The Go Programming Language

Command cgo

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

Using cgo with the go command

To use cgo write normal Go code that imports a pseudo-package "C". The Go code can then refer to types such as C.size_t, variables such as C.stdout, or functions such as C.putchar.

If the import of "C" is immediately preceded by a comment, that comment, called the preamble, is used as a header when compiling the C parts of the package. For example:

```
// #include <stdio.h>
// #include <errno.h>
import "C"
```

The preamble may contain any C code, including function and variable declarations and definitions. These may then be referred to from Go code as though they were defined in the package "C". All names declared in the preamble may be used, even if they start with a lower-case letter. Exception: static variables in the preamble may not be referenced from Go code; static functions are permitted.

See $GOROOT/misc/cgo/stdio and $GOROOT/misc/cgo/gmp for examples. See "C? Go? Cgo!" for an introduction to using cgo: https://golang.org/doc/articles/c_go_cgo.html.

CFLAGS, CPPFLAGS, CXXFLAGS, FFLAGS and LDFLAGS may be defined with pseudo #cgo directives within these comments to tweak the behavior of the C, C++ or Fortran compiler. Values defined in multiple directives are concatenated together. The directive can include a list of build constraints limiting its effect to systems satisfying one of the constraints (see https://golang.org/pkg/go/build/#hdr-Build_Constraints for details about the constraint syntax). For example:

```
// #cgo CFLAGS: -DPNG_DEBUG=1
// #cgo amd64 386 CFLAGS: -DX86=1
// #cgo LDFLAGS: -lpng
// #include <png.h>
import "C"
```

Alternatively, CPPFLAGS and LDFLAGS may be obtained via the pkg-config tool using a '#cgo pkg-config:' directive followed by the package names. For example:

```
// #cgo pkg-config: png cairo
// #include <png.h>
import "C"
```

The default pkg-config tool may be changed by setting the PKG_CONFIG environment variable.

For security reasons, only a limited set of flags are allowed, notably -D, -I, and -l. To allow additional flags, set CGO_CFLAGS_ALLOW to a regular expression matching the new flags. To disallow flags that would otherwise be allowed, set CGO_CFLAGS_DISALLOW to a regular expression matching arguments that must be disallowed. In both cases the regular expression must match a full argument: to allow -mfoo=bar, use CGO_CFLAGS_ALLOW='-mfoo.*', not just CGO_CFLAGS_ALLOW='-mfoo'. Similarly named variables control the allowed CPPFLAGS, CXXFLAGS, FFLAGS, and LDFLAGS.

Also for security reasons, only a limited set of characters are permitted, notably alphanumeric characters and a few symbols, such as '.', that will not be interpreted in unexpected ways. Attempts to use forbidden characters will get a "malformed #cgo argument" error.

When building, the CGO_CFLAGS, CGO_CPPFLAGS, CGO_CXXFLAGS, CGO_FFLAGS and CGO_LDFLAGS environment variables are added to the flags derived from these directives. Package-specific flags should be set using the directives, not the environment variables, so that builds work in unmodified environments. Flags obtained from environment variables are not subject to the security limitations described above.

All the cgo CPPFLAGS and CFLAGS directives in a package are concatenated and used to compile C files in that package. All the CPPFLAGS and CXXFLAGS directives in a package are concatenated and used to compile C++ files in that package. All the CPPFLAGS and FFLAGS directives in a package are concatenated and used to compile Fortran files in that package. All the LDFLAGS directives in any package in the program are concatenated and used at link time. All the pkg-config directives are concatenated and sent to pkg-config simultaneously to add to each appropriate set of command-line flags.

When the cgo directives are parsed, any occurrence of the string ${SRCDIR} will be replaced by the absolute path to the directory containing the source file. This allows pre-compiled static libraries to be included in the package directory and linked properly. For example if package foo is in the directory /go/src/foo:

```
// #cgo LDFLAGS: -L${SRCDIR}/libs -lfoo
```

Will be expanded to:

```
// #cgo LDFLAGS: -L/go/src/foo/libs -lfoo
```

When the Go tool sees that one or more Go files use the special import "C", it will look for other non-Go files in the directory and compile them as part of the Go package. Any .c, .s, or .S files will be compiled with the C compiler. Any .cc, .cpp, or .cxx files will be compiled with the C++ compiler. Any .f, .F, .for or .f90 files will be compiled with the fortran compiler. Any .h, .hh, .hpp, or .hxx files will not be compiled separately, but, if these header files are changed, the package (including its non-Go source files) will be recompiled. Note that changes to files in other directories do not cause the package to be recompiled, so all non-Go source code for the package should be stored in the package directory, not in subdirectories. The default C and C++ compilers may be changed by the CC and CXX environment variables, respectively; those environment variables may include command line options.

The cgo tool is enabled by default for native builds on systems where it is expected to work. It is disabled by default when cross-compiling. You can control this by setting the CGO_ENABLED environment variable when running the go tool: set it to 1 to enable the use of cgo, and to 0 to disable it. The go tool will set the build constraint "cgo" if cgo is enabled. The special import "C" implies the "cgo" build constraint, as though the file also said "// +build cgo". Therefore, if cgo is disabled, files that import "C" will not be built by the go tool. (For more about build constraints see https://golang.org/pkg/go/build/#hdr-Build_Constraints).

When cross-compiling, you must specify a C cross-compiler for cgo to use. You can do this by setting the generic CC_FOR_TARGET or the more specific CC_FOR_${GOOS}_${GOARCH} (for example, CC_FOR_linux_arm) environment variable when building the toolchain using make.bash, or you can set the CC environment variable any time you run the go tool.

The CXX_FOR_TARGET, CXX_FOR_${GOOS}_${GOARCH}, and CXX environment variables work in a similar way for C++ code.

Go references to C

Within the Go file, C's struct field names that are keywords in Go can be accessed by prefixing them with an underscore: if x points at a C struct with a field named "type", x._type accesses the field. C struct fields that cannot be expressed in Go, such as bit fields or misaligned data, are omitted in the Go struct, replaced by appropriate padding to reach the next field or the end of the struct.

The standard C numeric types are available under the names C.char, C.schar (signed char), C.uchar (unsigned char), C.short, C.ushort (unsigned short), C.int, C.uint (unsigned int), C.long, C.ulong (unsigned long), C.longlong (long long), C.ulonglong (unsigned long long), C.float, C.double, C.complexfloat (complex float), and C.complexdouble (complex double). The C type void* is represented by Go's unsafe.Pointer. The C types __int128_t and __uint128_t are represented by [16]byte.

A few special C types which would normally be represented by a pointer type in Go are instead represented by a uintptr. See the Special cases section below.

To access a struct, union, or enum type directly, prefix it with struct_, union_, or enum_, as in C.struct_stat.

The size of any C type T is available as C.sizeof_T, as in C.sizeof_struct_stat.

A C function may be declared in the Go file with a parameter type of the special name _GoString_. This function may be called with an ordinary Go string value. The string length, and a pointer to the string contents, may be accessed by calling the C functions

```
size_t _GoStringLen(_GoString_ s);
const char *_GoStringPtr(_GoString_ s);
```

These functions are only available in the preamble, not in other C files. The C code must not modify the contents of the pointer returned by _GoStringPtr. Note that the string contents may not have a trailing NUL byte.

As Go doesn't have support for C's union type in the general case, C's union types are represented as a Go byte array with the same length.

Go structs cannot embed fields with C types.

Go code cannot refer to zero-sized fields that occur at the end of non-empty C structs. To get the address of such a field (which is the only operation you can do with a zero-sized field) you must take the address of the struct and add the size of the struct.

Cgo translates C types into equivalent unexported Go types. Because the translations are unexported, a Go package should not expose C types in its exported API: a C type used in one Go package is different from the same C type used in another.

Any C function (even void functions) may be called in a multiple assignment context to retrieve both the return value (if any) and the C errno variable as an error (use _ to skip the result value if the function returns void). For example:

```
n, err = C.sqrt(-1)
_, err := C.voidFunc()
var n, err = C.sqrt(1)
```

Calling C function pointers is currently not supported, however you can declare Go variables which hold C function pointers and pass them back and forth between Go and C. C code may call function pointers received from Go. For example:

```
package main

// typedef int (*intFunc) ();
//
// int
// bridge_int_func(intFunc f)
// {
//              return f();
// }
//
// int fortytwo()
// {
//          return 42;
// }
import "C"
import "fmt"

func main() {
        f := C.intFunc(C.fortytwo)
        fmt.Println(int(C.bridge_int_func(f)))
        // Output: 42
}
```

In C, a function argument written as a fixed size array actually requires a pointer to the first element of the array. C compilers are aware of this calling convention and adjust the call accordingly, but Go cannot. In Go, you must pass the pointer to the first element explicitly: C.f(&C.x[0]).

Calling variadic C functions is not supported. It is possible to circumvent this by using a C function wrapper. For example:

```
package main

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

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

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

A few special functions convert between Go and C types by making copies of the data. In pseudo-Go definitions:

```
// Go string to C string
// The C string is allocated in the C heap using malloc.
// It is the caller's responsibility to arrange for it to be
// freed, such as by calling C.free (be sure to include stdlib.h
// if C.free is needed).
func C.CString(string) *C.char

// Go []byte slice to C array
// The C array is allocated in the C heap using malloc.
// It is the caller's responsibility to arrange for it to be
// freed, such as by calling C.free (be sure to include stdlib.h
// if C.free is needed).
func C.CBytes([]byte) unsafe.Pointer

// C string to Go string
func C.GoString(*C.char) string

// C data with explicit length to Go string
func C.GoStringN(*C.char, C.int) string

// C data with explicit length to Go []byte
func C.GoBytes(unsafe.Pointer, C.int) []byte
```

As a special case, C.malloc does not call the C library malloc directly but instead calls a Go helper function that wraps the C library malloc but guarantees never to return nil. If C's malloc indicates out of memory, the helper function crashes the program, like when Go itself runs out of memory. Because C.malloc cannot fail, it has no two-result form that returns errno.

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 {...}
```

```
//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. Go struct types are not supported; use a C struct type. Go array types are not supported; use a C pointer.

Go functions that take arguments of type string may be called with the C type _GoString_, described above. The _GoString_ type will be automatically defined in the preamble. Note that there is no way for C code to create a value of this type; this is only useful for passing string values from Go to C and back to Go.

Using //export in a file places a restriction on the preamble: since it is copied into two different C output files, it must not contain any definitions, only declarations. If a file contains both definitions and declarations, then the two output files will produce duplicate symbols and the linker will fail. To avoid this, definitions must be placed in preambles in other files, or in C source files.

Passing pointers

Go is a garbage collected language, and the garbage collector needs to know the location of every pointer to Go memory. Because of this, there are restrictions on passing pointers between Go and C.

In this section the term Go pointer means a pointer to memory allocated by Go (such as by using the & operator or calling the predefined new function) and the term C pointer means a pointer to memory allocated by C (such as by a call to C.malloc). Whether a pointer is a Go pointer or a C pointer is a dynamic property determined by how the memory was allocated; it has nothing to do with the type of the pointer.

Note that values of some Go types, other than the type's zero value, always include Go pointers. This is true of string, slice, interface, channel, map, and function types. A pointer type may hold a Go pointer or a C pointer. Array and struct types may or may not include Go pointers, depending on the element types. All the discussion below about Go pointers applies not just to pointer types, but also to other types that include Go pointers.

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.

C code may not keep a copy of a Go pointer after the call returns. This includes the _GoString_ type, which, as noted above, includes a Go pointer; _GoString_ values may not be retained by C code.

A Go function called by C code may not return a Go pointer (which implies that it may not return a string, slice, channel, and so forth). A Go function called by C code may take C pointers as arguments, and it may store non-pointer or C pointer data through those pointers, but it may not store a Go pointer in memory pointed to by a C pointer. A Go function called by C code may take a Go pointer as an argument, but it must preserve the property that the Go memory to which it points does not contain any Go pointers.

Go code may not store a Go pointer in C memory. C code may store Go pointers in C memory, subject to the rule above: it must stop storing the Go pointer when the C function returns.

These rules are checked dynamically at runtime. The checking is controlled by the cgocheck setting of the GODEBUG environment variable. The default setting is GODEBUG=cgocheck=1, which implements reasonably cheap dynamic checks. These checks may be disabled entirely using GODEBUG=cgocheck=0. Complete checking of pointer handling, at some cost in run time, is available via GODEBUG=cgocheck=2.

It is possible to defeat this enforcement by using the unsafe package, and of course there is nothing stopping the C code from doing anything it likes. However, programs that break these rules are likely to fail in unexpected and unpredictable ways.

Note: the current implementation has a bug. While Go code is permitted to write nil or a C pointer (but not a Go pointer) to C memory, the current implementation may sometimes cause a runtime error if the contents of the C memory appear to be a Go pointer. Therefore, avoid passing uninitialized C memory to Go code if the Go code is going to store pointer values in it. Zero out the memory in C before passing it to Go.

Special cases

A few special C types which would normally be represented by a pointer type in Go are instead represented by a uintptr. Those include:

1. The *Ref types on Darwin, rooted at CoreFoundation's CFTypeRef type.

2. The object types from Java's JNI interface:

```
jobject
jclass
jthrowable
jstring
jarray
jbooleanArray
jbyteArray
jcharArray
jshortArray
jintArray
jlongArray
jfloatArray
jdoubleArray
jobjectArray
jweak
```

3. The EGLDisplay type from the EGL API.

These types are uintptr on the Go side because they would otherwise confuse the Go garbage collector; they are sometimes not really pointers but data structures encoded in a pointer type. All operations on these types must happen in C. The proper constant to initialize an empty such reference is 0, not nil.

These special cases were introduced in Go 1.10. For auto-updating code from Go 1.9 and earlier, use the cftype or jni rewrites in the Go fix tool:

```
go tool fix -r cftype <pkg>
go tool fix -r jni <pkg>
```

It will replace nil with 0 in the appropriate places.

The EGLDisplay case were introduced in Go 1.12. Use the egl rewrite to auto-update code from Go 1.11 and earlier:

```
go tool fix -r egl <pkg>
```

Using cgo directly

Usage:

```
go tool cgo [cgo options] [-- compiler options] gofiles...
```

Cgo transforms the specified input Go source files into several output Go and C source files.

The compiler options are passed through uninterpreted when invoking the C compiler to compile the C parts of the package.

The following options are available when running cgo directly:

```
-V
        Print cgo version and exit.
-debug-define
        Debugging option. Print #defines.

-debug-gcc
        Debugging option. Trace C compiler execution and output.
-dynimport file
        Write list of symbols imported by file. Write to
        -dynout argument or to standard output. Used by go
        build when building a cgo package.
-dynlinker
        Write dynamic linker as part of -dynimport output.
-dynout file
        Write -dynimport output to file.
-dynpackage package
        Set Go package for -dynimport output.
-exportheader file
        If there are any exported functions, write the
        generated export declarations to file.
        C code can #include this to see the declarations.
-importpath string
        The import path for the Go package. Optional; used for
        nicer comments in the generated files.
-import_runtime_cgo
        If set (which it is by default) import runtime/cgo in
```

```
		generated output.
	-import_syscall
		If set (which it is by default) import syscall in
		generated output.
	-gccgo
		Generate output for the gccgo compiler rather than the
		gc compiler.
	-gccgoprefix prefix
		The -fgo-prefix option to be used with gccgo.
	-gccgopkgpath path
		The -fgo-pkgpath option to be used with gccgo.
	-godefs
		Write out input file in Go syntax replacing C package
		names with real values. Used to generate files in the
		syscall package when bootstrapping a new target.
	-objdir directory
		Put all generated files in directory.
	-srcdir directory
```