Introduction

First, https://pkg.go.dev/cmd/cgo is the primary cgo documentation.

There is also a good introduction article at https://go.dev/blog/cgo

The basics

If a Go source file imports `"C"`, it is using cgo. The Go file will have access to anything appearing in the comment immediately preceding the line `import "C"`, and will be linked against all other cgo comments in other Go files, and all C files included in the build process.

Note that there must be no blank lines in between the cgo comment and the import statement.

To access a symbol originating from the C side, use the package name `C`. That is, if you want to call the C function `printf()` from Go code, you write `C.printf()`. Since variable argument methods like printf aren't supported yet (issue 975), we will wrap it in the C method "myprint":

```go
package cgoexample

/*
#include <stdio.h>
#include <stdlib.h>

void myprint(char* s) {
    printf("%s\n", s);
}
*/
import "C"

import "unsafe"

func Example() {
    cs := C.CString("Hello from stdio\n")
    C.myprint(cs)
    C.free(unsafe.Pointer(cs))
}
```

Calling Go functions from C

It is possible to call both top-level Go functions and function variables from C code invoked from Go code using cgo.

Global functions

Go makes its functions available to C code through use of a special `//export` comment. Note: you can't define any C functions in preamble if you're using exports.

For example, there are two files, foo.c and foo.go: foo.go contains:

```go
package gocallback

import "fmt"

/*
#include <stdio.h>
extern void ACFunction();
*/
import "C"

//export AGoFunction
func AGoFunction() {
    fmt.Println("AGoFunction()")
}

func Example() {
    C.ACFunction()
}
```

foo.c contains:

```c
#include "_cgo_export.h"
void ACFunction() {
    printf("ACFunction()\n");
    AGoFunction();
}
```

Function variables

The following code shows an example of invoking a Go callback from C code. Because of the pointer passing rules Go code can not pass a function value directly to C. Instead it is necessary to use an indirection. This example uses a registry with a mutex, but there are many other ways to map from a value that can be passed to C to a Go function.

```go
package gocallback

import (
    "fmt"
    "sync"
)
```

```
/*
extern void go_callback_int(int foo, int p1);

// normally you will have to define function or variables
// in another separate C file to avoid the multiple definition
// errors, however, using "static inline" is a nice workaround
// for simple functions like this one.
static inline void CallMyFunction(int foo) {
    go_callback_int(foo, 5);
}
*/
import "C"

//export go_callback_int
func go_callback_int(foo C.int, p1 C.int) {
    fn := lookup(int(foo))
    fn(p1)
}

func MyCallback(x C.int) {
    fmt.Println("callback with", x)
}

func Example() {
    i := register(MyCallback)
    C.CallMyFunction(C.int(i))
    unregister(i)
}

var mu sync.Mutex
var index int
var fns = make(map[int]func(C.int))

func register(fn func(C.int)) int {
    mu.Lock()
    defer mu.Unlock()
    index++
    for fns[index] != nil {
        index++
    }
    fns[index] = fn
    return index
}

func lookup(i int) func(C.int) {
    mu.Lock()
    defer mu.Unlock()
    return fns[i]
}

func unregister(i int) {
    mu.Lock()
    defer mu.Unlock()
    delete(fns, i)
}
```

As of Go 1.17, the `runtime/cgo` package provides runtime/cgo.Handle mechanism and simplifies the above examples to:

```
package main

import (
    "fmt"
    "runtime/cgo"
)

/*
#include <stdint.h>

extern void go_callback_int(uintptr_t h, int p1);
static inline void CallMyFunction(uintptr_t h) {
    go_callback_int(h, 5);
}
*/
import "C"

//export go_callback_int
func go_callback_int(h C.uintptr_t, p1 C.int) {
    fn := cgo.Handle(h).Value().(func(C.int))
    fn(p1)
}

func MyCallback(x C.int) {
    fmt.Println("callback with", x)
}

func main() {
    h := cgo.NewHandle(MyCallback)
    C.CallMyFunction(C.uintptr_t(h))
    h.Delete()
}
```

Function pointer callbacks

C code can call exported Go functions with their explicit name. But if a C-program wants a function pointer, a gateway function has to be written. This is because we can't take the address of a Go function and give that to C-code since the cgo tool will generate a stub in C that should be called. The following example shows how to integrate with C code wanting a function pointer of a give type.

Place these source files under *$GOPATH/src/ccallbacks/*. Compile and run with:

```
$ gcc -c clibrary.c
$ ar cru libclibrary.a clibrary.o
$ go build
$ ./ccallbacks
Go.main(): calling C function with callback to us
C.some_c_func(): calling callback with arg = 2
C.callOnMeGo_cgo(): called with arg = 2
Go.callOnMeGo(): called with arg = 2
C.some_c_func(): callback responded with 3
```

goprog.go

```go
package main

/*
#cgo CFLAGS: -I .
#cgo LDFLAGS: -L . -lclibrary

#include "clibrary.h"

int callOnMeGo_cgo(int in); // Forward declaration.
*/
import "C"

import (
    "fmt"
    "unsafe"
)

//export callOnMeGo
func callOnMeGo(in int) int {
    fmt.Printf("Go.callOnMeGo(): called with arg = %d\n", in)
    return in + 1
}

func main() {
    fmt.Printf("Go.main(): calling C function with callback to us\n")
    C.some_c_func((C.callback_fcn)(unsafe.Pointer(C.callOnMeGo_cgo)))
}
```

cfuncs.go

```
package main

/*

#include <stdio.h>

// The gateway function
int callOnMeGo_cgo(int in)
{
    printf("C.callOnMeGo_cgo(): called with arg = %d\n", in);
    int callOnMeGo(int);
    return callOnMeGo(in);
}
*/
import "C"
```

clibrary.h

```
#ifndef CLIBRARY_H
#define CLIBRARY_H
typedef int (*callback_fcn)(int);
void some_c_func(callback_fcn);
#endif
```

clibrary.c

```
#include <stdio.h>

#include "clibrary.h"

void some_c_func(callback_fcn callback)
{
    int arg = 2;
    printf("C.some_c_func(): calling callback with arg = %d\n", arg);
    int response = callback(2);
    printf("C.some_c_func(): callback responded with %d\n", response);
}
```

Go strings and C strings

Go strings and C strings are different. Go strings are the combination of a length and a pointer to the first character in the string. C strings are just the pointer to the first character, and are terminated by the first instance of the null character, `'\0'`.

Go provides means to go from one to another in the form of the following three functions: * `func C.CString(goString string) *C.char` * `func C.GoString(cString *C.char) string` * `func C.GoStringN(cString *C.char, length C.int) string`

One important thing to remember is that `C.CString()` will allocate a new string of the appropriate length, and return it. That means the C string is not going to be garbage collected and it is up to **you** to free it. A standard way to do this follows.

```
// #include <stdlib.h>
import "C"
import "unsafe"
...
    var cmsg *C.char = C.CString("hi")
    defer C.free(unsafe.Pointer(cmsg))
    // do something with the C string
```

Of course, you aren't required to use `defer` to call `C.free()`. You can free the C string whenever you like, but it is your responsibility to make sure it happens.

Turning C arrays into Go slices

C arrays are typically either null-terminated or have a length kept elsewhere.

Go provides the following function to make a new Go byte slice from a C array:
* `func C.GoBytes(cArray unsafe.Pointer, length C.int) []byte`

To create a Go slice backed by a C array (without copying the original data), one needs to acquire this length at runtime and use a type conversion to a pointer to a very big array and then slice it to the length that you want (also remember to set the cap if you're using Go 1.2 or later), for example (see https://go.dev/play/p/XuC0xqtAIC for a runnable example):

```
import "C"
import "unsafe"
...
        var theCArray *C.YourType = C.getTheArray()
        length := C.getTheArrayLength()
        slice := (*[1 << 28]C.YourType)(unsafe.Pointer(theCArray))[:length:length]
```

With Go 1.17 or later, programs can use `unsafe.Slice` instead, which similarly results in a Go slice backed by a C array:

```
import "C"
import "unsafe"
...
        var theCArray *C.YourType = C.getTheArray()
        length := C.getTheArrayLength()
        slice := unsafe.Slice(theCArray, length) // Go 1.17
```

It is important to keep in mind that the Go garbage collector will not interact with the underlying C array, and that if it is freed from the C side of things, the behavior of any Go code using the slice is nondeterministic.

Common Pitfalls

Struct Alignment Issues

As Go doesn't support packed struct (e.g., structs where maximum alignment is 1 byte), you can't use packed C struct in Go. Even if your program passes compilation, it won't do what you want. To use it, you have to read/write the struct as byte array/slice.

Another problem is that some types has lower alignment requirement than their counterpart in Go, and if that type happens to be aligned in C but not in Go rules, that struct simply can't be represented in Go. An example is this (issue 7560):

```
struct T {
    uint32_t pad;
    complex float x;
};
```

Go's complex64 has an alignment of 8-byte, where as C has only 4-byte (because C treats the complex float internally as a `struct { float real; float imag; }`, not a basic type), this T struct simply doesn't have a Go representation. For this case, if you control the layout of the struct, move the complex float so that it is also aligned to 8-byte is better, and if you're not willing to move it, use this form will force it to align to 8-byte (and waste 4-byte):

```
struct T {
   uint32_t pad;
   __attribute__((align(8))) complex float x;
};
```

However, if you don't control the struct layout, you will have to define accessor C functions for that struct because cgo won't be able to translate that struct into equivalent Go struct.

`//export` and definition in preamble

If a Go source file uses any `//export` directives, then the C code in the comment may only include declarations (`extern int f();`), not definitions (`int f() { return 1; }` or `int n;`). Note: you can use `static inline` trick to work around this restriction for tiny functions defined in the preamble (see above for a complete example).

Windows

In order to use cgo on Windows, you'll also need to first install a gcc compiler (for instance, mingw-w64) and have gcc.exe (etc.) in your PATH environment variable before compiling with cgo will work.

environmental variables

Go os.Getenv() doesn't see variables set by C.setenv()

tests

_test.go files can't use cgo.