**Variables**

Variables are at the heart of the language and provide the ability to read from and write to memory. In Go, access to memory is type safe. This means the compiler takes type seriously and will not allow us to use variables outside the scope of how they are declared.

**Notes**

* The purpose of all programs and all parts of those programs is to transform data from one form to the other.
* Code primarily allocates, reads and writes to memory.
* Understanding type is crucial to writing good code and understanding code.
* If you don't understand the data, you don't understand the problem.
* You understand the problem better by understanding the data.
* When variables are being declared to their zero value, use the keyword var.
* When variables are being declared and initialized, use the short variable declaration operator

**Struct Types**

Struct types are a way of creating complex types that group fields of data together. They are a great way of organizing and sharing the different aspects of the data your program consumes.

A computer architecture’s potential performance is determined predominantly by its word length (the number of bits that can be processed per access) and, more importantly, memory size, or the number of words that it can access.

**Notes**

* We can use the struct literal form to initialize a value from a struct type.
* The dot (.) operator allows us to access individual field values.
* We can create anonymous structs.

[Understanding Type in Go](https://www.ardanlabs.com/blog/2013/07/understanding-type-in-go.html)

**Pointers**

Pointers provide a way to share data across program boundaries. Having the ability to share and reference data with a pointer provides the benefit of efficiency. There is only one copy of the data and everyone can see it changing. The cost is that anyone can change the data which can cause side effects in running programs.

**Notes**

* Use pointers to share data.
* Values in Go are always pass by value.
* "Value of", what's in the box. "Address of" ( **&** ), where is the box.
* The (\*) operator declares a pointer variable and the "Value that the pointer points to".

**Escape Analysis**

* When a value could be referenced after the function that constructs the value returns.
* When the compiler determines a value is too large to fit on the stack.
* When the compiler doesn’t know the size of a value at compile time.
* When a value is decoupled through the use of function or interface values.

[Language Mechanics On Stacks And Pointers](https://www.ardanlabs.com/blog/2017/05/language-mechanics-on-stacks-and-pointers.html)

[Using Pointers In Go](https://www.ardanlabs.com/blog/2014/12/using-pointers-in-go.html)

[Understanding Pointers and Memory Allocation](https://www.ardanlabs.com/blog/2013/07/understanding-pointers-and-memory.html)

**Constants**

Constants are a way to create a named identifier whose value can never change. They also provide an incredible amount of flexibility to the language. The way constants are implemented in Go is very unique.

**Notes**

* Constants are not variables.
* They exist only at compilation.
* Untyped constants can be implicitly converted where typed constants and variables can't.
* Think of untyped constants as having a Kind, not a Type.
* Learn about explicit and implicit conversions.
* See the power of constants and their use in the standard library.

[Introduction To Numeric Constants In Go](https://www.ardanlabs.com/blog/2014/04/introduction-to-numeric-constants-in-go.html)