Mobile Go, part 1: Calling Go functions from C

Part 1 of my quest to learn Go for use in my iOS and Android app. Here's how to call Go code from a C command-line app.

Why Go? And why start here?

While I've had my eye on Go for a few years now, I've always been skeptical— I like the idea of replacing the monstrosity that is C++, but Go didn't really seem like enough of an improvement to pursue. However, over the years it's been recommended to me again and again by some of my favorite engineers, including Bob Lee and Alan Donovan. So I finally started learning more about it, and decided to explore it by adding some Go code into a my company's mobile app.

Go developers seem pretty happy using ***go build*** to compile their command-line apps and servers. They don't have to worry about how their code gets built or linked—they just implement ***func main()*** and they're on their merry way. In this way, Google has made it easy for developers who fit into that use case—i.e., those building a standalone app built in Go. Unfortunately, it turns out I made things very complicated for myself by going outside this path ☺

Our smartphone app, which we develop for both iOS and Android, contains a lot of cross-platform C++ code. (Many people are surprised by this, and it should probably be the subject of another blog post.) This part of the codebase powers much of the app's core functionality—our ultrasonic audio signal codec, sound synthesis, image filters, and so on. While C++ is an unusual choice for developing mobile apps, it has allowed us to Write Once, Run Anywhere—at least partially—instead of having to write complex algorithms once in Objective C and then again in Java. It has not only saved us time and money, but also allowed us to concentrate our energy on one language and thus one way of doing things.

A beginner's journey

This blog post documents my path from not knowing Go at all, to making a very simple command-line application that crosses the Go/C boundary in both directions. It's a bit long, but I wanted to include all the code and error messages, so that developers encountering these issues can hopefully find this post, and have an easier time than I did ☺

(Hello, World)²

My first goal was simple: a cross-language Hello World command-line app. I imagined writing *hello.c*:

```
#include <stdio.h>

int main() {
    printf("Hello from C!\n");
    HelloFromGo();
    return 0;
}
```

And then *hello.go*:

```
package hello

import "fmt"

func HelloFromGo() {
   fmt.Printf("Hello from Go!\n")
}
```

Now I imagined this could be made to work pretty easily, and started diving into the docs. After some Googling it appeared that the relevant information was spread across a Go FAQ entry, a Go blog post, and the documentation for the *cgo* tool.

Let's start with the Go FAQ

A Google search led me immediately to the FAQ on golang.org.

> ***Do Go programs link with C/C++ programs?***

> *There are two Go compiler implementations, gc (the 6g program and friends) and gccgo. Gc uses a different calling convention and linker and can therefore only be linked with C programs using the same convention. There is such a C compiler but no C++ compiler. Gccgo is a GCC front-end that can, with care, be linked with GCC-compiled C or C++ programs.*

> *The cgo program provides the mechanism for a "foreign function interface" to allow safe calling of C libraries from Go code. SWIG extends this capability to C++ libraries.*

While it was good to know that Go and C++ are indeed interoperable, this wasn't very helpful, and in fact it left me with more questions than answers: *Why are there two Go compilers? What is the "6g program," and why isn't it* *mentioned anywhere else in the FAQ? And who are its friends? Is cgo one of those two compilers, or rather a third tool that works with those compilers? Do I really have to use goddamn SWIG? Yuck!*

Next up: the Go blog

Some more Googling led me to a promising a blog post from the Go team called C? Go? Cgo! It states:

> *Cgo lets Go packages call C code. Given a Go source file written with some special features, cgo outputs Go and C files that can be combined into a single Go package.*

Okay, we're definitely getting somewhere, but unfortunately it does not look very promising for calling out from C++ to Go—rather, the blog provides instructions for the *opposite* use case: calling C code *from* Go. I can understand why this is a popular thing to want, as it allows Go developers to use libraries written in C. However, this is not what we need. I continued hunting.

Next up: the cgo docs

From reading the blog post, it does seem like *cgo* can help us, even if the FAQ and blog post did not mention our use case at all. So I dove into the cgo docs:

> *Cgo enables the creation of Go packages that call C code.*

Okay… also not the right use case. But as I continued reading, something caught my eye:

> ***C references to Go***

> *Go functions can be exported for use by C code in the following way:*

> //export MyFunction
> func MyFunction(arg1, arg2 int, arg3 string) int64 {…}

> *It will be available in the C code as:*

> extern int64 MyFunction(int arg1, int arg2, GoString arg3);

> *found in the _cgo_export.h generated header, after any preambles copied from the cgo input files.*

Okay so it looks like we need to use *cgo* to generate some files, including *_cgo_export.h*. It sounds like this will let us call our HelloFromGo functions from our C main() function. Let's try it!

Adventures with cgo

```
$ go tool cgo hello.go
cannot find import "C"
```

Hmm, I saw something about this in the *cgo* command documentation. The "C" package contains C functions from whatever you're linking against… or something. Well, let's try importing it, and also adding the *//export* comment mentioned in the blog post.

```
package main

import "fmt"
import "C"

//export HelloFromGo
func HelloFromGo() {
   fmt.Printf("Hello from Go!")
}
```

And try to run *cgo* again:

```
$ go tool cgo hello.go
$
```

Wow it did something! And I have a new folder called *_obj*!

```
$ ls
_obj hello.c hello.go
$ ls _obj
_cgo_defun.c _cgo_export.c _cgo_export.h _cgo_flags
_cgo_gotypes.go _cgo_main.c hello.cgo1.go hello.cgo2.c
```

That's certainly a lot of files! There's not really much inside each file—really just some boilerplate. For example here's the _cgo_export.h that was mentioned in the blog post above:

```
/* Created by cgo - DO NOT EDIT. */

typedef signed char GoInt8;
typedef unsigned char GoUint8;
typedef short GoInt16;
typedef unsigned short GoUint16;
// etc...
typedef struct { char *p; GoInt n; } GoString;
typedef void *GoMap;
typedef void *GoChan;
typedef struct { void *t; void *v; } GoInterface;
typedef struct { void *data; GoInt len; GoInt cap; } GoSlice;

extern void HelloFromGo();
```

But hey, I see my HelloFromGo function, declared as a C function! This is looking really promising! But…

Adventures with go build

Okay, so what do we do with all these files? Well, I'm not sure, so I resort to the blog post again. And it turns out we don't need these files at all—and we don't even need to run *cgo* ourselves. At the end of the blog post, the author explains:

> *To build cgo packages, just use* ***go build*** *or* ***go install*** *as usual. The go tool recognizes the special "C" import and automatically uses cgo for those files.*

Well, this is kind of a roundabout way to explain things, but I can live with this. And really it explains the mysterious section of the cgo docs called "Using cgo directly," to which my initial reaction was: *Well duh, why* ***wouldn't*** *I use it directly?*

Let's delete that *_obj* folder and try building using ***go build:***

```
$ go build
# _/Users/keith/.../src
duplicate symbol _main in:
 $WORK/_/Users/keith/.../src/_obj/_cgo_main.o
 $WORK/_/Users/keith/.../src/_obj/hello.o
ld: 1 duplicate symbol for architecture x86_64
clang: error: linker command failed with exit code 1 (use -v to
see invocation)
```

Hmmmm… Clearly they're saying we have two main functions, but I don't know why this would be happening—I define main() once and only once, in *hello.c*. As you may recall, I didn't define main() in Go, because my goal here is to integrate Go code into my *existing* C++ app, which already has a main().

I'm feeling a little stuck, but I'd really love to get this working, and I had an idea: What if I define main() in Go, and then simply have it call my actual C main() function, which I can rename to main2()? Let's try it out.

A detour: calling C code from Go

Well, I didn't want to be doing this—calling C from Go—but it seems like it may be necessary. So here's my new *hello.go* file:

```
package main

import "fmt"
import "C"

//export HelloFromGo
func HelloFromGo() {
   fmt.Printf("Hello from Go!")
}

func main() {
   C.main2();
}
```

And my *hello.c*:

```
#include <stdio.h>
#include "_cgo_export.h"

int main2() {
    printf("Hi from C++\n");
    HelloFromGo();
    return 0;
}
```

Now let's try building again:

```
$ go build
# _/Users/keith/…/src
could not determine kind of name for C.main2
```

Well, this is understandable—I had a hunch that it wouldn't be so easy to call the C main2() function from Go. Because really, how would Go know what arguments it takes, or what its return value is? I imagine I need to create a header file and pass that into Go's compiler somehow. This should be easy, as the majority of the Go/C documentation provided by the Go team is for calling C *from* Go. Let me check the blog post again.

Looks like the first example doesn't call any *custom* C code, but rather calls some functions from the C standard library:

```
package rand

/*
#include <stdlib.h>
*/
import "C"

func Random() int {
   return int(C.random())
}

func Seed(i int) {
   C.srandom(C.uint(i))
}
```

The blog post explains how Go got access to those functions:

> *Cgo recognizes the comment above the import statement.… these are used as a header when compiling the C parts of the package. In this case those lines are just a single #include statement, but they can be almost any C code.*

Great! I'm hoping I can just put the main2() function declaration right in that special comment:

```
package main

import "fmt"
/*
int main2();
*/
import "C"

//export HelloFromGo
func HelloFromGo() {
   fmt.Printf("Hello from Go!\n")
}

func main() {
   C.main2()
}
```

Now I cross my fingers and…

```
$ go build
$
```

Wow! It worked?? Looks like it produced a binary file called *src* (oops, I guess I should've read up on Go package naming). What happens when I run it?

```
$ ./src
Hi from C++
Hello from Go!
$
```

Wow!!

OMG!!! Success!

It's a dream come true! And now I feel like I understand Go a little better. And all it took was about 8 hours of puttering around the Internet, filing bug reports, and trying not to scream ☺

Lessons learned:

- Go always generates a main() function, and there's nothing you can do about it. (Ian Lance Taylor from Google confirmed this for me in a comment on GitHub, and there is an open issue for it.)
- This means there's currently no supported way to fire up the Go runtime from your existing C++ app, without initializing your C++ app from Go itself. (But I have some ideas for how to work around this—will cover in a future blog post if it works ☺)
- This limitation not a huge deal, because you can make Go's main() call your app's main function—you just need to rename it from main() to something else.

- The Go documentation points you towards using a tool called *cgo* for Go/C interoperability, but in fact you don't need to use this or even know that it exists.

You can follow the Go team's progress on these issues, or post comments to show support:

- #256—Generate .so from Go code
- #2790—Embed Go in C program
- #4848—Allow invoking Go from C main program

(Also, Go developer Ian Lance Taylor has published a document called Go Execution Modes. He explains the work that needs to be done to allow Go to be called from C code in various ways.)

Stay tuned for the next post in this series!