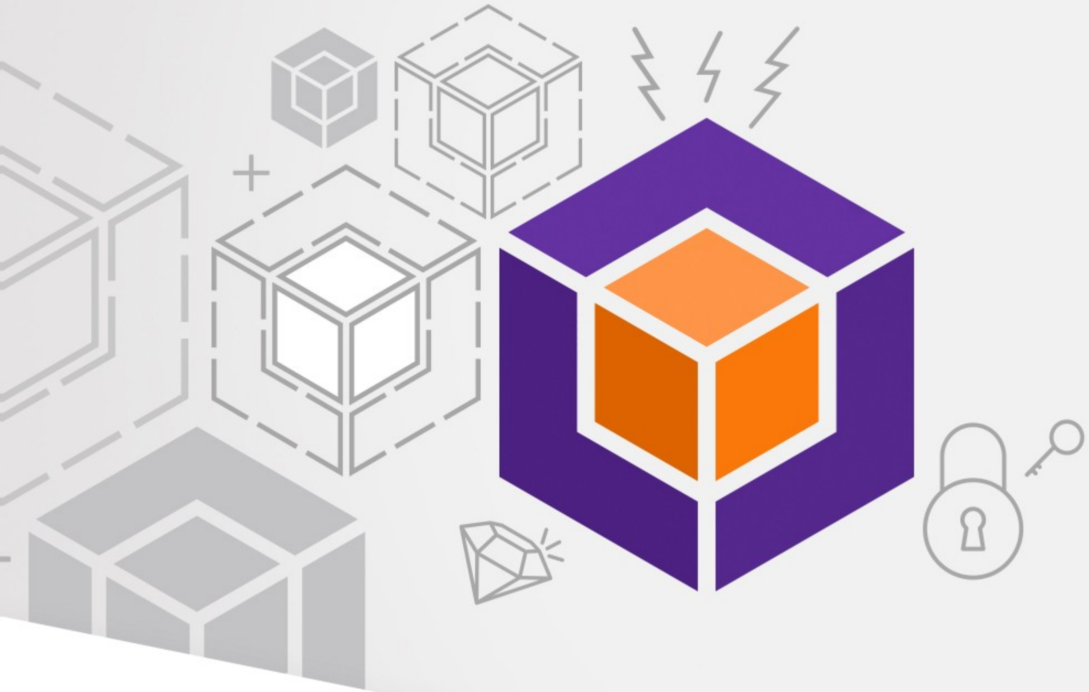
YottaDB

June 16, 2019

Hidden Dragons of CGO: Hard-learned Lessons from Writing Go Wrappers



Outline

- Introduction
- Review – YottaDB, Go, C
- Review – CGO
- Problem 1: Passing Data to Functions
- Problem 2: Passing Callbacks to Functions
- Problem 3: Garbage Collection
- Questions & Answers

YottaDB® – https://yottadb.com

- A mature, high performance, hierarchical key-value NoSQL database whose code base scales up to mission-critical applications like large real-time core-banking and electronic health records, and also scales down to run on platforms like the Raspberry Pi Zero, as well as everything in-between.
- *Rock Solid. Lightning Fast. Secure. Pick any three.*

Golang

- “Go is an open source programming language that makes it easy to build simple, reliable, and efficient software. “
  - https://golang.org/
- Programming language from Google
  - Static typing
  - Cooperative multithreading
  - Mix of C, Erlang, and “Java”

C

- Low-level programming language developed in 1972 by Dennis Ritchie
- Everwhere.
  - Everything speaks C; Linux machines, embedded systems, IOT devices, Android, …. everything
- Static typing, manual memory management
- Low-level primitives (casting bits, pointer math)

Golang - CGO

- https://golang.org/cmd/cgo/
  - A lot of code is written in C (GUI libraries, databases, math libraries, …); calling into that code open a whole new world of possibilities
  - C is not garbage collected; every C program is responsible for its own memory allocations (malloc) and frees; CGO attempts to work around this by creating rules

Golang - CGO

- Rule 1
  - “Go code may pass a Go pointer to C provided the Go memory to which it points does not contain any Go pointers. The C code must preserve this property: it must not store any Go pointers in Go memory, even temporarily. When passing a pointer to a field in a struct, the Go memory in question is the memory occupied by the field, not the entire struct. When passing a pointer to an element in an array or slice, the Go memory in question is the entire array or the entire backing array of the slice. “
    - https://golang.org/cmd/cgo/#hdr-Passing_pointers

Golang - CGO

- Rule 1 - OK

```
package main

import "fmt"

// int myFunc(int r1) {
//   return r1 + 42;
// }
import "C"

func main() {
  fmt.Printf("%v",
C.myFunc(22))
}
```

Golang - CGO

- Rule 1 - OK

```
package main

import "fmt"

// int myFunc(char c1) {
//   return c1 + 42;
// }
import "C"

func main() {
  fmt.Printf("%v\n",
C.myFunc(22))
}
```

Golang - CGO

- Rule 1 – Not OK
- Compilation error

```
cannot use "Hello
world!" (type
string) as type
*_Ctype_char in
argument to
_Cfunc_myFunc
```

```
package main

// #include <stdio.h>
// void myFunc(char
*c1) {
//  printf("Hello %s!\
n", c1);
// }
import "C"

func main() {
  C.myFunc("world!")
}
```

Golang - CGO

- Rule 1 – Not OK
- Compilation error

```
cannot use "Hello
world!" (type
string) as type
*_Ctype_char in
argument to
_Cfunc_myFunc
```

```
package main

// #include <stdio.h>
// void myFunc(char
*c1) {
//  printf("Hello %s!\
n", c1);
// }
import "C"

func main() {
  msg := "world!"
  C.myFunc(msg)
}
```

Golang - CGO

- Rule 1 – OK
- Correct? Yes
- Still garbage collected

```go
package main

import (
  "unsafe"
)

// #include <stdio.h>
// void myFunc(void *c1) {
//  printf("Hello %s!\n", (char*)c1);
// }
import "C"

func main() {
  msg := ([]byte)("world!")
  C.myFunc(unsafe.Pointer(&msg[0]))
}
```

Golang - CGO

- Rule 1 – Here be dragons!
  - Slice

```
type Buffer struct {
    buff []byte
}
```

Golang - CGO

- Rule 1 – Here be dragons!
  - Array

```
type Buffer struct {
    buff [255]byte
}
```

Golang - CGO

- Rule 1 – Here be dragons!
  - ?

```
type Buffer struct {
    buff []byte
}

func hello() {
    C.myFunc(&myBuff.buff[0])
}
```

Golang - CGO

- Rule 1 – Here be dragons!
  - A slice is a thing wrapper to an array; get reference to first element

```
type Buffer struct {
    buff []byte
}

func hello() {
    C.myFunc(&myBuff.buff[0])
}
```

Golang - CGO

- What about complex structures?

```
package main

// #include <stdio.h>
// typedef struct {
//   char *msg;
// } myStruct;
// void myFunc(myStruct *strct)
{
//   printf("Hello %s!\n",
strct->msg);
// }
import "C"

func main() {
  msg := C.myStruct{"world!"}
  C.myFunc(&msg)
}
```

Golang - CGO

- What about complex structures?
  - No go; compilation error

```
package main

// #include <stdio.h>
// typedef struct {
//   char *msg;
// } myStruct;
// void myFunc(myStruct *strct)
{
//   printf("Hello %s!\n",
strct->msg);
// }
import "C"

func main() {
  msg := C.myStruct{"world!"}
  C.myFunc(&msg)
}
```

Golang - CGO

- What about complex structures?

```go
package main

import "unsafe"

// #include <stdio.h>
// typedef struct {
//   char *msg;
// } myStruct;
// void myFunc(myStruct *strct) {
//   printf("Hello %s!\n", strct->msg);
// }
import "C"

func main() {
  val := ([]byte)("world!")
  msg := C.myStruct{(*C.char)(unsafe.Pointer(&val[0]))}
  C.myFunc(&msg)
}
```

Golang - CGO

- What about complex structures?
  - Compiles

```go
package main

import "unsafe"

// #include <stdio.h>
// typedef struct {
//   char *msg;
// } myStruct;
// void myFunc(myStruct *strct) {
//   printf("Hello %s!\n", strct->msg);
// }
import "C"

func main() {
  val := ([]byte)("world!")
  msg := C.myStruct{(*C.char)(unsafe.Pointer(&val[0]))}
  C.myFunc(&msg)
}
```

Golang - CGO

- What about complex structures?
  - Compiles
  - Will fail at runtime

```
package main

import "unsafe"

// #include <stdio.h>
// typedef struct {
//   char *msg;
// } myStruct;
// void myFunc(myStruct *strct) {
//   printf("Hello %s!\n", strct->msg);
// }
import "C"

func main() {
  val := ([]byte)("world!")
  msg := C.myStruct{(*C.char)(unsafe.Pointer(&val[0]))}
  C.myFunc(&msg)
}
```

Golang - CGO

- What about complex structures?
  - Compiles

```go
package main

// #include <stdio.h>
// typedef struct {
//   char *msg;
// } myStruct;
// void myFunc(myStruct *strct) {
//   printf("Hello %s!\n", strct->msg);
// }
import "C"

func main() {
  msg := C.myStruct{C.CString("world")}
  C.myFunc(&msg)
}
```

Golang - CGO

- What about complex structures?
  - Compiles
  - Runs

```
package main

// #include <stdio.h>
// typedef struct {
//   char *msg;
// } myStruct;
// void myFunc(myStruct *strct) {
//   printf("Hello %s!\n", strct->msg);
// }
import "C"

func main() {
  msg := C.myStruct{C.CString("world")}
  C.myFunc(&msg)
}
```

Golang - CGO

- What about complex structures?
  - Compiles
  - Runs
  - Leaks memory

```
package main

// #include <stdio.h>
// typedef struct {
//   char *msg;
// } myStruct;
// void myFunc(myStruct *strct) {
//   printf("Hello %s!\n", strct->msg);
// }
import "C"

func main() {
  msg := C.myStruct{C.CString("world")}
  C.myFunc(&msg)
}
```

Golang - CGO

- What about complex structures?
  - Manually free?

```
package main

// #include <stdio.h>
// typedef struct {
//   char *msg;
// } myStruct;
// void myFunc(myStruct *strct) {
//   printf("Hello %s!\n", strct->msg);
// }
import "C"

func main() {
  msg := C.myStruct{C.CString("world")}
  defer C.free(msg.msg)
  C.myFunc(&msg)
}
```

Golang - CGO

- What about complex structures?
  - Manually free?
  - Ugh.

```
package main

// #include <stdio.h>
// typedef struct {
//   char *msg;
// } myStruct;
// void myFunc(myStruct *strct) {
//   printf("Hello %s!\n", strct->msg);
// }
import "C"

func main() {
  msg := C.myStruct{C.CString("world")}
  defer C.free(msg.msg)
  C.myFunc(&msg)
}
```

Golang - CGO

- What about complex structures?
  - Manually free?
  - Ugh.
  - What about complex memory patterns?

```
package main

// #include <stdio.h>
// typedef struct {
//   char *msg;
// } myStruct;
// void myFunc(myStruct *strct) {
//   printf("Hello %s!\n", strct->msg);
// }
import "C"

func main() {
  msg := C.myStruct{C.CString("world")}
  defer C.free(msg.msg)
  C.myFunc(&msg)
}
```

Golang - CGO

- What about complex structures?
  - Finalizer?

```
// #include <stdio.h>
// #include <stdlib.h>
// typedef struct {
//   char *msg;
// } myStruct;
// void myFunc(myStruct *strct) {
//   printf("Hello %s!\n", strct->msg);
// }
import "C"

func main() {
  msg := C.myStruct{C.CString("world")}
  runtime.SetFinalizer(&msg, func(t *C.myStruct) {
    C.free(unsafe.Pointer(t.msg))
  })
  C.myFunc(&msg)
}
```

Golang - CGO

- What about complex structures?
  - Finalizer?
  - Makes Gophers angry

```
// #include <stdio.h>
// #include <stdlib.h>
// typedef struct {
//   char *msg;
// } myStruct;
// void myFunc(myStruct *strct) {
//   printf("Hello %s!\n", strct->msg);
// }
import "C"

func main() {
  msg := C.myStruct{C.CString("world")}
  runtime.SetFinalizer(&msg, func(t *C.myStruct) {
    C.free(unsafe.Pointer(t.msg))
  })
  C.myFunc(&msg)
}
```

Golang - CGO

- What about complex structures?
  - Finalizer?
  - Makes Gophers angry
  - Works?

```
// #include <stdio.h>
// #include <stdlib.h>
// typedef struct {
//   char *msg;
// } myStruct;
// void myFunc(myStruct *strct) {
//   printf("Hello %s!\n", strct->msg);
// }
import "C"

func main() {
  msg := C.myStruct{C.CString("world")}
  runtime.SetFinalizer(&msg, func(t *C.myStruct) {
    C.free(unsafe.Pointer(t.msg))
  })
  C.myFunc(&msg)
}
```

Golang - CGO

- Here be dragons!

```
// #include <stdio.h>
// #include <stdlib.h>
// typedef struct {
//  char *msg;
// } myStruct;
// void myFunc(char *msg) {
//  printf("Hello %s!\n", msg);
// }
import "C"

func main() {
  msg := C.myStruct{C.CString("world")}
  runtime.SetFinalizer(&msg, func(t *C.myStruct) {
    C.free(unsafe.Pointer(t.msg))
  })
  C.myFunc(msg.msg)
}
```

Golang - CGO

- Here be dragons!
  - Finalizer can be called after call into myFunc, before return

```go
// #include <stdio.h>
// #include <stdlib.h>
// typedef struct {
//   char *msg;
// } myStruct;
// void myFunc(char *msg) {
//   printf("Hello %s!\n", msg);
// }
import "C"

func main() {
  msg := C.myStruct{C.CString("world")}
  runtime.SetFinalizer(&msg, func(t *C.myStruct) {
    C.free(unsafe.Pointer(t.msg))
  })
  C.myFunc(msg.msg)
}
```

Golang - CGO

- Here be dragons!
  - Finalizer can be called after call into myFunc, before return
  - Keep alive

```
// #include <stdio.h>
// #include <stdlib.h>
// typedef struct {
//  char *msg;
// } myStruct;
// void myFunc(char *msg) {
//  printf("Hello %s!\n", msg);
// }
import "C"

func main() {
  msg := C.myStruct{C.CString("world")}
  runtime.SetFinalizer(&msg, func(t *C.myStruct) {
    C.free(unsafe.Pointer(t.msg))
  })
  C.myFunc(msg.msg)
  runtime.KeepAlive(&msg)
}
```

Golang - CGO

- Rule 2 – How to let the dragons talk take turns?
  - Callbacks in C are very common, especially with UI work
  - YottaDB uses callbacks for our transaction processing

Golang - CGO

- Callback – Not OK
  - Compilation error

```
package main

// void do_callback(void (*cb)()) {
//   cb();
// }
import "C"

func callback() {
  fmt.Printf("Here!\n")
}

func main() {
  C.do_callback(callback)
}
```

Golang - CGO

- Callback
  - Split into two files
  - Declare C func to return callback

```
package main

// void do_callback(void (*cb)())
{
//   cb();
// }
// extern void callback();
// void (*get_callback())() {
//   return callback;
// }
import "C"


func main() {
  C.do_callback(C.get_callback())
}
```

```
package main

import "fmt"

import "C"

//export callback
func callback() {
  fmt.Printf("Here!\n")
}
```

Golang - CGO

- Do we expect users to do this for every callback?
  - Error prone
  - Makes testing hard
    - Can’t import C in test code
  - Requires knowledge of C

Golang - CGO

- Solution; write a callback function which passes back an index to a hashmap which can store the Golang callback

Golang - CGO

```
var tpIndex uint64
var tpMap sync.Map

// YdbTpStWrapper is a private callback to wrap calls to the Go closure required
for TpST.
//export ydbTpStWrapper
func ydbTpStWrapper(tptoken uint64, errstr *C.ydb_buffer_t, tpfnparm
unsafe.Pointer) int32 {
    var errbuff BufferT

    index := *((*uint64)(tpfnparm))
    v, ok := tpMap.Load(index)
    if !ok {
        panic("YDB: Could not find callback routine")
    }
    errbuff.BufferTFromPtr((unsafe.Pointer)(errstr))
    return (v.(func(uint64, *BufferT) int32))(tptoken, &errbuff)
}
```

Golang - CGO

- Solution; write a callback function which passes back an index to a hashmap which can store the Golang callback
  - Some small performance hit
  - Weird to think about
  - How do we pass argument to callback function?

Golang - CGO

- Solution; write a callback function which passes back an index to a hashmap which can store the Golang callback
  - Some small performance hit
  - Weird to think about
  - How do we pass argument to callback function?
    - Closures

Golang - CGO

```
func helloWorld(i int) {
  (func() {
    fmt.Printf("i: %v", i)
  })()
}
```

Triggering garbage collection

- Issues won't show up until garbage collection is triggered
- Issues will only show up sometimes when we do GC
- How do we help things along?

Triggering garbage collection

- How do we help things along?
  - Flags to trigger more garbage collection
    - export GOGC=1
  - Flags to watch for Go pointers being passed
    - Export GODEBUG="cgocheck=2"

Triggering garbage collection

- How do we help things along?
  - Horrible. Mean. Ugly. Tests.

Thank You!

yottadb.com