Working with Packed C Structs in cgo

Liam Kelly Follow
May 21, 2018 · 7 min read

The Packed Gopher

Introduction

Cgo is amazing for integrating Go with existing code. Often older projects handcuff engineers to earlier design decisions, you can look for COBOL job reqs to see examples. Cgo allowed me to avoid this problem and start modernizing existing code in a segmented manner. There has only been one major problem I have had using cgo, and that is that it could not directly handle packed C structs and the legacy code I was working with was full of them.

This article is meant to be a deep dive into how I was able to get around this problem by just using standard Go libraries. I will first

briefly go over what a C packed struct is and why programmers use them. Next I will go over what happens when cgo tries to handle packed structs. I will then show how to get around cgo's limitation and give examples of manually packing and unpacking packed structs. Finally I will end with a discussion about the drawbacks to my approach and what a more integrated solution might look like.

Why pack structs in the first place?

C struct fields are allocated along memory bounds by default so that memory calls are faster, but this comes at the cost of potentially wasting some memory. For most developers, this is normally an acceptable tradoff; however, there are some cases where structure alignment can greatly affect performance and code complexity. There is amazing article here that explain C memory allocation and alignment in greater detail.

On some occasions a programmer may require that a C structs take up the minimal amount of memory, even if it reduces code speed. When a struct is stored in a way that takes up the least memory it is call 'packed' because there is no wasted padding and the fields are packed together. Compilers will implement this by adding a `#pragma pack(1)` declaration above the structure in question. This is seen more often in embedded C code and/or older networking code. Embedded code uses this to conserve what little memory they have. Old networking code uses it for a simple way to send a structure over the network. The networking code leverages the fact that you can always know where a structure field is in memory, no matter the CPU architecture, when the structure is packed. Packed structure are not that common, but should be looked out for when dealing with legacy code.

Cgo silently removes packed struct fields

Cgo does not handle packed structs correctly. Unfortunately cgo will also not error deterministically when you try to actually use a packed struct. The most common observable symptom is that Go will give a compiler error saying that one of the structures fields in not there. Some of the packed fields in the struct will work normally, so this bug can go hidden for some time.

The common ‘cgo packed struct’ error is displayed in the example and results below. There are two near identical C Structure declared in the cgo preamble, the only difference is that the `packed` structure is packed (`#pragma pack(1)`). The example simply creates an instance of each and then prints their fields and values.

Example

```
package main

/*
typedef struct{
        unsigned char a;
        char b;
        int c;
        unsigned int d;
        char e[10];
}unpacked;

#pragma pack(1)
typedef struct{
        unsigned char a;
        char b;
        int c;
        unsigned int d;
        char e[10];
}packed;

*/
import "C"
import (
        "fmt"

```

Results

```
Printing the structure of the unpacked struct
(main._Ctype_struct___1) {
 a: (main._Ctype_uchar) 0,
 b: (main._Ctype_char) 0,
```

```
 _: ([2]uint8) (len=2 cap=2) {
  00000000  00 00
|..|
 },
 c: (main._Ctype_int) 0,
 d: (main._Ctype_uint) 0,
 e: ([10]main._Ctype_char) (len=10 cap=10) {
  00000000  00 00 00 00 00 00 00 00  00 00
|..........|
 },
 _: ([2]uint8) (len=2 cap=2) {
  00000000  00 00
|..|
 }
}
Printing the structure of the packed struct
(main._Ctype_struct___0) {
 a: (main._Ctype_uchar) 0,
 b: (main._Ctype_char) 0,
 _: ([4]uint8) (len=4 cap=4) {
  00000000  00 00 00 00
|....|
 },
 _: ([4]uint8) (len=4 cap=4) {
  00000000  00 00 00 00
|....|
 },
 e: ([10]main._Ctype_char) (len=10 cap=10) {
  00000000  00 00 00 00 00 00 00 00  00 00
|..........|
 }
}
```

The results of this snippet shows us the error. The unpacked structure has all the fields shown (`a-e`) and also two empty fields (`_`) which are both byte arrays. The empty fields are used to represent the padding in the unpacked struct. The packed structure has had fields `c` and `d` replaced with empty fields with byte arrays the size of the type that should be there (sizeof(int)=4 bytes). If the replaced fields are referenced the code with not compile, but `a` , `b` , and `e` can be access normally.

Unpacking the struct manually

Cgo cannot handle packed structs directly, but that doesn't mean they cannot be handled. The cgo wiki gives a general guidance on how to deal with them:

> *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 you*

> *program passes compilation, it won't do what you want. To use it, you have to read/write the struct as byte array/slice.*

To read the packed struct as a byte slice we can use one of cgo's special functions: `GoBytes` . This function reads memory handled by C and places it into a Go byte slice. In order to use it we need to first get the C pointer to the struct. Then we need to convert it to a `unsafe.Pointer` . Finally we need to get the length of the memory we want to read by using another special cgo function: `C.sizeof_<c type>` where `<c type>` is replaced with the C type name. This function is meant to mimic the C `sizeof` operator. We use this function to tell `GoBytes` the size of the memory read. With all this information we can extract the binary data into a Go byte slice.

With the binary data of the C packed struct now in a Go byte slice, we need a structure to unpack it into. I found the most convenient way of doing this is creating a mirror Go structure. This is fairly straight forward for basic C structs, just replace the fields types with the Go version.

The final step is unpacking the Go byte slice into the mirrored Go struct. This is done mainly by using the `binary` package. The `binary.Read` function allows for an easy way to sequentially read the byte slice, via a buffer, into the Go structure. The only reason we can use this technique is because the C structure is packed, if it was not then the padding would need to be handled.

Example

```
package main

/*
#include "stdio.h"
#pragma pack(1)
typedef struct{
        unsigned char a;
        char b;
        int c;
        unsigned int d;
        char e[10];
}packed;

packed PackedInit(){
        packed p;
        p.a = 1;
        p.b = 2;
        p.c = 3;
        p.d = 4;
        p.e[0] = 'T';
        p.e[1] = 'E';
        p.e[2] = 'S';
        p.e[3] = 'T';
        p.e[4] = '1';
        p.e[5] = '2';
        p.e[6] = '3';
        p.e[7] = '\0';
        p.e[8] = '\0';
        p.e[9] = '\0';
        return p;
}

*/
import "C"
import (
        "bytes"
        "encoding/binary"
        "fmt"
        "unsafe"
)

//C.Packed is the same size of the packed struct in
```

```
42  //GoPack is the go version of the c packed structure
43  type GoPack struct {
```

Result

```
a: 1
b: 2
c: 3
d: 4
e: TEST123
```

Packing the structure manually

Packing the structure is just the reverse of unpacking expect with an additional final step. The mirrored Go structure can be packed the same way it was unpacked using a buffer and `binary.Write`. This produces a byte slice representing the packed structure, and the last step is to write the binary data to memory allocated for the C packed type. This is the tricky part because while cgo has a function to go from memory to a byte slice (`GoBytes`), there is no builtin function to go the other way.

To convert a byte slice to active memory we cast the pointer to the C packed struct to something that looks very odd : `(*[1<<20]C.uchar)(unsafe.Pointer(<c type>))` . This right part converts the pointer to the C type to an `unsafe.Pointer` that allows for it to be cast to any other type. The second part casts the unsafe pointer to a very large,(`1<<20 ==` 1048576) slice of C type bytes. This allows for Go to manipulate memory in that entire giant span of memory. You can easily access memory you shouldn't, but as long as you know what parts of memory have been allocated for the C struct everything should be fine. With the ability to access the C struct memory we copy the byte slice contents into it. After the memory has been written successfully, you should be able to use the packed C type with C functions correctly.

Example

```
package main

/*
#include "stdio.h"
#pragma pack(1)
typedef struct{
        unsigned char a;
        char b;
        int c;
        unsigned int d;
        char e[10];
}packed;

void PrintPacked(packed p){
        printf("From C\na:%d\nb:%d\nc:%d\nd:%d\ne:%s\n
}

*/
import "C"
import (
        "bytes"
        "encoding/binary"
        "unsafe"
)

//GoPack is the go version of the c packed structure
type GoPack struct {
        a uint8
        b int8
        c int32
        d uint32
        e [10]uint8
}

//Pack Produces a packed version of the go struct
func (g *GoPack) Pack(out unsafe.Pointer) {
        buf := &bytes.Buffer{}
        binary.Write(buf, binary.LittleEndian, g.a)
        binary.Write(buf, binary.LittleEndian, g.b)
```

Result

```
From C
a:1
b:2
c:3
d:4
e:TEST123
```

Discussion

While this method works, it is far from perfect. Firstly it is not easy to automate because developers must manually create the mirrored Go type. Also the Go mirrored type may be harder to implement if the original C structure has more complex fields. Secondly the packing and unpacking methods are fairly redundant and could be reflectively implemented . Finally the packing method opens up the possibility of writing to unauthorized memory, which should make every developer a little nervous. The approach works, but with the exception of the second point, there are no existing way to get around these drawbacks.

After thinking about these drawbacks, I think that there are two practical additions to cgo that would greatly aid my approach. The first is to not explicitly error on C packed structs. At a minimum a warning at compile time warning be generated even if the warnings are not enabled by default. This would prevent developers from unknowingly using packed structs and having a hidden bug in their code. The second addition would be adding a function that does the reverse of `GoBytes` . I got around this, but having a built in function would be safer and could help prevent a programmer accidentally writing to unauthorized memory. I feel like these two additions do not alter cgo in any significant way and could make it safer in the long run.

A more ideal solution would involve modifying cgo to allow for general C AST parsing and code generation. This could allow for the generation of pure Go mirror types instead of cgo types, as well as built in functions to convert back to C types (just like we did with Pack and Unpack). Cgo is pretty complex (generating DWARF files then parsing them) and this does not seem like an easy addition, but the effort may be worth it if it helps the greater developer community adopt Go.