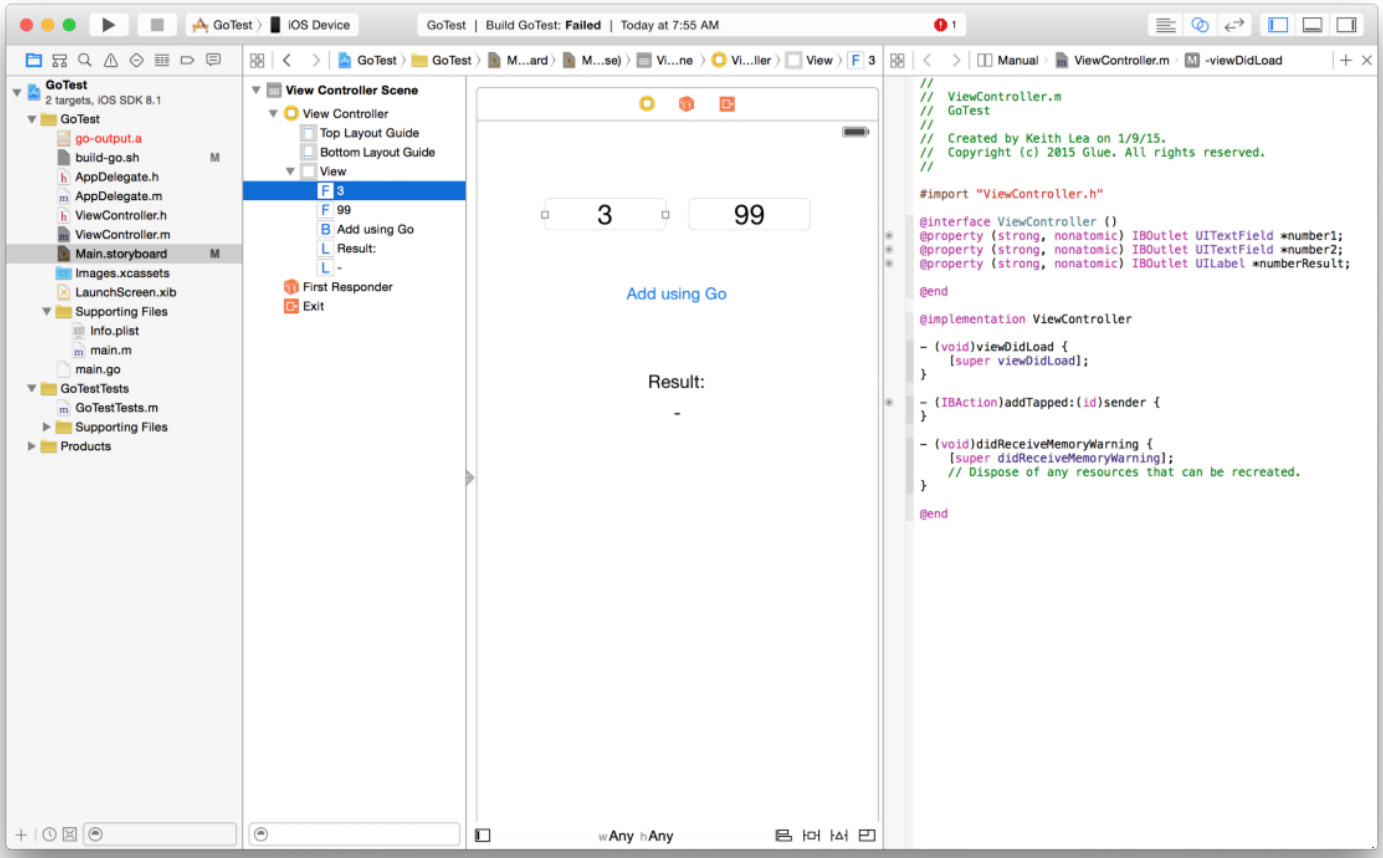
Mobile Go, part 3: Calling Go from Objective C

Part 3 of my quest to learn Go for use in my iOS and Android app. Here's how to call into Go from your iOS app.

Calling Go functions from Objective C

After finally figuring out how to build an iOS app with Go in part 2 of this series, it's time to actually start developing my app. I mentioned that my first goal is a simple calculator, where the calculation logic is implemented in Go, but controlled by a native iOS GUI.

Here's the pretty terrible calculator UI that I'll start with:



As I try to recall everything I learned in part 1 about C⟷Go interoperability, I am reminded of all those *.h and *.c files generated when I thought I needed to build with *Cgo*. That is, before I realized *Cgo* was invoked automatically by *go build*. I'm guessing that if I'm trying to call Go from C, I'll need those header files generated by Go. (And probably the *.c files, too, as I assume Go has some special code to manage the transition from C back into Go.)

As I revisit the Cgo website, I see the following:

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

```
//export MyFunction
func MyFunction(arg1, arg2 int, arg3 string) int64 {...}
//export MyFunction2
func MyFunction2(arg1, arg2 int, arg3 string) (int64, *C.char) {...}
```

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

```
extern int64 MyFunction(int arg1, int arg2, GoString arg3);
extern struct MyFunction2_return MyFunction2(int arg1, int arg2, GoString arg3);
```

> *found in the _cgo_export.h generated header, after any preambles copied from the cgo input files. Functions with multiple return values are mapped to functions returning a struct. Not all Go types can be mapped to C types in a useful way.*

That's right! So where do I find this _cgo_export.h? Well, maybe we can just include it and hope for the best. Let's create our add function in *main.go*:

```
package main

import "fmt"
import "os"
```

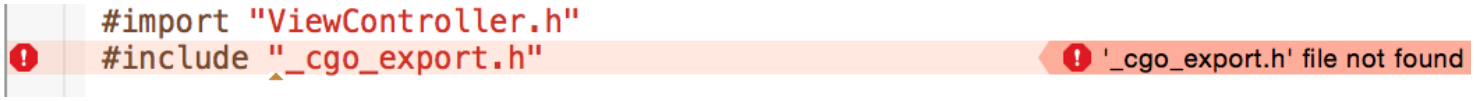
```
/*
#cgo LDFLAGS: -framework UIKit -framework Foundation -framework CoreGraphics
extern void iosmain(int argc, char *argv[]);
*/
import "C"

//export AddUsingGo
func AddUsingGo(a int, b int) int {
   return a + b
}
```

```
func main() {
   fmt.Fprintf(os.Stderr, "Hello from Go! I'm in func main()!\n")
   C.iosmain(0, nil)
}
```

And now let's try to call it in our view controller:

```
#import “ViewController.h”
#include “_cgo_export.h”

@interface ViewController ()
@property (strong, nonatomic) IBOutlet UITextField *number1;
@property (strong, nonatomic) IBOutlet UITextField *number2;
@property (strong, nonatomic) IBOutlet UILabel *numberResult;
@end

@implementation ViewController

- (void)viewDidLoad {
   [super viewDidLoad];
}

- (IBAction)addTapped:(id)sender {
    int result = AddUsingGo(self.number1.text.intValue,
                            self.number2.text.intValue);
    self.numberResult.text = @(result).stringValue;
}

@end
```

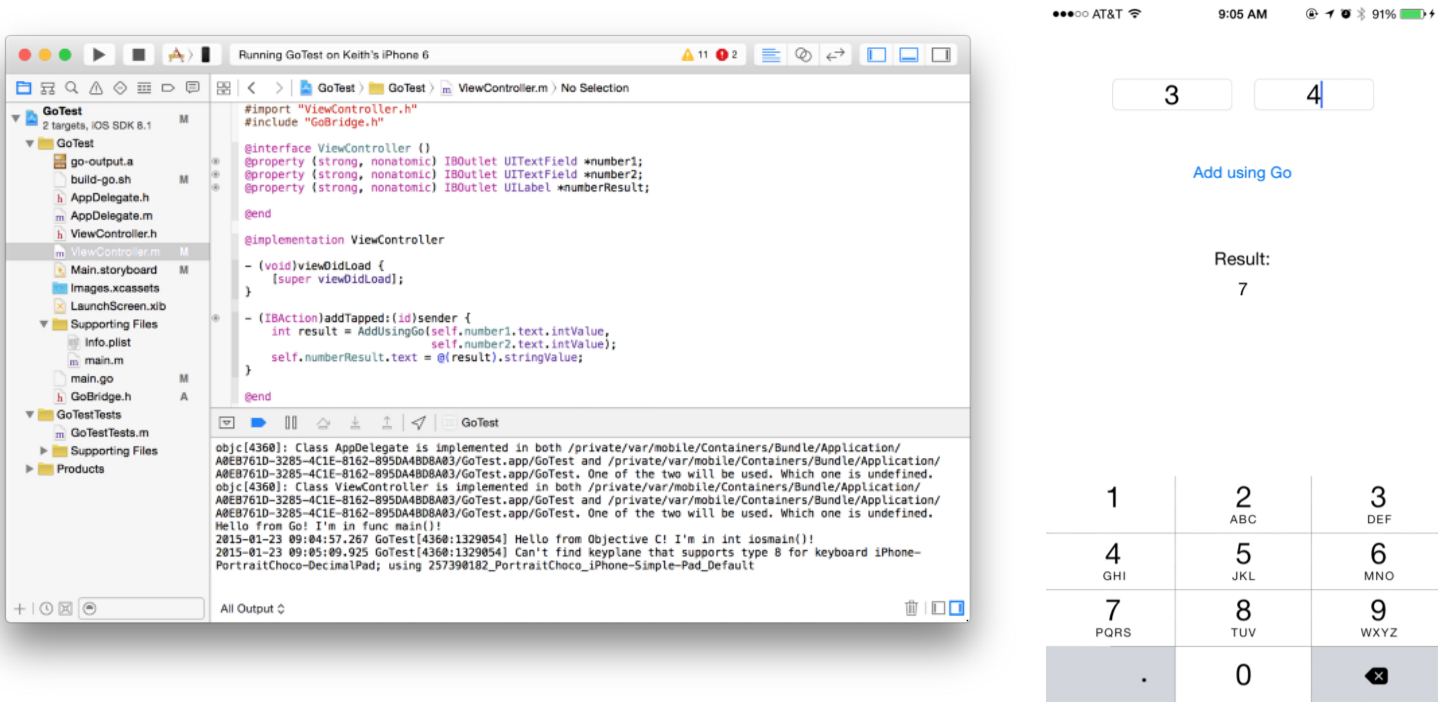
Well, I probably could’ve predicted this:



I’m assuming Xcode can’t find *_cgo_export.h* because it is generated during *go build* as an intermediate file. Well, I have a hacky idea, just to get it working right now. Couldn’t I just create a header file that contains the *AddUsingGo* declaration? Let’s call it *GoBridge.h*:

```
int AddUsingGo(int a, int b);
```

And now we can include *GoBridge.h* in our view controller… fingers crossed…



Wow! That's a screenshot from my actual iPhone! It works!! I built a calculator in Go!

Including the generated Cgo headers in Objective C

While I'm thrilled that it works, this is not a sustainable long-term solution. As the complexity of the Go code increases, we're not going to want to be generating function declarations by hand for every Go function we'd like to use in our iOS project.

So we need to find a way to include *_cgo_export.h* in our Xcode build. Yuck.

I think it's time to tackle a slightly creepy issue we saw in part 2 of this series. When running our Go app, we see this printed in the console (not the build log—the console):

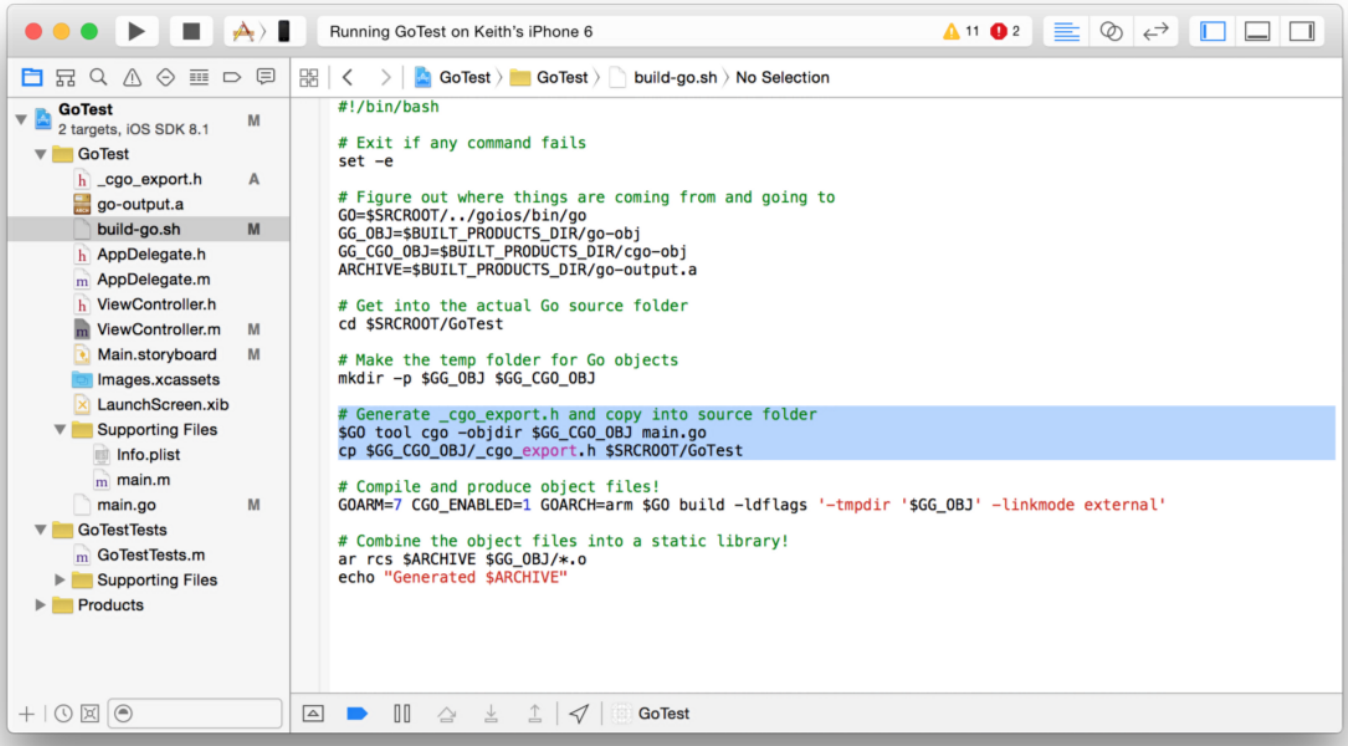
```
objc[4360]: Class AppDelegate is implemented in both
/private/var/mobile/Containers/Bundle/Application/A0EB761D-3285-
4C1E-8162-895DA4BD8A03/GoTest.app/GoTest and
/private/var/mobile/Containers/Bundle/Application/A0EB761D-3285-
4C1E-8162-895DA4BD8A03/GoTest.app/GoTest. One of the two will be
used. Which one is undefined.
```

```
objc[4360]: Class ViewController is implemented in both
/private/var/mobile/Containers/Bundle/Application/A0EB761D-3285-
4C1E-8162-895DA4BD8A03/GoTest.app/GoTest and
/private/var/mobile/Containers/Bundle/Application/A0EB761D-3285-
4C1E-8162-895DA4BD8A03/GoTest.app/GoTest. One of the two will be
used. Which one is undefined.
```

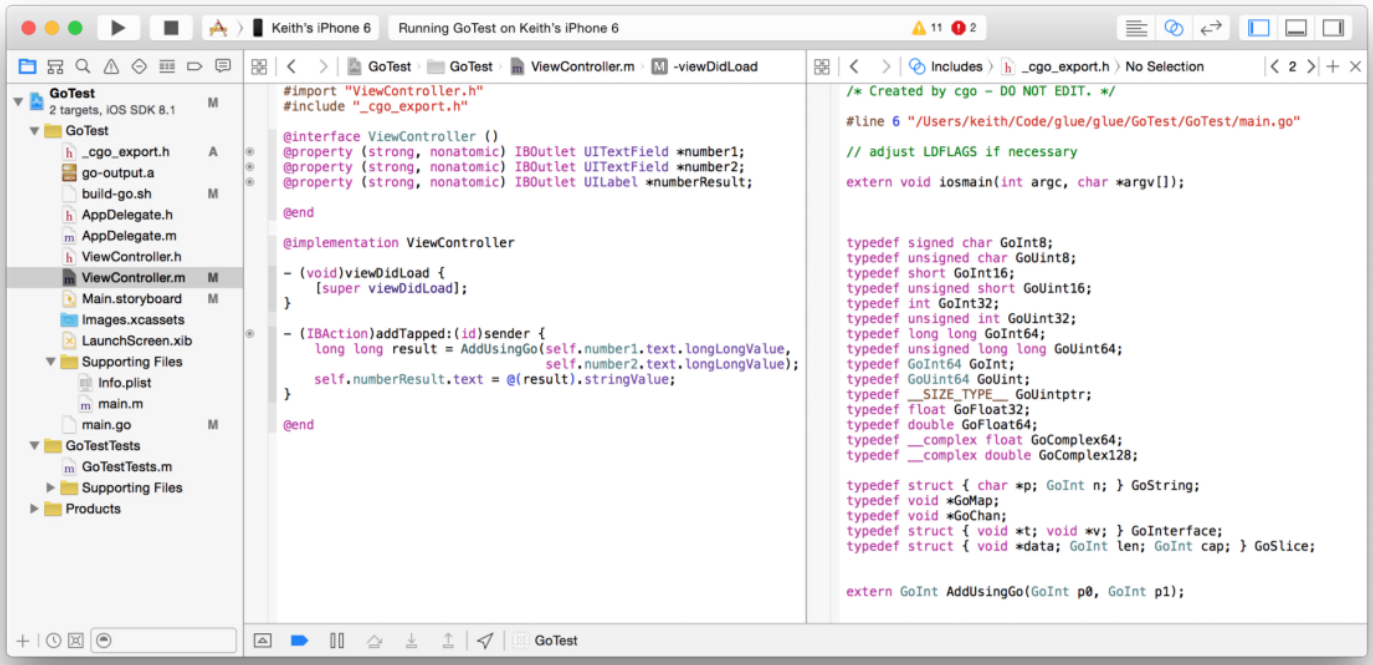
Well, I think they're trying to scare us with that "undefined" comment—and it worked. Undefined behavior is bad, and I'm guessing it's due to the fact that *go build* and Xcode are *both* building object files for our Objective C code.

If we could get *go build* to **just** compile our Go code, and output *_cgo_export.h* and some **.o* files, we'd be in good shape. I imagine Xcode is probably pretty good at building Objective C projects, and I'd like to let it do its thing. And let *go build* do its thing. And then merge those things into a thing.

I think we can generate *_cgo_export.h* pretty easily by adding a *cgo* call to our build script we created in part 2 of this series:



Now we can run the build and add the file to our project. Look—we can now include *_cgo_export.h*!



Removing duplicate symbols

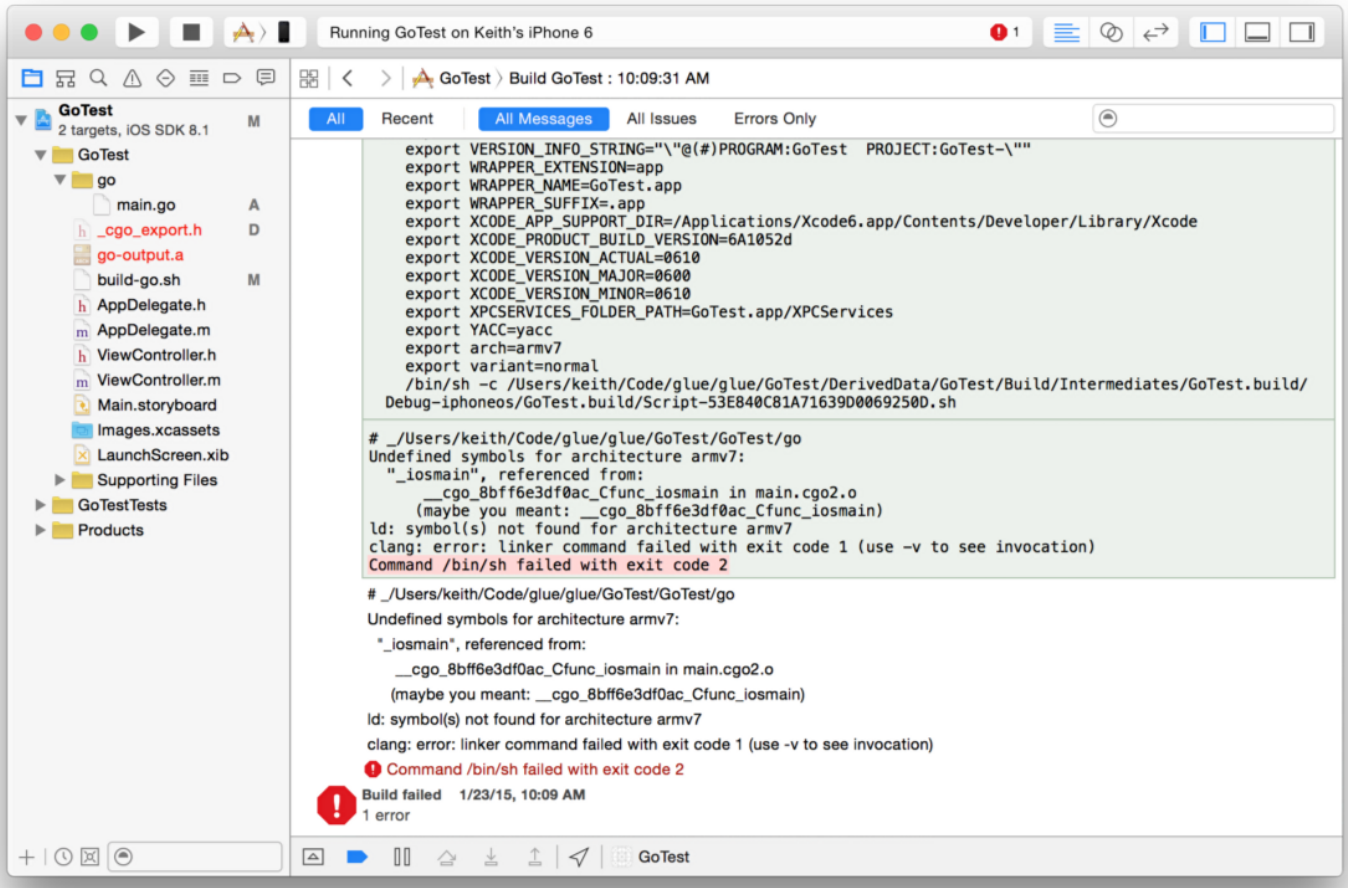
That was easier than I thought, but we're not done yet. When we run our build, we still get that "undefined" warning in the console:

```
objc[4448]: Class AppDelegate is implemented in both
/private/var/mobile/Containers/Bundle/Application/719267F5-B203-
41D8-9729-85AD94AF2FB5/GoTest.app/GoTest and
/private/var/mobile/Containers/Bundle/Application/719267F5-B203-
41D8-9729-85AD94AF2FB5/GoTest.app/GoTest. One of the two will be
used. Which one is undefined.
```

```
objc[4448]: Class ViewController is implemented in both
/private/var/mobile/Containers/Bundle/Application/719267F5-B203-
41D8-9729-85AD94AF2FB5/GoTest.app/GoTest and
/private/var/mobile/Containers/Bundle/Application/719267F5-B203-
41D8-9729-85AD94AF2FB5/GoTest.app/GoTest. One of the two will be
used. Which one is undefined.
```

As I mentioned earlier, I think this is because *go build* is compiling our **.m* files as well as our **.go* files. And I'd like it to stop doing that. I could probably look in *go build*'s temp folder and figure out how to exclude those object files from being packaged into the *go-output.a* static library. But I think there's an even easier way: let's just move the Go code into its own folder, and run *go build* from there! Then it won't even have the **option** to build our **.m* files.

There's only one problem:



It looks like *go build* is having trouble linking the Go binary—the problem is that our Go code calls *iosmain()*, which is declared in *main.m*, which we are explicitly trying to exclude from the Go build process. Hmmmm.

Could we just move *main.m* into the Go folder, and prevent Xcode from compiling it? After all, it's a pretty simple file:

```
#import <UIKit/UIKit.h>
#import “AppDelegate.h”


int iosmain(int argc, char * argv[]) {
  @autoreleasepool {
    NSLog(@”Hello from Objective C! I’m in int iosmain()!”);
    return UIApplicationMain(argc, argv, nil,
        NSStringFromClass([AppDelegate class]));
  }
}
```

Well, we **could** move it, and remove the *UIKit.h* and*AppDelegate.h* imports, and replace *NSStringFromClass([AppDelegate class])* with *@“AppDelegate”* (which is what *NSStringFromClass* returns). But that sounds kind of disgusting, and not sustainable. Eventually we’re probably going to want our Go code to be able to call into our Objective C code for more than just *iosmain()*. At which point Go is going to need to build our whole project.

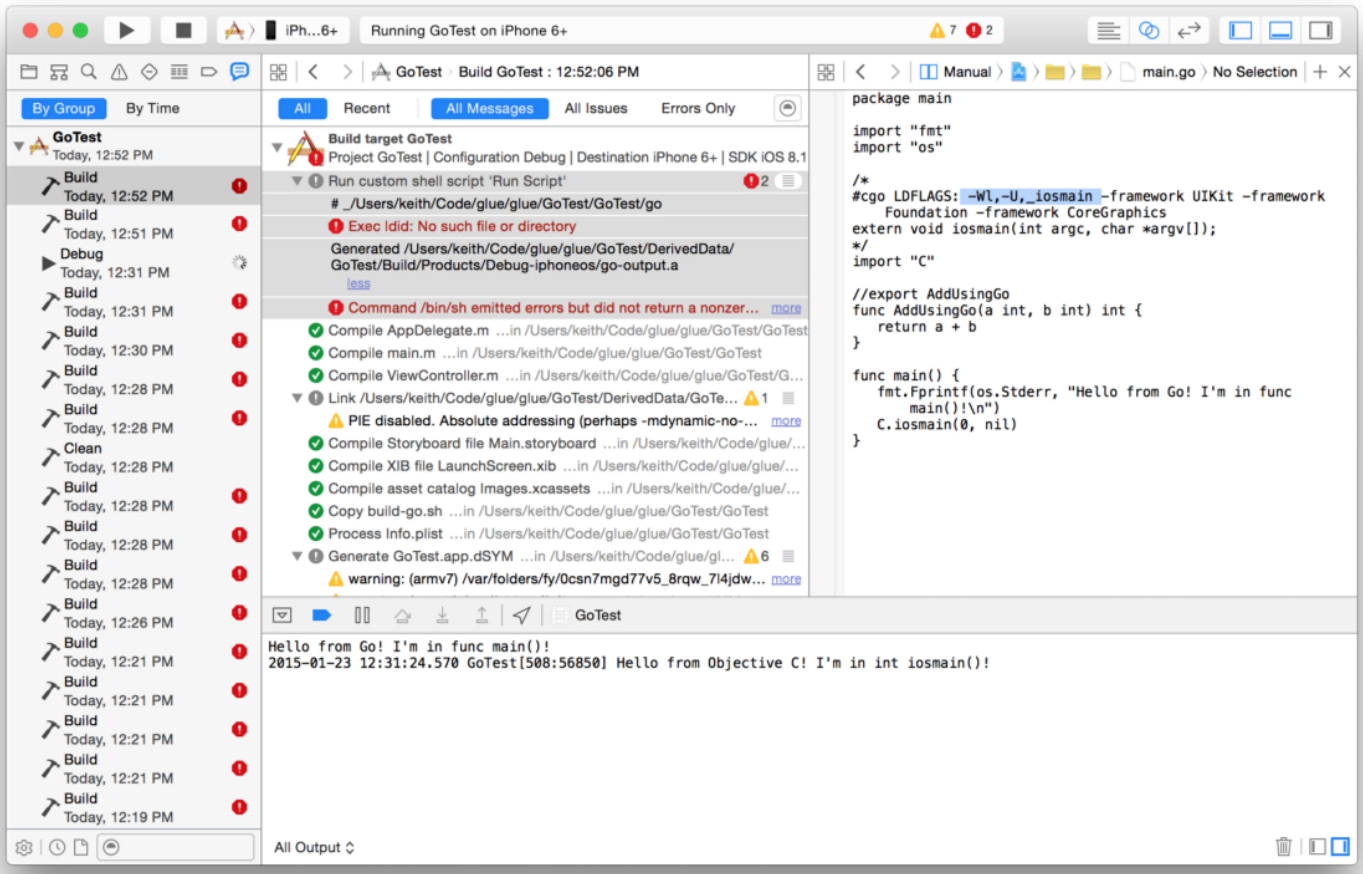
I wish we could just tell Go to skip the linking phase! We don’t need the fully linked binary anyway—we’re just using the **.o* files from go build’s temp folder, and letting Xcode do the linking. I’ve filed a request with the Go team to allow this, but in the meantime, is there a way to make the Go linker ignore this undefined symbol?

Since we know that *ld* is causing the error, and it’s being run by *clang*, it seems like we should be able to pass some kind of command-line flag telling it to ignore the unresolved symbol at link time. Luckily, Go provides several ways to modify the linker settings. Un-luckily, it’s not very clear how these interact. There’s the *-ldflags* option on *go build*, and the *-extldflags* flag, and there’s the *LDFLAGS=…* line in our special *Cgo* comment, the one above *import “C”* in our Go source file. After lots of research, and yes, a bit of pain, I figured out that the only way to pass arguments to *ld (*which is called by *clang (*which is called by *cgo (*which is called by *go build)))* is to add to the *LDFLAGS* above that *import “C”* line.

So what flags do we want to pass? Well, I found a StackOverflow post explaining that we can just pass *-U symbol_name* to *ld*. It’s not ideal—I’d rather not have to manually specify all the symbols to ignore—but it’s better than nothing. There’s only one problem: we don’t call *ld* directly—unfortunately the *LDFLAGS* argument is passed to *clang*, presumably because clang calls out to *ld* itself. Luckily, after a little more hunting, I discovered a

blog post mentioning how to pass arguments through *clang* to *ld*: just pass *-Wl,-U,symbol_name* to *clang* itself, and it will relay to *ld*.

Let's try it!



Whew! What a relief! It builds and runs again! And no more warnings!

Conclusions

You can indeed call Go code from Objective C, as long as you configure the build correctly:

- I recommend having *go build* compile your Go code—and nothing else. Then, theonly outputs we need are the intermediate **.o* files, and the *_cgo_exports.h* header file.

- This means moving Go code into its own subfolder, so *go build* didn't greedily build all the Objective C and C++ code in the project.
- Unfortunately *go build* needs to fully link the binary it's producing, but this is not possible due to *Cgo*'s Go→C calling system. Because *go build* doesn't have access to the object files Xcode is generating, we need to pass a special flag to the *Cgo* linker telling it to ignore undefined symbols.
- *Cgo* uses *clang* to compile and link, which means we need to use special syntax to pass arguments directly to *ld*.

Stay tuned for the rest of the series!