Go Pointers

Ian Harris

December 14, 2018

1 Basic Data Types in Go • Pointers

So, in this module, we're going to talk about basic data types, and we're going to start with Pointers, which maybe is an unusual place to start the discussion of data types, but that's where we're starting, because people who are taking these courses generally already know something about programming. So, let's go straight to these Pointers and talk about them.

1.1 Pointer

A pointer is an address to some data in memory. So, I got to say every variable is located in memory somewhere, in some data staying in memory somewhere. Also, functions and so on, they're all in memory somewhere. A pointer is the address of that in memory. Tips give virtual address but that doesn't matter to us too much right now.

1.1.1 Pointer Operators • `&` • `*`

So, with Pointers, there are two main operators that are associated with Pointers. The "ampersand" operator right there, that returns the address of the variable or the function, whatever the name is referring to, and the "star" operator which dereferencing, does the opposite of the ampersand. It returns the data at the address. So, the ampersand operator, if you put that in front of a variable, the name of a variable, that will return you the address of that variable. The star operator goes the other way. If you put that in front of a Pointer, to some address, put that in front of an address, it will return you the data at that address. So, it's important to understand this ampersand operator and the star operator are opposites of one another.

Pointer Example So, I'll give you an example, take the look at this code, little piece of code, we define our variable `x` is an integer, it's equal to one. Then, `y` is an integer and it's not initialized, so that would mean it would be by default initialized to zero. Then, a var `ip`. So, `ip` is not declared to be an `int`, it's a `* int`. Right? So, that means `ip` is actually a pointer since there's a star operator in front of the `int`, `ip` is declared to be a pointer to an integer. So, `ip` is the `pointer`, but it is not an actual integer, is a pointer to an integer. So then, if we go on, `ip` equals `* x`. `x` is actually an integer. Right? Integer whose value is one. So, there is a number one sitting in memory somewhere, and `x` is a reference to it, the name of that. `* x` is the address in memory where I can find that value one. So, `ip` is now equal to that address. So, whatever the address of that one, is rather, of `x`, whatever that address is, `ip` is the pointer to that address, it is the address. Then, in the next line, I say `y` equals `* ip`. So, remember that in this little example, `ip` is actually a pointer and the data at that address that `ip` is pointing to, is the value one. Now, `* ip`, `*` does dereferencing. `*` says, `*` returns the value, the data at that address. So, `y` is now equal to the data at the address that `ip` is pointing to. Now, if you remember for line for `ip` is pointed to what `x` is pointing to. Right? So, `ip` is pointing to the value one, that data in memory. So `y`, if `y` is equal to `* ip`, `y` is equal to one. So, this now sets `y` equal to one. It is just a little example here, just trying to show how the ampersand and the star operators work opposite to one another.

So, these are pointers, and pointers exist. These pointers are basically, if you now see same type of implementation.

1.2 `new()`

Now, there's another function called `new()`. It's another way to create a variable. `new()`, instead of returning a variable, it returns a pointer to the variable. So, the `new()` function creates a variable and it returns a pointer to that variable. So, this is unlike, if we're just declaring a variable, right? That also creates a variable, but `new()` explicitly returns a pointer to a variable. So, the variable is initialized to zero by default with `new()`. So, for instance here, if I say our pointer equals `new int`, and then `* pointer` equals three, right? Pointer was `new int`, `new int` returns me a pointer to an integer, and that `ptr` is equal to that pointer. Then, I can set the value of that integer by referring to `* ptr`, right? Because `* ptr` is the value that `ptr` is pointing to. If I say `* ptr` equals three, then the value three is placed at the address specified by `ptr`.