CHAPTER 16

Here Be Dragons: Reflect, Unsafe, and Cgo

The edges of the known world are scary. Ancient maps would fill in the unexplored areas with pictures of dragons and lions. In the previous sections, I have emphasized that Go is a safe language, with typed variables to make clear what sort of data you are using and garbage collection to manage memory. Even the pointers are tame; you can't abuse them in the way that C and C++ do.

All those things are true, and for the vast majority of the Go code that you'll write, you can be assured that the Go runtime will protect you. But there are escape hatches. Sometimes your Go programs need to venture out into less defined areas. In this chapter, you're going to look at how to handle situations that can't be solved with normal Go code. For example, when the type of the data can't be determined at compile time, you can use the reflection support in the `reflect` package to interact with and even construct data. When you need to take advantage of the memory layout of data types in Go, you can use the `unsafe` package. And if there is functionality that can be provided only by libraries written in C, you can call into C code with `cgo`.

You might be wondering why these advanced concepts appear in a book targeted at those new to Go. There are two reasons. First, developers searching for a solution to a problem sometimes discover (and copy and paste) techniques they don't fully understand. It's best to know a bit about advanced techniques that can cause problems before you add them to your codebase. Second, these tools are fun. Because they allow you to do things that aren't normally possible with Go, it feels a bit exciting to play with them and see what you can do.

Reflection Lets You Work with Types at Runtime

One of the things that people who use Go like about it is that it is a statically typed language. Most of the time, declaring variables, types, and functions in Go is pretty straightforward. When you need a type, a variable, or a function, you define it:

```
type Foo struct {
  A int
  B string
}

var x Foo

func DoSomething(f Foo) {
  fmt.Println(f.A, f.B)
}
```

You use types to represent the data structures you know you need when you write your programs. Since types are a core part of Go, the compiler uses them to make sure that your code is correct. But sometimes, relying on only compilation-time information is a limitation. You might need to work with variables at runtime using information that didn't exist when the program was written. Maybe you're trying to map data from a file or network request into a variable. In those situations, you need to use *reflection*. Reflection allows you to examine types at runtime. It also provides the ability to examine, modify, and create variables, functions, and structs at runtime.

This leads to the question of when this functionality is needed. If you look at the Go standard library, you can get an idea. Its uses fall into one of a few general categories:

- Reading and writing from a database. The `database/sql` package uses reflection to send records to databases and read data back.
- Go's built-in templating libraries, `text/template` and `html/template`, use reflection to process the values that are passed to the templates.
- The `fmt` package uses reflection heavily, as all those calls to `fmt.Println` and friends rely on reflection to detect the type of the provided parameters.
- The `errors` package uses reflection to implement `errors.Is` and `errors.As`.
- The `sort` package uses reflection to implement functions that sort and evaluate slices of any type: `sort.Slice`, `sort.SliceStable`, and `sort.SliceIsSorted`.
- The last main usage of reflection in the Go standard library is for marshaling and unmarshaling data into JSON and XML, along with the other data formats defined in the various `encoding` packages. Struct tags (which I will talk about soon) are accessed via reflection, and the fields in structs are read and written using reflection as well.

Most of these examples have one thing in common: they involve accessing and formatting data that is being imported into or exported out of a Go program. You'll often see reflection used at the boundaries between your program and the outside world.

As you take a look at the techniques you can implement with reflection, keep in mind that this power has a price. Using reflection is quite a bit slower than performing the identical operation without it. I'll talk more about this in "Use Reflection Only if It's Worthwhile" on page 424. Even more importantly, code that uses reflection is both more fragile and more verbose. Many functions and methods in the `reflect` package panic when passed the wrong type of data. Be sure to leave comments in your code to explain what you are doing so that it's clear to future reviewers (including yourself).

Another use of the `reflect` package in the Go standard library is testing. In "Slices" on page 39, I mentioned a function that you can find in the `reflect` package called `DeepEqual`. It's in the `reflect` package because it takes advantage of reflection to do its work. The `reflect.DeepEqual` function checks whether two values are "deeply equal" to each other. This is a more thorough comparison than what you get if you use `==` to compare two things, and it's used in the standard library as a way to validate test results. It can also compare things that can't be compared using `==`, like slices and maps. Most of the time, you don't need `DeepEqual`. Since the release of Go 1.21, it's faster to use `slices.Equal` and `maps.Equal` to check for equality in slices and maps.

Types, Kinds, and Values

Now that you know what reflection is and when you might need it, let's see how it works. The `reflect` package in the standard library is the home for the types and functions that implement reflection in Go. Reflection is built around three core concepts: types, kinds, and values.

Types and kinds

A *type* is exactly what it sounds like. It defines the properties of a variable, what it can hold, and how you can interact with it. With reflection, you are able to query a type to find out about these properties using code.

You get the reflection representation of the type of a variable with the `TypeOf` function in the `reflect` package:

```
vType := reflect.TypeOf(v)
```

The `reflect.TypeOf` function returns a value of type `reflect.Type`, which represents the type of the variable passed into the `TypeOf` function. The `reflect.Type`

type defines methods with information about a variable's type. I can't cover all the methods, but here are a few.

The `Name` method returns, not surprisingly, the name of the type. Here is a quick example:

```
var x int
xt := reflect.TypeOf(x)
fmt.Println(xt.Name())     // returns int
f := Foo{}
ft := reflect.TypeOf(f)
fmt.Println(ft.Name())     // returns Foo
xpt := reflect.TypeOf(&x)
fmt.Println(xpt.Name())    // returns an empty string
```

You start with a variable `x` of type `int`. You pass it to `reflect.TypeOf` and get back a `reflect.Type` instance. For primitive types like `int`, `Name` returns the name of the type, in this case the string "int" for your `int`. For a struct, the name of the struct is returned. Some types, like a slice or a pointer, don't have names; in those cases, `Name` returns an empty string.

The `Kind` method on `reflect.Type` returns a value of type `reflect.Kind`, which is a constant that says what the type is made of—a slice, a map, a pointer, a struct, an interface, a string, an array, a function, an int, or some other primitive type. The difference between the kind and the type can be tricky to understand. Remember this rule: if you define a struct named Foo, the kind is `reflect.Struct` and the type is "Foo."

The kind is very important. One thing to be aware of when using reflection is that almost everything in the `reflect` package assumes that you know what you are doing. Some of the methods defined on `reflect.Type` and other types in the `reflect` package make sense for only certain kinds. For example, there's a method on `reflect.Type` called `NumIn`. If your `reflect.Type` instance represents a function, it returns the number of input parameters for the function. If your `reflect.Type` instance isn't a function, calling `NumIn` will panic your program.

In general, if you call a method that doesn't make sense for the kind of the type, the method call panics. Always remember to use the kind of the reflected type to know which methods will work and which ones will panic.

Another important method on `reflect.Type` is `Elem`. Some types in Go have references to other types and `Elem` is how to find out the contained type. For example, let's use `reflect.TypeOf` on a pointer to an `int`:

```
var x int
xpt := reflect.TypeOf(&x)
fmt.Println(xpt.Name())        // returns an empty string
fmt.Println(xpt.Kind())        // returns reflect.Pointer
fmt.Println(xpt.Elem().Name()) // returns "int"
fmt.Println(xpt.Elem().Kind()) // returns reflect.Int
```

That gives you a `reflect.Type` instance with a blank name and a kind of `reflect.Pointer`. When the `reflect.Type` represents a pointer, `Elem` returns the `reflect.Type` for the type the pointer points to. In this case, the `Name` method returns "int," and `Kind` returns `reflect.Int`. The `Elem` method also works for slices, maps, channels, and arrays.

There are methods on `reflect.Type` for reflecting on structs. Use the `NumField` method to get the number of fields in the struct, and get the fields in a struct by index with the `Field` method. That returns each field's structure described in a `reflect.StructField`, which has the name, order, type, and struct tags on a field. Let's look at a quick example, which you can run on The Go Playground or in the *sample_code/struct_tag* directory in the Chapter 16 repository:

```
type Foo struct {
    A int    `myTag:"value"`
    B string `myTag:"value2"`
}

var f Foo
ft := reflect.TypeOf(f)
for i := 0; i < ft.NumField(); i++ {
    curField := ft.Field(i)
    fmt.Println(curField.Name, curField.Type.Name(),
        curField.Tag.Get("myTag"))
}
```

You create an instance of type `Foo` and use `reflect.TypeOf` to get the `reflect.Type` for `f`. Next you use the `NumField` method to set up a `for` loop to get the index of each field in `f`. Then you use the `Field` method to get the `reflect.StructField` struct that represents the field, and then you can use the fields on `reflect.StructField` to get more information about the field. This code prints out the following:

```
A int value
B string value2
```

There are many more methods in `reflect.Type`, but they all follow the same pattern, allowing you to access the information that describes the type of a variable. You can look at the `reflect.Type` documentation in the standard library for more information.

Values

In addition to examining the types of variables, you can also use reflection to read a variable's value, set its value, or create a new value from scratch.

You use the `reflect.ValueOf` function to create a `reflect.Value` instance that represents the value of a variable:

```
vValue := reflect.ValueOf(v)
```

Since every variable in Go has a type, `reflect.Value` has a method called `Type` that returns the `reflect.Type` of the `reflect.Value`. There's also a `Kind` method, just as there is on `reflect.Type`.

Just as `reflect.Type` has methods for finding out information about the type of a variable, `reflect.Value` has methods for finding out information about the value of a variable. I'm not going to cover all of them, but let's take a look at how to use a `reflect.Value` to get the value of the variable.

I'll start by demonstrating how to read your values back out of a `reflect.Value`. The `Interface` method returns the value of the variable as `any`. When you put the value returned by `Interface` into a variable, you have to use a type assertion to get back to a usable type:

```
s := []string{"a", "b", "c"}
sv := reflect.ValueOf(s)        // sv is of type reflect.Value
s2 := sv.Interface().([]string) // s2 is of type []string
```

While `Interface` can be called for `reflect.Value` instances that contain values of any kind, you can use special case methods if the kind of the variable is one of the built-in, primitive types: `Bool`, `Complex`, `Int`, `Uint`, `Float`, and `String`. There's also a `Bytes` method that works if the type of the variable is a slice of bytes. If you use a method that doesn't match the type of the `reflect.Value`, your code will panic. If you are unsure about the type of the `Value`, methods can preemptively check: the `CanComplex`, `CanFloat`, `CanInt`, and `CanUint` methods validate that the `Value` is of one of the numeric types, and `CanConvert` checks for other types.

You can use reflection to set the value of a variable as well, but it's a three-step process.

First, you pass a pointer to the variable into `reflect.ValueOf`. This returns a `reflect.Value` that represents the pointer:

```
i := 10
iv := reflect.ValueOf(&i)
```

Next, you need to get to the actual value to set it. You use the `Elem` method on `reflect.Value` to get to the value pointed to by the pointer that was passed into `reflect.ValueOf`. Just as `Elem` on `reflect.Type` returns the type that's pointed to by

a containing type, `Elem` on `reflect.Value` returns the value that's pointed to by a pointer or the value that's stored in an interface:

```
ivv := iv.Elem()
```

Finally, you get to the actual method that's used to set the value. Just as there are special-case methods for reading primitive types, there are special-case methods for setting primitive types: `SetBool`, `SetInt`, `SetFloat`, `SetString`, and `SetUint`. In the example, calling `ivv.SetInt(20)` changes the value of `i`. If you print out `i` now, you will get 20:

```
ivv.SetInt(20)
fmt.Println(i) // prints 20
```

For all other types, you need to use the `Set` method, which takes a variable of type `reflect.Value`. The value that you are setting it to doesn't need to be a pointer, because you are just reading this value, not changing it. And just as you can use `Interface()` to read primitive types, you can use `Set` to write primitive types.

The reason you need to pass a pointer to `reflect.ValueOf` to change the value of the input parameter is that it is just like any other function in Go. As I discussed in "Pointers Indicate Mutable Parameters" on page 125, you use a parameter of a pointer type to indicate that you want to modify the value of the parameter. When you modify the value, you dereference the pointer and then set the value. The following two functions follow the same process:

```
func changeInt(i *int) {
    *i = 20
}

func changeIntReflect(i *int) {
    iv := reflect.ValueOf(i)
    iv.Elem().SetInt(20)
}
```

Attempting to set a `reflect.Value` to a value of the wrong type will produce a panic.

If you don't pass a pointer to a variable to `reflect.ValueOf`, you can still read the value of the variable using reflection. But if you try to use any of the methods that can change the value of a variable, the method calls will (not surprisingly) panic. The `CanSet` method on `reflect.Value` will tell you if calling `Set` will produce a panic.

Make New Values

Before you learn how to best use reflection, there's one more thing to cover: how to create a value. The `reflect.New` function is the reflection analogue of the `new`

function. It takes in a `reflect.Type` and returns a `reflect.Value` that's a pointer to a `reflect.Value` of the specified type. Since it's a pointer, you can modify it and then assign the modified value to a variable by using the `Interface` method.

Just as `reflect.New` creates a pointer to a scalar type, you can also use reflection to do the same thing as the `make` keyword with the following functions:

```
func MakeChan(typ Type, buffer int) Value

func MakeMap(typ Type) Value

func MakeMapWithSize(typ Type, n int) Value

func MakeSlice(typ Type, len, cap int) Value
```

Each of these functions takes in a `reflect.Type` that represents the compound type, not the contained type.

You must always start from a value when constructing a `reflect.Type`. However, a trick lets you create a variable to represent a `reflect.Type` if you don't have a value handy:

```
var stringType = reflect.TypeOf((*string)(nil)).Elem()

var stringSliceType = reflect.TypeOf([]string(nil))
```

The variable `stringType` contains a `reflect.Type` that represents a `string`, and the variable `stringSliceType` contains a `reflect.Type` that represents a `[]string`. That first line can take a bit of effort to decode. What you are doing is converting `nil` to a pointer to `string`, using `reflect.TypeOf` to make a `reflect.Type` of that pointer type, and then calling `Elem` on that pointer's `reflect.Type` to get the underlying type. You have to put `*string` in parentheses because of the Go order of operations; without the parentheses, the compiler thinks that you are converting `nil` to `string`, which is illegal.

For the `stringSliceType`, it's a bit simpler since `nil` is a valid value for a slice. All you have to do is type conversion of `nil` to a `[]string` and pass that to `reflect.Type`.

Now that you have these types, you can see how to use `reflect.New` and `reflect.MakeSlice`:

```
ssv := reflect.MakeSlice(stringSliceType, 0, 10)

sv := reflect.New(stringType).Elem()
sv.SetString("hello")

ssv = reflect.Append(ssv, sv)
```

```
ss := ssv.Interface().([]string)
fmt.Println(ss) // prints [hello]
```

You can try out this code for yourself on The Go Playground or in the *sample_code/reflect_string_slice* directory in the Chapter 16 repository.

Use Reflection to Check If an Interface's Value Is nil

As I talked about in "Interfaces and nil" on page 164, if a `nil` variable of a concrete type is assigned to a variable of an interface type, the variable of the interface type is not `nil`. This is because a type is associated with the interface variable. If you want to check whether the value associated with an interface is `nil`, you can do so with reflection by using two methods—`IsValid` and `IsNil`:

```
func hasNoValue(i any) bool {
    iv := reflect.ValueOf(i)
    if !iv.IsValid() {
        return true
    }
    switch iv.Kind() {
    case reflect.Pointer, reflect.Slice, reflect.Map, reflect.Func,
         reflect.Interface:
        return iv.IsNil()
    default:
        return false
    }
}
```

The `IsValid` method returns `true` if `reflect.Value` holds anything other than a `nil` interface. You need to check this first because calling any other method on `reflect.Value` will (unsurprisingly) panic if `IsValid` is `false`. The `IsNil` method returns `true` if the value of the `reflect.Value` is `nil`, but it can be called only if the `reflect.Kind` is something that *can* be `nil`. If you call it on a type whose zero value isn't `nil`, it (you guessed it) panics.

You can see this function in use on The Go Playground or in the *sample_code/no_value* directory in the Chapter 16 repository.

Even though it is possible to detect an interface with a `nil` value, strive to write your code so that it performs correctly even when the value associated with an interface is `nil`. Reserve this code for situations where you have no other options.

Use Reflection to Write a Data Marshaler

As mentioned earlier, reflection is what the standard library uses to implement marshaling and unmarshaling. Let's see how it's done by building a data marshaler for ourselves. Go provides the `csv.NewReader` and `csv.NewWriter` functions to read a CSV file into a slice of slice of strings and to write a slice of slice of strings out to a

CSV file, but nothing in the standard library can automatically map that data to the fields in a struct. The following code is going to add that missing functionality.

The examples here have been cut down a bit to fit, reducing the number of supported types. You can find the complete code on The Go Playground or in the *sample_code/csv* directory in the Chapter 16 repository.

You'll start by defining your API. As with other marshalers, you'll define a struct tag that specifies the name of a field in the data to map it to a field in a struct:

```
type MyData struct {
    Name   string `csv:"name"`
    Age    int    `csv:"age"`
    HasPet bool   `csv:"has_pet"`
}
```

The public API consists of two functions:

```
// Unmarshal maps all of the rows of data in a slice of slice of strings
// into a slice of structs.
// The first row is assumed to be the header with the column names.
func Unmarshal(data [][]string, v any) error

// Marshal maps all of the structs in a slice of structs to a slice of slice
// of strings.
// The first row written is the header with the column names.
func Marshal(v any) ([][]string, error)
```

You'll start with `Marshal`, writing the function and then looking at the two helper functions it uses:

```
func Marshal(v any) ([][]string, error) {
    sliceVal := reflect.ValueOf(v)
    if sliceVal.Kind() != reflect.Slice {
        return nil, errors.New("must be a slice of structs")
    }
    structType := sliceVal.Type().Elem()
    if structType.Kind() != reflect.Struct {
        return nil, errors.New("must be a slice of structs")
    }
    var out [][]string
    header := marshalHeader(structType)
    out = append(out, header)
    for i := 0; i < sliceVal.Len(); i++ {
        row, err := marshalOne(sliceVal.Index(i))
        if err != nil {
            return nil, err
        }
        out = append(out, row)
    }
```

```
    return out, nil
}
```

Since you can marshal a struct of any type, you need to use a parameter of type `any`. This isn't a pointer to a slice of structs, because you are only reading from your slice, not modifying it.

The first row of your CSV is going to be the header with the column names, so you get those column names from the struct tags on fields in the struct's type. You use the `Type` method to get the `reflect.Type` of the slice from the `reflect.Value`, and then call the `Elem` method to get the `reflect.Type` of the elements of the slice. You then pass this to `marshalHeader` and append the response to your output.

Next, you iterate through each element in the struct slice using reflection, passing the `reflect.Value` of each element to `marshalOne`, appending the result to your output. When you finish iterating, you return your slice of slice of `string`.

Look at the implementation of your first helper function, `marshalHeader`:

```
func marshalHeader(vt reflect.Type) []string {
    var row []string
    for i := 0; i < vt.NumField(); i++ {
        field := vt.Field(i)
        if curTag, ok := field.Tag.Lookup("csv"); ok {
            row = append(row, curTag)
        }
    }
    return row
}
```

This function simply loops over the fields of the `reflect.Type`, reads the `csv` tag on each field, appends it into a `string` slice, and returns the slice.

The second helper function is `marshalOne`:

```
func marshalOne(vv reflect.Value) ([]string, error) {
    var row []string
    vt := vv.Type()
    for i := 0; i < vv.NumField(); i++ {
        fieldVal := vv.Field(i)
        if _, ok := vt.Field(i).Tag.Lookup("csv"); !ok {
            continue
        }
        switch fieldVal.Kind() {
        case reflect.Int:
            row = append(row, strconv.FormatInt(fieldVal.Int(), 10))
        case reflect.String:
            row = append(row, fieldVal.String())
        case reflect.Bool:
            row = append(row, strconv.FormatBool(fieldVal.Bool()))
        default:
            return nil, fmt.Errorf("cannot handle field of kind %v",
```

```
                                    fieldVal.Kind())
        }
    }
    return row, nil
}
```

This function takes in a `reflect.Value` and returns a `string` slice. You create the `string` slice, and for each field in the struct, you switch on its `reflect.Kind` to determine how to convert it to a `string`, and append it to the output.

Your simple marshaler is now complete. Let's see what you have to do to unmarshal:

```
func Unmarshal(data [][]string, v any) error {
    sliceValPointer := reflect.ValueOf(v)
    if sliceValPointer.Kind() != reflect.Pointer {
        return errors.New("must be a pointer to a slice of structs")
    }
    sliceVal := sliceValPointer.Elem()
    if sliceVal.Kind() != reflect.Slice {
        return errors.New("must be a pointer to a slice of structs")
    }
    structType := sliceVal.Type().Elem()
    if structType.Kind() != reflect.Struct {
        return errors.New("must be a pointer to a slice of structs")
    }

    // assume the first row is a header
    header := data[0]
    namePos := make(map[string]int, len(header))
    for i, name := range header {
        namePos[name] = i
    }

    for _, row := range data[1:] {
        newVal := reflect.New(structType).Elem()
        err := unmarshalOne(row, namePos, newVal)
        if err != nil {
            return err
        }
        sliceVal.Set(reflect.Append(sliceVal, newVal))
    }
    return nil
}
```

Since you are copying data into a slice of any kind of struct, you need to use a parameter of type `any`. Furthermore, because you are modifying the value stored in this parameter, you *must* pass in a pointer to a slice of structs. The `Unmarshal` function converts that slice of structs pointer to a `reflect.Value`, then gets the underlying slice, and then gets the type of the structs in the underlying slice.

As I said earlier, the code assumes that the first row of data is a header with the names of the columns. You use this information to build up a map, so you can associate the `csv` struct tag value with the correct data element.

You then loop through all the remaining `string` slices, creating a new `reflect.Value` using the `reflect.Type` of the struct, call `unmarshalOne` to copy the data in the current `string` slice into the struct, and then add the struct to your slice. After iterating through all the rows of data, you return.

All that remains is looking at the implementation of `unmarshalOne`:

```
func unmarshalOne(row []string, namePos map[string]int, vv reflect.Value) error {
    vt := vv.Type()
    for i := 0; i < vv.NumField(); i++ {
        typeField := vt.Field(i)
        pos, ok := namePos[typeField.Tag.Get("csv")]
        if !ok {
            continue
        }
        val := row[pos]
        field := vv.Field(i)
        switch field.Kind() {
        case reflect.Int:
            i, err := strconv.ParseInt(val, 10, 64)
            if err != nil {
                return err
            }
            field.SetInt(i)
        case reflect.String:
            field.SetString(val)
        case reflect.Bool:
            b, err := strconv.ParseBool(val)
            if err != nil {
                return err
            }
            field.SetBool(b)
        default:
            return fmt.Errorf("cannot handle field of kind %v",
                              field.Kind())
        }
    }
    return nil
}
```

This function iterates over each field in the newly created `reflect.Value`, uses the `csv` struct tag on the current field to find its name, looks up the element in the `data` slice by using the `namePos` map, converts the value from a `string` to the correct type, and sets the value on the current field. After all fields have been populated, the function returns.

Now that you have written your marshaler and unmarshaler, you can integrate with the existing CSV support in the Go standard library:

```
data := `name,age,has_pet
Jon,"100",true
"Fred ""The Hammer"" Smith",42,false
Martha,37,"true"
`
r := csv.NewReader(strings.NewReader(data))
allData, err := r.ReadAll()
if err != nil {
    panic(err)
}
var entries []MyData
Unmarshal(allData, &entries)
fmt.Println(entries)

//now to turn entries into output
out, err := Marshal(entries)
if err != nil {
    panic(err)
}
sb := &strings.Builder{}
w := csv.NewWriter(sb)
w.WriteAll(out)
fmt.Println(sb)
```

Build Functions with Reflection to Automate Repetitive Tasks

Another thing that Go lets you do with reflection is create a function. You can use this technique to wrap existing functions with common functionality without writing repetitive code. For example, here's a factory function that adds timing to any function that's passed into it:

```
func MakeTimedFunction(f any) any {
    ft := reflect.TypeOf(f)
    fv := reflect.ValueOf(f)
    wrapperF := reflect.MakeFunc(ft, func(in []reflect.Value) []reflect.Value {
        start := time.Now()
        out := fv.Call(in)
        end := time.Now()
        fmt.Println(end.Sub(start))
        return out
    })
    return wrapperF.Interface()
}
```

This function takes in any function, so the parameter is of type `any`. It then passes the `reflect.Type` that represents the function into `reflect.MakeFunc`, along with a closure that captures the start time, calls the original function using reflection, captures the end time, prints out the difference, and returns the value calculated by the

original function. The value returned from `reflect.MakeFunc` is a `reflect.Value`, and you call its `Interface` method to get the value to return. Here's how you use it:

```
func timeMe(a int) int {
    time.Sleep(time.Duration(a) * time.Second)
    result := a * 2
    return result
}

func main() {
    timed:= MakeTimedFunction(timeMe).(func(int) int)
    fmt.Println(timed(2))
}
```

You can run a more complete version of this program on The Go Playground or in the *sample_code/timed_function* directory in the Chapter 16 repository.

While generating functions is clever, be careful when using this feature. Make sure that it's clear when you are using a generated function and what functionality it is adding. Otherwise, you will be making it harder to understand the flow of data through your program. Furthermore, as I'll discuss in "Use Reflection Only if It's Worthwhile" on page 424, reflection makes your programs slower, so using it to generate and invoke functions seriously impacts performance unless the code you are generating is already performing a slow operation, like a network call. Remember, reflection works best when it's used to map data in and out of the edge of your programs.

One project that follows these rules for generated functions is my SQL mapping library Proteus. It creates a type-safe database API by generating a function from a SQL query and a function field or variable. You can learn more about Proteus in my GopherCon 2017 talk, "Runtime Generated, Typesafe, and Declarative: Pick Any Three," and you can find the source code on GitHub.

You Can Build Structs with Reflection, but Don't

You can make one more thing with reflection, and it's weird. The `reflect.StructOf` function takes in a slice of `reflect.StructField` and returns a `reflect.Type` that represents a new struct type. These structs can be assigned only to variables of type `any`, and their fields can only be read and written using reflection.

For the most part, this is a feature of academic interest only. To see a demo of how `reflect.StructOf` works, look at the memoizer function on The Go Playground or in the *sample_code/memoizer* directory in the Chapter 16 repository. It uses dynamically generated structs as the keys to a map that caches the output of a function.

Reflection Can't Make Methods

You've seen all the things that you can do with reflection, but there's one thing you can't make. While you can use reflection to create new functions and new struct types, there's no way to use reflection to add methods to a type. This means you cannot use reflection to create a new type that implements an interface.

Use Reflection Only if It's Worthwhile

Although reflection is essential when converting data at the boundaries of Go, be careful using it in other situations. Reflection isn't free. To demonstrate, let's implement `Filter` by using reflection. You looked at a generic implementation in "Generic Functions Abstract Algorithms" on page 187, but let's see what a reflection-based version looks like:

```
func Filter(slice any, filter any) any {
    sv := reflect.ValueOf(slice)
    fv := reflect.ValueOf(filter)

    sliceLen := sv.Len()
    out := reflect.MakeSlice(sv.Type(), 0, sliceLen)
    for i := 0; i < sliceLen; i++ {
        curVal := sv.Index(i)
        values := fv.Call([]reflect.Value{curVal})
        if values[0].Bool() {
            out = reflect.Append(out, curVal)
        }
    }
    return out.Interface()
}
```

You use it like this:

```
names := []string{"Andrew", "Bob", "Clara", "Hortense"}
longNames := Filter(names, func(s string) bool {
    return len(s) > 3
}).([]string)
fmt.Println(longNames)

ages := []int{20, 50, 13}
adults := Filter(ages, func(age int) bool {
    return age >= 18
}).([]int)
fmt.Println(adults)
```

This prints out the following:

```
[Andrew Clara Hortense]
[20 50]
```

Your reflection-using filter function isn't difficult to understand, but it's certainly longer than a custom-written or generic function. Let's see how it performs on an Apple Silicon M1 with 16 GB of RAM on Go 1.20 when filtering 1,000 element slices of strings and `int`s, compared to custom-written functions:

```
BenchmarkFilterReflectString-8     5870  203962 ns/op  46616 B/op  2219 allocs/op
BenchmarkFilterGenericString-8   294355    3920 ns/op  16384 B/op     1 allocs/op
BenchmarkFilterString-8          302636    3885 ns/op  16384 B/op     1 allocs/op
BenchmarkFilterReflectInt-8        5756  204530 ns/op  45240 B/op  2503 allocs/op
BenchmarkFilterGenericInt-8      439100    2698 ns/op   8192 B/op     1 allocs/op
BenchmarkFilterInt-8             443745    2677 ns/op   8192 B/op     1 allocs/op
```

You can find this code in the *sample_code/reflection_filter* directory in the Chapter 16 repository so you can run it yourself.

This benchmark demonstrates the value of using generics when you can. Reflection is over 50 times slower than a custom or generic function for string filtering and roughly 75 times slower for `int`s. It uses significantly more memory and performs thousands of allocations, which creates additional work for the garbage collector. The generic version provides the same performance as the custom-written functions without having to write multiple versions.

A more serious downside to the reflection implementation is that the compiler can't stop you from passing in a wrong type for either the `slice` or the `filter` parameter. You might not mind a few thousand nanoseconds of CPU time, but if someone passes in a function or slice of the wrong type to `Filter`, your program will crash in production. The maintenance cost might be too high to accept.

Some things can't be written using generics, and you'll need to fall back to reflection. CSV marshaling and unmarshaling requires reflection, as does the memoizing program. In both cases, you need to work with an unknown number of values of different (and unknown) types. But do make sure that reflection is essential before employing it.

unsafe Is Unsafe

Just as the `reflect` package allows you to manipulate types and values, the `unsafe` package allows you to manipulate memory. The `unsafe` package is very small and very odd. It defines several functions and one type, none of which act like the types and functions found in other packages.

Given Go's focus on memory safety, you might wonder why `unsafe` even exists. Just as you used `reflect` to translate text data between the outside world and Go code, you use `unsafe` to translate binary data. There are two main reasons for using `unsafe`. A 2020 paper by Diego Elias Costa et al. called "Breaking Type-Safety in Go: An

Empirical Study on the Usage of the unsafe Package" surveyed 2,438 popular Go open source projects and found the following:

- 24% of the studied Go projects use `unsafe` at least once in their codebase.
- The majority of `unsafe` usages were motivated by integration with operating systems and C code.
- Developers frequently use `unsafe` to write more efficient Go code.

The plurality of the uses of `unsafe` are for system interoperability. The Go standard library uses `unsafe` to read data from and write data to the operating system. You can see examples in the `syscall` package in the standard library or in the higher-level `sys` package. You can learn more about how to use `unsafe` to communicate with the operating system in a great blog post written by Matt Layher.

The `unsafe.Pointer` type is a special type that exists for a single purpose: a pointer of any type can be converted to or from `unsafe.Pointer`. In addition to pointers, `unsafe.Pointer` can also be converted to or from a special integer type, called `uintptr`. As with any other integer type, you can do math with it. This allows you to walk into an instance of a type, extracting individual bytes. You can also perform pointer arithmetic, just as you can with pointers in C and C++. This byte manipulation changes the value of the variable.

There are two common patterns in `unsafe` code. The first is a conversion between two types of variables that are normally not convertible. This is performed using a series of type conversions with `unsafe.Pointer` in the middle. The second is reading or modifying the bytes in a variable by converting a variable to an `unsafe.Pointer`, converting the `unsafe.Pointer` to a pointer, and then copying or manipulating the underlying bytes. Both of these techniques require you to know the size (and possibly the location) of the data being manipulated. The `Sizeof` and `Offsetof` functions in the `unsafe` package provide this information.

Using Sizeof and Offsetof

Some of the functions in `unsafe` reveal how the bytes in various types are laid out in memory. The first one you'll look at is `Sizeof`. As the name implies, it returns the size in bytes of whatever is passed into it.

While the sizes for numeric types are fairly obvious (an `int16` is 16 bits or 2 bytes, a `byte` is 1 byte, and so on), other types are a bit more complicated. For a pointer, you get the size of the memory to store the pointer (usually 8 bytes on a 64-bit system), not the size of the data that the pointer points to. This is why `Sizeof` considers any slice to be 24 bytes long on a 64-bit system; it's implemented as two `int` fields (for length and capacity) and a pointer to the data for the slice. Any `string` is 16

bytes long on a 64-bit system (an `int` for length and a pointer to the contents of the string). Any `map` on a 64-bit system is 8 bytes, since within the Go runtime, a `map` is implemented as a pointer to a rather complicated data structure.

Arrays are value types, so their size is calculated by multiplying the length of the array by the size of each element in the array.

For a struct, the size is the sum of the sizes of the fields, plus some adjustments for *alignment*. Computers like to read and write data in regular-sized chunks and they really don't want a value to start in one chunk and end in another. To make this happen, the compiler adds padding between fields so they line up properly. The compiler also wants the entire struct to be properly aligned. On a 64-bit system, it will add padding at the end of the struct to bring its size up to a multiple of 8 bytes.

Another function in `unsafe`, `Offsetof`, tells you the position of a field within a struct. Let's use `Sizeof` and `Offsetof` to look at the effects of field order on the size of a struct. Say you have two structs:

```
type BoolInt struct {
    b bool
    i int64
}

type IntBool struct {
    i int64
    b bool
}
```

Running this code

```
fmt.Println(unsafe.Sizeof(BoolInt{}),
    unsafe.Offsetof(BoolInt{}.b),
    unsafe.Offsetof(BoolInt{}.i))

fmt.Println(unsafe.Sizeof(IntBool{}),
    unsafe.Offsetof(IntBool{}.i),
    unsafe.Offsetof(IntBool{}.b))
```

produces the following output:

```
16 0 8
16 0 8
```

The size of both types is 16 bytes. When the `bool` comes first, the compiler puts 7 bytes of padding between `b` and `i`. When the `int64` comes first, the compiler puts 7 bytes of padding after `b` to make the struct's size a multiple of 8 bytes.

You can see the effect of field order when there are more fields:

```
type BoolIntBool struct {
    b  bool
    i  int64
    b2 bool
}

type BoolBoolInt struct {
    b  bool
    b2 bool
    i  int64
}

type IntBoolBool struct {
    i  int64
    b  bool
    b2 bool
}
```

Printing out the sizes and offsets for these types produces the following output:

```
24 0 8 16
16 0 1 8
16 0 8 9
```

Putting the `int64` field between the two `bool` fields produces a struct that's 24 bytes long, since both `bool` fields need to be padded to 8 bytes. Meanwhile, grouping the `bool` fields together produces a struct that's 16 bytes long, the same as the structs with only two fields. You can validate this on The Go Playground or in the *sample_code/sizeof_offsetof* directory in the Chapter 16 repository.

Although only of academic interest most of the time, this information is useful in two situations. The first is in programs that manage large amounts of data. You can sometimes achieve significant savings in memory usage simply by reordering the fields in heavily used structs in order to minimize the amount of padding needed for alignment.

The second situation occurs when you want to map a sequence of bytes directly into a struct. You'll look at that next.

Using unsafe to Convert External Binary Data

As was mentioned earlier, one of the main reasons people use `unsafe` is for performance, especially when reading data from a network. If you want to map the data into or out of a Go data structure, `unsafe.Pointer` gives you a very fast way to do so. You can explore this with a contrived example. Imagine a *wire protocol* (a specification indicating which bytes are written in which order when communicating over a network) with the following structure:

- Value: 4 bytes, representing an unsigned, big-endian 32-bit int
- Label: 10 bytes, ASCII name for the value
- Active: 1 byte, boolean flag to indicate whether the field is active
- Padding: 1 byte, because you want everything to fit into 16 bytes

Data sent over a network is usually sent in big-endian format (most significant bytes first), often called *network byte order*. Since most CPUs in use today are little-endian (or bi-endian running in little-endian mode), you need to be careful when reading or writing data to a network.

The code for this example is in the *sample_code/unsafe_data* directory in the Chapter 16 repository.

You define a data structure whose in-memory layout matches the wire protocol:

```
type Data struct {
    Value  uint32   // 4 bytes
    Label  [10]byte // 10 bytes
    Active bool     // 1 byte
    // Go pads this with 1 byte to make it align
}
```

You use `unsafe.Sizeof` to define a constant that represents its size:

```
const dataSize = unsafe.Sizeof(Data{}) // sets dataSize to 16
```

(One of the weird things about `unsafe.Sizeof` and `Offsetof` is that you can use them in `const` expressions. The size and layout of a data structure in memory is known at compile time, so the results of these functions are calculated at compile time, just like a constant expression.)

Say you just read the following bytes off the network:

```
[0 132 95 237 80 104 111 110 101 0 0 0 0 0 1 0]
```

You're going to read these bytes into an array of length 16 and then convert that array into the `struct` described previously.

With safe Go code, you could map it like this:

```
func DataFromBytes(b [dataSize]byte) Data {
    d := Data{}
    d.Value = binary.BigEndian.Uint32(b[:4])
    copy(d.Label[:], b[4:14])
    d.Active = b[14] != 0
    return d
}
```

Or, you could use `unsafe.Pointer` instead:

```
func DataFromBytesUnsafe(b [dataSize]byte) Data {
    data := *(*Data)(unsafe.Pointer(&b))
    if isLE {
        data.Value = bits.ReverseBytes32(data.Value)
    }
    return data
}
```

The first line is a little confusing, but you can take it apart and understand what's going on. First, you take the address of the byte array and convert it to an `unsafe.Pointer`. Then you convert the `unsafe.Pointer` to a `(*Data)` (you have to put `(*Data)` in parentheses because of Go's order of operations). You want to return the struct, not a pointer to it, so you dereference the pointer. Next, you check your flag to see if you are on a little-endian platform. If so, you reverse the bytes in the `Value` field. Finally, you return the value.

Why did you use an array instead of a slice for the parameter? Remember, arrays, like structs, are value types; the bytes are allocated directly. That means you can map the values in `b` directly into the `data` struct. A slice is composed of three parts: a length, a capacity, and a pointer to the actual values. You'll see how to map a slice into a struct using `unsafe` in a bit.

How do you know if you are on a little-endian platform? Here's the code you're using:

```
var isLE bool

func init() {
    var x uint16 = 0xFF00
    xb := *(*[2]byte)(unsafe.Pointer(&x))
    isLE = (xb[0] == 0x00)
}
```

As I discussed in "Avoiding the init Function if Possible" on page 239, you should avoid using `init` functions, except when initializing a package-level value whose value is effectively immutable. Since the endianness of your processor isn't going to change while your program is running, this is a good use case.

On a little-endian platform, the bytes that represent `x` will be stored as [00 FF]. On a big-endian platform, `x` is stored in memory as [FF 00]. You use `unsafe.Pointer` to convert a number to an array of bytes, and you check what the first byte is to determine the value of `isLE`.

Likewise, if you wanted to write your `Data` back to the network, you could use safe Go:

```
func BytesFromData(d Data) [dataSize]byte {
    out := [dataSize]byte{}
    binary.BigEndian.PutUint32(out[:4], d.Value)
    copy(out[4:14], d.Label[:])
    if d.Active {
        out[14] = 1
    }
    return out
}
```

Or you could use `unsafe`:

```
func BytesFromDataUnsafe(d Data) [dataSize]byte {
    if isLE {
        d.Value = bits.ReverseBytes32(d.Value)
    }
    b := *(*[dataSize]byte)(unsafe.Pointer(&d))
    return b
}
```

If the bytes are stored in a slice, you can use the `unsafe.Slice` function to create a slice from the contents of `Data`. The `unsafe.SliceData` function is used to create a `Data` instance from the data stored in a slice:

```
func BytesFromDataUnsafeSlice(d Data) []byte {
    if isLE {
        d.Value = bits.ReverseBytes32(d.Value)
    }
    bs := unsafe.Slice((*byte)(unsafe.Pointer(&d)), dataSize)
    return bs
}

func DataFromBytesUnsafeSlice(b []byte) Data {
    data := *(*Data)((unsafe.Pointer)(unsafe.SliceData(b)))
    if isLE {
        data.Value = bits.ReverseBytes32(data.Value)
    }
    return data
}
```

The first parameter to `unsafe.Slice` requires two casts. The first cast converts a pointer of the `Data` instance to an `unsafe.Pointer`. Then you need to cast again to a pointer of the type of data that you want the slice to hold. For a slice of bytes, you use `*byte`. The second parameter is the length of the data.

The `unsafe.SliceData` function takes in a slice and returns a pointer to the type of the data contained in the slice. In this case, you passed in a `[]byte`, so it returned a

`*byte`. You then use `unsafe.Pointer` as a bridge between `*byte` and `*Data` to convert the contents of the slice into a `Data` instance.

Is this worth it? Here are the timings on an Apple Silicon M1 (which is little-endian):

```
BenchmarkBytesFromData-8              548607271  2.185 ns/op   0 B/op 0 allocs/op
BenchmarkBytesFromDataUnsafe-8       1000000000  0.8418 ns/op  0 B/op 0 allocs/op
BenchmarkBytesFromDataUnsafeSlice-8   91179056  13.14 ns/op   16 B/op 1 allocs/op
BenchmarkDataFromBytes-8              538443861  2.186 ns/op   0 B/op 0 allocs/op
BenchmarkDataFromBytesUnsafe-8       1000000000  1.160 ns/op   0 B/op 0 allocs/op
BenchmarkDataFromBytesUnsafeSlice-8 1000000000  0.9617 ns/op  0 B/op 0 allocs/op
```

Two things stand out in this chart. First, the conversion from a struct to a slice is by far the slowest operation, and it's the only one that allocates memory. This isn't surprising, since the slice's data needs to escape to the heap when it is returned from the function. Allocating memory on the heap is almost always slower than using memory on the stack. Converting from a slice to a struct is very fast, though.

If you are working with arrays, using `unsafe` is about 2–2.5 times faster than the standard approach. If you have a program with many of these kinds of conversions, or if you are trying to map a very large and complicated data structure, using these low-level techniques is worthwhile. But for the vast majority of programs, stick with the safe code.

Accessing Unexported Fields

Here's another bit of magic you can do with `unsafe`, but it's something to use only as a last resort. You can combine reflection and `unsafe` to read and modify unexported fields in structs. Let's see how.

First, define a struct in one package:

```
type HasUnexportedField struct {
    A int
    b bool
    C string
}
```

Normally, code outside this package is unable to access `b`. However, let's see what you can do from another package by using `unsafe`:

```
func SetBUnsafe(huf *one_package.HasUnexportedField) {
    sf, _ := reflect.TypeOf(huf).Elem().FieldByName("b")
    offset := sf.Offset
    start := unsafe.Pointer(huf)
    pos := unsafe.Add(start, offset)
    b := (*bool)(pos)
    fmt.Println(*b) // read the value
    *b = true       // write the value
}
```

You use reflection to access type information about field `b`. The `FieldByName` method returns a `reflect.StructField` instance for any field on a struct, even unexported ones. The instance includes the offset of its associated field. You then convert `huf` into an `unsafe.Pointer` and use the `unsafe.Add` method to add the offset to the pointer in order to find the location of `b` within the struct. All that remains is casting the `unsafe.Pointer` returned by `Add` to a `*bool`. Now you can read the value of `b` or set its value. You can try out this code in the *sample_code/unexported_field_access* directory in the Chapter 16 repository.

Using unsafe Tools

Go is a language that values tooling, and a compiler flag can help you find misuse of `Pointer` and `unsafe.Pointer`. Run your code with the flag `-gcflags=-d=checkptr` to add additional checks at runtime. Like the race checker, it's not guaranteed to find every `unsafe` problem and it does slow your program. However, it's a good practice while testing your code.

If you want to learn more about `unsafe`, read through the package documentation.

The `unsafe` package is powerful and low-level! Avoid using it unless you know what you are doing and you need the performance improvements that it provides.

Cgo Is for Integration, Not Performance

Just like reflection and `unsafe`, `cgo` is most useful at the border between Go programs and the outside world. Reflection helps integrate with external textual data, `unsafe` is best used with operating system and network data, and `cgo` is best for integrating with C libraries.

Despite being nearly 50 years old, C is still the *lingua franca* of programming languages. All the major operating systems are primarily written in either C or C++, which means that they are bundled with libraries written in C. It also means that nearly every programming language provides a way to integrate with C libraries. Go calls its foreign function interface (FFI) to C `cgo`.

As you have seen many times, Go is a language that favors explicit specification. Go developers sometimes deride automatic behaviors in other languages as "magic." However, using `cgo` feels a bit like spending time with Merlin. Let's take a look at this magical glue code.

You'll start with a very simple program that calls C code to do some math. The source code is in GitHub in the *sample_code/call_c_from_go* directory in the Chapter 16 repository. First, here's the code in *main.go*:

```
package main

import "fmt"

/*
    #cgo LDFLAGS: -lm
    #include <stdio.h>
    #include <math.h>
    #include "mylib.h"

    int add(int a, int b) {
        int sum = a + b;
        printf("a: %d, b: %d, sum %d\n", a, b, sum);
        return sum;
    }
*/
import "C"

func main() {
    sum := C.add(3, 2)
    fmt.Println(sum)
    fmt.Println(C.sqrt(100))
    fmt.Println(C.multiply(10, 20))
}
```

The *mylib.h* header is in the same directory as your *main.go*:

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

There is also *mylib.c*:

```
#include "mylib.h"

int multiply(int a, int b) {
    return a * b;
}
```

Assuming you have a C compiler installed on your computer, all you need to do is compile your program with `go build`:

```
$ go build
$ ./call_c_from_go
a: 3, b: 2, sum 5
5
10
200
```

What's going on here? The standard library doesn't have a real package named `C`. Instead, `C` is an automatically generated package whose identifiers mostly come from

the C code embedded in the comments that immediately precede it. In this example, you declare a C function called `add`, and `cgo` makes it available to your Go program as the name `C.add`. You can also invoke functions or global variables that are imported into the comment block from libraries via header files, as you can see when you call `C.sqrt` from `main` (imported from *math.h*) or `C.multiply` (imported from *mylib.h*).

In addition to the identifier names that appear in the comment block (or are imported into the comment block), the `C` pseudopackage also defines types like `C.int` and `C.char` to represent the built-in C types and functions, such as `C.CString` to convert a Go string to a C string.

You can use more magic to call Go functions from C functions. Go functions can be exposed to C code by putting an `//export` comment before the function. You can see this in use in the *sample_code/call_go_from_c* directory in the Chapter 16 repository. In *main.go*, you export the doubler function:

```
//export doubler
func doubler(i int) int {
    return i * 2
}
```

When you export a Go function, you can no longer write C code directly in the comment before the `import "C"` statement. You can only list function headers:

```
/*
    extern int add(int a, int b);
*/
import "C"
```

Put your C code into a *.c* file in the same directory as your Go code and include the magic header `"cgo_export.h"`. You can see this in the *example.c* file:

```
#include "_cgo_export.h"

int add(int a, int b) {
    int doubleA = doubler(a);
    int sum = doubleA + b;
    return sum;
}
```

Running this program with `go build` give the following output:

```
$ go build
$ ./call_go_from_c
8
```

So far, this seems pretty simple, but one major stumbling block arises when using `cgo`: Go is a garbage-collected language, and C is not. This makes it difficult to integrate nontrivial Go code with C. While you can pass a pointer into C code, you cannot directly pass something that contains a pointer. This is very limiting, as things like

strings, slices, and functions are implemented with pointers and therefore cannot be contained in a struct passed into a C function.

That's not all: a C function cannot store a copy of a Go pointer that lasts after the function returns. If you break these rules, your program will compile and run, but it may crash or behave incorrectly at runtime when the memory pointed to by the pointer is garbage collected.

If you find yourself needing to pass an instance of a type that contains a pointer from Go to C and then back into Go, you can use a `cgo.Handle` to wrap the instance. Here's a short example. You can find the source code in the *sample_code/handle* directory in the Chapter 16 repository.

First you have your Go code:

```
package main

/*
    #include <stdint.h>

    extern void in_c(uintptr_t handle);
*/
import "C"

import (
    "fmt"
    "runtime/cgo"
)

type Person struct {
    Name string
    Age  int
}

func main() {
    p := Person{
        Name: "Jon",
        Age:  21,
    }
    C.in_c(C.uintptr_t(cgo.NewHandle(p)))
}

//export processor
func processor(handle C.uintptr_t) {
    h := cgo.Handle(handle)
    p := h.Value().(Person)
    fmt.Println(p.Name, p.Age)
    h.Delete()
}
```

And this is the C code:

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

void in_c(uintptr_t handle) {
    processor(handle);
}
```

In the Go code, you're passing a `Person` instance into the C function `in_c`. This function in turn calls the Go function `processor`. You can't safely pass a `Person` into C via `cgo`, because one of its fields is a `string`, and every `string` contains a pointer. To make this work, use the `cgo.NewHandle` function to convert `p` to a `cgo.Handle`. You then cast the `Handle` to a `C.uintptr_t` so you can pass it to the C function `in_c`. The function takes in a single parameter of type `uintptr_t`, which is a C type that is analogous to Go's `uintptr` type. The `in_c` function calls the Go function `process`, which also takes in a single parameter of type `C.uintptr_t`. It casts this parameter to a `cgo.Handle`, and then uses a type assertion to convert the `Handle` to a `Person` instance. You print out the fields in `p`. Now that you are done using the `Handle`, you call the `Delete` method to delete it.

In addition to working with pointers, other `cgo` limitations exist. For example, you cannot use `cgo` to call a variadic C function (such as `printf`). Union types in C are converted into byte arrays. And you cannot invoke a C function pointer (but you can assign it to a Go variable and pass it back into a C function).

These rules make using `cgo` nontrivial. If you have a background in Python or Ruby, you might think that using `cgo` is worth it for performance reasons. Those developers write the performance-critical parts of their programs in C. The speed of NumPy is due to the C libraries that are wrapped by Python code.

In most cases, Go code is many times faster than Python or Ruby, so the need to rewrite algorithms in a lower-level language is greatly reduced. You might think that you could save `cgo` for those situations when you do need additional performance gains, but unfortunately, using `cgo` to make your code faster is difficult. Because of the mismatches in the processing and memory models, calling a C function from Go is roughly 29 times slower than a C function calling another C function.

At CapitalGo 2018, Filippo Valsorda gave a talk, called "Why cgo is slow." Unfortunately, the talk wasn't recorded, but slides are available. They explain why `cgo` is slow and why it will not be made appreciably faster in the future. In his blog post "CGO Performance in Go 1.21," Shane Hansen measured the overhead of a `cgo` call in Go 1.21 at roughly 40ns on an Intel Core i7-12700H.

Since `cgo` isn't fast, and it isn't easy to use for nontrivial programs, the only reason to use `cgo` is if you must use a C library and there is no suitable Go replacement. Rather than writing `cgo` yourself, see if a third-party module already provides the wrapper. For example, if you want to embed SQLite in a Go application, look at GitHub. For ImageMagick, check out this repository.

If you find yourself needing to use an internal C library or third-party library that doesn't have a wrapper, you can find additional details on how to write your integration in the Go documentation. For information on the sorts of performance and design trade-offs that you are likely to encounter when using `cgo`, read Tobias Grieger's blog post called "The Cost and Complexity of Cgo."

Exercises

Now it's time to test yourself to see if you can write small programs using reflection, `unsafe`, and `cgo`. Answers for the exercises are found in the *exercise_solutions* directory in the Chapter 16 repository.

1. Use reflection to create a simple minimal string-length validator for struct fields. Write a `ValidateStringLength` function that takes in a struct and returns an error if one or more of the fields is a string, has a struct tag called `minStrlen`, and the length of the value in the field is less than the value specified in the struct tag. Nonstring fields and string fields that don't have the `minStrlen` struct tag are ignored. Use `errors.Join` to report all invalid fields. Be sure to validate that a struct was passed in. Return `nil` if all fields are of the proper length.
2. Use `unsafe.Sizeof` and `unsafe.Offsetof` to print out the size and offsets for the `OrderInfo` struct defined in *ch16/tree/main/sample_code/orders*. Create a new type, `SmallOrderInfo`, that has the same fields, but reordered to use as little memory as possible.
3. Copy the C code at *ch16/tree/main/sample_code/mini_calc* into your own module and use `cgo` to call it from a Go program.

Wrapping Up

In this chapter, you've learned about reflection, `unsafe`, and `cgo`. These features are probably the most exciting parts of Go because they allow you to break the rules that make Go a boring, type-safe, memory-safe language. More importantly, you've learned *why* you would want to break the rules and why you should avoid doing so most of the time.

You've completed your journey through Go and how to use it idiomatically. As with any graduation ceremony, it's time for a few closing words. Let's look back at what was said in the preface. "[P]roperly written, Go is boring….Well-written Go programs tend to be straightforward and sometimes a bit repetitive." I hope you can now see why this leads to better software engineering.

Idiomatic Go is a set of tools, practices, and patterns that makes it easier to maintain software across time and changing teams. That's not to say the cultures around other languages don't value maintainability; it just may not be their highest priority. Instead, they emphasize things like performance, new features, or concise syntax. There is a place for these trade-offs, but in the long run, I suspect Go's focus on crafting software that lasts will win out. I wish you the best as you create the software for the next 50 years of computing.