The Go Programming Language

Mooly Sagiv

Based on a presentation by

Rob Pike(Google)

Content

- What is wrong with C
- Go’s goals
- History and status
- A tour of Go
- Critiqe

The C Programming Language

- Originally developed by Dennis Ritchie 1969-73 at Bell Labs
- Used for implementing Unix
- Became the standard system programming language
- Keys to success:
  - Simplicity and elegance
  - Availability
  - Performance
    - No more manual assembly code
  - Documentation Brian Kernighan & Dennis Ritchie
- Extended to C++

Object Orientation in C

```
class Vehicle extends object {
  int position = 10;
  void move(int x)
   {
     position = position + x ;
   }
```

```
struct  Vehicle {
  int position ;
       }
  void New_V(struct Vehicle *this)
  {
     this→position = 10;
  }
  void move_V(struct Vehicle *this, int x)
 {
    this→position=this→position + x;
  }
```

What is wrong with C?

- Type safety
  - Pointer arithmetic, casts, unions, no bound checking, free
- Ugly syntax
  - Mainly for historical reasons
  - Influenced Java
- Unpredicted side-effects
- Low level control constructs (break, goto, continue)
- Lacks support for modularity, concurrency and dynamic environments
  - typedefs and #include are just macros

Type Safety (1)

```
int *x = (*int ) 0x 7777;
int y = *x ;
```

Type Safety (2)

```
union u { int i;
          int * p;
    }
u.i =  0x 7777;
int y = u.p ;
```

Type Safety (3)

```
int x;
int a[2];
x= 5;
a[2] = 7;
printf(“%d”, x) ;
```

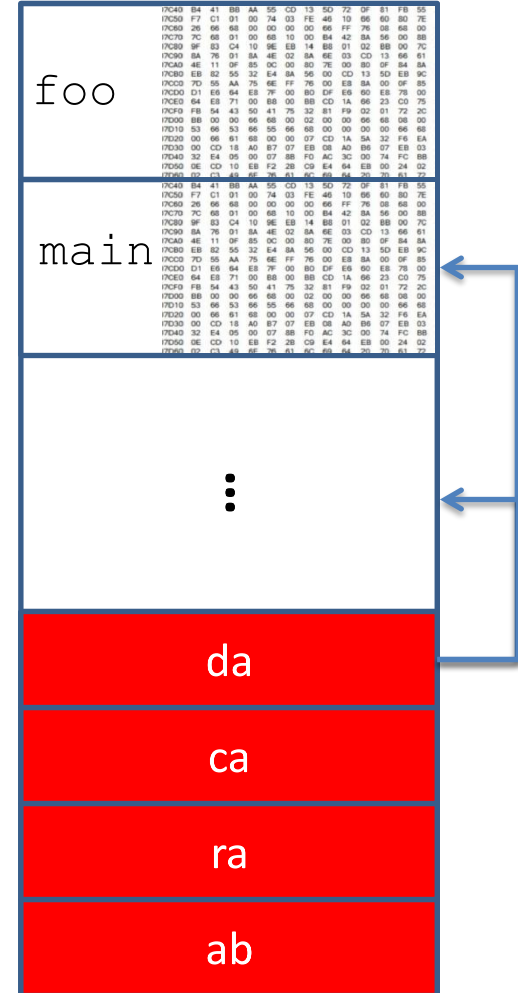
Buffer Overrun Exploits

```
void foo (char *x) {
  char buf[2];
  strcpy(buf, x);
}
int main (int argc, char *argv[]) {
  foo(argv[1]);
}
```

source code

```
> ./a.out abracadabra
Segmentation fault
```

terminal



memory

Pointer Errors (1)

```
a =  malloc(…) ;
b = a;
free (a);
c = malloc (…);
if  (b == c)  printf(“unexpected equality”);
```

Pointer Errors (2)

```
char* ptr = malloc(sizeof(char));
 *ptr = 'a';
free(ptr);
free(ptr);
```

Pointer Errors (3)

```
int* foo() {
  int x = 5 ;
  return &x;
}
```

String Errors

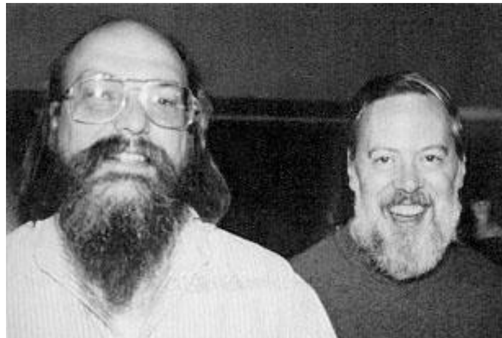
```
char c[8],a[7] ;
c[0] = ‘a’ ;
c[1] = ‘b’;
c[2] = ‘a’ ;
strcpy(a, c) ;
```

The Go Programming Language

Robert Griesemer

Ken Thompson

Rob Pike



Javascrtipt V8, Chubby, ETH

B, Unix, Regexp, ed, Plan 9, Berkeley,Bell

Unix, Plan 9, Bell

Hello, world

```
package main

import "fmt“

func main() {
fmt.Printf("Hello, 世界\n");
}
```

Who

- Robert Griesemer, Ken Thompson, and Rob Pike started the project in late 2007
- By mid 2008 the language was mostly designed and the implementation (compiler, runtime) starting to work
- Ian Lance Taylor and Russ Cox joined in 2008
- Lots of help from many others

Why

- Go fast!
- Make programming fun again

Our changing world

- No new major systems language since C
- But much has changed
  - sprawling libraries & dependency chains
  - dominance of networking
  - client/server focus
  - massive clusters
  - the rise of multi-core CPUs
- Major systems languages were not designed with all these factors in mind

Construction speed

- It takes too long to build software
- The tools are slow and are getting slower
- Dependencies are uncontrolled
- Machines have stopped getting faster
- Yet software still grows and grows
- If we stay as we are, before long software construction will be unbearably slow

Type system tyranny

- Robert Griesemer: “Clumsy type systems drive people to dynamically typed languages”
- Clunky typing: Taints good idea with bad implementation
- Makes programming harder
  - think of C's **const:** well-intentioned but awkward in practice
- Hierarchy is too stringent:
  - Types in large programs do not easily fall into hierarchies
- Programmers spend too much time deciding tree structure and rearranging inheritance
- You can be productive or safe, not both

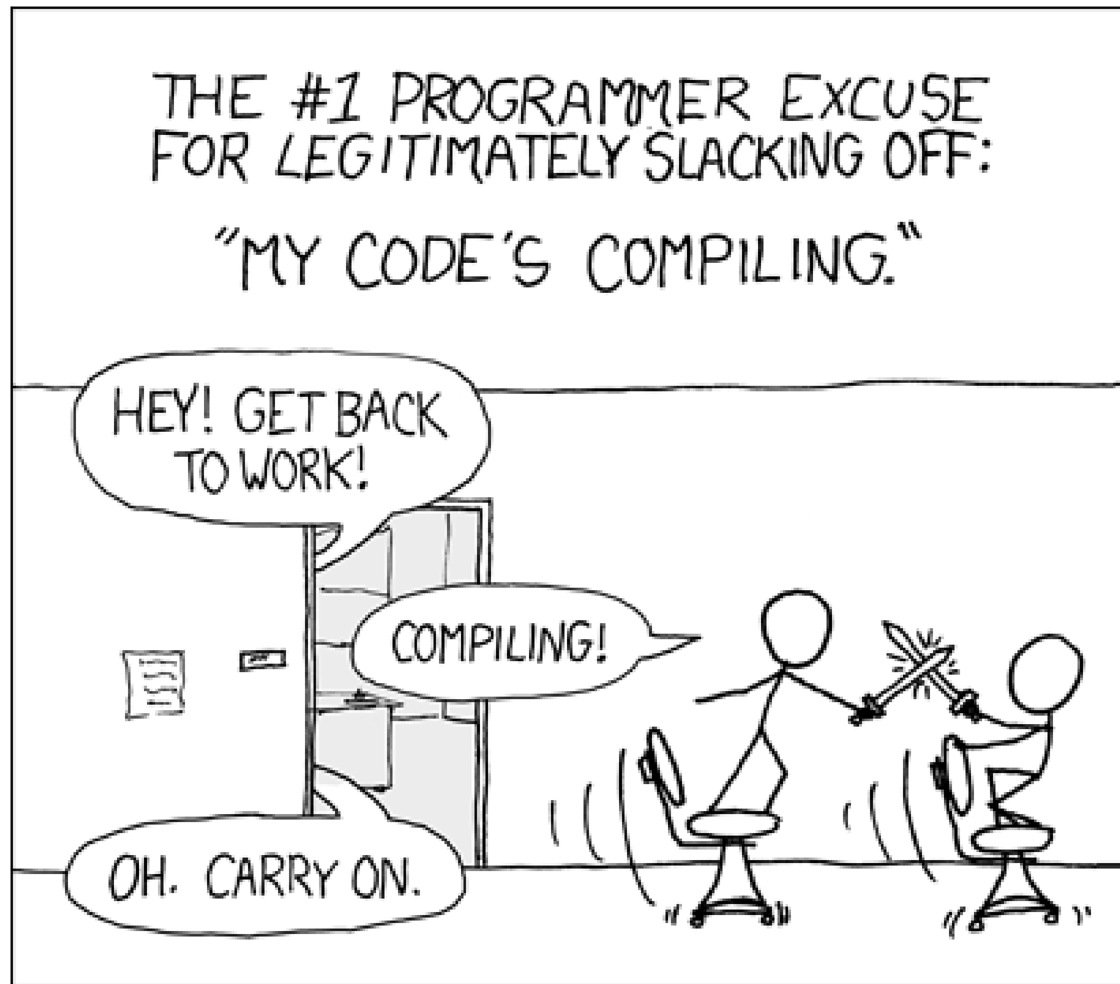
Why a new language?

- These problems are endemic and linguistic.
- New libraries won't help
  - Adding anything is going in the wrong direction
- Need to start over, thinking about the way programs are written and constructed

Goals

- The efficiency of a statically-typed compiled language with the ease of programming of a dynamic language
- Safety: type-safe and memory-safe
- Good support for concurrency and communication
- Efficient, latency-free garbage collection
- High-speed compilation

As xkcd observes



The image is licensed under a Creative Commons Attribution-NonCommercial 2.5 License

Design principles

- Keep concepts orthogonal
  - A few orthogonal features work better than a lot of overlapping ones
- Keep the grammar regular and simple
  - Few keywords, parsable without a symbol table
- Reduce typing
  - Let the language work things out
  - No stuttering; don't want to see **foo.Foo *myFoo = new foo.Foo(foo.FOO_INIT)**
  - Avoid bookkeeping
  - But keep things safe.
  - Keep the type system clear.
  - No type hierarchy. Too clumsy to write code by constructing type hierarchies
- It can still be object-oriented

The big picture

- Fundamentals:
  - Clean, concise syntax.
  - Lightweight type system.
  - No implicit conversions: keep things explicit.
  - Untyped unsized constants: no more **0x80ULL.**
  - Strict separation of interface and implementation
- Run-time
  - Garbage collection
  - Strings, maps, communication channels
  - Concurrency
- Package model
  - Explicit dependencies to enable faster builds

New approach: Dependencies

- Construction speed depends on managing dependencies
- Explicit dependencies in source allow
  - fast compilation
  - fast linking
- The Go compiler pulls transitive dependency type info from the object file but only what it needs.
- If **A.go** depends on **B.go** depends on **C.go**
  - compile **C.go, B.go, then A.go**
  - to compile **A.go,** compiler reads **B.o** not **C.o**
- At scale, this can be a huge speedup

New approach: Concurrency

- Go provides a way to write systems and servers as concurrent, garbage-collected processes
- (goroutines) with support from the language and runtime
- Language takes care of goroutine management, memory management
- Growing stacks, multiplexing of goroutines onto threads is done automatically
- Concurrency is hard without garbage collection
- Garbage collection is hard without the right language

Basics

```
const N = 1024 // just a number
const str = “this is a 日本語 string\n”
var x, y *float
var ch = '\u1234

/* Define and use a type, T. */
type T struct { a, b int }
var t0 *T = new(T);
t1 := new(T); // type taken from expr

// Control structures:
// (no parens, always braces)
if len(str) > 0 { ch = str[0] }
```

Program structure

```
package main
import "os"
import "flag"
var nFlag = flag.Bool("n", false, `no \n`)
func main() {
    flag.Parse();
    s := "";
    for i := 0; i < flag.NArg(); i++ {
    if i > 0 { s += " " }
    s += flag.Arg(i)
    }
    if !*nFlag { s += "\n" }
     os.Stdout.WriteString(s);
}
```

Constants

```
type TZ int

const (
UTC TZ = 0*60*60;
EST TZ = -5*60*60; // and so on
)

// iota enumerates:
const (
bit0, mask0 uint32 = 1<<iota, 1<<iota - 1;
bit1, mask1 uint32 = 1<<iota, 1<<iota - 1;
bit2, mask2; // implicitly same text
)

// high precision:
const Ln2= 0.693147180559945309417232121458\
176568075500134360255254120680009
const Log2E= 1/Ln2 // precise reciprocal
```

Values and types

```
weekend := [] string { "Saturday", "Sunday“ }

timeZones := map[string]TZ {
    "UTC":UTC, "EST":EST, "CST":CST, //...
}

func add(a, b int) int { return a+b }

type Op func (int, int) int

type RPC struct {
  a, b int;
  op Op;
  result *int;
}
rpc := RPC{ 1, 2, add, new(int); }
```

Methods

```
type Point struct {
    X, Y float // Upper case means exported
}

func (p *Point) Scale(s float) {
    p.X *= s; p.Y *= s; // p is explicit
}

func (p *Point) Abs() float {
    return math.Sqrt(p.X*p.X + p.Y*p.Y)
}

x := &Point{ 3, 4 };

x.Scale(5);
```

Methods for any user type

```
package main
import "fmt"
type TZ int
const (
    HOUR TZ = 60*60; UTC TZ = 0*HOUR; EST TZ = -5*HOUR; //...
)

var timeZones = map[string]TZ { "UTC": UTC, "EST": EST, }
func (tz TZ) String() string { // Method on TZ (not ptr)
   for name, zone := range timeZones {
     if tz == zone { return name }
    }
   return fmt.Sprintf("%+d:%02d", tz/3600, (tz%3600)/60);
  }
func main() {
fmt.Println(EST); // Print* know about method String()
fmt.Println(5*HOUR/2);
}
```

Interfaces

```
type Magnitude interface {
  Abs() float; // among other things
}

var m Magnitude;

m = x; // x is type *Point, has method Abs()

mag := m.Abs();

type Point3 struct { X, Y, Z float }
func (p *Point3) Abs() float {
   return math.Sqrt(p.X*p.X + p.Y*p.Y + p.Z*p.Z)
}

m = &Point3{ 3, 4, 5 };

type Polar struct { R, θ float }
func (p Polar) Abs() float { return p.R }

m = Polar{ 2.0, PI/2 };
mag += m.Abs();
```

Interfaces for generality

- Package io defines the Writer interface:

```
type Writer interface {
Write(p []byte) (n int, err os.Error)
}
```

- Any type with that method can be written to: files, pipes, network connections, buffers, ... On the other hand, anything that needs to write can just specify io.Writer.
- For instance, fmt.Fprintf takes io.Writer as first
- argument
- For instance, bufio.NewWriter takes an io.Writer in, buffers it, satisfies io.Writer out
- And so on…

Putting it together

```
package main
import (
  "bufio";
   "fmt";
    "os";
)
func main() {
   // unbuffered
  fmt.Fprintf(os.Stdout, "%s, ", "hello");
  // buffered: os.Stdout implements io.Writer
  buf := bufio.NewWriter(os.Stdout);
   // and now so does buf.
  fmt.Fprintf(buf, "%s\n", "world!");
  buf.Flush();
}
```

Concurrency

- Sequential machines are not getting much faster
  - No more free lunch
- Multicore is the answer
  - But a big effort on the programmer

Multithreading is hard

- Dataraces
- Weak memory drastically increases complexity
- No silver bullet solution

Java Data Races

```
public class Example extends Thread {
    private static int cnt = 0; // shared state
     public void run() {
          int y = cnt;
          cnt = y + 1; }
     public static void main(String args[]) {
         Thread t1 = new Example();
         Thread t2 = new Example();
         t1.start();
         t2.start();
    }
}
```

Go’s approach to concurrency

- Specialized goroutines which are executed in parallel
- Communication via Channels

Communication channels

```
var c chan string;

c = make(chan string);

c <- "Hello"; // infix send                    greeting := <-c; // prefix receive
```

goroutines

```
x := longCalculation(17); // runs too long

c := make(chan int);
func wrapper(a int, c chan int) {
    result := longCalculation(a);
    c <- result;
}

go wrapper(17, c);

// do something for a while; then...

x := <-c;
```

A multiplexed server

```
type Request struct {
    a, b int;
    replyc chan int; // reply channel inside the Request
}
type binOp func(a, b int) int
func run(op binOp, req *request) {
    req.replyc <- op(req.a, req.b)
}
func server(op binOp, service chan *request) {
   for {
      req := <-service; // requests arrive here
      go run(op, req); // don't wait for op
     }
}
func StartServer(op binOp) chan *request {
    reqChan := make(chan *request);
    go server(op, reqChan);
return reqChan;
}
```

The client

```
// Start server; receive a channel on which
// to send requests.
server := StartServer(
                    func(a, b int) int {return a+b});

// Create requests
req1 := &Request{23,45, make(chan int)};
req2 := &Request{-17,1<<4, make(chan int)};

// Send them in arbitrary order
server <- req1; server <- req2

// Wait for the answers in arbitrary order
fmt.Printf("Answer2: %d\n", <-req2.replyc);
fmt.Printf("Answer1: %d\n", <-req1.replyc);
```

Select

- Like a switch statement in which the cases are communications
- A simple example uses a second channel to tear down the server

```
func server(op binOp, service chan *request, quit chan bool) {
  for {
     select {
        case req := <-service:
           go run(op, req); // don't wait
        case <-quit:
           return;
      }
    }
}
```

Missing

- package construction a-la-ML
- Initialization
- Reflection
- dynamic types
- Embedding
- Iterators
- Testing

Libraries

- OS, I/O, files math (sin(x) etc.)
- strings, Unicode, regular expressions
- reflection
- command-line flags, logging hashes, crypto
- networking, HTTP, RPC
- HTML (and more general) templates
- …

Language Tools

Tool	Description
go build	builds Go binaries using only information in the source files themselves, no separate makefiles
go test	unit testing and microbenchmarks
go fmt	Preprint the code
go get	Retrieve and install remote packages
go vet	Static analyzer looking for potential errors in code
go run	Build and executing code
go doc	display documentation or serving it via HTTP
go rename	rename variables, functions, and so on in a type-safe way
go generate	A standard way to invoke code generators

Notable Users

- Docker, a set of tools for deploying Linux containers
- Doozer, a lock service by managed hosting provider Heroku
- Juju, a service orchestration tool by Canonical, packagers of Ubuntu Linux
- Syncthing, an open-source file synchronization client/server application
- Packer, a tool for creating identical machine images for multiple platforms from a single source configuration
- Ethereum, a shared world computing platform

Companies

- Google, for many projects, notably including download server
- Netflix, for two portions of their server architecture
- Dropbox, migrated some of their critical components from Python to Go
- CloudFlare, for their delta-coding proxy Railgun, their distributed DNS service, as well as tools for cryptography, logging, stream processing, and accessing SPDY sites
- SoundCloud, for "dozens of systems"
- The BBC, in some games and internal projects
- Novartis, for an internal inventory system
- Splice, for the entire backend (API and parsers) of their online music collaboration platform
- Cloud Foundry, a platform as a service
- CoreOS, a Linux-based operating system that utilizes Docker containers
- Couchbase, Query and Indexing services within the Couchbase Server
- Replicated, Docker based PaaS for creating enterprise, installable software
- MongoDB, tools for administering MongoDB instances

Some Critique

- Missing generics
- Missing algebraic data types
- Limited type inference
- Missing polymorphism
- Extendibility

Summary

- Designing a system's programming language is challenging
- Tradeoffs
  - Performance
  - Safety
  - Generality
  - Compiler speed
- Go takes a huge step from C
  - Is it enough?
  - Will it be adapted

Interesting PLs

Statically Typed

- C
- Java
- ML/Ocaml/F#
- Haskel
- C++
- C#
- Scala
- Go
- Rust

Dynamically Typed

- Lisp
- Scheme
- Python
- Javascript
- Lua
- Ruby