**PROJECT PART 1 FALL 2016**

**LEARNING AND EXPERIMENTATION**

**GO LANGUAGE**

1. **PARADIGM OF GO LANGUAGE**

“Go was born out of frustration with existing languages and environments for systems programming.” Programming had become too difficult and the choice of languages was partly to blame. One had to choose either efficient compilation, efficient execution, or ease of programming; all three were not available in the same mainstream language. Programmers who could were choosing ease over safety and efficiency by moving to dynamically typed languages such as Python and JavaScript rather than C++ or, to a lesser extent, Java.

Go is an attempt to combine the ease of programming of an interpreted, dynamically typed language with the efficiency and safety of a statically typed, compiled language. It also aims to be modern, with support for networked and multicore computing. Finally, working with Go is intended to be fast: it should take at most a few seconds to build a large executable on a single computer.”

Go is compiled, statically typed language. Go language ancestors include C language syntax, Communication Sequential Process by C.A.R Hoare’s concurrency model. It also have significant input from Pascal, Modula and Oberon family. Such design decision make Go language to adhere to above mentioned goal of Go language.

1. **HISTORY OF GO PROGRAMMING LANGUAGE**

Go originated as an experiment by Google engineers Robert Griesemer, Rob Pike, and Ken Thompson to design a new programming language that would resolve common criticisms of other languages while maintaining their positive characteristics. The new language was to:

* To be statically typed, scalable to large systems (as [Java](https://en.wikipedia.org/wiki/Java_(programming_language)) and [C++](https://en.wikipedia.org/wiki/C%2B%2B));
* To be [productive](https://en.wikipedia.org/wiki/Programming_productivity) and readable, without too many mandatory keywords and repetition ("light on the page" like [dynamic languages](https://en.wikipedia.org/wiki/Dynamic_language))
* For not requiring Tooling, but support it well.
* Support networking and multiprocessing.

In an interview with the designers of Go language, all three of the language designers cited their shared dislike of C++'s complexity as a primary motivation for designing a new language.

**Latest on Go:**

Go 1.7 added "one tiny [language](https://en.wikipedia.org/wiki/Syntax_(programming_languages)) change" and one port to [MacOS 10.12 Sierra](https://en.wikipedia.org/wiki/MacOS_Sierra) plus some experimental ports, e.g. for [Linux on Z Systems](https://en.wikipedia.org/wiki/Linux_on_z_Systems) (Linux/s390x). Some library changes apply, and e.g. [Unicode 9.0](https://en.wikipedia.org/wiki/Unicode_9.0) is now supported.

1. **ELEMENTS OF GO**

The elements of Go programming language form the basic building blocks of the language. The following below listed keywords, datatypes form the most important elements of the language.

***Reserved Keywords***

The following list shows the reserved words in Go. These reserved words may not be used as constant or variable or any other identifier names.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Break | default | func | interface | Select |
| Case | defer | go | map | Struct |
| Chan | else | goto | package | Switch |
| Const | fallthrough | if | range | Type |
| continue | for | import | return | Var |

***Datatypes***

In the Go programming language, data types refer to an extensive system used for declaring variables or functions of different types. The type of a variable determines how much space it occupies in storage and how the bit pattern stored is interpreted.

The types in Go can be classified as follows:

|  |  |
| --- | --- |
| **S.No.** | **Types and Description** |
| 1 | **Boolean Types**  They are Boolean types and consists of the two predefined constants: (a) true (b) false |
| 2 | **Numeric Types**  They are again arithmetic types and they represent a) integer types or b) floating point values throughout the program. |
| 3 | **string types:**  A string type represents the set of string values. Its value is a sequence of bytes. Strings are immutable types that is once created, it is not possible to change the contents of a string. The predeclared string type is string. |
| 4 | **Derived types:**  They include (a) Pointer types, (b) Array types, (c) Structure types, (d) Union types and (e) Function types f) Slice types g) Function types h) Interface types i) Map types j) Channel Types |

The array types and structure types are referred to collectively as the aggregate types. The type of a function specifies the set of all functions with the same parameter and result types.

***Integer Types***

The predefined architecture-independent integer types are:

|  |  |
| --- | --- |
| **S.N.** | **Types and Description** |
| 1 | **uint8**  Unsigned 8-bit integers (0 to 255) |
| 2 | **uint16**  Unsigned 16-bit integers (0 to 65535) |
| 3 | **uint32**  Unsigned 32-bit integers (0 to 4294967295) |
| 4 | **uint64**  Unsigned 64-bit integers (0 to 18446744073709551615) |
| 5 | **int8**  Signed 8-bit integers (-128 to 127) |
| 6 | **int16**  Signed 16-bit integers (-32768 to 32767) |
| 7 | **int32**  Signed 32-bit integers (-2147483648 to 2147483647) |
| 8 | **int64**  Signed 64-bit integers (-9223372036854775808 to 9223372036854775807) |

***Floating Types***

The predefined architecture-independent float types are:

|  |  |
| --- | --- |
| **S.N.** | **Types and Description** |
| 1 | **float32**  IEEE-754 32-bit floating-point numbers |
| 2 | **float64**  IEEE-754 64-bit floating-point numbers |
| 3 | **complex64**  Complex numbers with float32 real and imaginary parts |
| 4 | **complex128**  Complex numbers with float64 real and imaginary parts |

The value of an n-bit integer is n bits and is represented using two's complement arithmetic operations.

***Other Numeric Types***

There is also a set of numeric types with implementation-specific sizes:

|  |  |
| --- | --- |
| **S.N.** | **Types and Description** |
| 1 | **Byte**  same as uint8 |
| 2 | **Rune**  same as int32 |
| 3 | **Uint**  32 or 64 bits |
| 4 | **Int**  same size as uint |
| 5 | **Uintptr**  an unsigned integer to store the uninterpreted bits of a pointer value |

**4. A DESCRIPTION OF THE SYNTAX OF THE LANGUAGE**

Languages outside the C family usually use a distinct type syntax in declarations. Although it's a separate point, the name usually comes first, often followed by a colon.Thus, our examples above become something like (in a fictional but illustrative language)

**x: int  
p: pointer to int  
a: array[3] of int**

These declarations are clear, if verbose - you just read them left to right. Go takes its cue from here, but in the interests of brevity it drops the colon and removes some of the keywords:

**x int  
p \*int  
a [3]int**

There is no direct correspondence between the look of [3]int and how to use a in an expression. You gain clarity at the cost of a separate syntax.

Now let’s consider functions. Let's transcribe the declaration for main as it would read in Go, although the real main function in Go takes no arguments:

**funcmain(argcint, argv []string) int**

Superficially that's not much different from C, other than the change from char arrays to strings, but it reads well from left to right:

function main takes an int and a slice of strings and returns an int.

Drop the parameter names and it's just as clear - they're always first so there's no confusion.

**funcmain(int, []string) int**

One merit of this left-to-right style is how well it works as the types become more complex. Here's a declaration of a function variable (analogous to a function pointer in C):

**f func(func(int,int) int, int) int**

Or if f returns a function:

**f func(func(int,int) int, int) func(int, int) int**

It still reads clearly, from left to right, and it's always obvious which name is being declared - the name comes first.

The distinction between type and expression syntax makes it easy to write and invoke closures in Go:

**sum := func(a, b int) int { return a+b } (3, 4)**

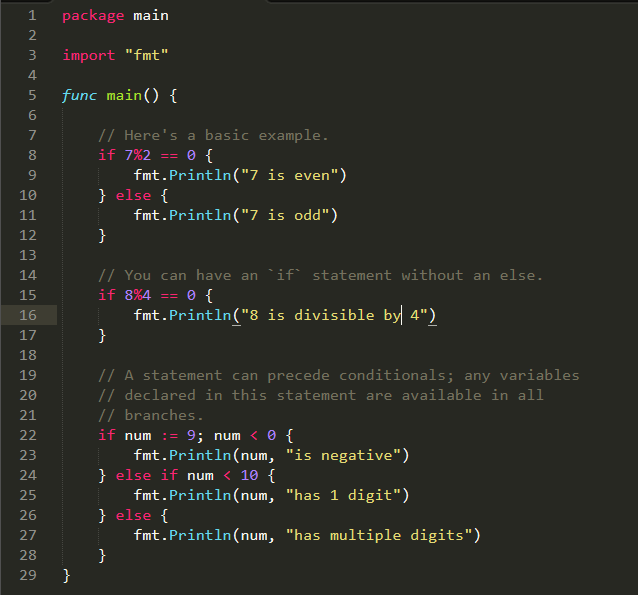
***ABSTRACTIONS OF THE GO LANGUAGE:***

***Conditional Control:***

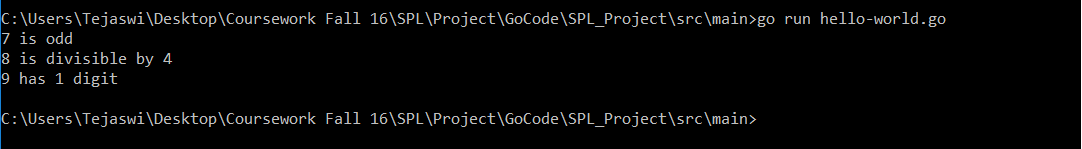
**if / else**

* Branching with if and else in Go is straight-forward.
* You can have an if statement without an else.
* A statement can precede conditionals; any variables declared in this statement are available in all branches.
* Note that you don’t need parentheses around conditions in Go, but that the braces are required.
* There is no [ternary if](http://en.wikipedia.org/wiki/%3F:) in Go, so you’ll need to use a full if statement even for basic conditions.
* Here’s a basic example:

***Code Snippet:***



***Output:***



## *Switch*

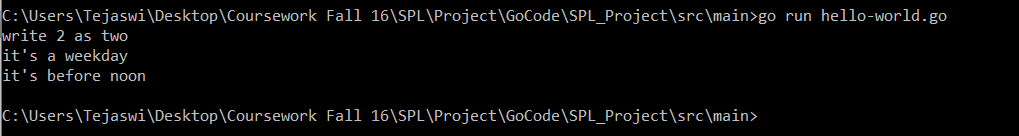
* Switch statements express conditionals across many branches.
* You can use commas to separate multiple expressions in the same case statement. We use the optional default case in this example as well.
* switch without an expression is an alternate way to express if/else logic. Here we also show how the case expressions can be non-constants.

Here’s an example:

***Code Snippet***:



***Output:***



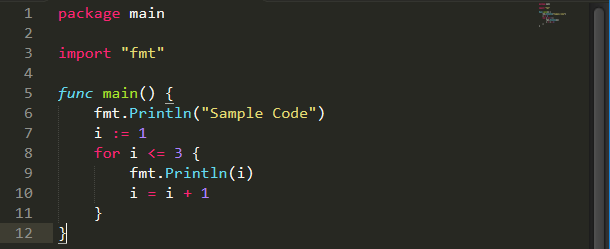
***Loops:***

Go has only one looping construct, i.e., **for**.

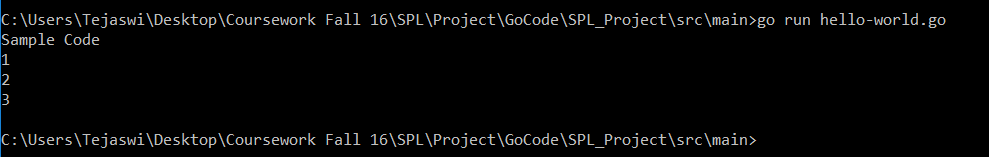
There are 3 variants for the **for** loop:

* 1. The most basic type, with single condition

***Code Snippet:***

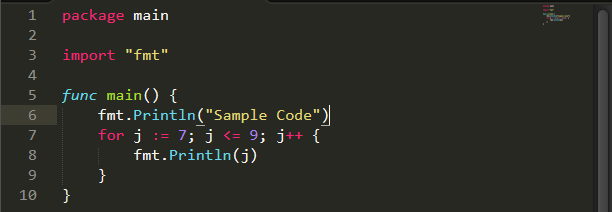


***Output:***

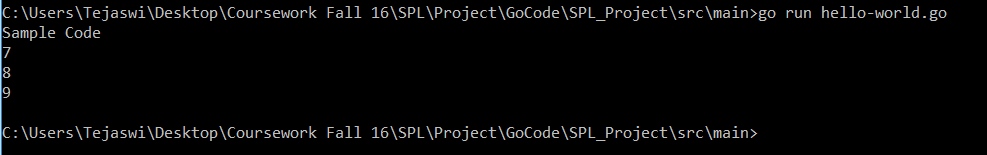


* 1. A classic initial/condition/after for loop.

***Code Snippet:***

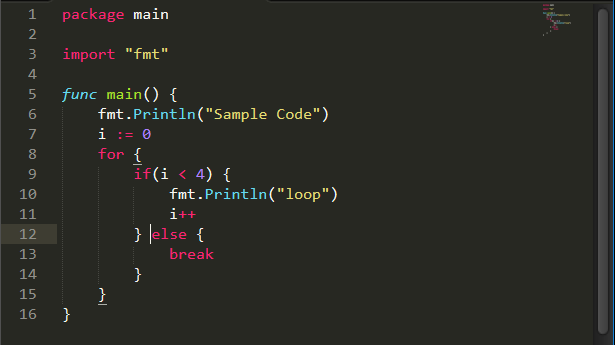


***Output:***

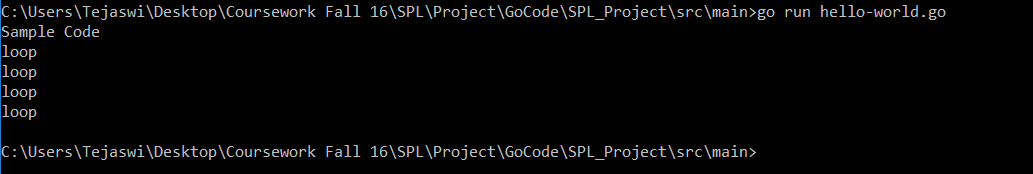


* 1. for without a condition will loop repeatedly until you break out of the loop or return from the enclosing function.

***Code Snippet:***



***Output:***

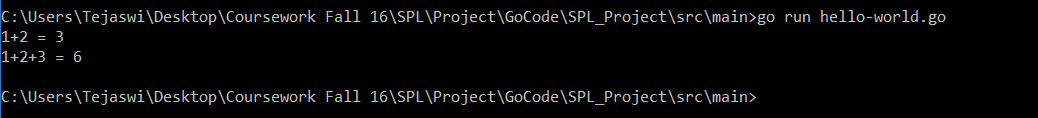


**5. FUNCTIONS**

* Functions are central in Go.
* Go requires explicit returns, i.e. it won’t automatically return the value of the last expression.
* When you have multiple consecutive parameters of the same type, you may omit the type name for the like-typed parameters up to the final parameter that declares the type.
* Call a function just as you’d expect, with name(args).
* Here’s a function that takes two int arguments and returns their sum as an int.



***Output:***

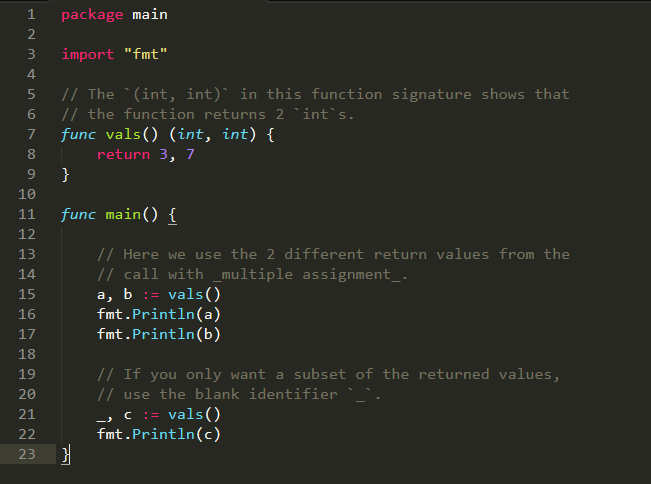


***Other features of functions in GO:***

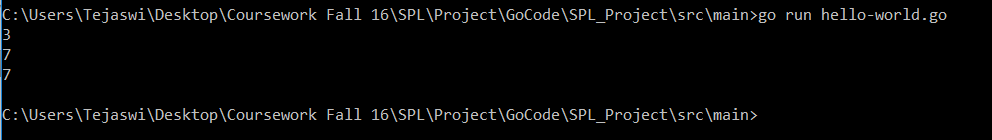
1. **Multiple Return Values:**

* Go has built-in support for multiple return values. This feature is used often in idiomatic Go, for example to return both result and error values from a function.
* The (int, int) in this function signature shows that the function returns 2 ints.
* If you only want a subset of the returned values, use the blank identifier \_.
* Here we use the 2 different return values from the call with multiple assignment.

***Code Snippet:***



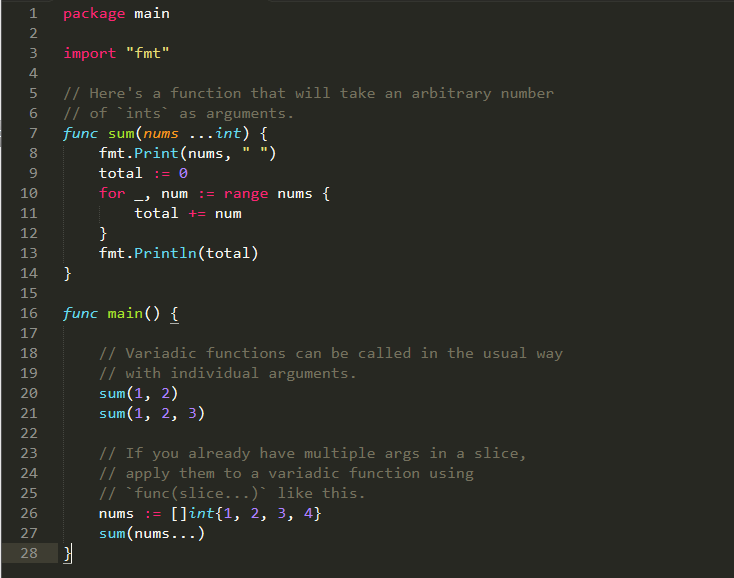
***Output:***



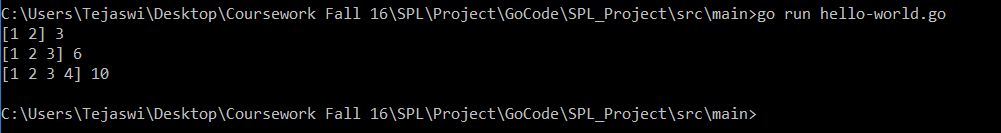
1. **Variadic functions**

* Variadic functions can be called with any number of trailing arguments. For example, fmt.Println is a common variadic function.
* Variadic functions can be called in the usual way with individual arguments.
* If you already have multiple args in a slice, apply them to a variadic function using func(slice...) like this.
* Here’s a function that will take an arbitrary number of ints as arguments.

***Code Snippet:***



***Output:***

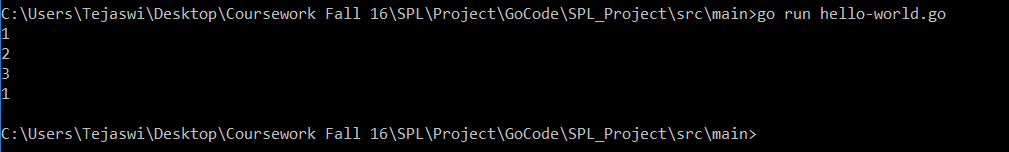


* 1. **Closures**
* Go supports anonymous functions, which can form closures. Anonymous functions are useful when you want to define a function inline without having to name it.
* In the code snippet below, the function intSeq returns another function, which we define anonymously in the body of intSeq. The returned function closes over the variable i to form a closure.
* We call intSeq, assigning the result (a function) to nextInt. This function value captures its own i value, which will be updated each time we call nextInt.
* See the effect of the closure by calling nextInt a few times.
* To confirm that the state is unique to that particular function, create and test a new one.

***Code Snippet:***



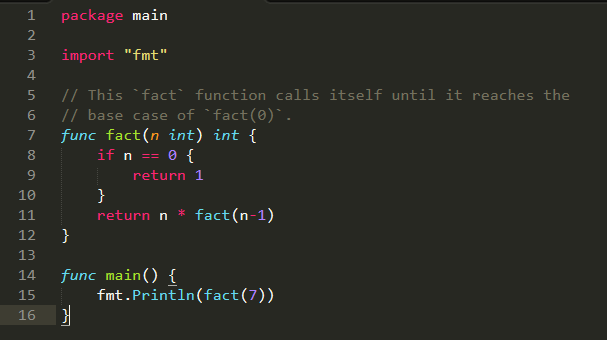
***Output:***



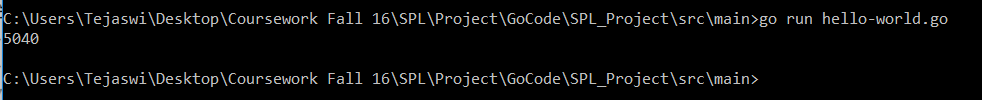
**4. Recursion**

* Go supports recursive functions.
* Here’s a classic factorial example. The fact function calls itself until it reaches the base case of fact (0).

***Code Snippet:***



***Output:***

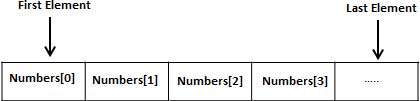


**6. ARRAYS**

Go programming language provides a data structure called the array, which can store a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables. A specific element in an array is accessed by an index.

All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.



***Declaring Arrays***

To declare an array in Go, a programmer specifies the type of the elements and the number of elements required by an array as follows:

**varvariable\_name[SIZE]variable\_type**

This is called a *single-dimensional* array. The **arraySize** must be an integer constant greater than zero and **type** can be any valid Go data type. For example, to declare a 10-element array called **balance** of type float32, use this statement:

**var balance [10] float32**

Now *balance* is avariable array which is sufficient to hold upto 10 float numbers.

***Initializing Arrays***

You can initialize array in Go either one by one or using a single statement as follows:

**var balance =[5]float32{1000.0,2.0,3.4,7.0,50.0}**

The number of values between braces { }can not be larger than the number of elements that we declare for the array between square brackets [ ].

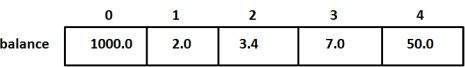
If you omit the size of the array, an array just big enough to hold the initialization is created. Therefore, if you write:

**var balance =[]float32{1000.0,2.0,3.4,7.0,50.0}**

You will create exactly the same array as you did in the previous example. Following is an example to assign a single element of the array:

balance[4]=50.0

The above statement assigns element number 5th in the array with a value of 50.0. All arrays have 0 as the index of their first element which is also called base index and last index of an array will be total size of the array minus 1. Following is the pictorial representation of the same array we discussed above:

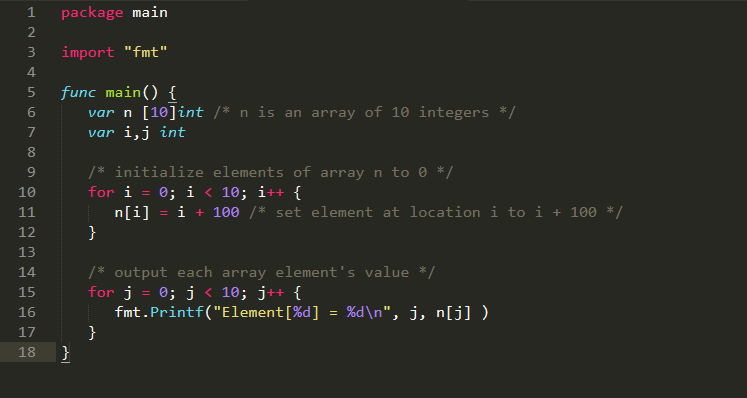


***Accessing Array Elements***

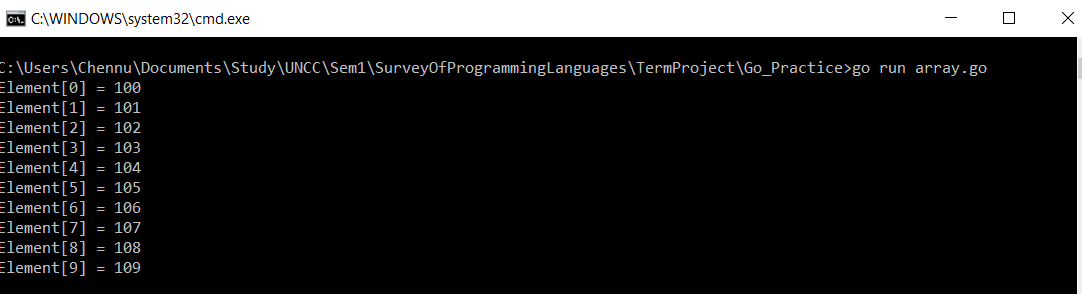
An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array. For example:

float32 salary =balance[9]

The above statement will take 10th element from the array and assign the value to salary variable. Following is an example which will use all the above mentioned three concepts viz. declaration, assignment and accessing arrays:



When the above code is compiled and executed, it produces the following result:



**7. POINTERS**

Pointers in Go programming language are easy and fun to learn. Some Go Programming tasks are performed more easily with pointers. Also, other tasks such as reference, cannot be performed without pointers.

***Definition:***

A pointer is a variable whose value is the address of another variable, i.e. direct address of the memory location. Like any variables or constant, you must declare a pointer before you can use it for storing any variable address.

The general form of pointer is:

**varvar\_name \*var\_type**

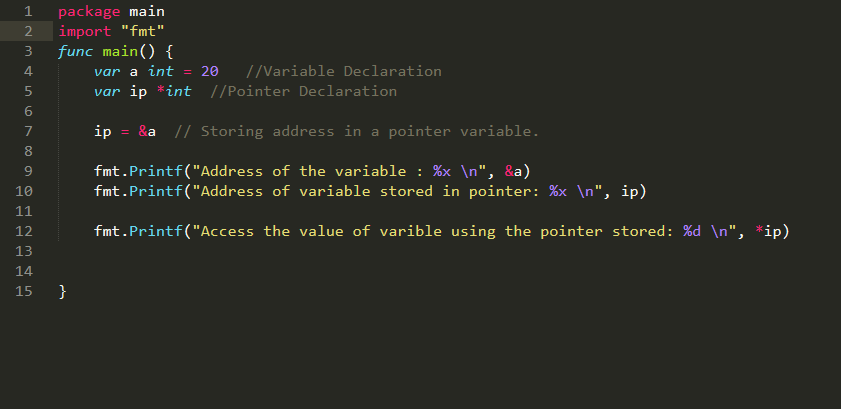
where **type** is the pointer’s base type; it must be a valid C data type and **var\_name**indicates the name of the pointer. An \* is used to designate that it is a pointer.

***How to Use Pointers:***

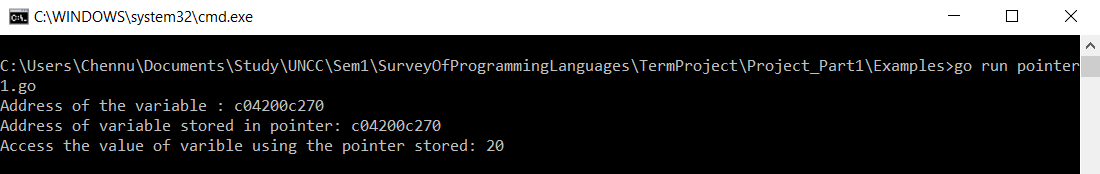
There are few operations, which we frequently perform with pointers:

1. We define pointer variables.
2. Assign the address of a variable to a pointer
3. Access the value at the address stored in the pointer variable.

To understand the concepts of the pointers let us illustrate with an example:



***Output:***



***Nil Pointers in Go:***

Go compiler assigns a nil value to a pointer variable in case you do no have exact address to be assigned. This is done at the time of variable declaration. A pointer that is assigned nil is called as **nil pointer.**

The nil pointer is a constant with a value zero defined in standard libraries.

By default, if you don’t initialize the pointer, its value is set to nil.

For example: varptr \*int

Here when we print ptr the output would be 0.

***Pointers in Detail:***

Pointers have many but easy concepts and they are very important to Go programming. The following concepts of pointers should be clear to a Go programmer:

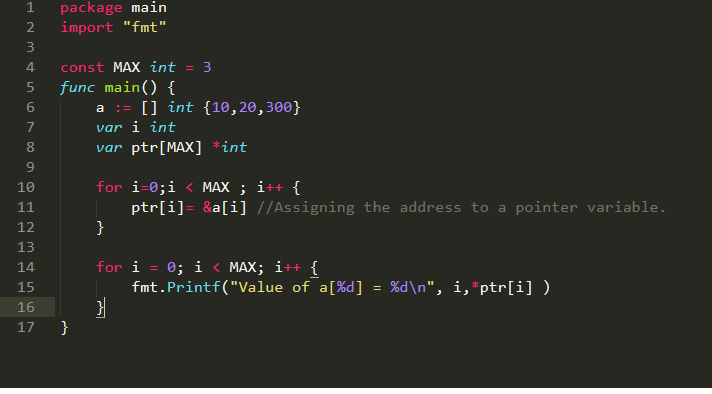
|  |  |
| --- | --- |
| **Concept** | **Description** |
| Go – Array of pointers | You can define arrays to hold several pointers. |
| Go – Pointer to pointer | Go allows you to have pointer on a pointer and so on. |
| Passing pointers to functions in Go | Passing an argument by reference or by address both enable the passed argument to be changed in the calling function by the called function. |

***Array of Pointers:***

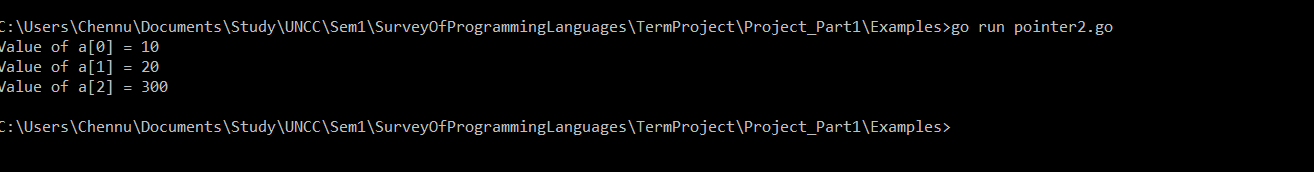
There may be a situation when we want to maintain an array, which can store pointers to an int, string or any other datatype available. The following statements declares an array of pointers to an integer.

**varptr [MAX] \*int;**

This declares **ptr**as an array of MAX integer pointers. Thus, each element in ptr, now holds a pointer to an int value. The following example makes use of three integers, which will be stored in an array of pointers as follows:



***Output:***



***Comparison with C Pointers:***

The concepts involved in C programming language are same with the pointers. But the only difference is we can access the pointer of an array just by declaring its variable name. But in Go, it is mandatory to mention & operator before an array element to access the address of each element of an array.

***Go- Pointer to a Pointer:***

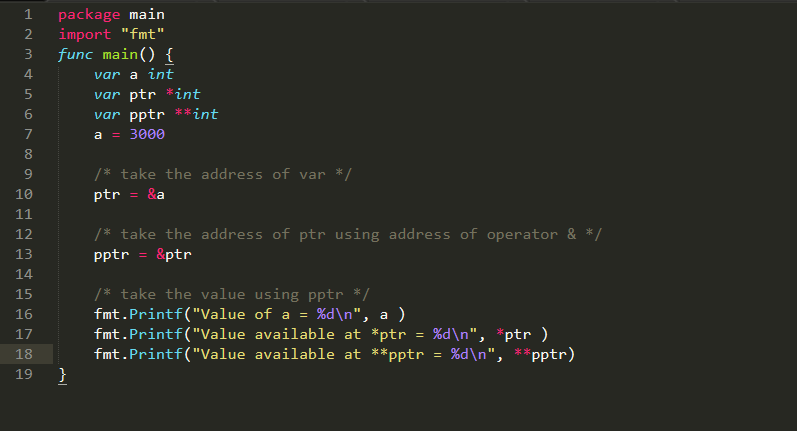
A pointer to a pointer is a form of chain of pointers. Normally, a pointer contains the address of a variable. When we define a pointer to a pointer, the first pointer contains the address of the second pointer, which points to the location that contains the actual value as shown below:



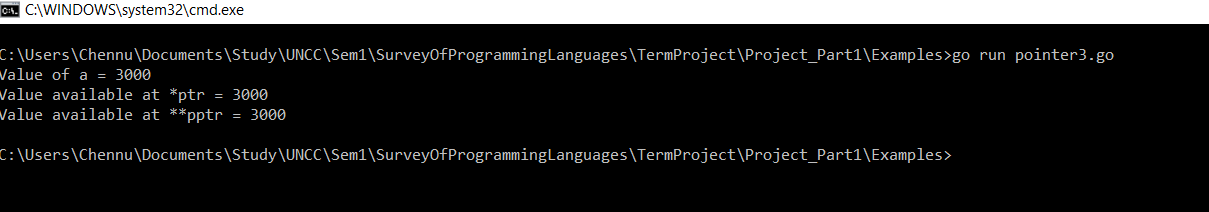
A variable that is a pointer to a pointer must be declared as such. This is done by placing an additional asterisk in front of its name. For example, the following statement declares a pointer to a pointer of type int:

**varptr \*\*int;**

When a target value is indirectly pointed to by a pointer to a pointer, accessing that value requires that the asterisk operator be applied twice, as is shown in the following example:



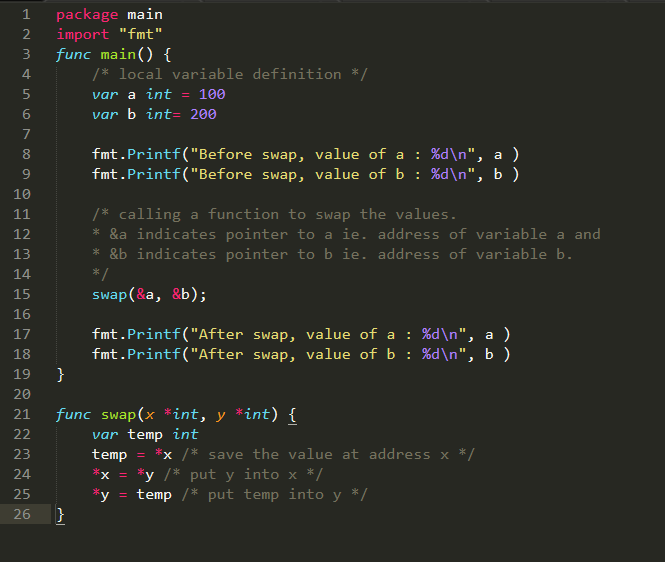
***Output:***



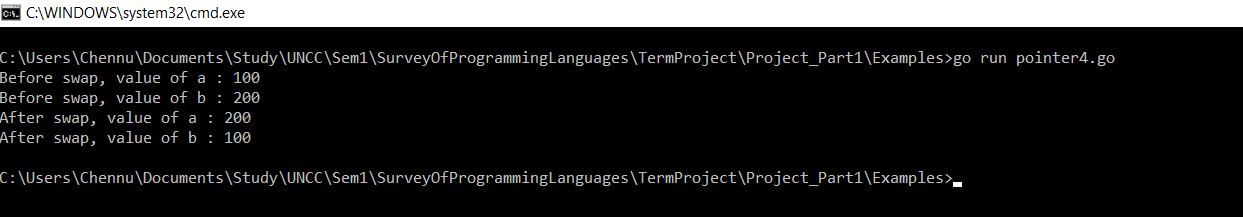
***Go – Passing Pointers to a Function:***

Go programming language allows you to pass a pointer to a function. To do so, simply declare the function parameter as a pointer type.

In the following example, we pass two pointers to a function and change the value inside the function which reflects back in the calling function. This can be illustrated with the following example:



***Output:***



**8. STRUCTURES**

Structure is a user defined data type available in Go programming, which allows to combine data items of different kind.

Structures are used to represent a record. For example, consider a record of an Employee. It contains following attributes of different data types.

* employeeId
* employeeName
* designation
* projectName

In this kind of scenario, structures are highly useful.

***Defining aStructure:***

To define a structure, you must use **type** and **struct** statements. The struct statement defines a new data type, with multiple members for your program. The type statement binds a name with the type which is struct in our case. The format of the struct statement is as follows:

type struct\_variable\_type struct {

member definition;

member definition;

...

member definition;

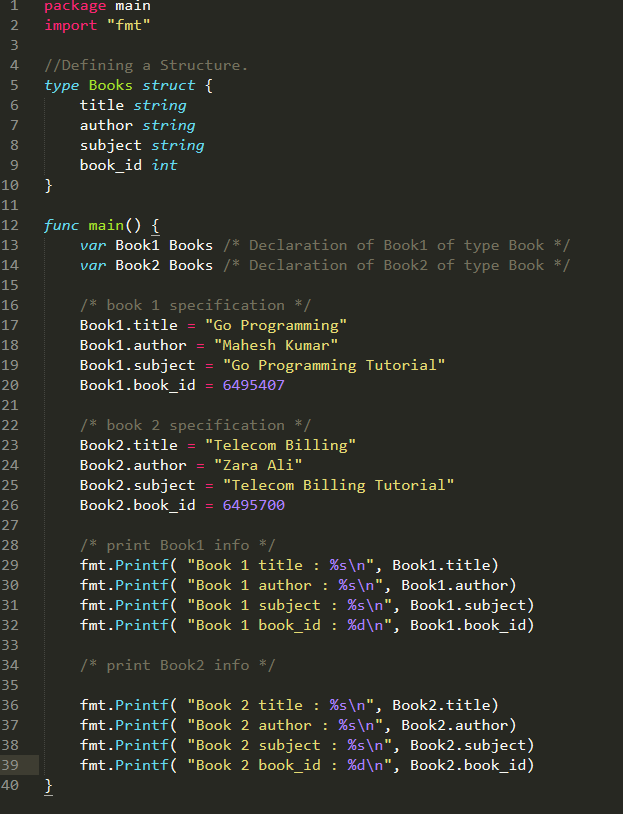
}

Once a structure type is defined, it can be used to declare variables of that type using the following syntax:

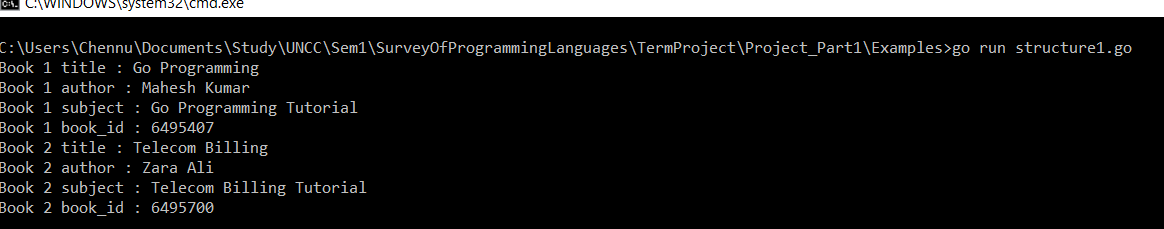
**Variable\_name := structure\_variable\_type {value1, value2…valuen}**

***Accessing Structure Variables:***

To access any member of a structure, we use the **member access operator (.)**. The member access operator is coded as a period between the structure variable name and the structure member that we wish to access. You would use **struct** keyword to define variables of structure type. The following example explains how to use a structure:



***Output:***

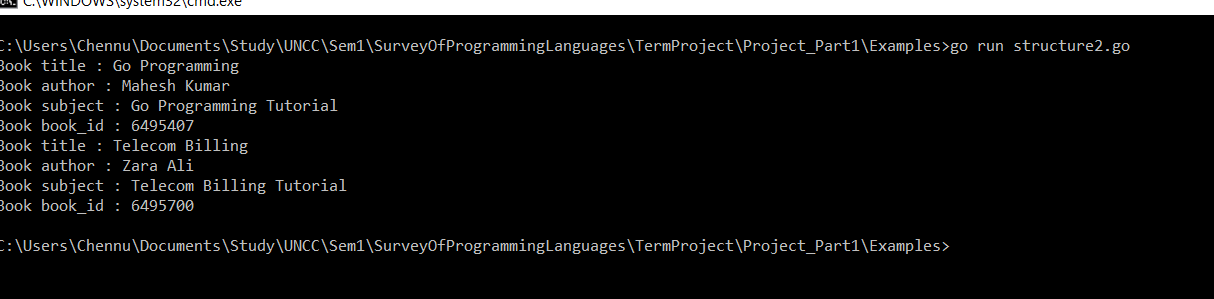


***Structures as Function Arguments:***

A structure can be passed as a function argument like the way we pass any other variable or any pointer. Consider the following example to illustrate the same:



***Output:***



***Pointers to Structures:***

A pointer to a structure can be defined as the same way we define any pointers:

**varstruct\_pointer \*Books**

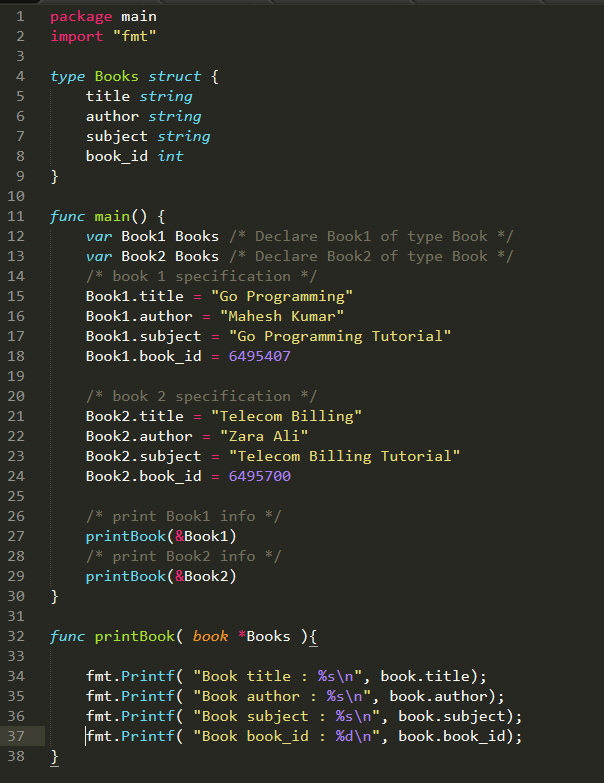
Now, you can store the address of a structure variable in the above defined pointer variable. To find the address of a structure variable, place the “*&”* operator before the structure name as follows:

**struct\_pointer = &Book1;**

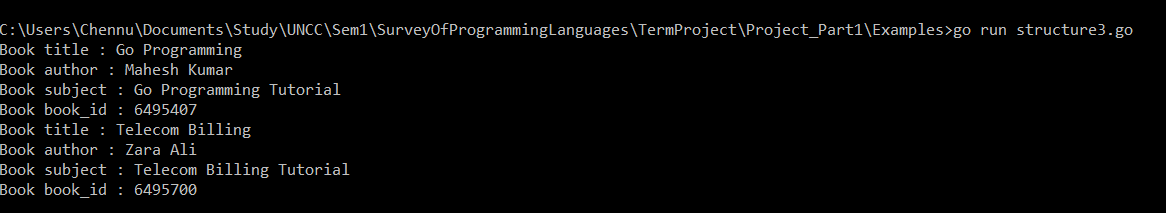
To access the members of a structure using a pointer to that structure, you must use the "." operator as follows:

**struct\_pointer.title;**

Consider the example of a Book Record:



***Output:***

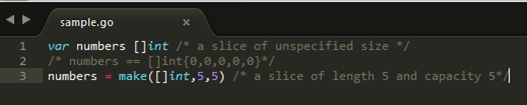


**9. SLICES**

Go Slice is an abstraction over Go Array. As Go Array allows us to define type of variables that can hold several data items of the same kind but it do not provide any inbuilt method to increase size of it dynamically or get a sub-array of its own. Slices covers this limitation. It provides many utility functions required on Array and is widely used in Go programming.

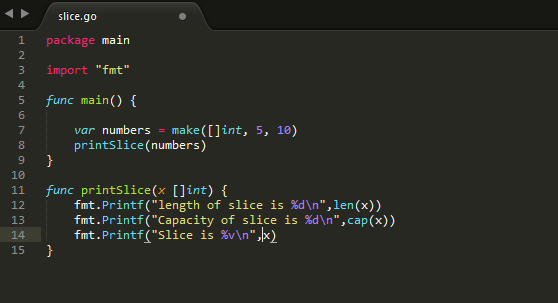
***Defining a Slice:***

To define a slice, we can declare it as an array without specifying size or use **make** function to create the one.

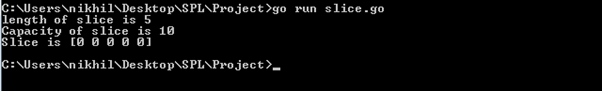


***len() and cap() functions***

**len()** function returns the elements presents in the slice whereas **cap()** function returns the capacity of slice as how many elements it can be accommodate. Following is an example to explain usage of slice.

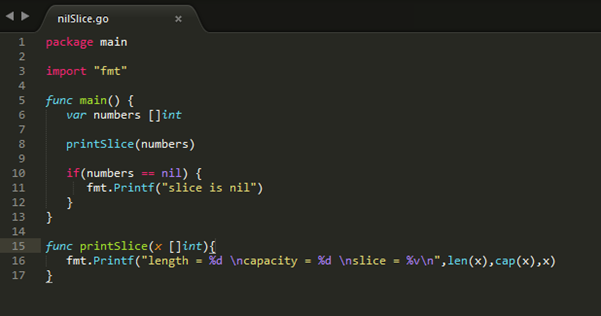


***Output:***

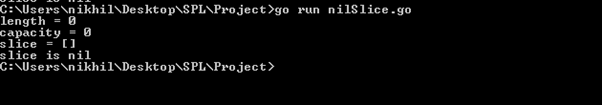
******

***Nil Slice***

If a slice is declared with no inputs the by default, it is initialized as nil. Its length and capacity are zero. Following is an example:



***Output:***

******

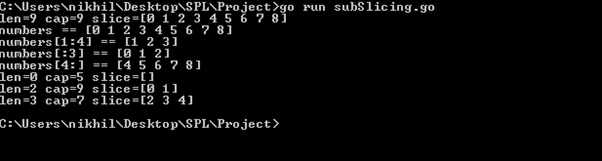
***Sub-slicing***

Slice allows lower-bound and upper bound to be specified to get the subslice of it using

**[lower-bound: upper-bound]**. Following is an example:



***Output:***

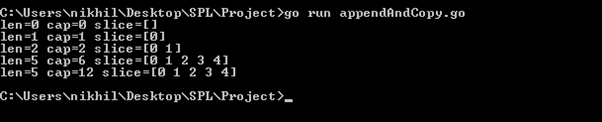
******

***append() and copy() functions***

Slice allows increasing the capacity of a slice using **append()** function. Using **copy()** function, contents of a source slice are copied to destination slice. Following is the example:



***Output:***

******

**10. RANGE**

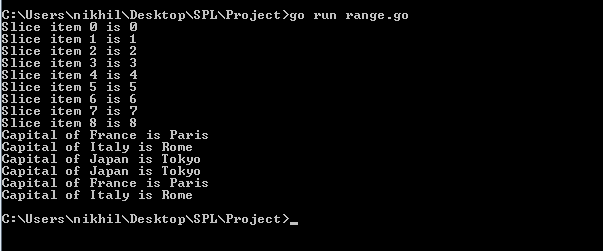
The **range** keyword is used in **for** loop to iterate over items of an array, slice, channel or map. With array and slices, it returns the index of the item as integer. With maps, it returns the key of the next key-value pair. Range either returns one value or two. If only one value is used on the left of a range expression, it is the 1st value in the following table.

|  |  |  |
| --- | --- | --- |
| **Range expression** | **1st Value** | **2nd Value(Optional)** |
| Array or slice a [n]E | index iint | a[i] E |
| String s string type | index iint | rune int |
| map m map[K]V | key k K | value m[k] V |
| channel c chan E | element e E | none |

***Example:***

******

***Output:***

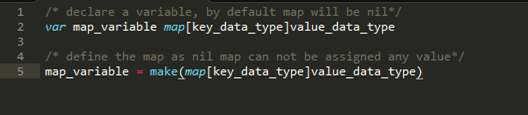
******

**11. MAPS**

Go provides another important data type map which maps unique keys to values. A key is an object that we use to retrieve a value later. Given a key and a value, we can store the value in a Map object. After value is stored, we can retrieve it by using its key.

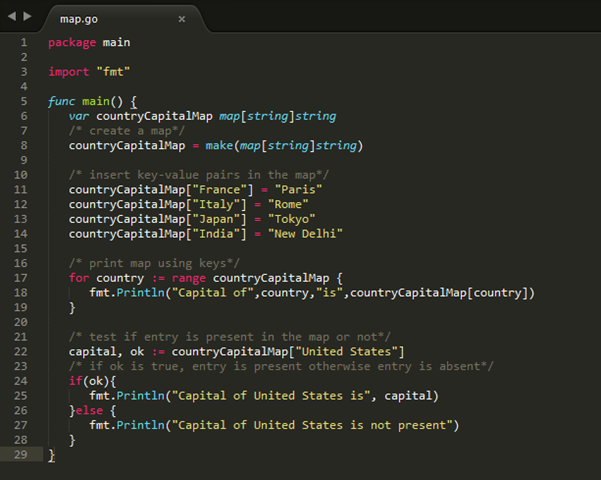
***Defining a map***

We must use **make** function to create a map.

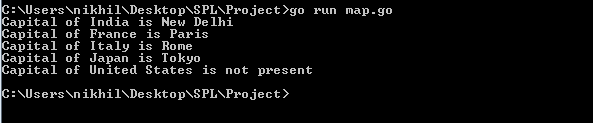
******

***Example:***

Following example illustrates creation and usage of map.

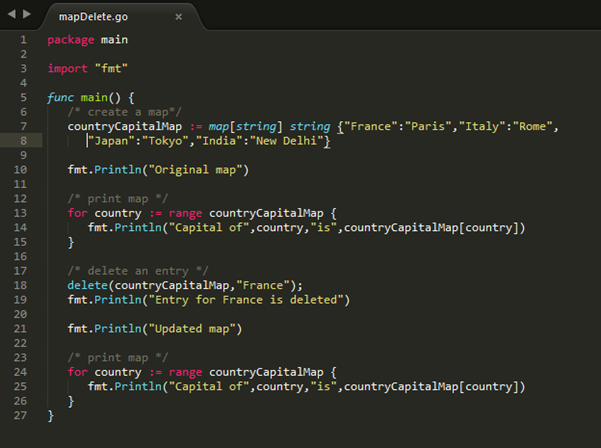


***Output:***

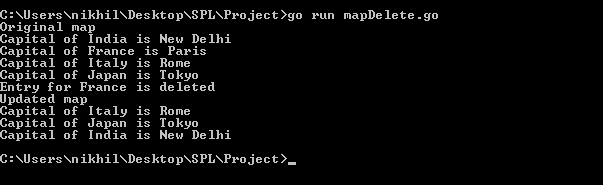
******

***delete() function***

delete() function is used to delete an entry from the map. It requires map and corresponding key which is to be deleted. Following is the example:

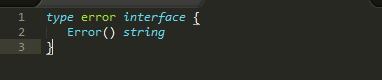


***Output:***

******

**12. ERROR HANDLING**

Go programming provides a pretty simple error handling framework with inbuilt error interface type of following declaration:



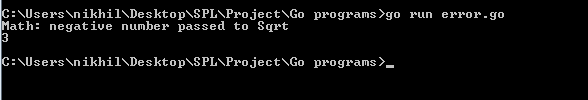
Functions normally return error as last return value. We use **errors.New** to construct a basic error message.

***Example***

The following example prints an error message when the input is a negative number.



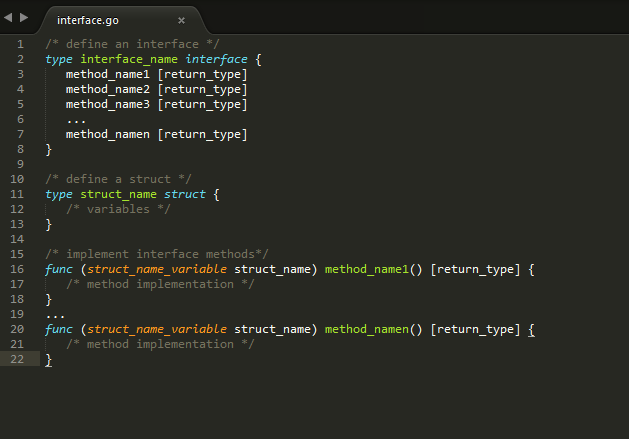
***Output***



**13. INTERFACES**

Go programming provides another data type called interfaces which represents a set of method signatures. struct data type implements these interfaces to have method definitions for the method signature of the interfaces.

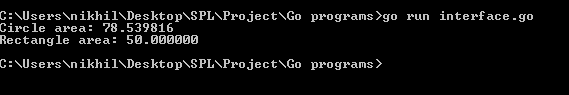
***Syntax***



***Example***



***Output***



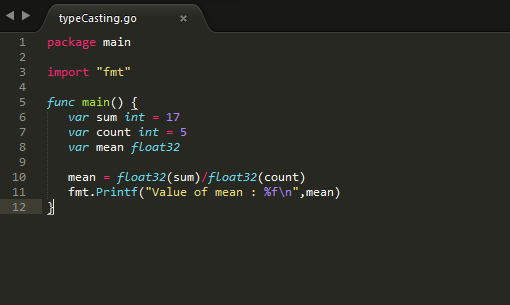
**14. TYPE CASTING**

Type casting is a way to convert a variable from one data type to another data type. For example, if you want to store a long value into a simple integer then you can type cast long to int. You can convert values from one type to another using the **cast operator** as following:

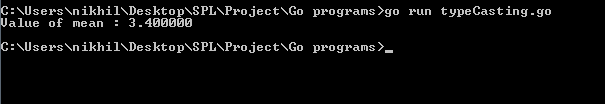
**type\_name(expression)**

***Example***

Consider the following example where the cast operator causes the division of one integer variable by another to be performed as a floating number operation.



***Output***



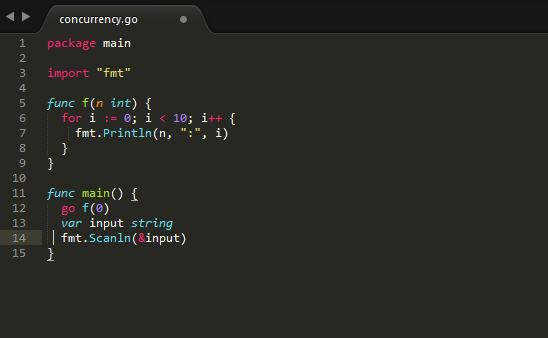
**15. CONCURRENCY**

Large programs are often made up of many smaller sub-programs. For example, a web server handles requests made from web browsers and serves up HTML web pages in response. Each request is handled like a small program.

It would be ideal for programs like these to be able to run their smaller components at the same time (in the case of the web server to handle multiple requests). Making progress on more than one task simultaneously is known as concurrency. Go has rich support for concurrency using goroutines and channels.

***Goroutines***

A goroutine is a function that is capable of running concurrently with other functions. To create a goroutine we use the keyword go followed by a function invocation:



This program consists of two goroutines. The first goroutine is implicit and is the main function itself. The second goroutine is created when we call go f(0). Normally when we invoke a function our program will execute all the statements in a function and then return to the next line following the invocation. With a goroutine we return immediately to the next line and don't wait for the function to complete. Therefore, the call to the Scanln function has been included; without it the program would exit before being given the opportunity to print all the numbers.

***Channels***

Channels provide a way for two goroutines to communicate with one another and synchronize their execution. Here is an example program using channels:



A channel type is represented with the keyword **chan** followed by the type of the things that are passed on the channel (in this case we are passing strings). The <- (left arrow) operator is used to send and receive messages on the channel. **c <- "ping"** means send "ping".

**msg := <- c** means receive a message and store it in msg. The **fmt** line could also have been written like this: **fmt.Println(<-c)** in which case we could remove the previous line.

Using a channel like this synchronizes the two goroutines. When pinger attempts to send a message on the channel it will wait until printer is ready to receive the message. The program will take turns printing “ping” and “pong”.

***Output***



***Channel Direction***

We can specify a direction on a channel type thus restricting it to either sending or receiving. For example, pinger's function signature can be changed to this:

**funcpinger(c chan<- string)**

Now c can only be sent to. Attempting to receive from c will result in a compiler error. Similarly, we can change printer to this:

**funcprinter(c <-chan string)**

A channel that doesn't have these restrictions is known as bi-directional. A bi-directional channel can be passed to a function that takes send-only or receive-only channels, but the reverse is not true.

**16. EVOLUTION OF LANGUAGE’S WRITABILITY, READABILITY AND RELIABILITY**

***Readability***

***Simplicity***

* Golanguage has less number of keywords i.e 25 compared to 32 keyword in C language.
* Go language has less feature multiplicity, so that every programming style is consistent
* Developer who can read C language will not have hard time getting used to Go language syntax.

***Orthogonality***

Go language is highly orthogonal as it takes many of the design cues from UNIX’s philosophy. UNIX philosophy revolves around following quotes

* + - 1. Do one thing well
      2. Keep things simple
      3. Maintain standard way to communicate

As Go language adheres to such principle we have high orthogonality with Go language.

Go is made up small tools, and we can make use of such tools and get started.

***Data type***

Go language has separate 32 bit and 64 bit data types such as int32 float64. As in past there were many issues, security loopholes generated due to use of incorrect data types.

Since the creators of Go language were part of UNIX and C language team. They created Go language such that they would not repeat the mistakes.

***Syntax Design***

Go language has nice syntax, although it is not as playful and enjoyable as python is but it better than C, Java.

***Writability***

* As Go language has small number of keywords and small modules in Go language interoperate nicely with each other, this makes Go language possess properties such as simplicity and orthogonality.
* Go language has redefined the meaning of object oriented, such as there is not inheritance but we have composition instead, have composition provides high abstraction. This help us in adhering to DRY principle.
* Go language has high expressivity, programs written in Go language are small and they are easy to read. We less lines of code compared to other language like java and C, Go language can achieve much more.

***Reliability***

* + Go language is static compiled language we have type checking at compilation phase. As mentioned earlier the creators of Go language made such they don’t repeat that were made during designing of C and UNIX systems.
  + Go language has pointer involved, thus on the other hand there is scope to have some side effect if pointers not used properly.
  + Go language has this property that the code that person A writes and person B write, given the same logic they both would end up writing code in similar structure. This structure is defined by creators of Go language. This makes go code more readable and thus a cost of less freedom we have much easy to read code. This helps in uncovering bugs, having less obfuscated code. Thus making project more reliable.

***Cost***

* + Training Programmers: Go language is new to market, there is some cost involved in training programmers. Since Go language has less keywords, strictness in coding style, and having its creators from UNIX background. This makes Go language easier to imbibe.
  + Maintenance of Software: Code written in Go language is much readable compared to C and java. Furthermore, in Go language we tend to create small tools and use them with each other. Thus, maintaining software tends to be much simpler than java and C.
  + Compilation:Go language is statically compiled language. The compiler provided with go is fast, as mentioned by Rob Pike in one of his talks “The time required to compile go code, takes less time than some interpreted language would take to execute one line of code.”
  + Executing Program: Go language being statically compiled language is par on performance when compared to python. Instead of threads Go language has goroutines which Rob Pike mentioned were lighter than threads, thus in SMP processors we can get most out of Go language.
  + Language Implementation System: Go language is open source, it is licensed under three clauses BSD license that are mentioned below.
    1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
    2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
    3. Neither the name of Google Inc. nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

***Other Criteria***

* + Portability: Go language is an open source project, thus this language comes for free. Go has target Linux, OX, Windows and various UNIX and BSD versions. Go has lesser number of keywords associated with it, thus there is less work on side of programmer and on compiler. Go language has a compiler which creates binary which is created using systems resources, thus not very portable. Also some package can be system dependent.
  + Generality: As mentioned by Professor Dewan, modern programming languages instead of trying to create language that would serve wide spectrum of domain, modern language tend to be specific and try solving the problem .
  + Well Definedness: Go language has nice FAQ section [https://Go language.org/doc/faq](https://golang.org/doc/faq) . Go language is well defined, earlier they used to release version frequently, but then Industry had issue that they had to upgrade Go language, and were incurring high maintenance cost. Thus now we have stable Go language from 1.0 and onwards.

**17. MAJOR STRENGTHS AND WEAKNESS OF GO LANGUAGE**

Every language has its own merits and demerits. Soit’s difficult to compare a programming language on few specific parameters. Following are few of the strengths and weaknesses of Go Programming languages:

**STRENGTHS:**

1. Go compiles very quickly.
2. Go supports concurrency at the language level.
3. Functions are first class objects in Go.
4. Go has garbage collection.
5. Strings and maps are built into the language.
6. Error Handling.
7. It comes up with an inbuilt web server which means you don't need Apache or any other web server. This is a direct advantage of Go.
8. Setting up and building an app in Go is very easy.

**WEAKNESS:**

1. The packages distributed with Go are useful, but there are still some libraries that are still not present, notably UI toolkit.
2. There is no support for generics in Go, although there are many discussions around it.

**18. OVERVIEW OF THE PROGRAMS INCLUDED**

* Programs illustrating the functionality of pointers and structures.
* Programs which help in understanding the basic concepts of loops, functions, recursion
* Programs explaining the functionality of slices, range and maps.

These programs mostly demonstrate the features of Data and control abstractions in Go language. For example: Arrays and Structures are a part of the Structural Data Abstractions. Concepts of loops, Functions, Conditional Statements are Structural Control Abstractions.

**19. UNIQUE FEATURES IN GO LANGUAGE**

1) Go Array allows us to increase the size of the array dynamically or get a sub array of its own. This feature which is not present in many traditional languages like Java, C++ etc., is used widely in many Go applications.

2) Go allows us to increase the capacity of the slice dynamically by using a built in append() function. The built in copy() function allows us to copy the contents of an array to another array without using much code. (An example of data abstraction).