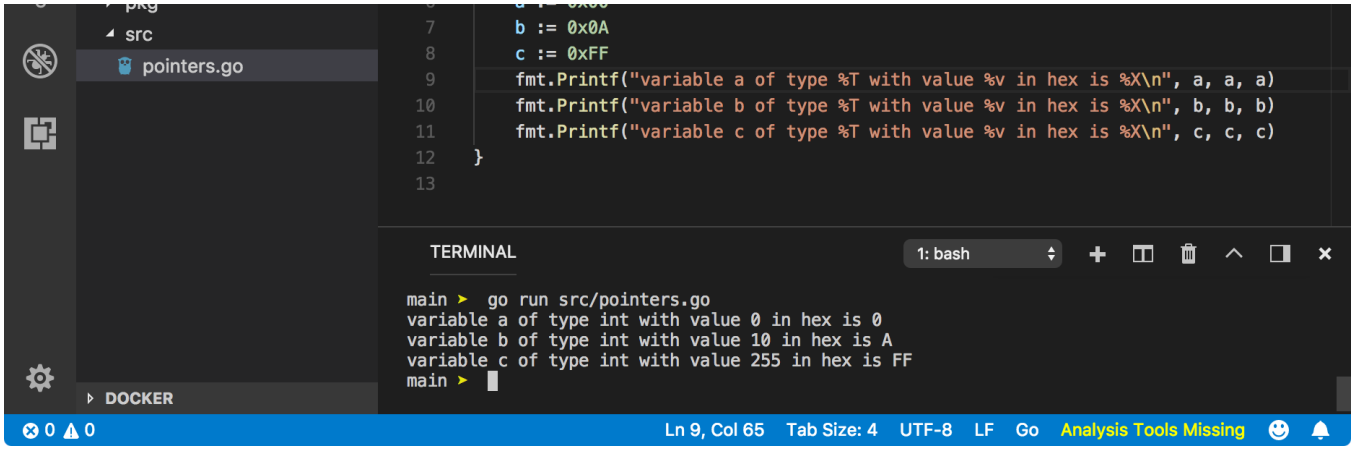
Pointers in Go

A pointer is a variable that stores the memory address of another variable. Pointers have the power to mutate data they are pointing.

Uday Hiwarale Follow
Oct 19, 2018 · 6 min read

Before we begin talking about pointers, let's learn a thing or two about `hexadecimal` numbers. A hexadecimal number is a number with base `16`. If you are a web developer, then you are using them for a long time, because mostly; colors are represented in hex format. For example, white is represented as `#FFFFFF` and black as `#000000`.

In Go, you can save a hexadecimal number in a variable and go provides a literal expression to represent a hexadecimal number. If a number starts with `0x` then it's a hexadecimal number.



https://play.golang.org/p/NnAJ5go4dfz

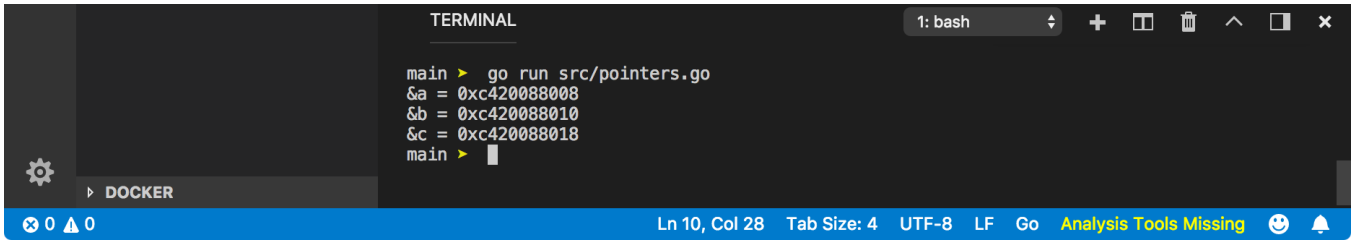
From the above example, we found out that even values represented in hexadecimal are saved in decimal with data type `int`.

But why we are learning about hexadecimal numbers when we are talking about pointers. Well, let's talk about memory address first.

When you are declaring a variable, Go compiler will allocate some memory for it in RAM and depending on its type, it will allocate a specific size of memory to store the data. That memory will have some memory address so that go can find that variable's value when asked for it. These memory addresses are represented in hexadecimal values.

☛ How to access the memory address of a variable?

To access an address of a variable memory, go provides a simple operator `&` (*ampersand*) which if used in front of the variable name, returns the memory address.



https://play.golang.org/p/15P6SHOBavA

We saw this in **slice** lesson when we were proving that how two slices can represent values from the same array. Anyway, in the above example, using `&` operator, we found the memory address of the variable `a`, `b` and `c`.

You may see different memory addresses, which I don't need to explain because you are smart enough to find out why.

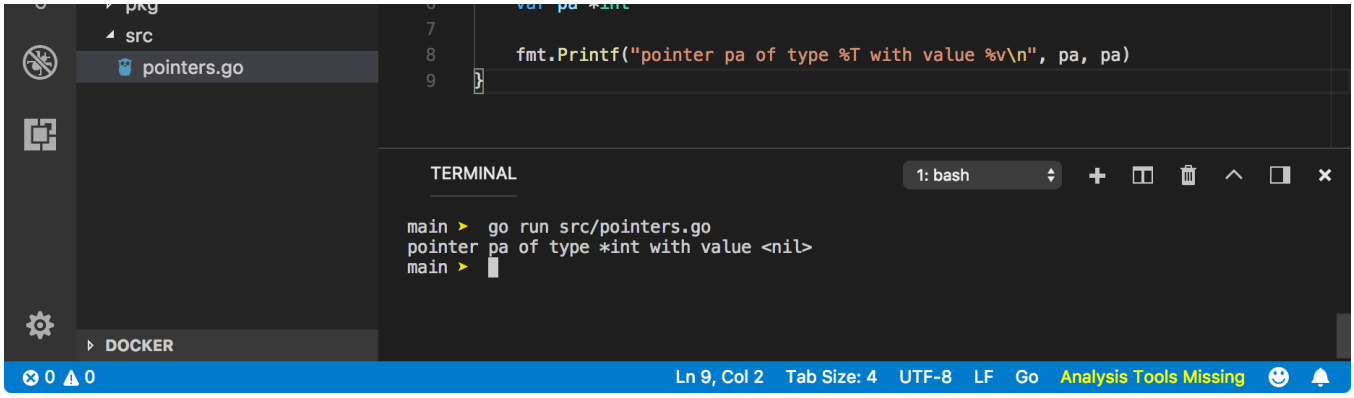
☛ What is a pointer?

A pointer is a variable which points to the memory location of another variable.

As we saw earlier, we can save a hexadecimal value to a variable, but underneath it is saved in a decimal value of type `int`. But there is a catch, even though you could save a memory address in a variable, it's not a pointer or it does not point to the memory location of another variable. It is just a value and it has no recollection of where it came from.

That's where `pointer` comes into play. Not only it saves the memory address of another variable but it also knows where that memory is located and how to find out the value stored in that memory. Unlike saving hex value in a variable which has type `int`, `pointer` has the data type of `*int` if it points to the memory address of integer data and `*string` if it points to the memory address of string data.

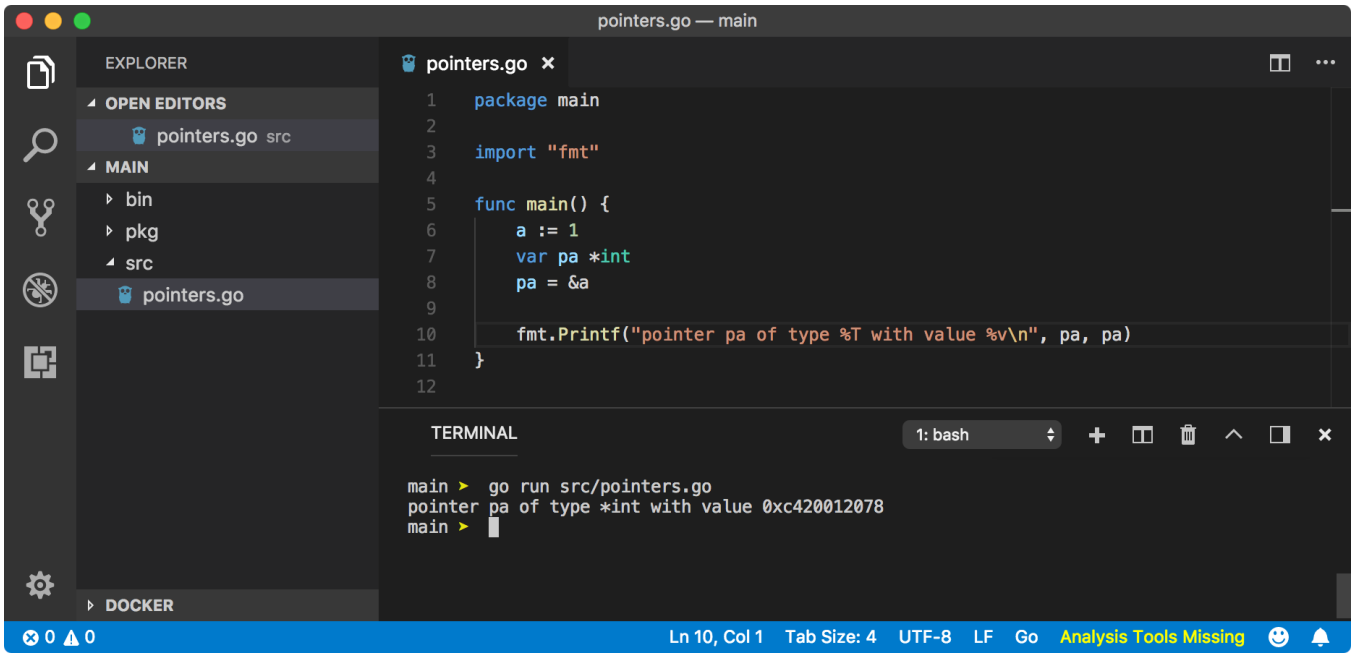
Syntax to create a pointer or define a pointer is `var p *Type` where `Type` is a data type. Let's create a simple pointer.



https://play.golang.org/p/77BocN1gVXu

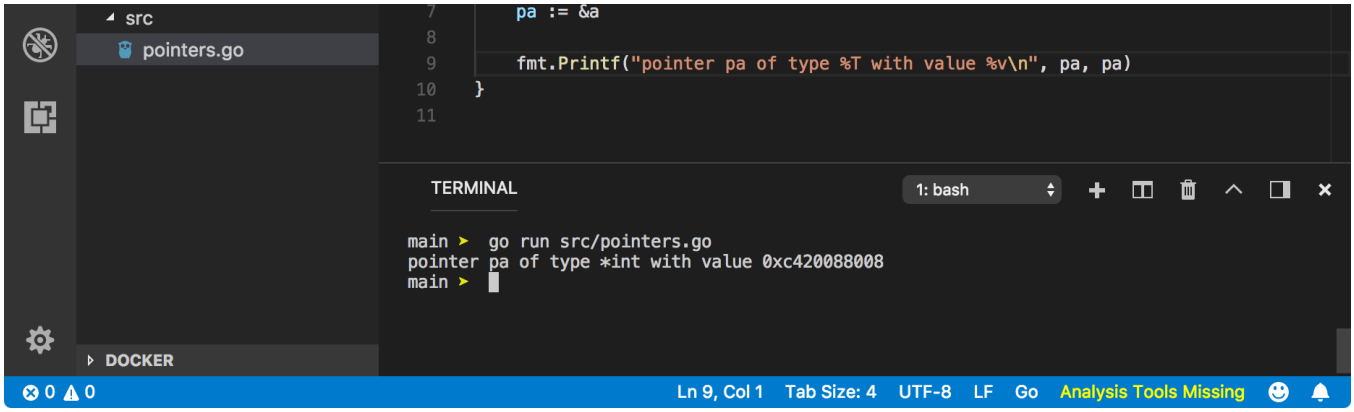
Which proves that pointer points to the data type of `int` and zero value of a pointer is `nil` which means it is not pointing to any variable or memory at this moment. We discussed `zero-value` of a pointer in `slice` lesson.

So let's create a variable of type `int` and make `pa` point to it.



https://play.golang.org/p/LHlGgDwSpH6

Above program would also be written using shorthand syntax like



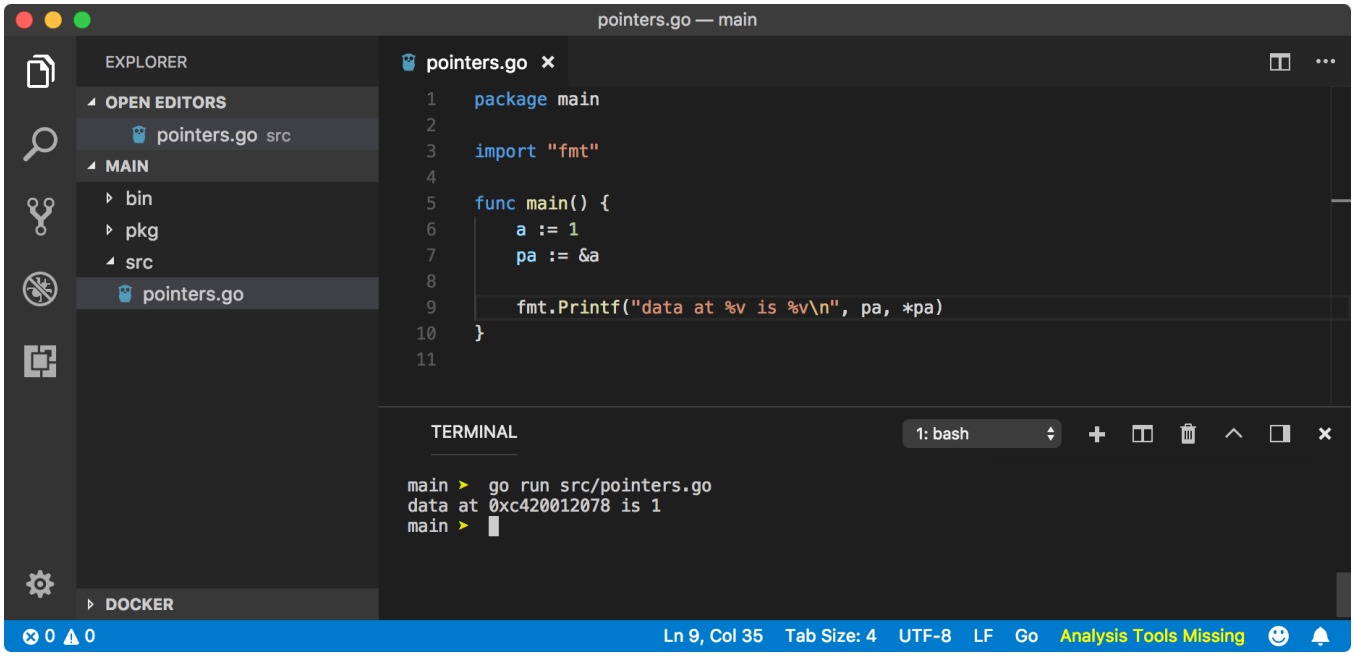
https://play.golang.org/p/la95cObwyiy

go will interpret that we are trying to create a pointer because we are assigning memory address of a variable to another variable (*using* `&` *operator*).

Which proves that pointer `pa` is not empty and pointing some memory location. It does not explicitly mention which variable it points to though. But it can find out the value in that memory location.

☛ Dereferencing the pointer

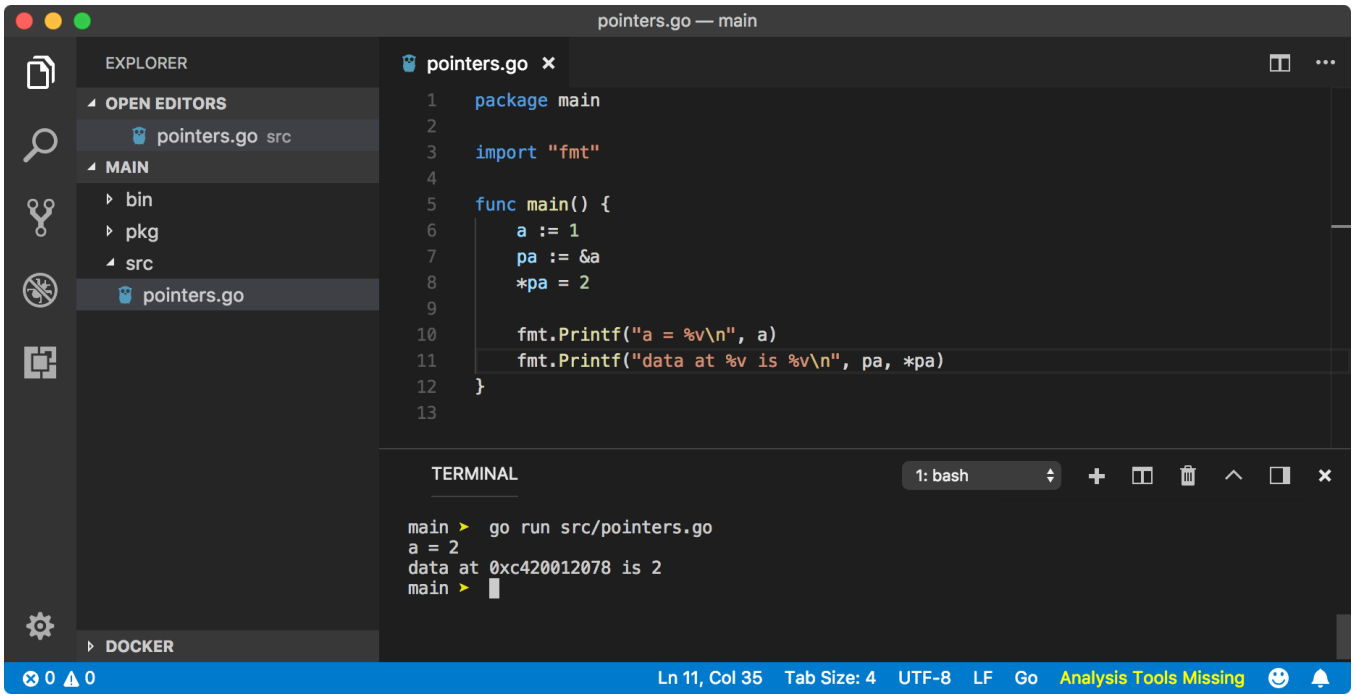
To find out value a pointer it points to, we need to use `*` operator, also called dereferencing operator which if placed before a pointer variable (*like* `&` *operator to get memory address*), returns the value in that memory.



https://play.golang.org/p/TnS5HySFnA9

☛ Changing the value of a pointer

As we saw, we can access data at the memory location pointed by a `pointer`, we can also change the value at that memory location, instead of assigning the new value to the variable.



https://play.golang.org/p/yv9QmgmVQCK

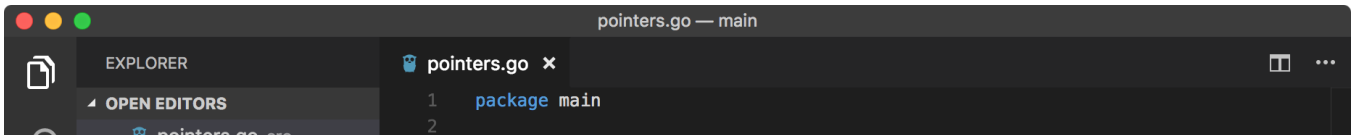
Why the value of `a` changed, because we changed the value at the memory location of the variable `a`.

☛ `new` function

Go provides `new` function which allocates memory and returns a pointer. Syntax of `new` function is

```
func new(Type) *Type
```

The first argument is a type, not a value, and the value returned is a pointer to a newly allocated zero value of that type. We can use this return value to save in a pointer.



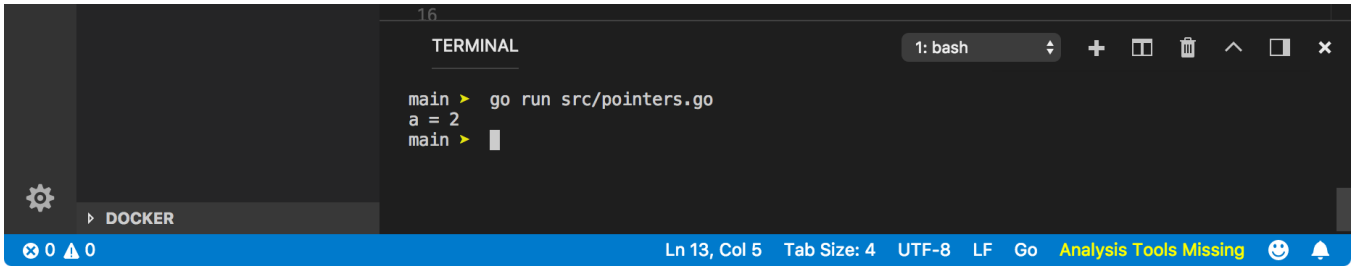
https://play.golang.org/p/WCVmwQCS6cG

Did you expect data at memory location created by go to be `nil`. Well, since `zero-value` of a pointer is `nil` which means pointer is not pointing to any memory but when the pointer points it to a memory location, memory cannot be empty (or `nil`), it must hold some data. go stores `zero-value` of data type passed in `new` function and returns the memory address of it. So, if you are interested in `pointer` only, you can use `new` function instead of creating a new variable as making a pointer points to it.

> *Hence definition of `A pointer is a variable which stores the memory address of another variable` is strictly not true. `A pointer is a variable which stores a memory address` is more accurate.*

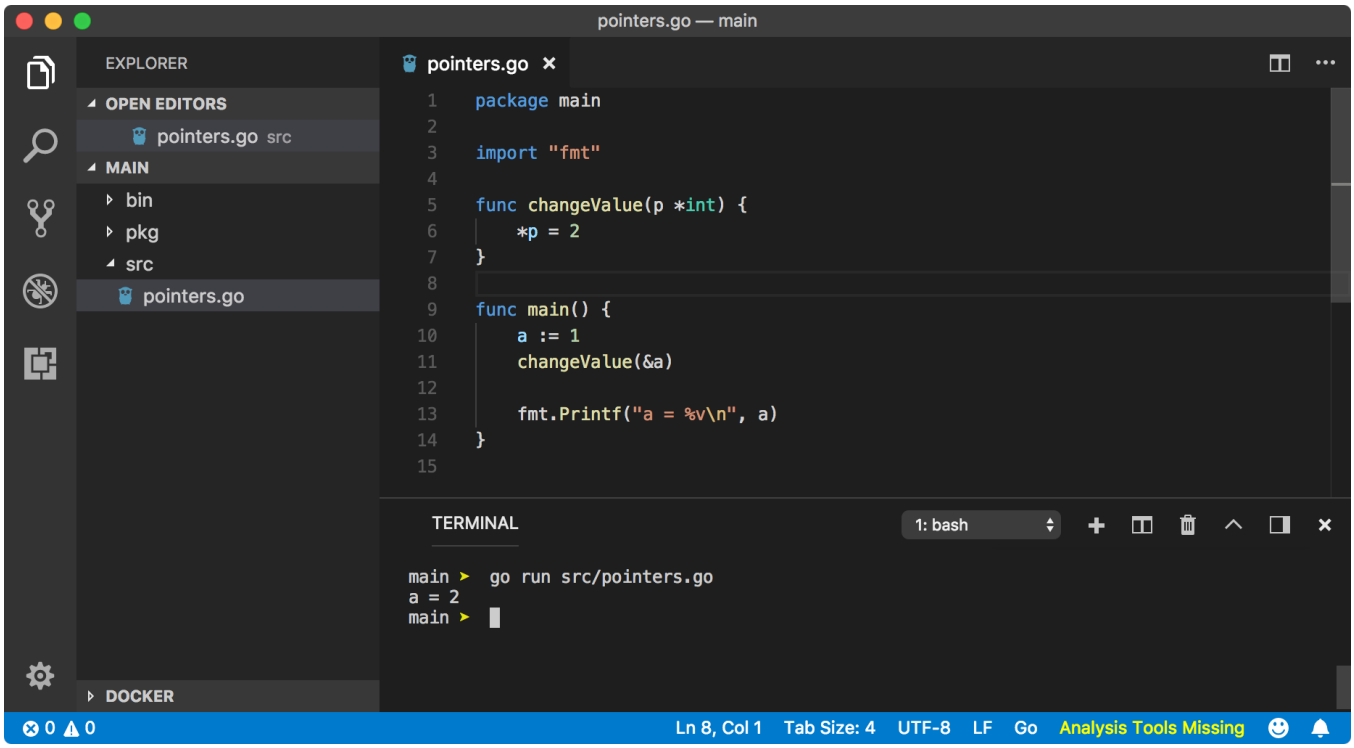
☛ Passing a pointer to a function

Like a variable, you can pass a pointer to a function. There are two ways you can do this. Either create a pointer and then pass it to the function or just pass an address of a variable.



https://play.golang.org/p/CZof2zNT9Jo

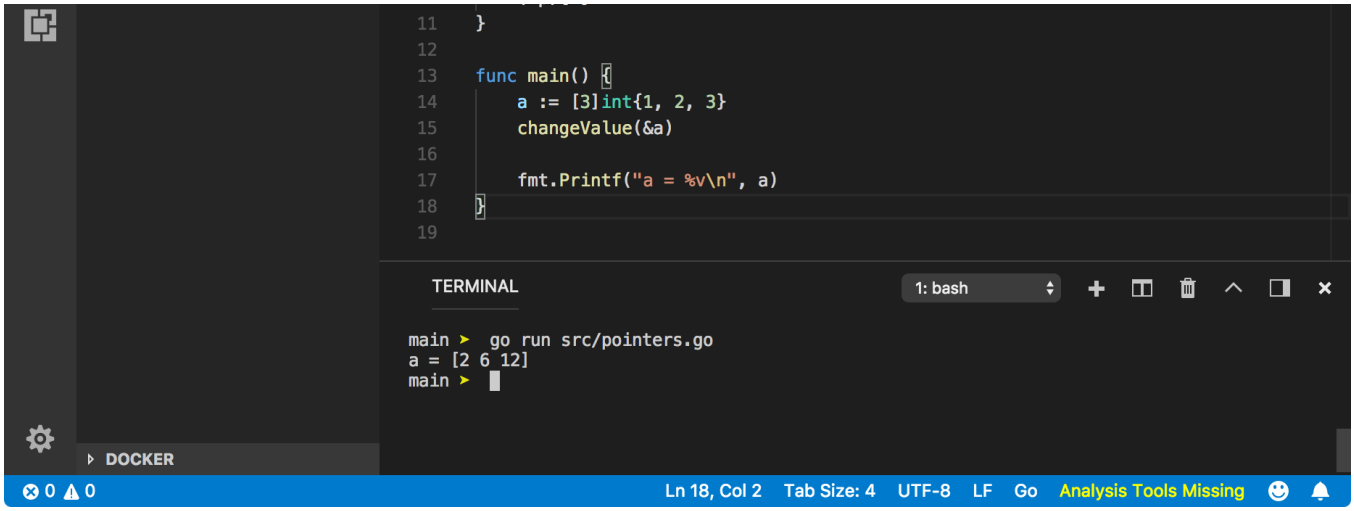
We could write above program like



https://play.golang.org/p/kr-LTGyjZs0

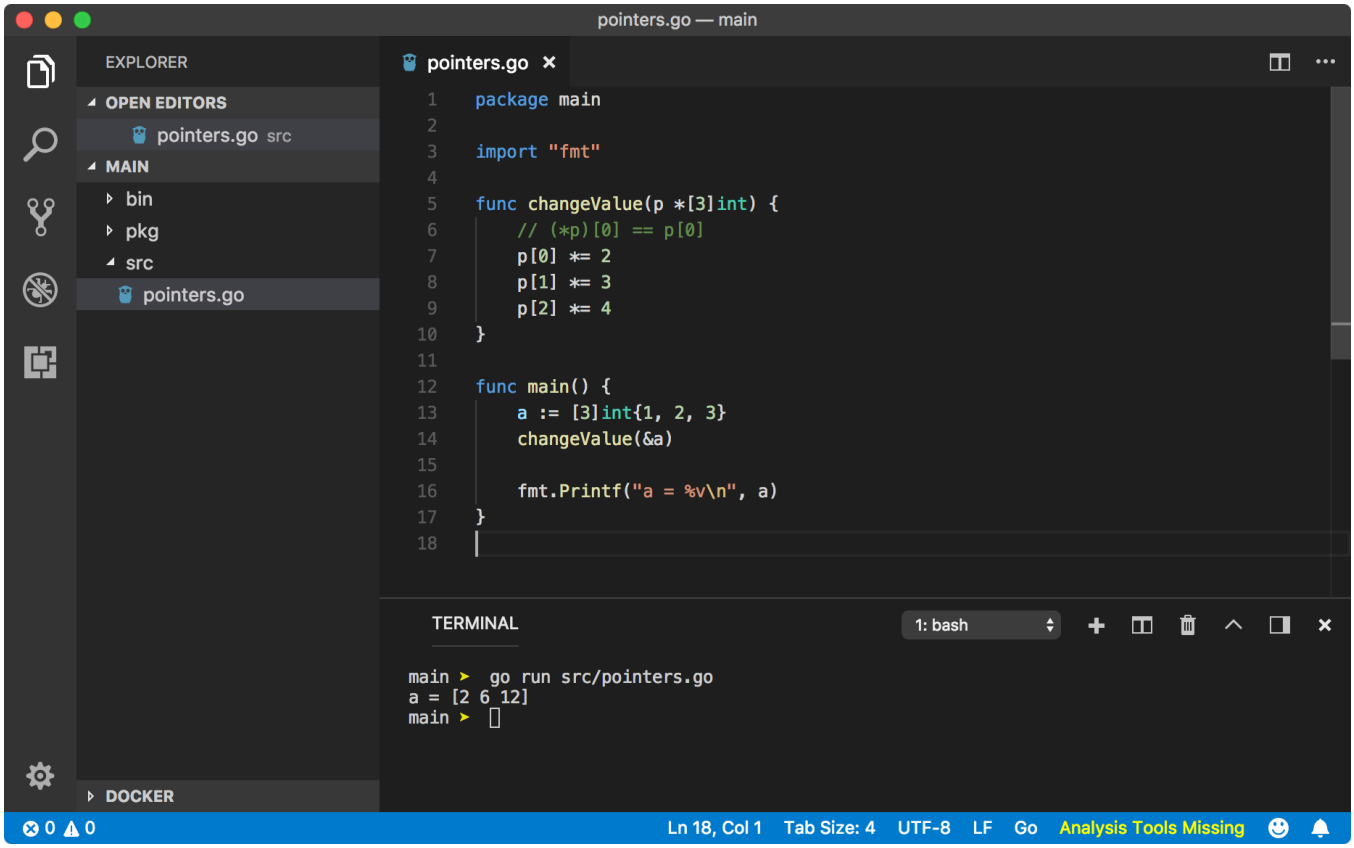
In the above program, the syntax of `changeValue` function instructs Go that we are expecting a pointer `p`, specifically `*int` part of the argument.

You can pass a pointer of a composite data types like `array` in function.



https://play.golang.org/p/wp2QxYOSSBC

We could also write above program using shorthand syntax provided by Go to access data from array pointer



https://play.golang.org/p/GLa6BJ7bvc_G

But passing array pointer as function parameter is not idiomatic to Go. As we saw in `slice`, we can modify a slice inside a function and see changes in the original slice. So if that's what you want as we did in the above program, go for it.

☛ Pointer arithmetic

Unlike `C` where a pointer can be incremented or decremented, Go does not allow pointer arithmetic. If you tried to do so, Go will throw compilation error.