Memory Management in Go: The good, the bad and the ugly

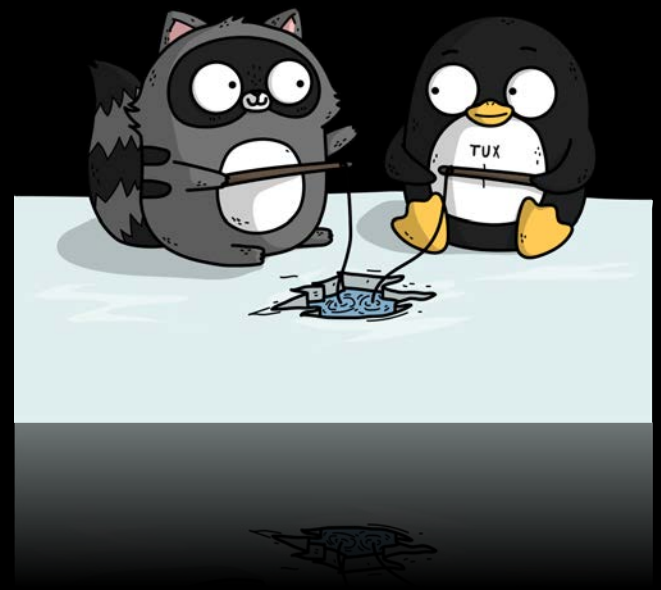
Liam Hampton

Sr. Cloud Advocate @ Microsoft

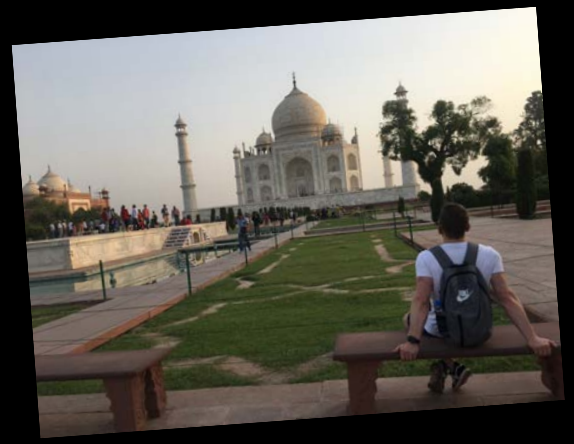
@liamchampton

Agenda

- About me
- Introduction to memory management
- Go's memory model
- Managing memory in Go
- Good / bad code examples
- Memory management in other languages
- Top Tips
- Conclusion



Microsoft



- Microsoft Sr. Cloud Advocate
- Auth0 Ambassador
- DevNetwork Advisory Board Member
- I write Go code
- I travel the world

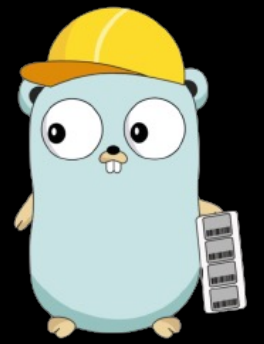
@liamchampton

The Learning Goal(s)

1. Understand the Go memory model
2. Understand how to manage memory in Go


@liamchampton

Introduction to memory management

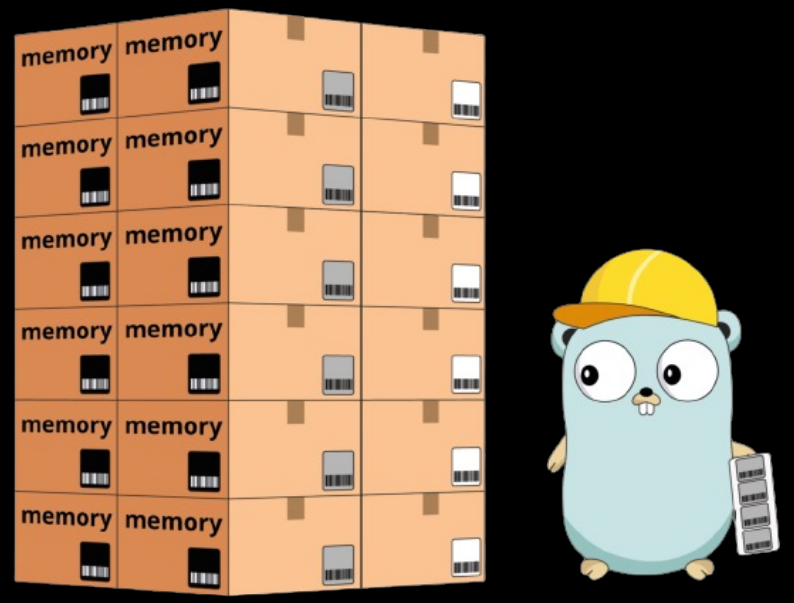


@liamchampton

Overview

Memory management

“Memory management keeps track of each memory location, regardless of either it is allocated to some process, or it is free.”



Overview

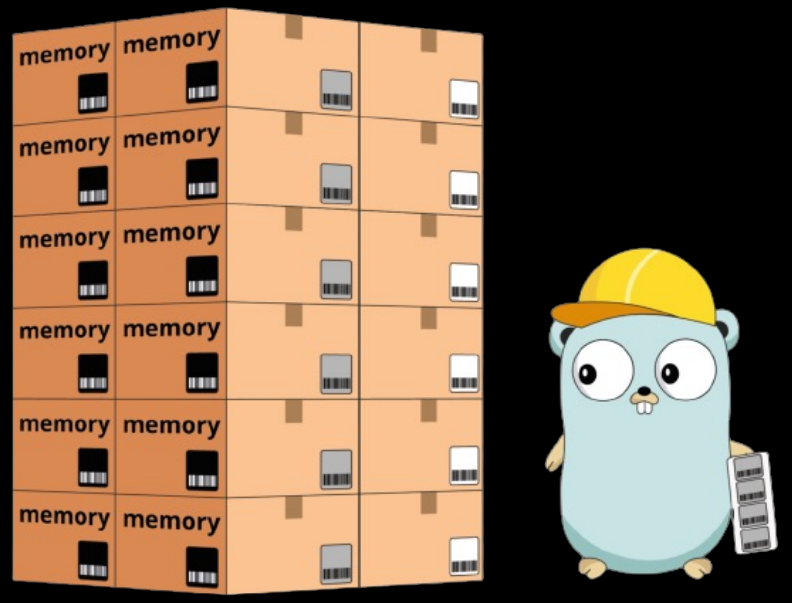
Memory management

"Memory management keeps track of each memory location, regardless of either it is allocated to some process, or it is free."

Why is it important?

Prevents memory leaks, program crashes and a slow down of your system

You must also avoid buffer overflows as this could lead to security vulnerabilities



Stack vs Heap

Stack

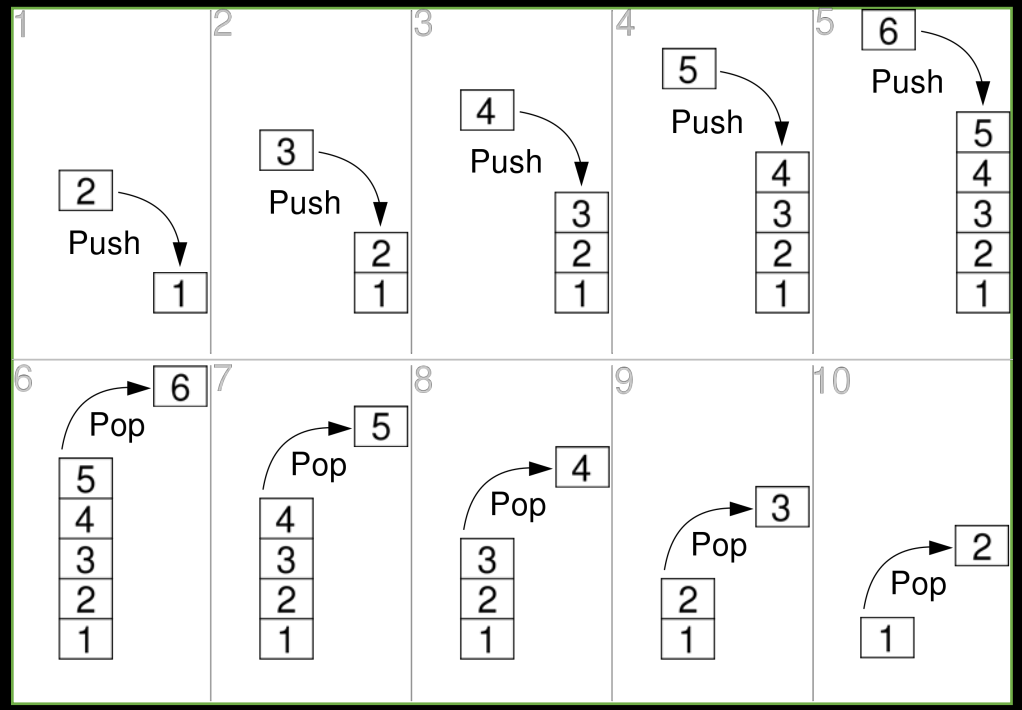
Stores local vars and function call frames

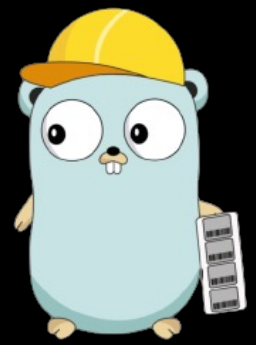
Last In First Out (LIFO)

Typically a fixed size

Allocated at runtime

Fast and efficient but is limited in size





@liamchampton

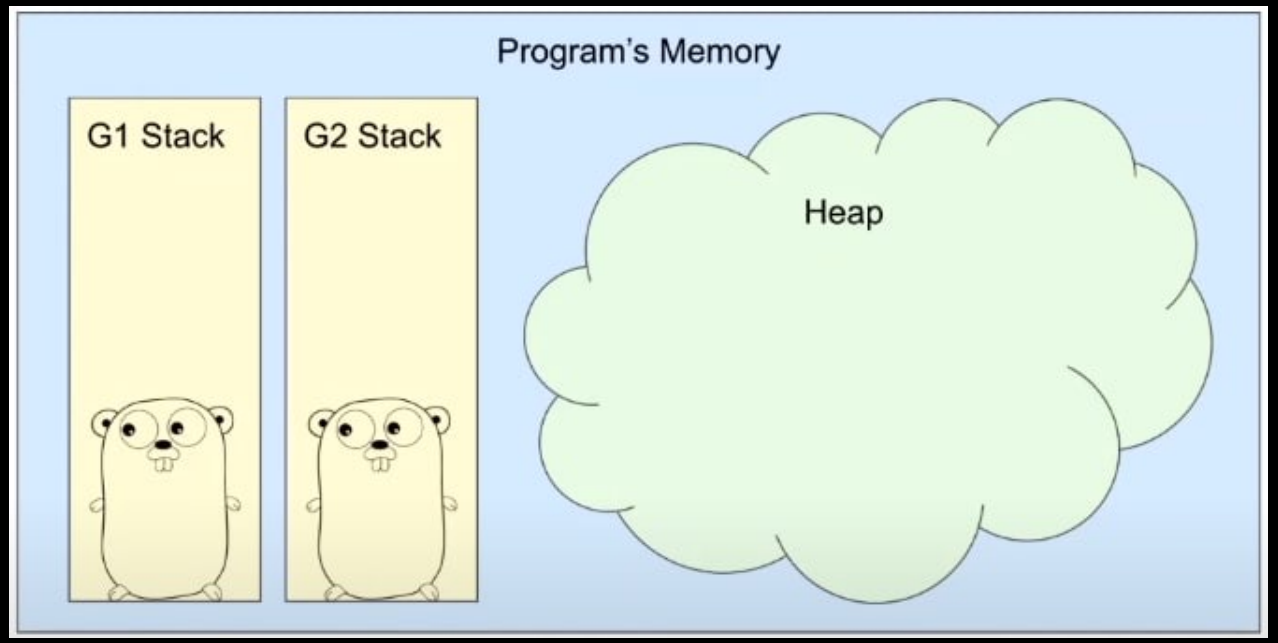
Heap

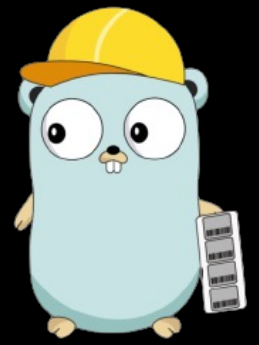
Stores dynamically allocated memory

Grow and shrink during the execution of a program

Slower than the stack = less efficient

Much larger capacity






@liamchampton

Stack vs Heap

Stack : short-lived data

Heap : long lived data

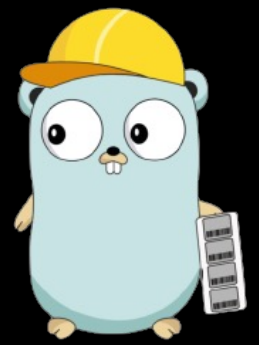
Go’s memory model


@liamchampton

Garbage collector

What is it?

Automatically attempts reclaim memory which was allocated by the program but is no longer referenced



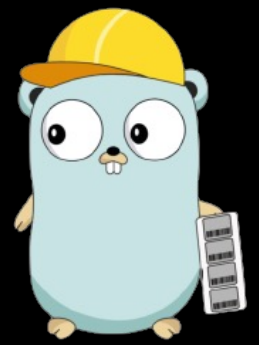

@liamchampton

Garbage collector

What is it?

Automatically attempts reclaim memory which was allocated by the program but is no longer referenced

No need to manually manage memory




@liamchampton

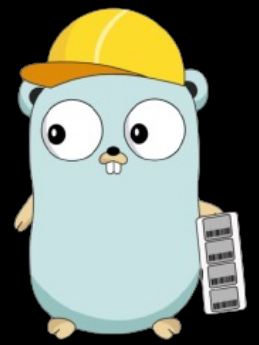
Garbage collector

What is it?

Automatically attempts reclaim memory which was allocated by the program but is no longer referenced

No need to manually manage memory

Reduces security and leak risks




@liamchampton

Goroutines & Channels

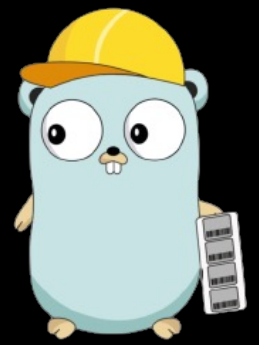
What is a goroutine?

A lightweight execution thread and a function that executes concurrently with the rest of the program

Very cheap with low overheads when compared to traditional threads

Syntax:

go foo()



@liamchampton

Goroutines & Channels

What is a goroutine?

A lightweight execution thread and a function that executes concurrently with the rest of the program

Very cheap with low overheads when compared to traditional threads

Syntax:

go foo()

What is a channel?

Channels are a built-in feature that allows go routines to communicate in a thread-safe manner

"Communication over channels" to synchronise access to shared memory between goroutines

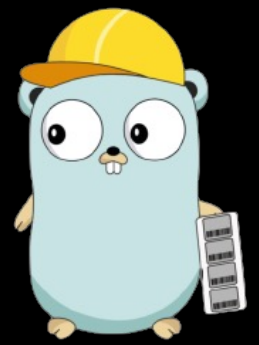
They prevent race conditions, locks and other synchronisation issues

Syntax:

chan keyword

Write: *c <- x*

Read: *<-c*



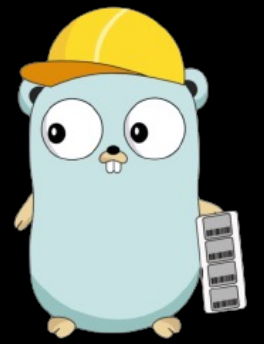
@liamchampton

Memory model summary

- Ensures the program does not run out of memory by utilising the garbage collector
- Allows goroutines to communicate safely

... Therefore, perfect to write / run concurrent and parallel code

Managing memory in Go




@liamchampton

Two ways you can help manage memory

The **“new”** function

- used to allocate memory for a variable of a given type

- It takes a type as an argument and returns a pointer to a newly allocated **zero value type**.

Example

ptr := new(int)

**ptr = 0*

Two ways you can help manage memory

The **“new”** function

- used to allocate memory for a variable of a given type

- It takes a type as an argument and returns a pointer to a newly allocated **zero value type**.

Example

ptr := new(int)

**ptr = 0*

The **“make”** function

- Used to allocate memory for data structures **(slices / maps / channels)**

- Initialises the memory to a useful default value, unlike the ”new” function.

Example:

slice := make([]int, 3, 5)

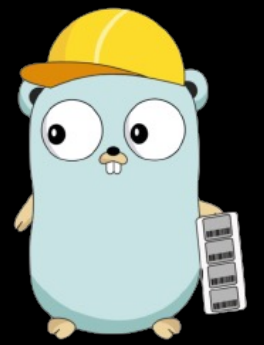
@liamchampton

Two ways you can help manage memory

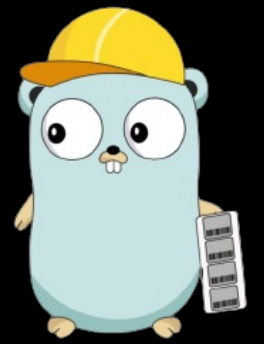
When to use them?

Use **”new”** to create a var and initialise it later

Use **“make”** when to create a data structure and use it right away



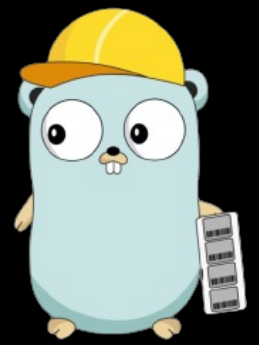
What is a memory leak?



Memory leak

What is it?

It is when memory is **no longer needed** but is also **not freed up** causing the program to eventually run out of memory / crash



Memory leak

What is it?

It is when memory is **no longer needed** but is also **not freed up** causing the program to eventually run out of memory / crash

Scenarios:

- Not properly terminating a goroutine, causing it to continue to hold on to the allocated memory
- Assigning a global variable and never using it again
- An infinite loop that creates objects and never releases them

Memory leak

What is it?

It is when memory is **no longer needed** but is also **not freed up** causing the program to eventually run out of memory / crash

Scenarios:

- Not properly terminating a goroutine, causing it to continue to hold on to the allocated memory
- Assigning a global variable and never using it again
- An infinite loop that creates objects and never releases them

Tools:

"pprof" – built-in package that can be used to profile and analyse the memory utilisation of a program

Memory leak

What is it?

It is when memory is **no longer needed** but is also **not freed up** causing the program to eventually run out of memory / crash

Scenarios:

- Not properly terminating a goroutine, causing it to continue to hold on to the allocated memory
- Assigning a global variable and never using it again
- An infinite loop that creates objects and never releases them

Tools:

"pprof" – built-in package that can be used to profile and analyse the memory utilisation of a program

How can YOU help?

- Be vigilant when using global variables and understand the code you are writing

- Use "defer" keyword to help reduce leaks with files, sockets and database connections

@liamchampton

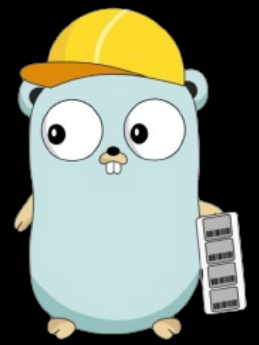
Code examples

Good example – defer

```
file, err := os.Open("file.txt") // open the file
if err != nil {
    log.Fatal(err)
}
defer file.Close()
```

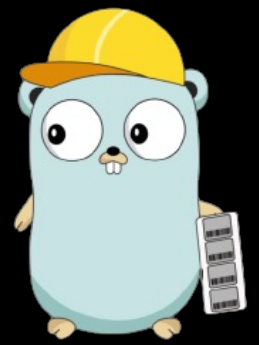
Good example – defer

```
file, err := os.Open("file.txt") // open the file
if err != nil { // check for an error
    log.Fatal(err)
}
defer file.Close()
```



Good example – defer

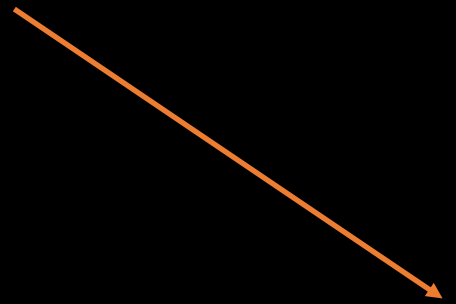
```
file, err := os.Open("file.txt") // open the file
if err != nil { // check for an error
    log.Fatal(err)
}
defer file.Close() // defer the closure
```




@liamchampton

Good example – defer

```
file, err := os.Open("file.txt") // open the file
if err != nil { // check for an error
    log.Fatal(err)
}
defer file.Close() // defer the closure
```



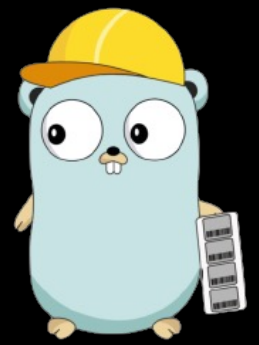
Schedules the file.Close() to execute after the surrounding function

File is closed even if the function errors!


@liamchampton

Good example – garbage collector

```
type MyStruct struct {
    data []byte
}

func main() {
    var myStruct MyStruct
    myStruct.data = make([]byte, 100000000)
}
```

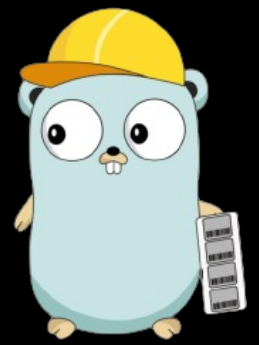


Good example – garbage collector

```
type MyStruct struct {
    data []byte
}

func main() {
    var myStruct MyStruct
    myStruct.data = make([]byte, 100000000)
}
```

Good example – garbage collector

```
type MyStruct struct {
    data []byte
}

func main() {
    var myStruct MyStruct
    myStruct.data = make([]byte, 100000000)
}
```

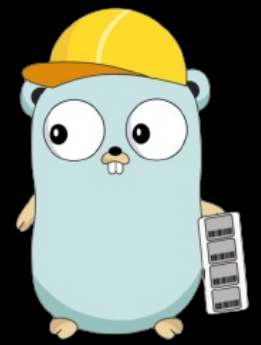


Good example – garbage collector

```
type MyStruct struct {
    data []byte
}

func main() {
    var myStruct MyStruct
    myStruct.data = make([]byte, 100000000)
}
```

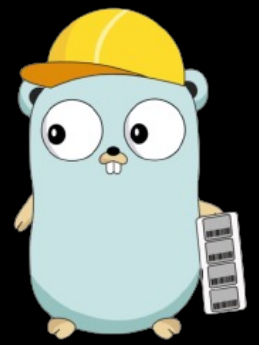
Once the function ends, the GC will reclaim the 100MB memory that was used by myStruct

@liamchampton



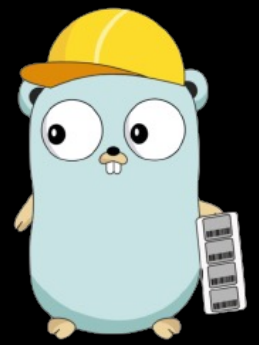
Bad example

```
var data []byte

func main() {
    data = make([]byte, 100000000)
    // Do some processing
    // ...
}
```



Bad example

```
var data []byte

func main() {
    data = make([]byte, 100000000)
    // Do some processing
    // ...
}
```



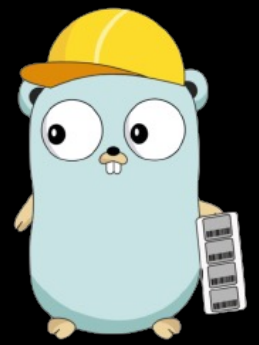

@liamchampton

Fixed

```
var data []byte

func main() {
    data = make([]byte, 100000000)
    // Do some processing
    // ...
}
```

```
func main() {
    data := make([]byte, 100000000)
    // Do some processing
    // ...
}
```

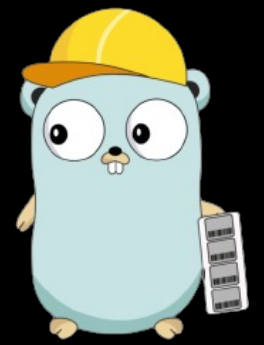
Give the ‘data’ variable a local scope so it will be cleaned when the function exits



Bad example

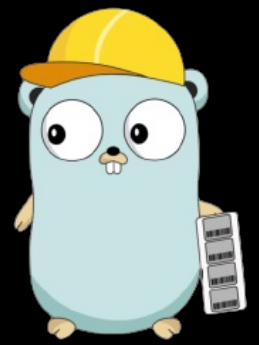
```
func recursion(n int) {
    if n == 0 {
        return
    }
    recursion(n-1)
}

func main() {
    recursion(1000000)
}
```

@liamchampton



Bad example

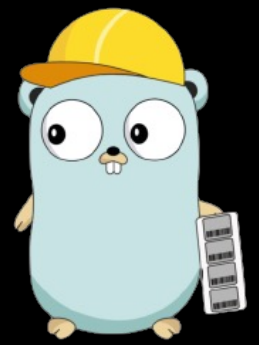
```
func recursion(n int) {
    if n == 0 {
        return
    }
    recursion(n-1)
}

func main() {
    recursion(1000000)
}
```



Bad example

```
func recursion(n int) {
    if n == 0 {
        return
    }
    recursion(n-1)
}

func main() {
    recursion(1000000)
}
```

That is a lot of loops! 😱

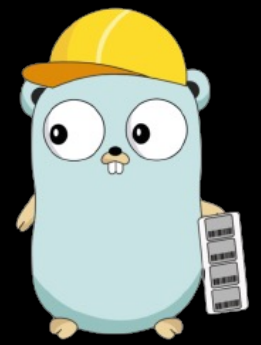


Fixed

```
func recursion(n int) {
    if n == 0 {
        return
    }
    recursion(n-1)
}

func main() {
    recursion(1000000)
}
```

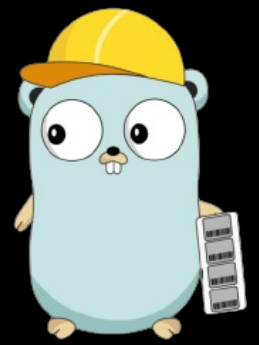
```
func recursion(n int) {
    if n == 0 {
        return
    }
    recursion(n-1)
}

func main() {
    recursion(1000)
}
```


@liamchampton



Go routines

https://go.dev/play/p/gwtTDGaLZ0g




@liamchampton

Channels

https://go.dev/play/p/Oj1A93xPA7t

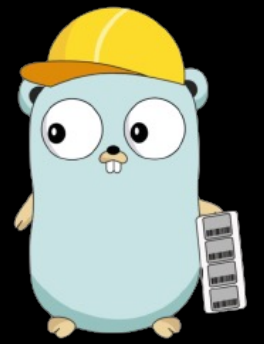

@liamchampton

Pointers / References

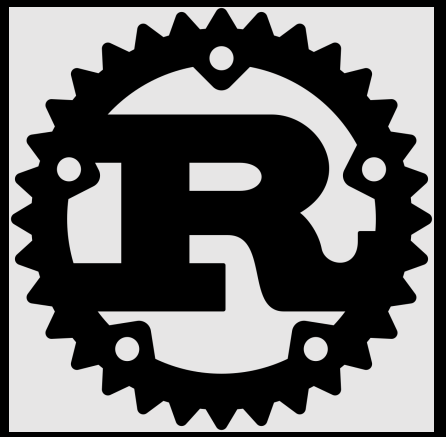
https://go.dev/play/p/q4r7sJSG4gX


@liamchampton

Memory management in other languages



Rust



Uses “ownership” and “borrowing” approach to model and manage memory

Every value has a singular variable that’s considered the “owner” and when the owner goes out of scope, the value it owns will be dropped – this prevents data races and undefined behaviours etc.

It is predominantly the developer's responsibility to allocate and deallocate memory usage

```
fn main() {
    let s = String::from("Hello");
    let len = calculate_length(&s);
    println!("The length of '{}' is {}.", s, len);
}

fn calculate_length(s: &String) -> usize {
    s.len()
}
// output : The length of 'Hello' is 5.
```

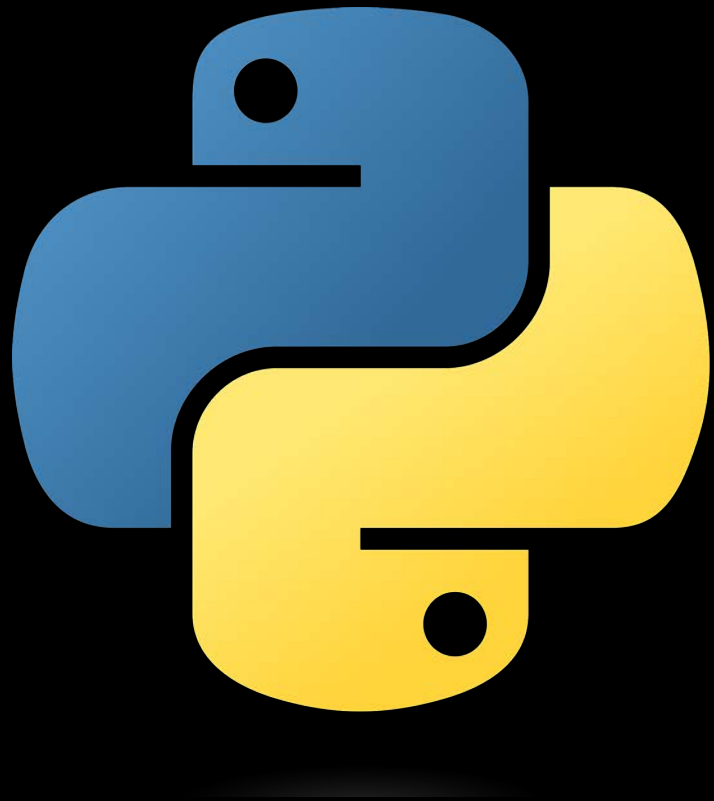
@liamchampton

Python

Uses a built-in garbage collector that uses a technique called **“reference counting”**

“Cyclic garbage collector” and periodically checks for unreachable objects and frees up their memory = a delay in object becoming unreachable and when its memory is freed up

Python has a memory manager used for allocation and deallocation of memory for large objects (arrays / lists etc)



Java

Similar to Go and uses both a stack and heap

Garbage collector manages the memory on the heap and uses a technique called **“mark and sweep”**

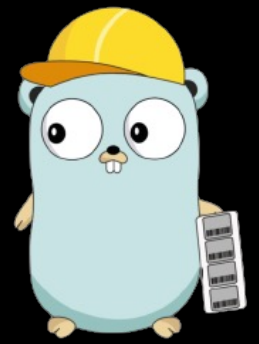
Built in memory manager allowing for explicit control of the memory usage

Tips for effective memory management

@liamchampton

Top tips...

1. Use the “defer” keyword



Top tips...

1. Use the “defer” keyword

2. Use the garbage collector wisely

@liamchampton

Top tips...

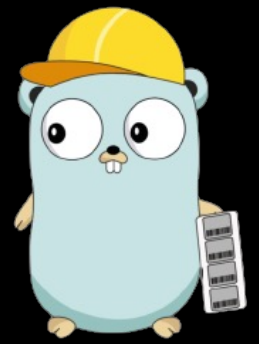
1. Use the “defer” keyword

2. Use the garbage collector wisely

3. Monitor memory utilisation


@liamchampton

Conclusion!




@liamchampton

Conclusion

Memory management is complicated!

Garbage collector handles the most part of it for you

Memory management is different across languages

Leaks are BAD!

@liamchampton

Thank You – Let’s Connect!

@liamchampton