008048
Process 10908 has exited with status 0
Detaching
```



# Escape Analysis

Mecanismo que decide se a variável escapa do stack e deve ser alocada no heap em **tempo de compilação**.

Durante a compilação da aplicação podemos passar argumentos para o **GC** explicar o processo de escape analysis em um determinado pacote ou arquivo, o que pode ser usado em análises para otimização de uso de memória.

```
go build -gcflags "-m -m" .
```

# Memory Benchmarking

```
go test -run none -bench . -benchtime 3s -benchmem
```

# Memory Profiling

```
go test -run none -bench . -benchtime 3s -benchmem -memprofile mem.out  
go tool pprof -alloc_space .\exemplo1.test.exe .\mem.out  
list BenchmarkCreatePersonStack  
list BenchmarkCreatePersonHeap
```

# Tracing

```
go test -run TestCreatePersonStack -trace traceStack.out
```

```
go test -run TestCreatePersonHeap -trace traceHeap.out
```

```
go tool trace .\traceStack.out
```

```
go tool trace .\traceHeap.out
```

# Otimização

Sempre otimizar o código para:

- integridade;
- facilidade de entendimento;
- simplicidade.

Só partir pra otimização de performance quando identificar a necessidade através de medição.

**“Premature optimization is the root of all evil.” - Donald Knuth**

# REFERENCIAS

System Programming Essentials with Go - Alex Rios, 2024 ([Packt Publishing](#))

Deep dive into the escape analysis in Go - Jalex Chang ([GopherCon Taiwan 2020](#))

Escape analysis and Memory Profiling - William Kennedy ([GopherCon Singapore 2017](#))

Understanding Allocations in Go - James Kirk, 2020 ([Medium](#))

<https://tip.golang.org/doc/gc-guide>

# OBRIGADO!

LinkedIn

<https://www.linkedin.com/in/auber-mardegan>

