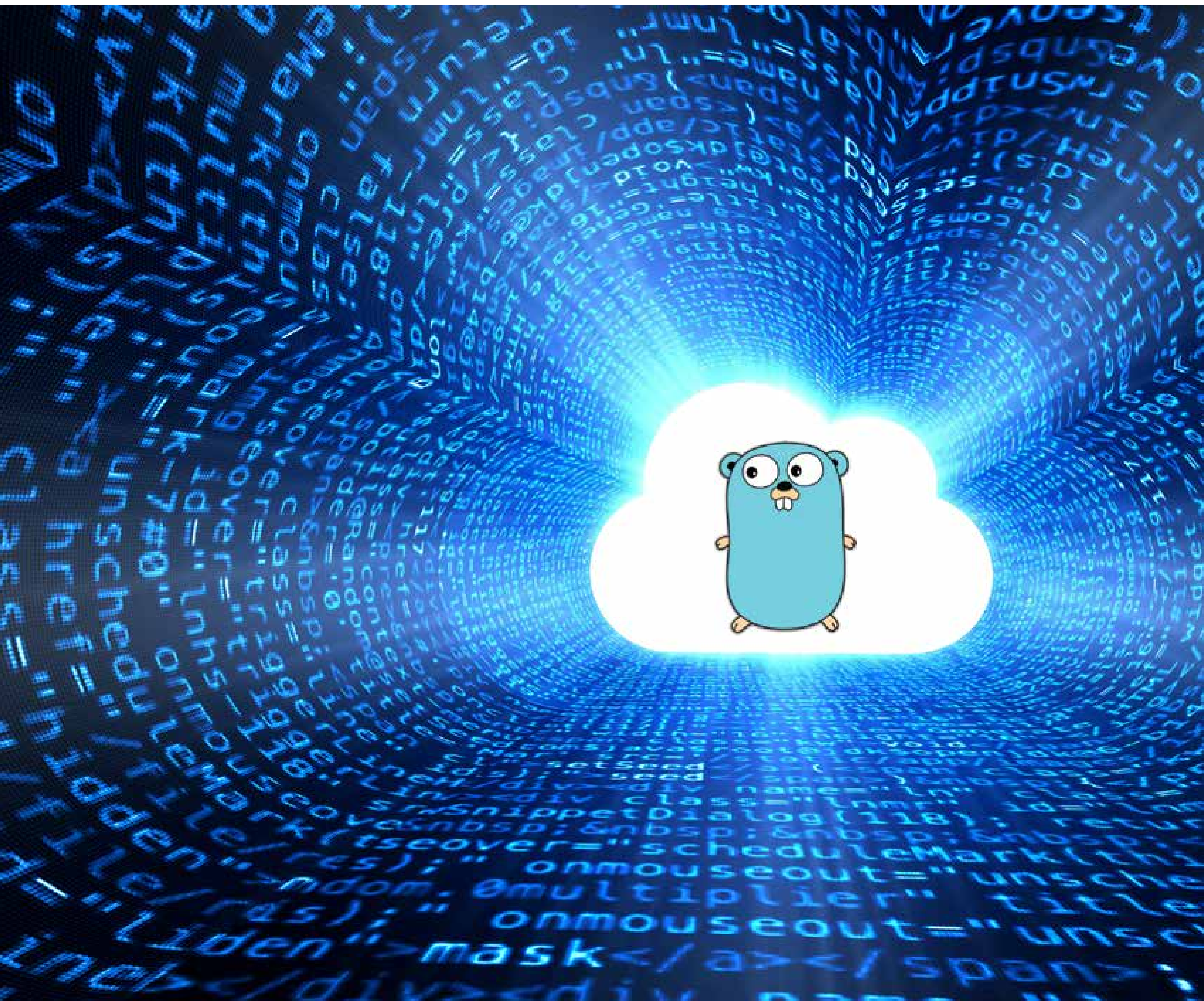
Mphasis
The Next Applied



Mphasis Digital POV

Use Go for System Programming, Distributed Systems and Cloud Workloads

PoV by

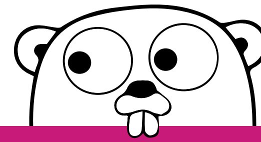
Aniruddha Chakrabarti

AVP Digital, Mphasis

Abstract

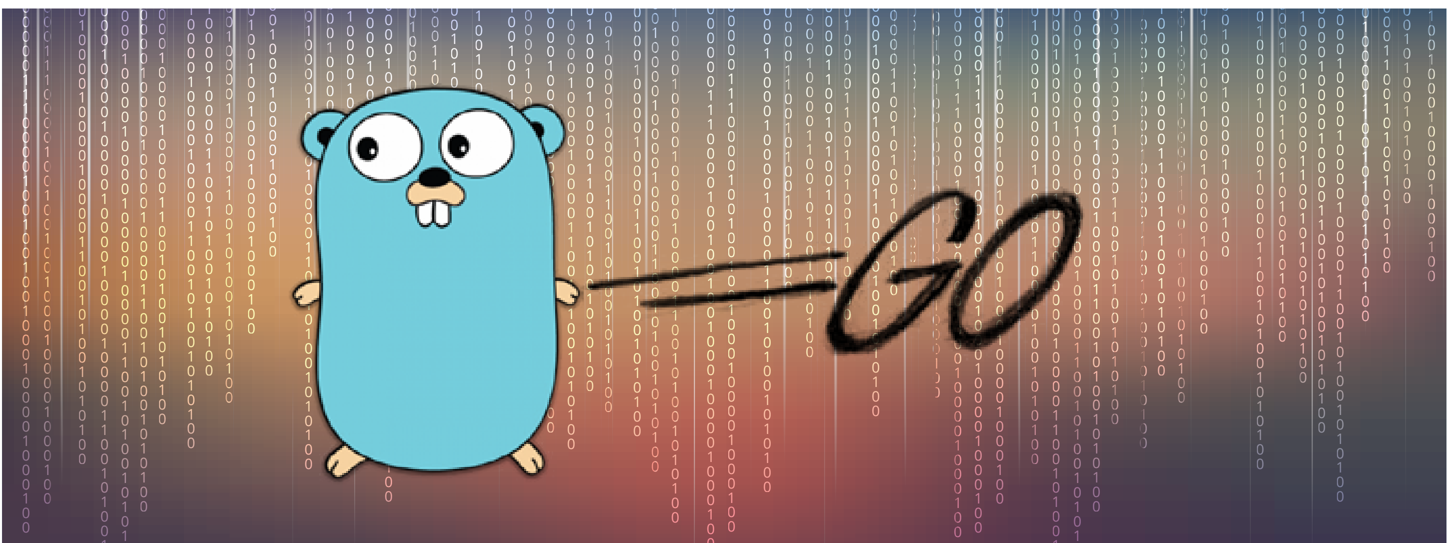
C and C++ are widely accepted and used as de-facto standard for systemprogramming language.In spite of the tremendous success and adoption of C and C++, there was always need for a more modern system programming language that would not only have performance characteristics of C and C++, but also the flexibility of other modern languages such as Python, Pascal, Modula and Smalltalk.

In recent past (since 2000) we have seen three moderately successful system programming languages being born – D (in 2001), Go (in 2009) and Rust (in 2012). Among them, while D is a close follower of C and C++, if not predecessor and hence the name D, Go takes a unique and differentiated approach than C and C++. We believe this is the strongest reason behind Go's success.



When you think about system programming, think Go - developed at Google for advanced technology demands

Go is also well suited for web development. There are multiple web development frameworks and middle tiers for Go that have received success in recent times, including Martini, Gin, Negroni, Revel, Beego, Gorilla and Net/HTTP. Go is also supported by multiple PaaS platforms including Google Cloud Platform (App Engine), Cloud Foundry (including Pivotal Web Service) and Heroku. AWS Elastic Beanstalk also supports deploying Go applications. We strongly believe that in future Go would come out as a strong alternate of C and C++ in the area of system programming, web development, large scale distributed systems and cloud workloads.



Overview of Go

Go or Golang is an open source programming language developed at Google, specially targeted at system programming and for building large scale distributed systems. It is natively compiled (does not need a VM to run), statically typed language following the footsteps of system programming languages such as C and C++. But Go is garbage collected, unlike C and C++.

Go is designed with the philosophy of "Less is exponentially more". In a blog post dated June 2012, Rob Pike, one of the designers of the language, reported how a language with less but well-designed features is far better than a language.

Go is not an object-oriented language. While Go support types, methods and interfaces, it does not support classes, inheritance, overloading of methods or operators.

History of Go

- Go language was originated as an outcome of an experiment by Robert Griesemer, Rob Pike and Ken Thompson at Google, to design a new system programming language in 2007
- Officially announced in November 2009, it is used in some of Google's production systems as well as by other firms
- Version 1.0 was released in March 2012 and between 2012 and 2015, multiple versions were released
- Version 1.5 was released in August 2015, which is the current stable version

New Programming Languages

Programming Language	Backed by / Developed at	First public release	Success	Paradigm	Suitable for
Go (also called golang)	Google	November 2009	High	Procedural - Designed from ground up with concurrency in mind. Supports Actor model based concurrency.	System programming, large scale distributed systems, cloud workloads, web back ends
D	Facebook	December 2001	Medium	Procedural, Object-oriented	System programming, large scale distributed systems, web back ends
Rust	Mozilla	2010	Medium	Procedural, Object-oriented with functional programming constructs.	System programming
R	Bell Laboratories	Late 90s	High	Object-oriented with functional and procedural constructs.	Statistical computing, Big Data analytics
Scala	Open Source	2004	High	Object oriented with influences functional programming Programming. Supports actor model based concurrency.	General purpose
Groovy	Apache Foundation	2004	Medium	Object oriented programming language with dynamic typing – designed for scripting in mind	General purpose, scripting focused
Clojure	Open Source	2007	Low	Functional – Is a dialect of Lisp	General purpose
Dart	Google	2011	Low	Object-oriented with scripting and functional programming constructs. Designed from group up for web development. Supports Actor model based concurrency.	Web development
TypeScript	Microsoft	2012	Medium	Scripting language with Object oriented constructs. Designed from group up for web development	Web development
F#	Microsoft	2005	Medium	Functional programming language that supports object oriented concepts	General purpose
Ceylon	Red Hat	2011	Low	Object oriented	General purpose
Swift	Apple	2014	High	Object oriented with influences from functional programming	General purpose

Other System Programming Languages

Language	Originator	Birth Date	Influenced by	Used for
ESPOL	Burroughs Corporation	1961	Algol 60	MCP
PL/I	IBM , SHARE	1964	Algol, FORTRAN , some COBOL	Multics
PL360	Niklaus Wirth	1968	Algol 60	Algol W
C	Dennis Ritchie	1969	BCPL	Unix
PL/S	IBM	196x	PL/I	OS/360
BLISS	Carnegie Mellon University	1970	Algol-PL/I(5)	VMS (portions)
PL/8	IBM	197x	PL/I	AIX
PL-6	Honeywell. Inc	197x	PL/I	CP-6
SYMPL	CDC	197x	JOVIAL	NOS subsystems, most complters, FSE editor
C++	Bjarne Stroustrup	1979	C. Simula	See C++ Applications[6]
Ada	Jean Ichbiah. S. Tucker Taft	1983	Algol 68, Pascal, C++, Java, Effel	Embedded systems, OS kernels, compilers, games, simulations, Cubesat, air traffic control, and avionics
D	Digital Mars	2001	C++	XomB
Go	Google	2009	C, Pasacal, CSP	Some Google systems.[7] Docker, Kubernetes, CoreOS[a]
Rust	Mozilla Research[8]	2012	C++, Haskell, Erlang, Ruby	Servo layput engine

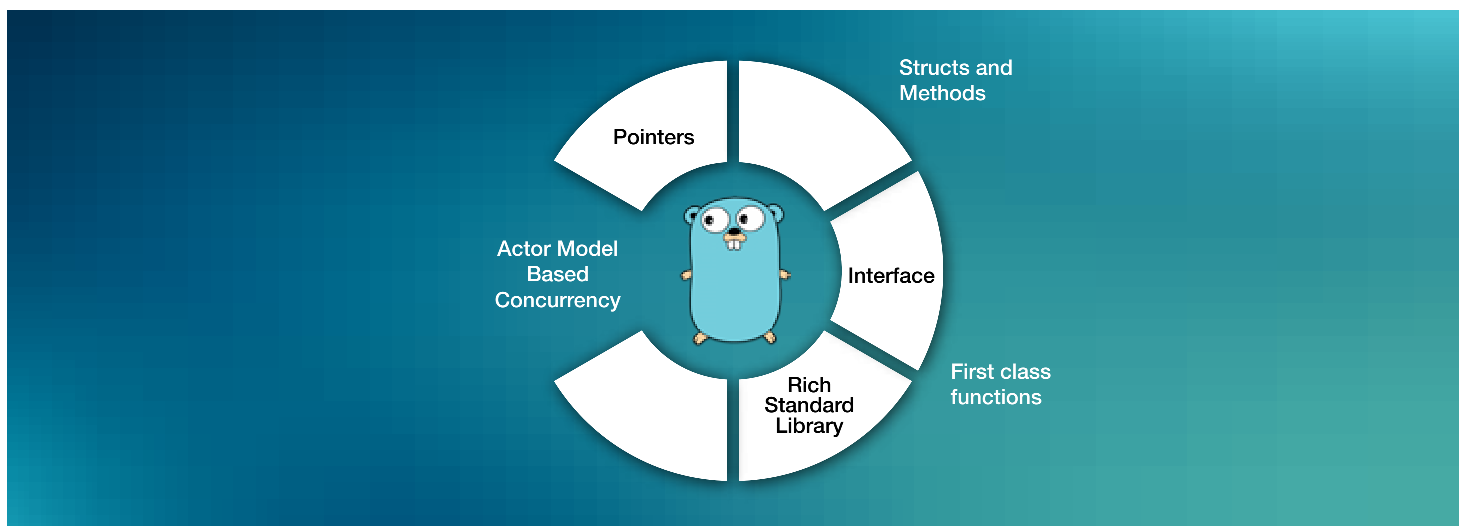
System Programming Languages Developed after 2000

Go Usage

- Many Google web properties and systems including YouTube, Kubernetes containers and download server dl.google.com
- Docker, a set of tools for deploying Linux containers
- Dropbox, migrated some of their critical components from Python to Go
- SoundCloud, for many of their systems
- Cloud Foundry, a platform as a service (PaaS)
- Couchbase, Query and Indexing services within the Couchbase Server
- MongoDB, tools for administering MongoDB instances
- ThoughtWorks, some tools and applications around continuous delivery and instant messages
- SendGrid, a transactional email delivery and management service
- The BBC, in some games and internal projects
- Novartis, for an internal inventory system

For a complete list of apps and systems that use Go refer-
https://en.wikipedia.org/wiki/Go_(programming_language)#Notable_users and *https://github.com/golang/go/wiki/GoUsers*

Features that made Go a success



First class functions

Go supports first class functions - Functions could be assigned to variables and passed as parameter to other functions. They could be even returned from other functions. Go supports anonymous functions, multiple return values from functions and all concepts supported by functional programming languages.

Anonymous Functions

Anonymous Functions are functions without name – they are typically assigned to variable.

```
func main(){
	add := func(x int, y int) int {return x+y}
	var sub = func(x int, y int) int {return x-y}

	fmt.Println(add(20,30))
	fmt.Println(sub(20,30))
}
```

Functions can be passed to other functions as parameters. In the below example higherOrderFunc is a higher order function that accepts another function (f func(x int, y int) int) as parameter -

```
func main(){
  add := func(x int, y int) int {return x+y}
  var sub = func(x int, y int) int {return x-y}

  fmt.Println(add(20,30))
  fmt.Println(sub(20,30))

  fmt.Println(higherOrderFunc(add, 30, 20))
  fmt.Println(higherOrderFunc(sub, 30, 20))
}

func higherOrderFunc(f func(x int, y int) int, num1 int, num2 int) int{
    return f(num1 * num1, num2 * num2)
}
```

Now the function definition for higherOrderFunc is a bit complicated and could get even more complicated if it accepts multiple functions as input parameter. This could be simplified by defining a user defined function type.

```
func main(){
    add := func(x int, y int) int {return x+y}
    var sub = func(x int, y int) int {return x-y}

    fmt.Println(add(20,30))
    fmt.Println(sub(20,30))

    fmt.Println(higherOrderFunc(add, 30, 20))
    fmt.Println(higherOrderFunc(sub, 30, 20))
}

type HigherFunc func(x int, y int) int                // user defined function type

func higherOrderFunc(f HigherFunc, num1 int, num2 int) int{
    return f(num1 * num1, num2 * num2)
}
```

Similarly a higher order function could return another function –

```
func main(){
  var result = higherOrderFunc2()
  fmt.Println(result(30, 20))                // 50
}

func higherOrderFunc() func(x int, y int)int {
  add := func(x int, y int) int {return x+y}
  return add
}
```

Go can also return multiple values from a function –

```
func swap(x, y int) (int, int){
  return y, x
}

var x,y int = 10,20
fmt.Println(x,y)                // 10 20

var p,q int = swap(x,y)
fmt.Println(p,q)                // 20 10
```

Structs and Methods

Structs

Go does not support Class, but supports Struct. Structs in Go are typed collections of named fields similar to other programming languages like C++ and Java. Structs can have methods apart from fields. Struct allows grouping of fields (data) and methods (behavior).

```
type Employee struct {
    name            string          // field name of type string
    age             int             // field age of type int
    salary          float32         // field salary of type float32
    designation     string          // field designation of type string
}
```

Structs could be initialized either using new function (Option 1) or using JavaScript like object literal notation (Option 2)

```
// Option 1 – using new function
emp := new(Employee)
emp.name = "Ken Thompson"
emp.age = 50
emp.salary = 12345.678
emp.designation = "Distinguished Engineer"
```

```
// Option 2 – using JavaScript like object literal notation
emp := Employee{}
emp.name = "Ken Thompson"
emp.age = 50
emp.salary = 12345.678
emp.designation = "Distinguished Engineer"
```

Another alternate is to specify field values as parameters in the same order fields are declared

```
// Option 3 – parameters should be in the same order fields are declared
emp := Employee{"Ken Thompson", 50, 12345.678, "Distinguished Engineer"}
```

Note:
; are optional in Go similar to JavaScript.
Local variables could be assigned without specifying their types using := instead of =

Structs can have arrays and other child structs as fields

```
type Employee struct{
    Name            string
    Age             int
    Salary          float32
    Skills          [4]string           // Array field
    HomeAddress     Address                     // Nested Child struct as property

}

type Address struct{
    StreetAddress string
    City string
    Country string
}

func main(){
    address := Address{"MG Road", "Blore", "IN"}
    skills := [4]string {"C","Go","Rwust"}
    emp := Employee{"Aniruddha", 40, 123.456, skills, address}
    fmt.Println(emp)                // {Aniruddha 40 123.456 [C Go Rust] {MG Road Blore IN}}
    fmt.Println(emp.Skills)                                         // [C Go Rust]
    fmt.Println(emp.HomeAddress.StreetAddress)                      // M G Road
}
```

Methods

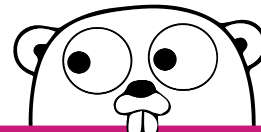
Apart from member fields, Go structs support defining methods (member functions). Methods of the struct are actually defined outside the struct declaration, which is unique in Go.

```
type Employee struct {
	name		string
	age		int
	designation	string
}

// Displayis declaredas a method for Employeestruct
func (emp Employee) Display() {
	fmt.Println("Name:", emp.name, ", Designation:", emp.designation)
}

func main(){
	emp := Employee{name:"Bill G", age:55, designation:"President"}
	emp.Display()			// prints Name: Bill G , Designation: President
}
```

Methods can accept parameter similar to normal functions –

```
type Employee struct {
	name		string
	age		int
	designation	string
}

// Method for struct Employee that accepts a string input
func (emp Employee) Display(message string) {
	fmt.Println(message, emp.name, ", Designation:", emp.designation)
}

func main(){
	emp := Employee{name:"Bill G", age:55, designation:"President"}
	emp.Display("Hello")
}
```

Interfaces



Go simplifies the process and reduces a lot of effort from programming perspective.

Go support interfaces, just like C++, Java and C# do. Interfaces are named collections of method signatures. They could be treated as blueprint as they do not provide implementation. Types that implement an interface is responsible for providing the actual implementation. Go is unique in the fact that types do not have to explicitly use “implements” or other similar declarations to specify that they are implementing an interface. To implement an interface, all that types have to do is provide implementation of methods defined in interface.

```
// An interface called Human is declared with two methods – Walk and Talk
type Human interface {
	Walk(miles int)
	Talk()
}

// A type called Employee is declared which would implement the interface Human
// Note that the type does not specify that it would implement the interface
type Employee struct {
	name		string
	designation	string
}
```

```
// Implementation of methods of Human interface
func (emp Employee) Walk(miles int){
	fmt.Println("I can walk", miles, "miles")
}

// Implementation of methods of Human interface
func (emp Employee) Talk(){
	fmt.Println("I can talk. My name is", emp.name, "and I am a", emp.designation)
}

func main(){
	var emp Human = Employee{name:"Rob Pike", designation:"Architect"}
	emp.Walk(10)			// I can walk 10 miles
	emp.Talk()			// I can talk. My name is Rob Pike and I am a Architect
}
```

The type implementing the interface has to implement all the methods specified in the interface. Also, all the method signatures should match. If both of these conditions are not matched then compiler complains -

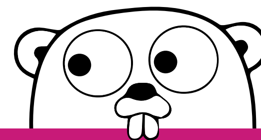
```
type Contractor struct {
	name string
}

func (cont Contractor) Walk(miles string){
	fmt.Println("I can walk", miles, "miles")
}
func main(){
	var cont Human = Contractor{name:"John Smith"}
	cont.Walk("7")
}
```

Go compiler throws the following error for the above example –

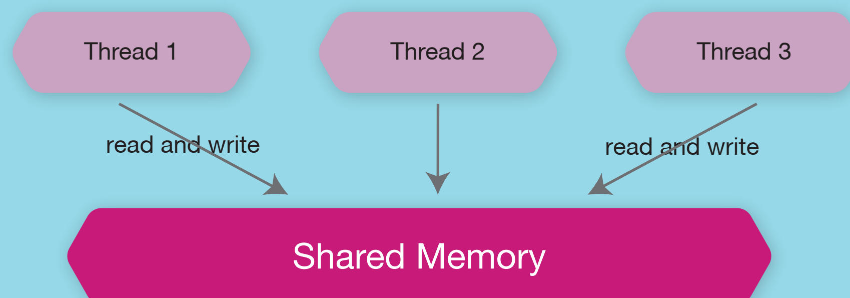
```
cannot use Contractor literal (type Contractor) as type Human in assignment:
		Contractor does not implement Human (missing Talk method)
cannot use "7" (type string) as type int in argument to cont.Walk
```

Concurrency (Goroutines and Channels)

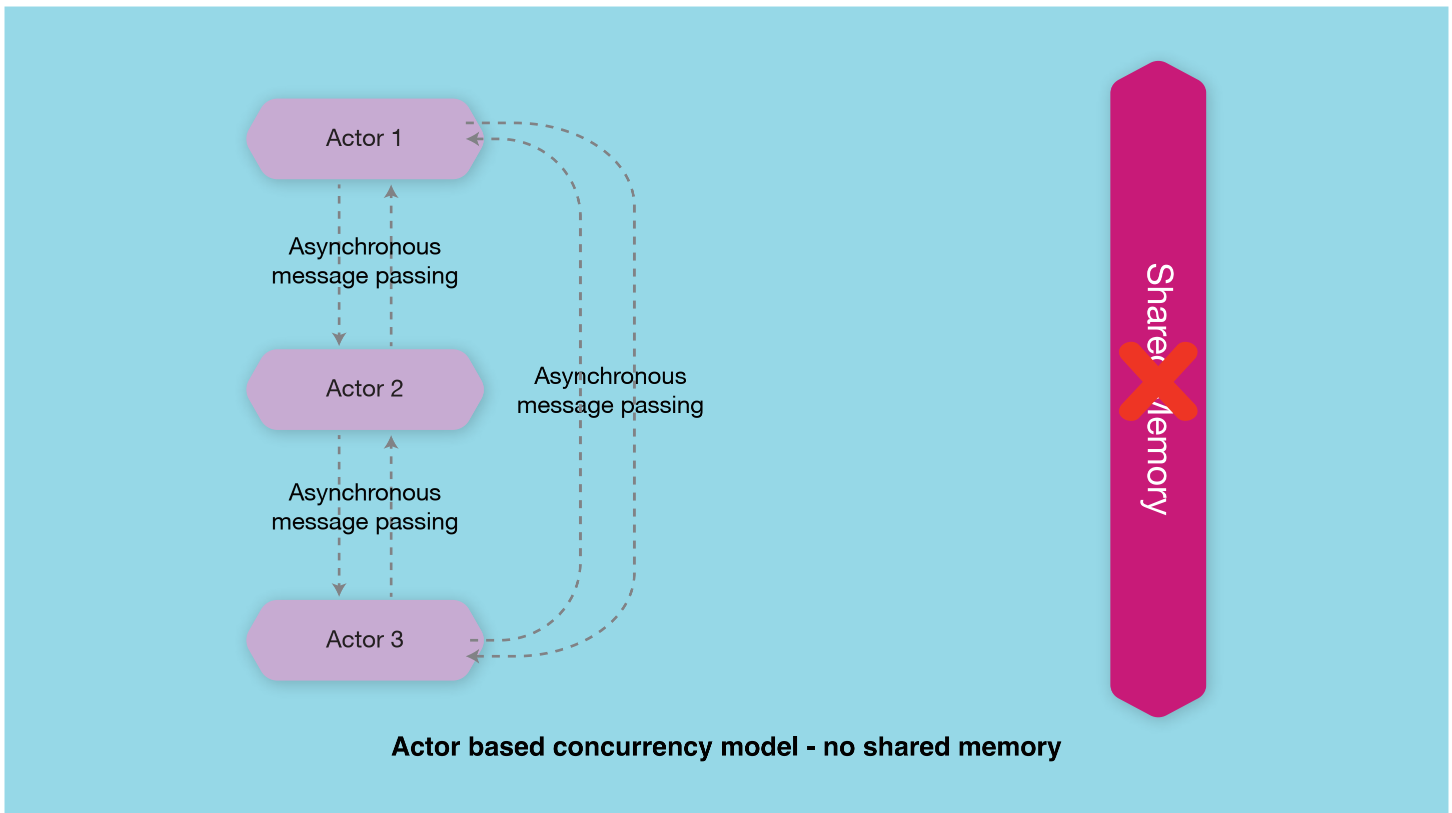
Go has been designed from ground up, keeping concurrency and parallelism in mind. Go's concurrency is based on a concept called Communicating Sequential Processes (CSP) that was introduced by C.A.R. Hoare in a seminal paper in 1978. CSP is similar to the Actor model of currency, which was made popular by Erlang. CSP takes a different approach to concurrency, compared to thread / lock based concurrency. While in thread based concurrency, multiple threads use shared memory to communicate among themselves, in CSP, concurrent processes or actor or routines use message passing to communicate. They do not use shared memory thus avoiding synchronization and synchronization primitive (lock, mutex, monitor, semaphore etc) related issue.



Parallelism and concurrency are the key elements in Go design



Thread based concurrency model with shared memory

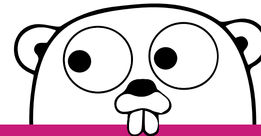


Actor based concurrency model - no shared memory

Do not communicate by sharing memory; instead, share memory by communicating – GoLang website

Goroutine

Go uses goroutines for concurrency. A goroutine is a function that executes concurrently with other goroutines in the same address space; it's quite lightweight as compared to OS threads and is managed by Go runtime. Multiple goroutines could be executed concurrently thus bringing in concurrency.



Goroutines make Go easy to use and help it keep concurrency intact.

Any function could be executed as a goroutine by prefixing it with go keyword –

```
func main() {
	// add function is called as goroutine
	go add(20, 10)

	fmt.Println("add executed")
	fmt.Scanln()
}

func add(x int, y int){
	fmt.Println(x+y)
}
```

Channel

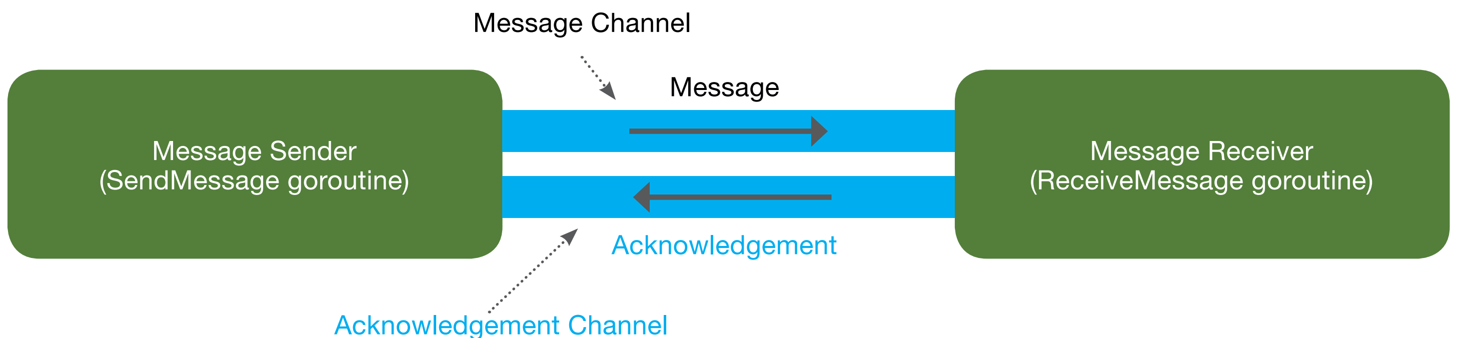
Channels are the conduit through which messages could be sent and received. For sending and receiving messages using channel <- operator is used.

Channels could be created using make function. While creating the channel the type of message to be passed using the channel is specified –

```
// Declare and allocate a channel
channel := make(chan string)

// Send a message across the channel
channel <- "message"

// Receive message from the channel
msg:= <- channel
```

The below example creates two goroutines – one of them behaves as message sender and the other is message receiver. The message sender then sends the message via a channel and the message receiver receives the message.

```
func main(){
	channel := make(chan string)

go SendMessage(channel)
go ReceiveMessage(channel)

	fmt.Scanln();
}

// goroutine that sends the message
func SendMessage(channel chan string) {
	for {
	channel <- "sending message @" + time.Now().String()
		time.Sleep(2 * time.Second)
	}
}

// goroutine that receives the message
func ReceiveMessage(channel chan string) {
   for {
	message := <- channel
		fmt.Println(message)
   }
}
```



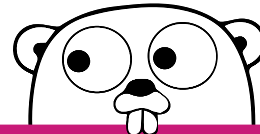

```
func main() {
	// Channel used to send the message from sender to receiver
	msgChannel := make(chan string)
	// Channel used to acknowledge the message receipt by receiver
	ackChannel := make(chan string)

	go SendMessage(msgChannel, ackChannel)
	go ReceiveMessage(msgChannel, ackChannel)

	fmt.Scanln()
}
```

```
// goroutine that sends the message
func SendMessage(msgChannel chan string, ackChannel chan string) {
	for {
		// Send the message to message channel
		msgChannel <- "Sending message @" + time.Now().String()
		time.Sleep(2 * time.Second)

// Receive the acknowledgement from acknowledgement channel
		ack := <-ackChannel
		fmt.Println(ack)
	}
}

// goroutine that receives the message
func ReceiveMessage(msgChannel chan string, ackChannel chan string) {
	for {
		// Receive the message from message channel
		message := <-msgChannel
		fmt.Println(message)

		// Send the acknowledgement to acknowledge channel
		ackChannel <- "Message received @" + time.Now().String()
	}
}
```

Pointers

Unlike some of the new programming languages such as Java, C#, Ruby, Python and Swift, Go supports pointers. Pointers in Go behave much the same way as C and C++. Pointers reference a location in memory where a value is stored rather than the value itself.

A pointer is represented using the * (asterisk) character followed by the type of the stored value. The & (ampersand) operator is used to denote the address of a variable. In the below example, a pointer is declared that points to an existing string variable (city). The address of the pointer and the value it's pointing to is printed.



Go is one of the most advanced programming languages for a reason - it supports the use of pointers.

```
var city string = "Bangalore"

// Declare a pointer that points to the address of city variable
ptrToCity := &city

// Print the address of the pointer and value the pointer is pointing to
fmt.Println("&ptrToCity = ", &ptrToCity);		// prints the address of the pointer
fmt.Println("*ptrToCity = ", *ptrToCity);// prints the value it's pointing to
```

Changing the value of the pointer changes the value of the original value it is pointing to -

```
var city string = "Bangalore"
var ctr int = 10

// Declare two pointers pointing to city and ctr variables
ptrToCity := &city
ptrToCtr := &ctr

fmt.Println("city = ", city, ", *ptrToCity = ", *ptrToCity);
fmt.Println("ctr = ", ctr, ", *ptrToCtr = ", *ptrToCtr);
```

```
// Change the values of the pointers
*ptrToCity = "Kolkata"
*ptrToCtr = 20

// The original variables to which pointers are pointing is also changed
fmt.Println("city = ", city, ", *ptrToCity = ", *ptrToCity);
fmt.Println("ctr = ", ctr, ", *ptrToCtr = ", *ptrToCtr);
```

Rich standard library

Go comes with a large no of packages that provide common functionality such as File handling, IO, String handling, Cryptography etc. Below is a list of popularly used Go packages -

Sl. No	Description	Package Name
1	String manipulation	Strings
2	Input and Output	io, bytes
3	Files and Folders	os, path/filepath
4	Errors	Errors
5	Containers & Sort	container/list
6	Hashes and Cryptography	hash, crypto
7	Encoding	encoding/sob
8	Allows interacting with Go's runtime system, such as functions to control goroutines	Runtime
9	Synchronization Primitives	Sync
10	Server communication, RPC, HTTP, SMTP etc.	net, http/rpc/jsonrpc
11	Math library	Math
12	Zip, Archive, Compress	archive, compress
13	Database related	database, sql
14	Debugging	Debug
15	Automated testing	Testing

Other highlights of Go

1. Go is statically typed
2. Natively compiled, but garbage collected
3. Go is statically typed with optional type inference
4. Supports Defer, Panic and Recover for error handling
5. Does not support classes and inheritance
6. Does not support overloading of methods and operators
7. Go compiler compiles source code very fast.
8. Does not support Generics

Conclusion

Go has evolved as a major programming language suitable for not only system programming, but also for cloud workloads. Go is supported by multiple cloud platforms, including Google Cloud Platform (App Engine), Cloud Foundry (including Pivotal Web Service), Heroku and AWS Elastic Beanstalk. Microsoft also added "experimental support" for Go into its Azure cloud service. Go is becoming a popular language of choice for system programming, web development, large scale distributed systems and cloud workloads.

Further Reading

- Less is exponentially more: A blog by Rob Pike - http://commandcenter.blogspot.de/2012/06/less-is-exponentially-more.html
- Go website - https://golang.org
- Go FAQ - https://golang.org/doc/faq#Is_Go_an_object-oriented_language
- Go By Example - https://gobyexample.com/
- An Introduction to Programming in Go - https://www.golang-book.com/books/intro
- Little Go Book - http://openmymind.net/assets/go/go.pdf
- Effective Go - https://golang.org/doc/effective_go.html
- Communicating Sequential Processes - https://en.wikipedia.org/wiki/Communicating_sequential_processes

Copyright ©: The brand names mentioned in this POV (Go, D, Rust, C, C++, AWS, Heroku, Google Cloud Platform, Cloud Foundry, Pivotal Web Service, Azure etc.) belong to their respective owners.

Aniruddha Chakrabarti

Associate Vice President, Digital, Mphasis

Aniruddha has 16+ years of IT experience spread across systems integration, technology consulting, IT outsourcing and product development. He has extensive experience of delivery leadership, solution architecture, presales, technology architecture and program management of large scale distributed systems.

As AVP, Digital in Mphasis, Aniruddha is responsible for Presales, Solutions, RFP/RFI and Capability Development of Digital Practice. Before as Sr. Manager & Sr. Principal Architect in Accenture, he was responsible for presales, architecture and leading large delivery teams. He had played delivery leadership and architecture focused roles in Microsoft, Target, Misys and Cognizant.

His interests include digital, cloud, mobility, IoT, cognitive computing, NLP, distributed systems, web, open source, .NET, Java, programming languages and NoSQL. His industry experience spans Retail, Healthcare, Capital Markets, Insurance, Travel, Hospitality, Pharma and Medical Technology.

About Mphasis

Mphasis (BSE: 526299; NSE: MPHASIS) applies next-generation technology to help enterprises transform businesses globally. Customer centricity is foundational to Mphasis and is reflected in the Mphasis' Front2Back™ Transformation approach. Front2Back™ uses the exponential power of cloud and cognitive to provide hyper-personalized ($C = X2C^{2}_{TM} = 1$) digital experience to clients and their end customers. Mphasis' Service Transformation approach helps 'shrink the core' through the application of digital technologies across legacy environments within an enterprise, enabling businesses to stay ahead in a changing world. Mphasis' core reference architectures and tools, speed and innovation with domain expertise and specialization are key to building strong relationships with marquee clients. To know more, please visit www.mphasis.com

For more information, contact: marketinginfo@mphasis.com

USA
460 Park Avenue South
Suite #1101
New York, NY 10016, USA
Tel.: +1 212 686 6655
Fax: +1 212 683 1690

UK
88 Wood Street
London EC2V 7RS, UK
Tel.: +44 20 8528 1000
Fax: +44 20 8528 1001

INDIA
Bagmane World Technology Center
Marathahalli Ring Road
Doddanakundhi Village
Mahadevapura
Bangalore 560 048, India
Tel.: +91 80 3352 5000
Fax: +91 80 6695 9942

VAL 18/01/19 US Letter BASIL 4020

Copyright © Mphasis Corporation. All rights reserved.

www.mphasis.com