Module Two

Variables and Data Types

In Go, the code does exactly what it says on the page.
Andrew Gerrand

Go doesn't implicitly anything.
Steven in golang-nuts

2.1 Introduction

Programming languages generally tend to fall into one of two categories. On one hand we have the strictly typed compiled languages like C, C++ and Java, while on the other we have the untyped interpreted languages like Python, Ruby and JavaScript. Generally the languages within each category tend to resemble each other in how they manage variables – if you know C++ you can generally translate that knowledge into Java, for example, without too much problem.

Go is a strictly typed compiled language which handles variables similar to how Java and C++ does... sort of. Go generally is not as casual as other languages about casting and converting from one data type to another, a quirk that which can feel somewhat restrictive initially if this is something you commonly do while writing code.

The whole idea of implicit conversion of types is forbidden in Go. Other strictly typed languages permit implicit widening conversions (copying the value from a smaller size variable into a larger size variable such as a short to a long) without any extra syntax. Go does not even allow that relatively safe implicit conversion.

In this module, we will briefly introduce the basic data types. We are not going to go into detail about what they are since the Go data types are pretty standard and should be familiar to you from other C-style languages. We will only focus on the few things that Go does that are somewhat unique to the language.

The section on variables will assume you know how to use variables in other C-style languages and understand concepts like scope, what a local variable is and other fundamental programming concepts related to variables.

The last section deals with pointers since they are used extensively in Go. If you are a C or C++ programmer, this material will be quite familiar to you. If you are a Java programmer, you may have to adjust from thinking in terms of references to thinking in pointers. If you are unfamiliar with pointers, then welcome to what C programmers call “real programming.”.

2.2 Basic Data types

Data Types

Numeric				Non-numeric	
integer		floats	complex	string	bool
signed	unsigned	float32	complex64		
		float64	complex128		
int8	uint8				
int16	uint16				
int32	uint32				
int64	uint64				

int - either int32 or int64
uint - either uint32 or uint64

rune alias for int32
byte alias for uint8

Most of the data types are the same as in other programming languages. The suffix on the numeric types indicates the size of that type. The float types are defined as the IEEE-754 32 bit and 64 bit floating point numbers complete with NaN and the infinity values. The integer forms are either signed twos-complement forms or unsigned integers, much the same as in C. The main difference is that Go has dumped the traditional type names like double, short and long and replaced them with a rather utilitarian naming of the numeric types.

The one addition that most other languages do not have is the complex number type. A complex number is an ordered pair of two floats (a,b) that represent the complex number a + bi. Complex numbers don't occur that often outside of scientific work so we will not be delving into them in this course. As a mathematician, it grieves me to say that.

The bool values are as you would expect, true and false, with no numerical equivalent like that which exists in C and C++.

2.2.1 Strings

Strings are handled quite differently in Go than most other C-style languages. In C and C++ strings are pointers to arrays of integers while Java also treats strings more like an array or object than a basic built-in data type. Strings in Go are basic data types just like the numeric types.

Like Java, strings in Go are immutable and for the same reason: when a string is immutable then it can be allocated into contiguous memory without worrying about ever having to support re-sizing operations. This results in faster and more efficient string processing.

However strings are not quite as simple as in other C-style languages. Strings use UTF-8 encoding in Go and are actually arrays of what we call code points, which we can think of as characters in some alphabet, where each code point can be represented by 1 to 4 bytes of data. The aliases for rune and byte are specifically implemented to help us use UTF-8 strings: runes represent code points while bytes are exactly that.

2.2.2 The int and uint types

The uint and int types are very C-like since they parallel the notion of what an int is in C. The size of these two types is either 32 or 64 bits depending on the underlying architecture of the target machine. The programmer cannot override these and force these types be the size they prefer, that decision is made by the compiler and is out of the control of the programmer. If a specific size is wanted for int, then the corresponding int32 or int64 should be used, and similarly for the uint.

2.2.3 No implicit data conversions

Most programming languages, like Java, allow some sort of implicit conversion of data types. For example, the assignment of a smaller int variable into a larger one is generally allowed because it is considered safe. Similarly, when we divide an integer by a float, generally called a mixed mode operation where the integer is implicitly changed into a float before the division takes place.

So why are implicit conversions and mixed mode arithmetic not allowed in Go? Simply because this is a common source of errors in code and requires the implementation of a lot of behind the scene logic to resolve what sort of promotions or conversions should be done to produce a "correct" result for these sorts of expressions. The kinds of errors that result from these sorts of conversions are usually not syntactic errors that cause some compile time problems but are semantic errors that are not caught until unusual results start showing up during testing, or even worse, during production.

Most programmers have had experience with this sort of problem when data types are converted implicitly in some computation and wind up producing a different result than the one the programmer was expecting when writing the code.

Many programmers, myself included, always explicitly cast data types when we do conversions or mixed mode arithmetic because we know that this is a source of those “duh” errors we have a habit of making. Go just makes that explicit conversion mandatory to eliminate a whole class of problems.

From the Go Blog

> *In the early days of thinking about Go, we talked about a number of problems caused by the way C and its descendants let you mix and match numeric types. Many mysterious bugs, crashes, and portability problems are caused by expressions that combine integers of different sizes and "signedness".*

Nasty bugs lurk here. C has a set of rules called "the usual arithmetic conversions" and it is an indicator of their subtlety that they have changed over the years (introducing yet more bugs, retroactively). When designing Go, we decided to avoid this minefield by mandating that there is no mixing of numeric types. If you want to add i and u, you must be explicit about what you want the result to be.

This strictness eliminates a common cause of bugs and other failures. It is a vital property of Go. But it has a cost: it sometimes requires programmers to decorate their code with clumsy numeric conversions to express their meaning clearly.

2.3 Variable Declaration

Example 02-01 shows what is called the basic long form for initializing variables. The var keyword is required in this form. This is the only form of variable declaration that can be used for package level variable declarations. Why? The same basic reason that we use the func keyword to declare function definitions – it simplifies the lexer component of the compilation process, which will be necessary when we start using functions as first class object. Just like needing that opening "{" on the same line as the function declaration – it allows a much more efficient parsing process. We don't need this efficiency quite as much at the local level because the lexer has more contextual information to work with which means we can use a more compact form for defining local variables.

```
// Example 02-01  Declaring Variables
package main

import "fmt"

var x float32
var c complex64
var b bool
var str string

func main() {
  var message string = "x=%f c=%f b=%t str=|%s| i=%d \n"
  var i int
  fmt.Printf(message, x, c, b, str, i)
}
```

```
[Module02]$ go run ex02-01.go
x=0.000000 c=(0.000000+0.000000i) b=false str=|| i=0
```

The example uses the fmt.Printf() function which is from the standard fmt package. The function works a lot like the fprint() or sprintf() functions in other C-style languages so we won't discuss it any further here. If you are unfamiliar with printf() types, check the documentation for the fmt package in the Go documentation.

2.3.1 Default Zero Value

The output of example 02-01 demonstrates the Go principle that there can never be an uninitialized variable. If a variable is not explicitly initialized, then it is set to the zero value for that type. The zero value is the appropriate numeric zero for all the numeric types, false for Boolean values and the empty string for string values. In most C-style languages string variables are actually pointers or references so the undefined value of a string is the null pointer or nil or null or something similar. Since strings are basic data types, the zero value for strings is the empty string.

2.3.2 Explicit Initialization with Literals

Example 02-02 demonstrates how variables are declared and explicitly initialized using literals. One important thing to notice is that we may have to convert the literal to the specific type we need in order to do the initial assignment.

The reason for this is that while literals themselves don't have a type associated with them, they have a default type that is used if one is not explicitly provided. For example, the default type for the literal “0” is int, which means that when we use “0” to initialize a complex variable, we have to convert it first to “0.0 + 0.0i” which is the 0 value for a complex number.

```
// Example 02-02  Initializing Variables
package main

import "fmt"

var x float32 = 32.0
var c complex64 = complex64(0)
var b bool = true
var str string = "Hi"

func main() {
  var message string = "x=%f c=%f b=%t str=|%s| i=%d \n"
  var i int = 3
  fmt.Printf(message, x, c, b, str, i)
}
```

```
[Module02]$ go run ex02-02.go
x=32.000000 c=(0.000000+0.000000i) b=true str=|Hi| i=3
```

2.3.3 Implicit Initialization with Literals

If the literal being used to initialize a variable has a known type, then we don't need a type identifier for the variable, it is implied from the type of the initializing literal.

Consider the literal 0 – it could be a int8 or a uint16 or an int128. Because in Go a literal has a default type, the literal 0 defaults to an int in the absence of any other information. If we want the literal to take on any other possible type then we use the conversion operation to change the type.

However we cannot do impossible conversions, for example int64(true) will not work since we can't convert a non-numeric into a numeric.

Example 02-03 shows implicit initialization. Just as a note the “%T” in the fmt.Printf() statement prints out the type of the variable.

```
// Example 02-03  Implicit Initialization
package main

import "fmt"

var x float32 = 32.0
var c = 5.0 + 3.1i
var b = true
var str = "Hi"

func main() {
  var message = "x=%T c=%T b=%T str=%T i=%T \n"
  var i = 3
  fmt.Printf(message, x, c, b, str, i)
}
```

```
[Module02]$ go run ex02-03.go
x=float32 c=complex128 b=bool str=string i=int
```

Examples 02-04 and 02-05 show some variations on the syntax that can be used in declaring both package level and local variables.

If we are initializing a number of variables in a single statement as we do in the second "var" statement in the example, then we have to initialize all or none. We cannot have a mismatch between the number of variables and the number of initializers.

Another conservative decision Go has made is to make it a compile time error to declare or define a variable and then not use it. In other languages this is usually a warning usually but Go takes a more code safe approach and makes it an error.

```
// Example 02-04  Implicit Initialization
package main

import "fmt"

var x = float32(0)
var c = complex64(0)
var b = true
var str = "Hi"

func main() {
  var message = "x=%T c=%T b=%T str=%T i=%T \n"
  var i = uint8(0)
  fmt.Printf(message, x, c, b, str, i)
}
```

```
[Module02]$ go run ex02-04.go
x=float32 c=complex64 b=bool str=string i=uint8
```

```go
// Example 02-05  Variations
package main

import "fmt"

var i, j int8
var b, str, x = true, "hi", float32(45)

func main() {
  fmt.Printf("i=%T j=%T b=%T str=%T x=%T \n",
       i, j, b, str, x)
}
```

```
[Module02]$ go run ex02-05.go
i=int8 j=int8 b=bool str=string x=float32
```

2.4 Short Form Variable Declaration

When we are declaring variables inside a local scope (ie. not at the package level) we can use the short form of a declaration which is demonstrated in example 02-06

The differences between the long form and the short form are:

1. The var is omitted in the short form and the assignment operator ":=" is used instead of "=".
2. The values on the left hand side of the ":=" may be existing variables like “k” that are being assigned a new value or variables that are being created at that point in the code like “i”.
3. At least one of the variables on the left hand side of the ":=" must be a new declaration otherwise the “=” must be used instead of the “:=” operator.
4. There is no explicit type declaration, the type is always inferred from the initializing literal.
5. This form can only be used with local variables, never with variables at the package level.

Why the short form? The short form will be necessary for doing in-line variable declarations in constructs like for loops where the longer syntax using the "var" keyword just can't work syntactically. The short form can be thought of as more of an "in-line" sort of construct.

```
// Example 02-06  Short Form
package main

import "fmt"

var (
  k int = 1
  m int
)

func main() {

  i, j, k := 3, 4, 5
  x := 1.2

  fmt.Printf("i=%d j=%d x=%f k=%d \n", i, j, x, k)
}
```

```
[Module02]$ go run ex02-06.go
i=3 j=4 x=1.200000 k=5
```

2.5 Scope

There are five kinds of scope in Go, two of which (file and universal) are not really that important to us as programmers. The scopes are:

1. *Universal Scope:* This refers to symbols (like “int32”) that are accessible everywhere in any package in a Go program. We don't write code for this package.
2. *File Scope:* This usually refers to symbols that have the current file as their scope. The package and import statements have file scope since they are only valid in the current file. Aside from these two cases, we don't use file scope in our code in this course.
3. *Package scope:* Symbols which have package scope can be referenced by any code in that package, even if the code is in a different file. These are roughly the global variables of Go, or at least global to the package in which they are defined. Symbols with package scope are always defined with some initial keyword such as func, var or type.
4. *Function scope:* This scope should be familiar to all programmers. These are the variables that are defined in the body of a function or passed as parameters or, in the case of Go, are named return values (more about them later).
5. *Block scope:* Like C and other C-style languages, Go allows the definition of variables that are local to a block, which is a set of code statements delimited by braces { }. Blocks usually occur as the bodies of if statement, for loops or other similar constructs, although we can define a block almost anywhere we want to in a function body.

Example 02-07 on the next page demonstrates this concept.

1. The variable k has package scope indicated by the outermost green box.
2. The variable I has function scope which is indicated by the red box
3. The variable x has block scope indicated by the innermost blue box.

```go
// Example 02-07  Variable Scope
package main

import "fmt"

var k int = 1

func main() {

  i := 1

  {
      x := 1.2
      fmt.Printf("i=%d x=%f k=%d \n", i, x, k)
  }
}
```

2.5.1 Shadowing

We are assuming that everyone is familiar with the idea of variable shadowing since it occurs in all C-style programming languages in one form or another. Go allows variable shadowing without any warning or error from the compiler. This is illustrated in example 02-08

```
// Example 02-08  Shadowing
package main

import "fmt"

var k int = 1

func main() {

  fmt.Printf("Package Scope k=%d \n", k)
  k := 2
  fmt.Printf("Function Scope k=%d \n", k)
  {
       k := 3
       fmt.Printf("Block Scope k=%d \n", k)
  }
  fmt.Printf("Function Scope k=%d \n", k)
}
```

```
[Module02]$ go run ex02-08.go
Package Scope k=1
Function Scope k=2
Block Scope k=3
Function Scope k=2
```

One reason that shadowing of package variables is allowed without comment is that a package variable may be defined in a different file in the same package – not allowing it to be shadowed could cause some very cryptic appearing errors to the programmer who is trying to define a second variable with the same name as an existing variable at a lower level of scope.

2.6 Constants

Constants work the same way as variables both syntactically and semantically with the following differences.

1. Once a constant's value has been initialized, it may not be changed.
2. Defining a constant uses the keyword const instead of var.
3. Because of (2), the short form with ":=" cannot be used to define a constant
4. The value of a constant can be computed by a constant expression, which is a computation that can be done at compile time by the compiler.

Example 02-09 demonstrates the use of constants.

```
// Example 02-09  Constants
package main

import "fmt"

const a float32 = 1
const b = 4 / 3

func main() {
  const c string = "Hello Word"
  fmt.Printf("a=%T b=%T c=%T\n", a, b, c)
  fmt.Printf("a=%f b=%d c=%s\n", a, b, c)
}
```

```
[Module02]$ go run ex02-09.go
a=float32 b=int c=string
a=1.000000 b=1 c=Hello Word
```

2.6.1 iota

iota is an enum generator which was borrowed from a rather innovative programming language called APL. It is used to generate sets of related constants which represent some sort of integral enumeration.

Within a set of constants (which means they are all defined in the same const statement) the first time iota is used, it assigns the first constant the value 0. Then each constant following is assigned the next integer – 1, 2, 3 … etc.

This is illustrated in example 02-10.

```
// Example 02-10 iota and enums
package main

import "fmt"

const (
  a = iota
  b
  c
)

func main() {
  fmt.Printf("a=%T b=%T c=%T\n", a, b, c)
  fmt.Printf("a=%f b=%d c=%s\n", a, b, c)
}
```

```
[Module02]$ go run ex02-10.go
a=int b=int c=int
a=0 b=1 c=2
```

2.6.1 The Blank Variable

If we want the enum series to start at 1 instead of 0 we can use the blank variable "_" for the first enum constant. The blank variable is used extensively in Go when we need to assign a value to something, but want to ignore the value after the assignment. The blank variable is a placeholder used instead of a real variable, which we would then have to use in a statement to avoid a compiler error. The blank variable has no type and cannot be used in any context except as the target in a variable assignment statement. That means that the following statement is not allowed:

```
_ := 42
```

This is illustrated in example 02-11

```
// Example 02-11 iota and enums
package main

import "fmt"

const (
  _ = iota
  a
  b
  c
)

func main() {
  fmt.Printf("a=%T b=%T c=%T\n", a, b, c)
  fmt.Printf("a=%f b=%d c=%s\n", a, b, c)
}
```

```
[Module02]$ go run ex02-11.go
a=int b=int c=int
a=1 b=2 c=3
```

2.6.2 Generator Functions

We don't have to have just a series of sequential integers as an enum series, we can use iota in a generator expression to compute a sequence of enum values. In example 02-12 a simple expression is used to generate the even integers as an enum series. However, the generator function used must be a constant function which can be computed by the compiler at compile time.

```
// Example 02-12 Enums with Generator
package main

import "fmt"

const (
  a = iota * 2
  b
  c
)

func main() {
  fmt.Printf("a=%T b=%T c=%T\n", a, b, c)
  fmt.Printf("a=%f b=%d c=%s\n", a, b, c)
```

```
[Module02]$ go run ex02-12.go
a=int b=int c=int
a=0 b=2 c=4
```

2.7 Pointers

Go uses pointers in a manner quite similar to C. Go has pointer variables which can contain the memory address of a particular type of value. The notation for the addressing operators and de-referencing operators is the same as for C and C++.

For example if we declare the integer variable "i"

```
var i int
```

then we can declare p to be a pointer to an integer with the declaration

```
var p *int
```

They "type" of a pointer is the type of the variable it points to. If p is a pointer to an integer, then it cannot point to a string or a float.

We can assign the address of i to p by using the address-of operator &

```
p = &i
```

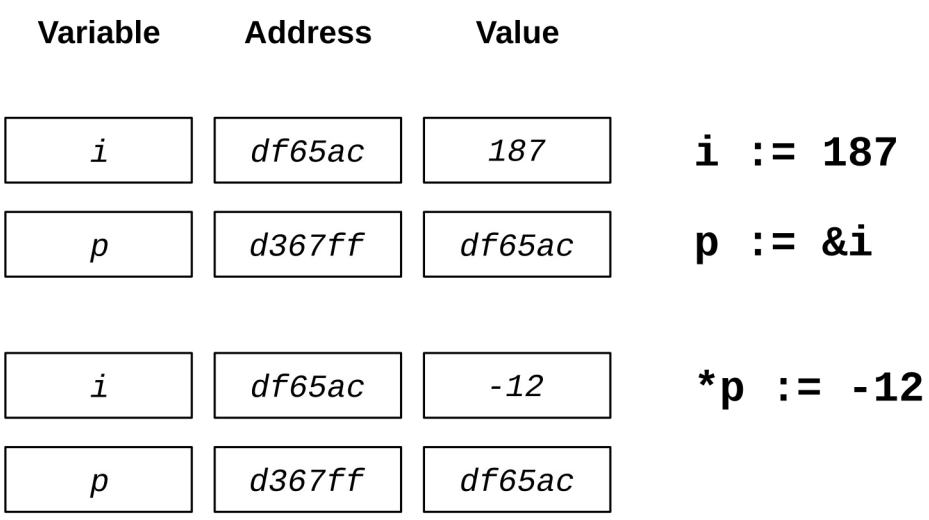
To access the contents of what p points to, we use the de-referencing operator “*”.
If we wanted to assign the value that p points to to another variable, then we would say

```
int j = *p
```

which is often read a “j is assigned the contents of p.”

Pointers are illustrated in example 02-13.

Go allows us to use assignment to assign one pointer to another, and allows the use of the == operator to see if two pointers point to the same memory location. However Go does not support the sort of pointer arithmetic that C and C++ do because it was decided that this provided minimal value to the programmer while permitting operations that historically have been a major source of problems in C and C++ programs. We will play with pointers more in the lab as well as seeing them later in the course quite at bit.



The diagram on the previous page is a graphical illustration of the example. The variable “i” is located at a memory address, which we are calling “df65ac” and the contents of that memory location are the value of “i” which ls the integer 187.

When “p” is assigned the address of i, it is implicitly takes on the type “pointer to int” and itself is a variable with a memory location. However the value of p is the address of the variable “i.” We can do a lot with p, we can assign it to another variable of type “pointer to int” for example, or we can assign it the address of a different int.

However we can also access the value of what “p” points to indirectly by using the “*p” notation, which can be read as “the value of the variable that p points to.” This is often called the indirection operator or the de-referencing operator. We can use this to change the value of “i” to -12, but notice that the value of “p” itself does not change.

```
// Example 02-13 Pointers

package main

import "fmt"

func main() {
  var i int = 187
  var p *int

  p = &i
  fmt.Println("i=", i, " &i or p =", p, " *p =", *p)
  *p = -12
  fmt.Println("i=", i, " &i or p =", p, " *p =", *p)
}
```

```
[Module02]$ go run ex02-13.go
i= 187  &i or p = 0xc82000a398  *p = 187
i= -12  &i or p = 0xc82000a398  *p = -12
```