Pointers

This lesson discusses how pointers are used in the Go language.

WE'LL COVER THE FOLLOWING

Introduction

Unlike Java and .NET, Go gives the programmers control over which data structure is a pointer and which is not; however, calculations with pointer values are not supported in Go programs. By giving the programmer control over basic memory layout, Go provides you the ability to control the total size of a given collection of data structures, the number of allocations, and the memory access patterns; all of which are important for building systems that perform well.

Pointers are important for performance and indispensable if you want to do systems programming close to the operating system and network. Because pointers are somewhat unknown to contemporary OO-programmers, we will explain them here and in the coming chapters in depth.

Pointers in Go

Programs store values in memory, and each memory block (or word) has an *address*, which is usually represented as a hexadecimal number, like 0x6b0820 or 0xf84001d7f0. Go has the *address-of operator* **&**, which, when placed before a variable, gives us the memory address of that variable. For example:

```
package main
import "fmt"
```

```
func main(){
    var i1 = 5

    fmt.Printf("An integer: %d, it's location in memory: %p\n", i1, &i1)
}
```

Memory Location of a Variable

The output of this example can be different every time because the value `&i1` can be different each time we execute this statement.

This address can be stored in a special data type called a *pointer*. In the above case, it is a pointer to an *int*. So `i1` is denoted by `*int`. If we call that pointer `intP`, we can declare it as:

```
var intP *int
```

Then, the following statement is true:

```
intP = &i1
```

So, `intP` stores the memory address of `i1` which means it points to the location of `i1`. In other words, it references the variable `i1`.

A pointer variable contains the memory address of *another* value which means it points to that value in memory. The size of a pointer is 4 bytes on 32-bit machines, and 8 bytes on 64-bit machines, regardless of the size of the value they point to. Of course, pointers can be declared to a value of any type, be it primitive or structured. The * is placed before the type of the value (prefixing), so the * is a *type modifier* here.

Using a pointer to refer to a value is called *indirection*. A newly declared pointer which has not been assigned to a variable has the nil value. A pointer variable is often abbreviated as ptr.

In an expression like:

```
var p *type
```

always leave a *space* between the *name of the pointer* and the *. The statement always leave a *space between the name of the pointer* and the *. The statement **var p*type** is syntactically correct, but in more complex expressions, it can

easily be mistaken for a multiplication.

The same symbol * can be placed before a pointer like `*intP`, and it gives the value which the pointer is pointing to and it is called the dereference (or contents or indirection) operator. Another way to say this is that the pointer is flattened. So for any variable `var`, the following is *true*:

```
var == *(&var)
```

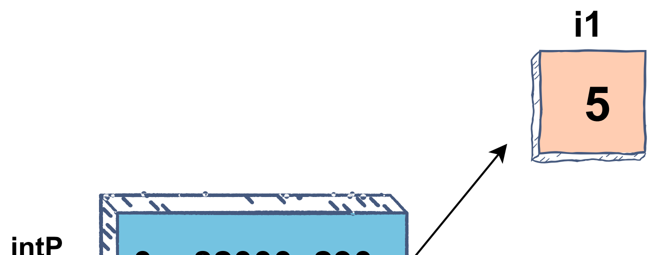
Now you have the basic concepts of pointers, you can understand the following program and its output.

```
package main
import "fmt"

func main() {
    var i1 = 5
    fmt.Printf("An integer: %d, its location in memory: %p\n", i1, &i1)
    var intP *int       // Pointer variable
    intP = &i1          // Storing address of i1 in pointer variable
    fmt.Printf("The value at memory location %p is %d\n", intP, *intP)
}
```

Pointers

As you can see, we declared a variable `i1` at line 5 and set the value of **5** to it. In the next line, we print its value along with its memory address with `&` operator as `&i1`. At **line** 7, we declared a pointer variable `intP` that is meant to have an address value. At **line 8**, we give `intP` the address of `i1` using `&i1`. Finally, at **line 9**, we print the address using `intP` and value at that address by dereferencing it as `*intP`, which is **5**.

You can represent the memory usage as:



Pointers and Memory Usage

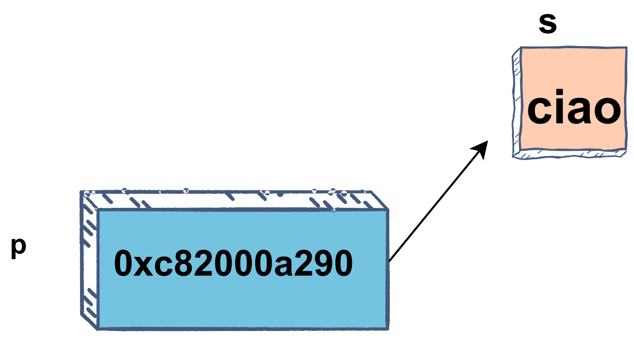
The following is another program that deals with strings and shows that assigning a new value to the pointer variable changes the value of the variable itself.

```go
package main
import "fmt"

func main() {
    s := "good bye"
    var p *string = &s
    *p = "ciao"         // changing the value at &s

    fmt.Printf("Here is the pointer p: %p\n", p)  // prints address
    fmt.Printf("Here is the string *p: %s\n", *p) // prints string
    fmt.Printf("Here is the string s: %s\n", s)   // prints same string
}
```

Changing Values with Pointers

As you can see, we declare a pointer variable `p` initially and set the address to it of a string variable `s`, at **line 6**. At **line** 7, we are dereferencing the `p` variable and changing value to **ciao**. It means the address that `p` holds which was `&s`, does not have an initial value of `s` anymore but a new value that is **ciao**. Then from **line 9** to **line 11**, we are printing to see the results, which show that dereferencing the `p` variable actually changes the `s` value from **good bye** to **ciao**.

You can represent the memory usage as:



Pointers and Memory Usage

You cannot take the address of a literal or a constant as the following code snippet shows:

```
package main

func main(){
    const i = 5
    ptr1 := &i //error: cannot take the address of i
    ptr2 := &10 //error: cannot take the address of 10
}
```

Go, like most other low level (system) languages like C, C++, and D, has the concept of pointers. But calculations with pointers (so-called pointer arithmetic, e.g., pointer + 2, to go through the bytes of a string or the positions in an array), which often lead to erroneous memory access in C and thus fatal crashes of programs, are not allowed in Go, making the language memory-safe.

One advantage of pointers is that you can pass a reference to a variable (for example, a parameter to a function) instead of passing a copy of that variable. Pointers are cheap to pass, only 4 or 8 bytes. When the program has to work with variables that occupy a lot of memory, or many variables, or both, working with pointers can reduce memory usage and increase efficiency. Pointer variables also persist in memory as long as there is at least 1 pointer pointing to them, so their lifetime is independent of the scope in which they were created.

On the other hand, because a pointer causes what is called an *indirection* (a shift in the processing to another address), using them unnecessarily could cause a performance decrease. Pointers can also point to other pointers, and this nesting can go arbitrarily deep so that you can have multiple levels of indirection. Still, in most cases, this will not contribute to the clarity of your code. As we will see, in many instances Go makes it easier for the programmer and will hide indirection like for example performing an automatic dereference.

Run the following program to see that nill pointer dereference is not allowed

Run the following program to see that nil pointer dereference is not allowed.

```go
package main

func main() {
    var p *int = nil    // Making a nil pointer
    *p = 0
}
// panic: runtime error: invalid memory address or nil pointer dereference
```

Nil Pointer Dereference

Oops! The program crashed. The reason is that we create a nil pointer at **line 4** and then try to dereference it at **line 5**. This is not allowed. Remember that a nil pointer dereference is *illegal* and makes a program *crash*.

That’s it about the basics of the Go language. In the next chapter, you’ll study some control structures, but before we jump to it, let’s take a little quiz to test our concepts.