

Go Get Them



What are we **GO**ing to talk about?

- Why use go?
- Function declaration in C and Go
- Unused imports and variables
- Concurrency
- Defer, Panic and Recover
- ; Semicolon
- ++/-- operator
- Compilation
- Function & method calls
- Goto statements
- Types
- OOP
- Inheritance
- Interface
- Duck typing
- Pointers
- Memory allocation
- New vs make
- Garbage collector
- Tidbits

Why use GO?

- Fast
- Simple
 - i.e. `while` , `foreach` and `for` loops are spelled '`for`' .
 - `for` loops have between 0 and 2 counters. no counters is a `while`, one counter is a `for`, two counters is a simultaneous `for` and `foreach`.
- Basically, a better C

Function declaration in C

- `returnType funcName (paramType paramName, ...){...}`
- i.e.
- `int foo (int* x){...}`

Function declaration in Go

- `func fN (pN1 [pN2...] pT , ...) (returnType no.1 ...){...}`
- Multiple return values!
- i.e.
- ```
func sumAtois(s t string) (bool , int){
 /** CODE **/
 return true, someInt
}
```
- P.S. c++14 recommended function declaration syntax is similar to GO's.  
Coincidence?  
`auto sunAtois(string s, string t) -> decltype({bool , int}) {...}`

# Let's get complex!

- What is the meaning in C++ ?
- `void (*(*f[]))()`

# Let's get complex!

- What is the meaning in C++ ?
- `void (*(*f[]))()`
- Defines f as an array of unspecified size of pointers to functions that return pointers to functions that return void

# Let's get complex!

- The equivalent in GO is

- $f \quad := \quad [][\text{func}()] \quad (\text{func}()) \quad \{\}$   
Implicit      2d      function      function      allocation  
definition   array   without   without  
and      arguments      arguments  
assignment      that returns      that returns  
The next      void  
token



# Unused import & variables

- GO won't compile your code if there is an unused import or variable
- This restriction accelerates build time
- A way around this restriction is to assign a variable to itself

```
3 func main(){
4 i := 5
5 i = i
6 }
```

```
1 package main
2
3 import "fmt"
4
5 func main(){
6 i := 5
7
8 }
```

Unused variable 'i' [more...](#) (Ctrl+F1 Alt+T)

# Concurrency

- Its always good for a programming language to have support for concurrency.
- In GO, running a function in parallel is as easy as adding the word 'go' before a function call. These functions are called routines, and are managed by GO's scheduler.
- GO also supports communication between routines using routes called channels.

# Defer, Panic and Recover

- Defer is a function call that will occur when the program is about to exit.
- If there are multiple defer calls, they will be executed in LIFO order.
- The main use of defer function is to catch exceptions and release the resources that were being used when the exception occurred.

# ; Semicolon

- “semicolons are for parsers, not for developers”
- There are no terminating semicolons (;) in GO (in some rare cases there are, but mostly not).
- In reality, semicolons are injected during the build phase to every line that can be the end of a statement.
- This forces the use of Java-style code formatting, but makes the code look more pleasant overall.

## ++/-- operator

- GO avoids the confusion caused by the ++ and -- operators by treating them like statements.
- In C, they were treated as expressions and so things like “a = a++ + ++a” were possible.
- In GO, ++ and -- are always postfix and don't return a value, so they have to appear on their own.
- This is much better than Python's approach, which is to abolish the ++ operator entirely.

# Compilation

- Thanks to its simple syntax, it's easy to build a compiler and an intelliSense (code-completion aid) for GO.
- GCC standardization.
- Optimization on assembly.

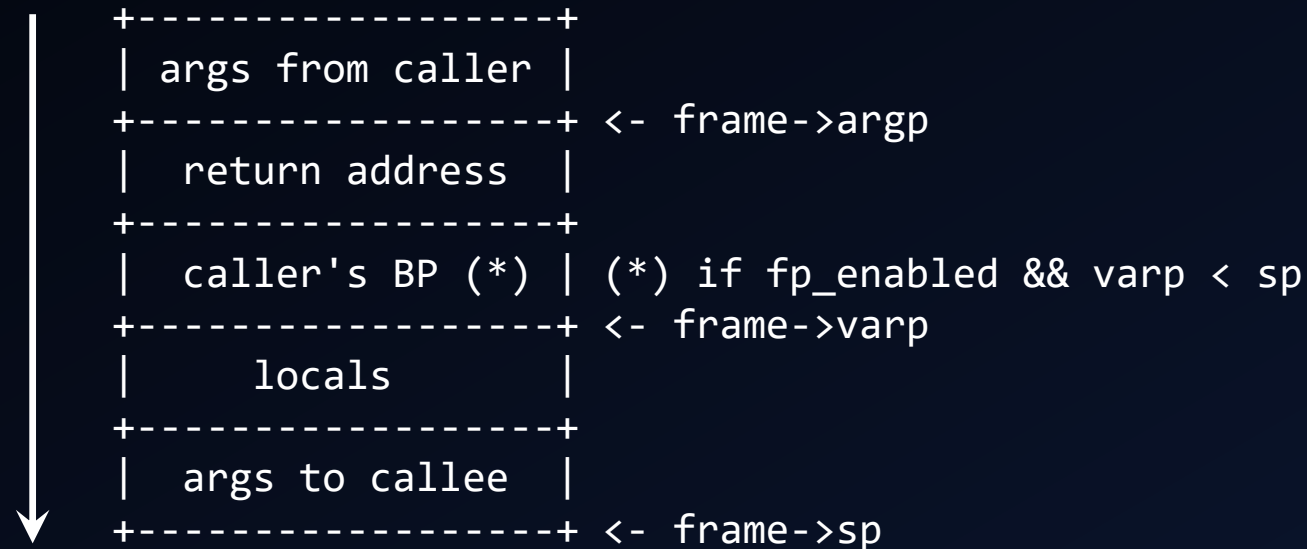
# Function & Method calls

- No default arguments.
- No function/operator overloading.
  - Ambiguous function calls can't happen because functions can have only one signature. Overloading isn't an option.
  - Why? Because the developers hate confusing code.
- No nested function declaration
  - Anonymous functions are allowed
- Positional arguments.
- Variadic support (unlimited passed arguments)
  - Watch me: <https://stackoverflow.com/a/35092570>
- There can be multiple arguments in both function calls and function returns.

# Function & Method calls

- Go implements stack-dynamic activation record instances, and its stack frame is built as follows:

source: src/runtime/stack.go  
Stack frame layout in x86 architecture





# Goto

- Goto statements are allowed, but only in the same scope

# Types

- Static & Strong.
- To avoid confusion and potentially disastrous mistakes, GO does not allow implied casting. all casting must be done explicitly.

# OOP

- GO is not object oriented.
- This makes the objects "feel" lighter, and also allows for more abstract code due to GO's interesting implementation of interfaces.

# Inheritance

- GO provides two features to replace class inheritance:
- Embedding, which can be viewed as an automated form of composition or delegation.
- Interfaces, which provide runtime polymorphism.

# Interface

- If a function returns an interface type, but that type was not instantiated, it will not be nil.
- The type will always have some kind of instance, even if it contains nothing. the only way to return nil from a function is to explicitly return nil.

# Duck Typing

- Go doesn't fully support duck typing
- Interfaces in GO take the form of "duck typing".
- If the object implements the functions of an interface, it can be referred to as that interface.
- This means that every data-type implements the blank interface: `interface{}`

# Pointers

- Pointers in GO are very similar to their C counterparts, but very different in several ways:
  - There is no pointer arithmetic. You can't change the address a pointer points to, except to assign a new address to it.
  - In C, pointers and arrays are essentially the same thing. In GO, arrays are values, and the lack of pointer arithmetic makes pointers to arrays very limited.
  - In C, often a null pointer is used to signify that something went wrong in a function. In GO, it is possible to return multiple values, so there is no need for a null pointer to signify errors.

# Memory Allocation

- Dynamic variables are allocated on the heap.
- Static variables are allocated on the stack.
  - If they aren't too big.
  - Big variables are allocated on the heap and are managed by the garbage collector.



# new vs make

- The `new(T)` function allocates zeroized storage for a new item of type `T` and returns its address, a value of type `*T`.
- In GO terminology, it returns a pointer to a newly allocated zero value of type `T`.
- The `make()` function, on the other hand, is a special built-in function that is used to initialize slices, maps, and channels.
- `make()` can only be used to initialize, and that, unlike the `new()` function, does not return a pointer.

# new vs make

```
1 var buf bytes.Buffer
2
3 // return a pointer to the value's address.
4 p := new(bytes.Buffer)
5
6 // Using make() to initialize a map.
7 m := make(map[string]bool, 0)
8 m := map[string]bool{}
```

# Garbage Collector

- GO's garbage collector uses Dijkstra's tri-color mark & sweep algorithm.
- The collector starts at the roots (globals) and works its way toward the leaves (last in the reference chain).
- A write barrier runs in parallel, and watches for new allocations that connect to pointers on the heap, in case they occur during the mark phase.
- Although this process is concurrent, there is a very short "Stop The World" phase to cover pointers on the stack.
  - usually under  $\mu 100$  seconds, often around  $\mu 10$  seconds.

# Tidbits

- Today, GO's compiler is written in GO.
  - The very first GO compiler was written in C.
- The developers hate functional-programming.
  - even “(cond)? s1 : s2”
- Support for imports directly from GitHub repositories
- Mostly, if-else-if... is translated to switch-case during compilation

# Questions?

