

**SURVEY OF PROGRAMMING LANGUAGES**

**PROJECT REPORT FALL 2016**

**GO LANGUAGE**

**Title: Contacts Directory**

**TEAM MEMBERS**

**CHAITANYA SRI KRISHNA LOLLA (800960353)**

**ABHISHEK CHANDRATRE (800961657)**

**TEJASWI KONDURI (800965883)**

**NAGAMANIKANTA SRIVENKATA JONNALAGADDA (800975714)**

## **Problem Definition:**

GoContacts is personal contact directory. It is aimed to manage csv file using web interface.

GoContacts provide following features.

1. User can load any csv contact file compatible with GoContacts.
2. User can add contacts.
3. User can delete contacts.
4. User can also restore deleted contacts.
5. User must be able to perform all the above actions using web interface.

## **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.

## **HOW GO HANDLES ABSTRACTION:**

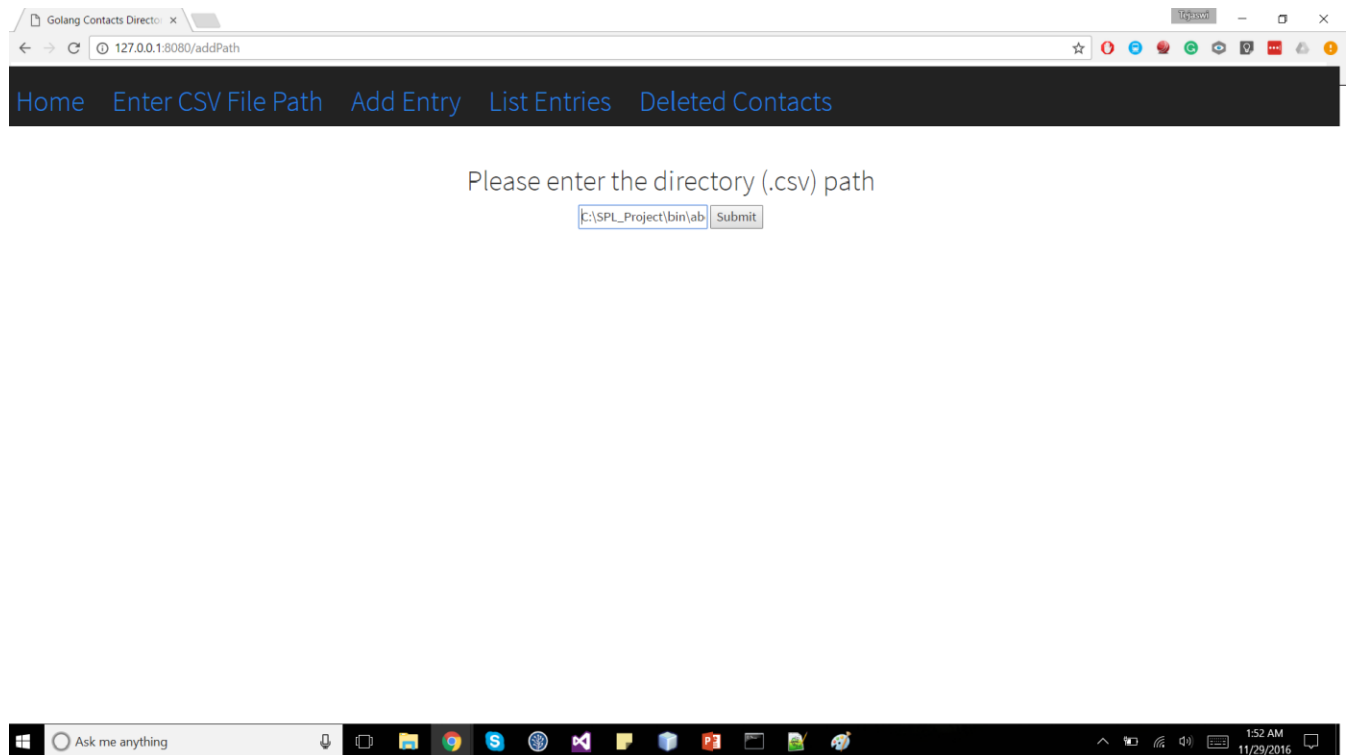
Go handles various layers of Abstraction using various features:

- It handles Basic Data Abstractions using the data types that are provided. Also, there is no need to explicitly declare the type.
- Golang handles Structural Data Abstractions using Arrays, Slices, Structures.
- It supports basic control abstractions for example: `sum := a + b`; Here there is no need to declare the sum separately. Golang internally assigns a valid datatype by itself.
- It supports Structured Control Abstractions using Structures, Pointers.
- It handles Unit Abstractions using Go Routines, Channels, Timers etc.

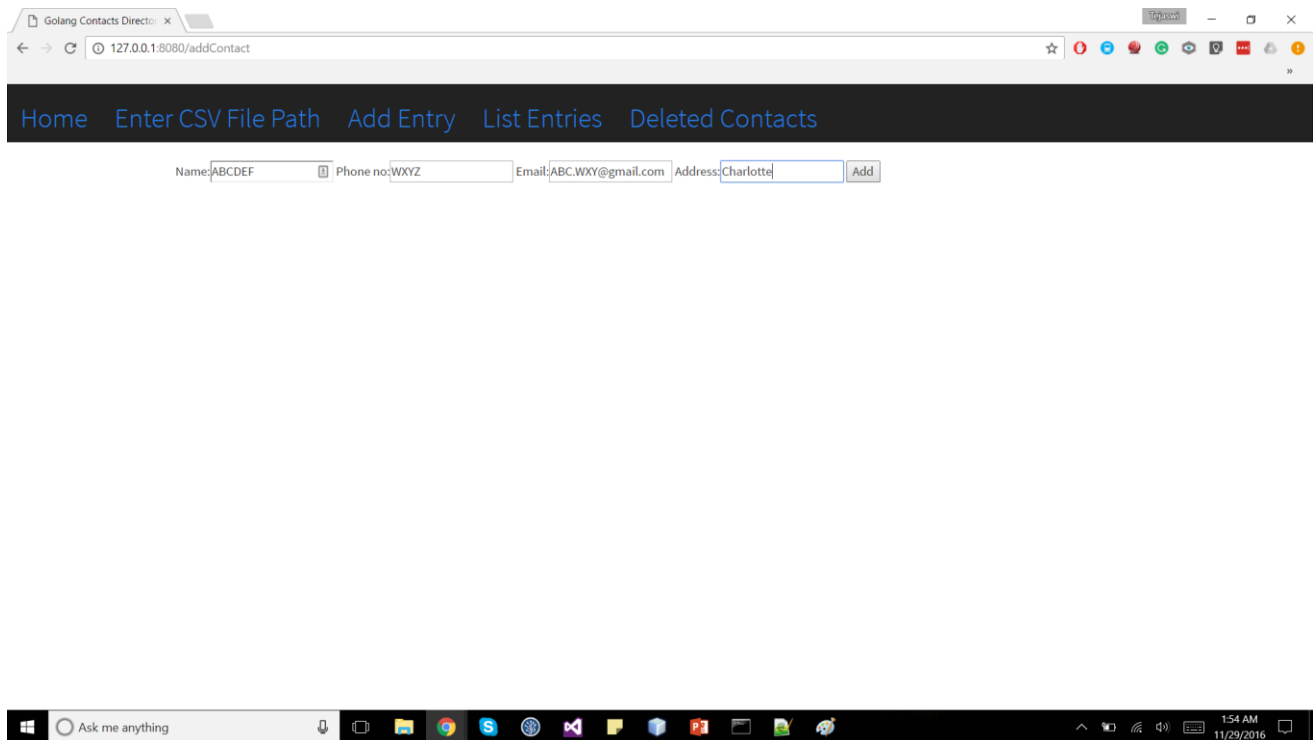
## **Steps to use the application (Contacts Directory):**

1) Enter a file path in the 'Enter CSV File Path' tab,

or Enter just the file name, the application will create a new file.

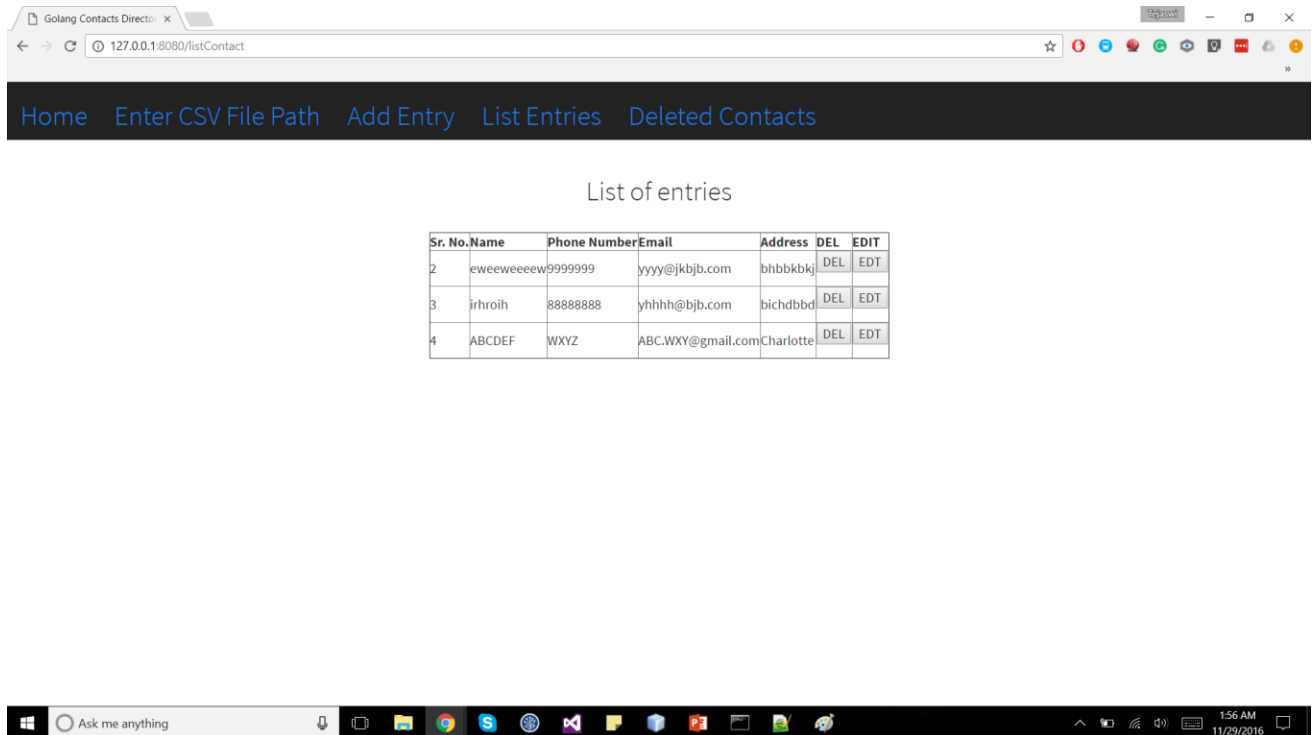


2) 'Add Entry' tab will add an entry in the directory.

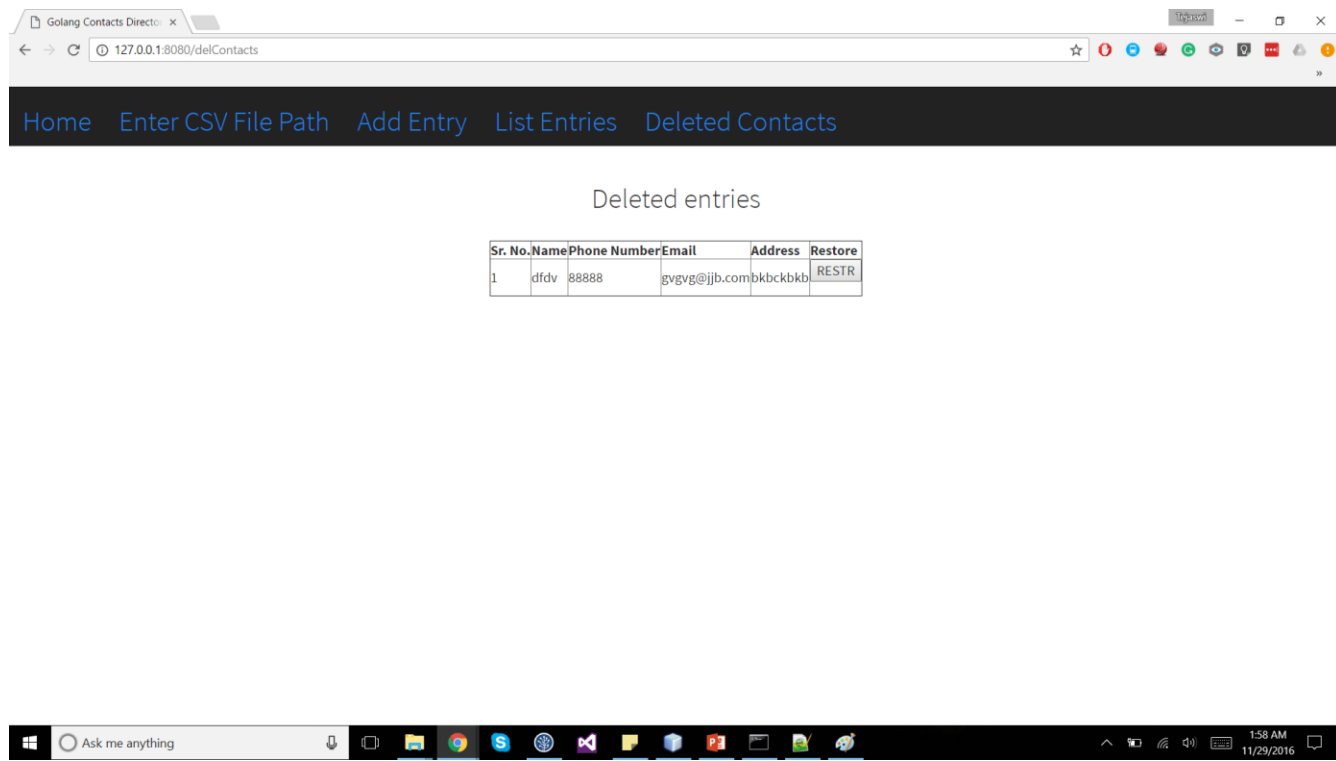


3) 'List Entries' tab will list all the entries in the directory.

Along with listings of contacts, there is also an option to Delete and Edit an entry.



4) 'Deleted Contacts' tab will show all the deleted contacts and have a Restore button, so that we can restore the deleted contacts.



## 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 in Go, so you'll need to use a full if statement even for basic conditions.
- Here's a basic example:

### Code Snippet:

```
1  package main
2
3  import "fmt"
4
5  func main() {
6
7      // Here's a basic example.
8      if 7%2 == 0 {
9          fmt.Println("7 is even")
10     } else {
11         fmt.Println("7 is odd")
12     }
13
14     // You can have an `if` statement without an else.
15     if 8%4 == 0 {
16         fmt.Println("8 is divisible by 4")
17     }
18
19     // A statement can precede conditionals; any variables
20     // declared in this statement are available in all
21     // branches.
22     if num := 9; num < 0 {
23         fmt.Println(num, "is negative")
24     } else if num < 10 {
25         fmt.Println(num, "has 1 digit")
26     } else {
27         fmt.Println(num, "has multiple digits")
28     }
29 }
```

### Output:



```
C:\Users\Tejaswi\Desktop\Coursework Fall 16\SPL\Project\GoCode\SPL_Project\src\main>go run hello-world.go
7 is odd
8 is divisible by 4
9 has 1 digit

C:\Users\Tejaswi\Desktop\Coursework Fall 16\SPL\Project\GoCode\SPL_Project\src\main>
```

## **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:

```
1  package main
2
3  import "fmt"
4  import "time"
5
6  func main() {
7
8      // Here's a basic `switch`.
9      i := 2
10     fmt.Print("write ", i, " as ")
11     switch i {
12     case 1:
13         fmt.Println("one")
14     case 2:
15         fmt.Println("two")
16     case 3:
17         fmt.Println("three")
18     }
19
20     // You can use commas to separate multiple expressions
21     // in the same `case` statement. We use the optional
22     // `default` case in this example as well.
23     switch time.Now().Weekday() {
24     case time.Saturday, time.Sunday:
25         fmt.Println("it's the weekend")
26     default:
27         fmt.Println("it's a weekday")
28     }
29
30     // `switch` without an expression is an alternate way
31     // to express if/else logic. Here we also show how the
32     // `case` expressions can be non-constants.
33     t := time.Now()
34     switch {
35     case t.Hour() < 12:
36         fmt.Println("it's before noon")
37     default:
38         fmt.Println("it's after noon")
39     }
40 }
```

### Output:

```
C:\Users\Tejaswi\Desktop\Coursework Fall 16\SPL\Project\GoCode\SPL_Project\src\main>go run hello-world.go
write 2 as two
it's a weekday
it's before noon

C:\Users\Tejaswi\Desktop\Coursework Fall 16\SPL\Project\GoCode\SPL_Project\src\main>
```

### 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:

```
1  package main
2
3  import "fmt"
4
5  func main() {
6      fmt.Println("Sample Code")
7      i := 1
8      for i <= 3 {
9          fmt.Println(i)
10         i = i + 1
11     }
12 }
```

### Output:

```
C:\Users\Tejaswi\Desktop\Coursework Fall 16\SPL\Project\GoCode\SPL_Project\src\main>go run hello-world.go
Sample Code
1
2
3

C:\Users\Tejaswi\Desktop\Coursework Fall 16\SPL\Project\GoCode\SPL_Project\src\main>
```

2) A classic initial/condition/after for loop.

**Code Snippet:**

```
1  package main
2
3  import "fmt"
4
5  func main() {
6      fmt.Println("Sample Code")
7      for j := 7; j <= 9; j++ {
8          fmt.Println(j)
9      }
10 }
```

**Output:**

```
C:\Users\Tejaswi\Desktop\Coursework Fall 16\SPL\Project\GoCode\SPL_Project\src\main>go run hello-world.go
Sample Code
7
8
9
C:\Users\Tejaswi\Desktop\Coursework Fall 16\SPL\Project\GoCode\SPL_Project\src\main>
```

- 3) for without a condition will loop repeatedly until you break out of the loop or return from the enclosing function.

**Code Snippet:**

```
1  package main
2
3  import "fmt"
4
5  func main() {
6      fmt.Println("Sample Code")
7      i := 0
8      for {
9          if(i < 4) {
10             fmt.Println("loop")
11             i++
12         } else {
13             break
14         }
15     }
16 }
```

**Output:**

```
C:\Users\Tejaswi\Desktop\Coursework Fall 16\SPL\Project\GoCode\SPL_Project\src\main>go run hello-world.go
Sample Code
loop
loop
loop
loop
C:\Users\Tejaswi\Desktop\Coursework Fall 16\SPL\Project\GoCode\SPL_Project\src\main>
```

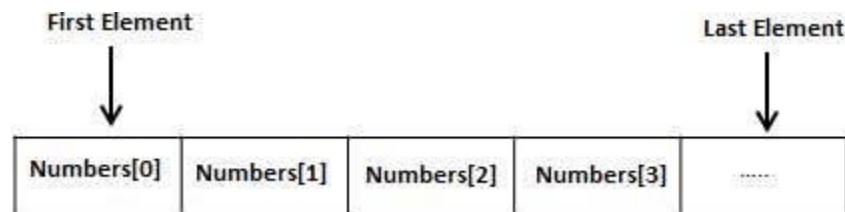
## The major data structures of Go:

### 1) 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:

```
var variable_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 a variable array which is sufficient to hold up to 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:

	0	1	2	3	4
balance	1000.0	2.0	3.4	7.0	50.0

### 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]
```

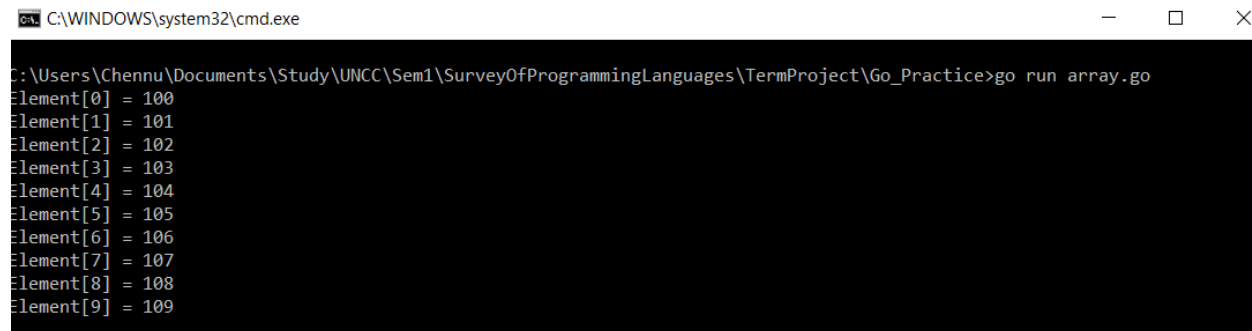
The above statement will take 10th element from the array and assign the value to salary variable. Following is an example which will use all the above mentioned three concepts viz. declaration, assignment and accessing arrays:

```

1  package main
2
3  import "fmt"
4
5  func main() {
6      var n [10]int /* n is an array of 10 integers */
7      var i,j int
8
9      /* initialize elements of array n to 0 */
10     for i = 0; i < 10; i++ {
11         n[i] = i + 100 /* set element at location i to i + 100 */
12     }
13
14     /* output each array element's value */
15     for j = 0; j < 10; j++ {
16         fmt.Printf("Element[%d] = %d\n", j, n[j] )
17     }
18 }

```

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



```

C:\WINDOWS\system32\cmd.exe
C:\Users\Chennu\Documents\Study\UNCC\Sem1\SurveyOfProgrammingLanguages\TermProject\Go_Practice>go run array.go
Element[0] = 100
Element[1] = 101
Element[2] = 102
Element[3] = 103
Element[4] = 104
Element[5] = 105
Element[6] = 106
Element[7] = 107
Element[8] = 108
Element[9] = 109

```



## 2) 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 a Structure:

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:

```

1  package main
2  import "fmt"
3
4  //Defining a Structure.
5  type Books struct {
6      title string
7      author string
8      subject string
9      book_id int
10 }
11
12 func main() {
13     var Book1 Books /* Declaration of Book1 of type Book */
14     var Book2 Books /* Declaration of Book2 of type Book */
15
16     /* book 1 specification */
17     Book1.title = "Go Programming"
18     Book1.author = "Mahesh Kumar"
19     Book1.subject = "Go Programming Tutorial"
20     Book1.book_id = 6495407
21
22     /* book 2 specification */
23     Book2.title = "Telecom Billing"
24     Book2.author = "Zara Ali"
25     Book2.subject = "Telecom Billing Tutorial"
26     Book2.book_id = 6495700
27
28     /* print Book1 info */
29     fmt.Printf( "Book 1 title : %s\n", Book1.title)
30     fmt.Printf( "Book 1 author : %s\n", Book1.author)
31     fmt.Printf( "Book 1 subject : %s\n", Book1.subject)
32     fmt.Printf( "Book 1 book_id : %d\n", Book1.book_id)
33
34     /* print Book2 info */
35
36     fmt.Printf( "Book 2 title : %s\n", Book2.title)
37     fmt.Printf( "Book 2 author : %s\n", Book2.author)
38     fmt.Printf( "Book 2 subject : %s\n", Book2.subject)
39     fmt.Printf( "Book 2 book_id : %d\n", Book2.book_id)
40 }

```

### **Output:**

```

C:\WINDOWS\system32\cmd.exe

C:\Users\Chennu\Documents\Study\UNCC\Sem1\SurveyOfProgrammingLanguages\TermProject\Project_Part1\Examples>go run structure1.go
Book 1 title : Go Programming
Book 1 author : Mahesh Kumar
Book 1 subject : Go Programming Tutorial
Book 1 book_id : 6495407
Book 2 title : Telecom Billing
Book 2 author : Zara Ali
Book 2 subject : Telecom Billing Tutorial
Book 2 book_id : 6495700

```

### 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:

```
1  package main
2  import "fmt"
3
4  type Books struct {
5      title string
6      author string
7      subject string
8      book_id int
9  }
10
11 func main() {
12     var Book1 Books /* Declare Book1 of type Book */
13     var Book2 Books /* Declare Book2 of type Book */
14
15     /* book 1 specification */
16     Book1.title = "Go Programming"
17     Book1.author = "Mahesh Kumar"
18     Book1.subject = "Go Programming Tutorial"
19     Book1.book_id = 6495407
20
21     /* book 2 specification */
22     Book2.title = "Telecom Billing"
23     Book2.author = "Zara Ali"
24     Book2.subject = "Telecom Billing Tutorial"
25     Book2.book_id = 6495700
26
27     /* print Book1 info */
28     printBook(Book1)
29
30     /* print Book2 info */
31     printBook(Book2)
32 }
33
34 func printBook( book Books ){
35
36     fmt.Printf( "Book title : %s\n", book.title);
37     fmt.Printf( "Book author : %s\n", book.author);
38     fmt.Printf( "Book subject : %s\n", book.subject);
39     fmt.Printf( "Book book_id : %d\n", book.book_id);
40 }
```

### Output:

```
C:\Windows\system32\cmd.exe
C:\Users\Chennu\Documents\Study\UNCC\Sem1\SurveyOfProgrammingLanguages\TermProject\Project_Part1\Examples>go run structure2.go
Book title : Go Programming
Book author : Mahesh Kumar
Book subject : Go Programming Tutorial
Book book_id : 6495407
Book title : Telecom Billing
Book author : Zara Ali
Book subject : Telecom Billing Tutorial
Book book_id : 6495700
C:\Users\Chennu\Documents\Study\UNCC\Sem1\SurveyOfProgrammingLanguages\TermProject\Project_Part1\Examples>
```

### Pointers to Structures:

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

**var struct\_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:

```
1  package main
2  import "fmt"
3
4  type Books struct {
5      title string
6      author string
7      subject string
8      book_id int
9  }
10
11 func main() {
12     var Book1 Books /* Declare Book1 of type Book */
13     var Book2 Books /* Declare Book2 of type Book */
14     /* book 1 specification */
15     Book1.title = "Go Programming"
16     Book1.author = "Mahesh Kumar"
17     Book1.subject = "Go Programming Tutorial"
18     Book1.book_id = 6495407
19
20     /* book 2 specification */
21     Book2.title = "Telecom Billing"
22     Book2.author = "Zara Ali"
23     Book2.subject = "Telecom Billing Tutorial"
24     Book2.book_id = 6495700
25
26     /* print Book1 info */
27     printBook(&Book1)
28     /* print Book2 info */
29     printBook(&Book2)
30 }
31
32 func printBook( book *Books ){
33
34     fmt.Printf( "Book title : %s\n", book.title);
35     fmt.Printf( "Book author : %s\n", book.author);
36     fmt.Printf( "Book subject : %s\n", book.subject);
37     |fmt.Printf( "Book book_id : %d\n", book.book_id);
38 }
```

**Output:**

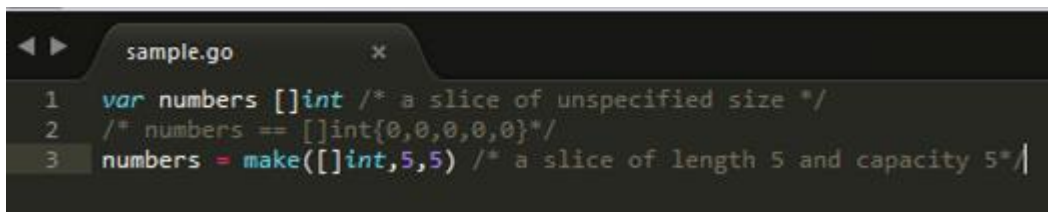
```
C:\Users\Chennu\Documents\Study\UNCC\Sem1\SurveyOfProgrammingLanguages\TermProject\Project_Part1\Examples>go run structure3.go
Book title : Go Programming
Book author : Mahesh Kumar
Book subject : Go Programming Tutorial
Book book_id : 6495407
Book title : Telecom Billing
Book author : Zara Ali
Book subject : Telecom Billing Tutorial
Book book_id : 6495700
```

### 3) 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:

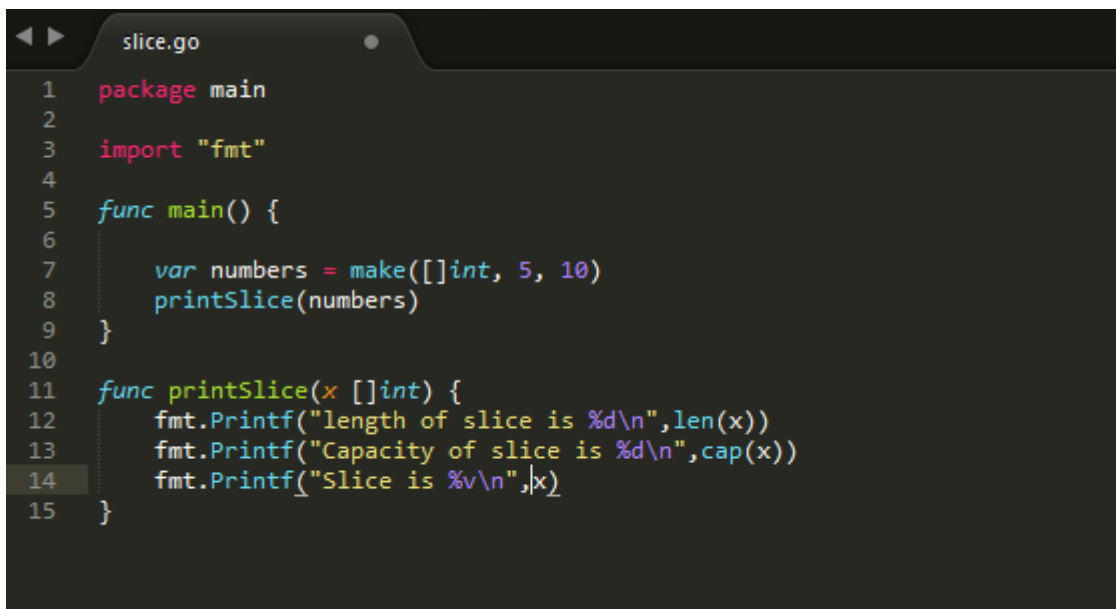
To define a slice, we can declare it as an array without specifying size or use **make** function to create the one.



```
1 var numbers []int /* a slice of unspecified size */
2 /* numbers == []int{0,0,0,0,0} */
3 numbers = make([]int,5,5) /* a slice of length 5 and capacity 5 */
```

#### 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.



```
1 package main
2
3 import "fmt"
4
5 func main() {
6
7     var numbers = make([]int, 5, 10)
8     printSlice(numbers)
9 }
10
11 func printSlice(x []int) {
12     fmt.Printf("length of slice is %d\n", len(x))
13     fmt.Printf("Capacity of slice is %d\n", cap(x))
14     fmt.Printf("Slice is %v\n", x)
15 }
```

### Output:

```
C:\Users\nikhil\Desktop\SPL\Project>go run slice.go
length of slice is 5
Capacity of slice is 10
Slice is [0 0 0 0 0]

C:\Users\nikhil\Desktop\SPL\Project>_
```

### 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:

```
1 package main
2
3 import "fmt"
4
5 func main() {
6     var numbers []int
7
8     printSlice(numbers)
9
10    if(numbers == nil) {
11        fmt.Printf("slice is nil")
12    }
13 }
14
15 func printSlice(x []int){
16     fmt.Printf("length = %d \ncapacity = %d \nslice = %v\n",len(x),cap(x),x)
17 }
```

### Output:

```
C:\Users\nikhil\Desktop\SPL\Project>go run nilSlice.go
length = 0
capacity = 0
slice = []
slice is nil
C:\Users\nikhil\Desktop\SPL\Project>
```

## 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:

```
1 package main
2 import "fmt"
3
4 func main() {
5     /* create a slice */
6     numbers := []int{0,1,2,3,4,5,6,7,8}
7     printSlice(numbers)
8
9     /* print the original slice */
10    fmt.Println("numbers ==", numbers)
11
12    /* print the sub slice starting from index 1(included) to index 4(excluded)*/
13    fmt.Println("numbers[1:4] ==", numbers[1:4])
14
15    /* missing lower bound implies 0*/
16    fmt.Println("numbers[:3] ==", numbers[:3])
17
18    /* missing upper bound implies len(s)*/
19    fmt.Println("numbers[4:] ==", numbers[4:])
20
21    numbers1 := make([]int,0,5)
22    printSlice(numbers1)
23
24    /* print the sub slice starting from index 0(included) to index 2(excluded) */
25    number2 := numbers[:2]
26    printSlice(number2)
27
28    /* print the sub slice starting from index 2(included) to index 5(excluded) */
29    number3 := numbers[2:5]
30    printSlice(number3)
31
32 }
33
34 func printSlice(x []int){
35     fmt.Printf("len=%d cap=%d slice=%v\n",len(x),cap(x),x)
36 }
```

## Output:

```
C:\Users\nikhil\Desktop\SPL\Project>go run subSlicing.go
len=9 cap=9 slice=[0 1 2 3 4 5 6 7 8]
numbers == [0 1 2 3 4 5 6 7 8]
numbers[1:4] == [1 2 3]
numbers[:3] == [0 1 2]
numbers[4:] == [4 5 6 7 8]
len=0 cap=5 slice=[]
len=2 cap=9 slice=[0 1]
len=3 cap=7 slice=[2 3 4]
C:\Users\nikhil\Desktop\SPL\Project>
```



### 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:

```
appendAndCopy.go
1 package main
2
3 import "fmt"
4
5 func main() {
6     var numbers []int
7     printSlice(numbers)
8
9     /* append allows nil slice */
10    numbers = append(numbers, 0)
11    printSlice(numbers)
12
13    /* add one element to slice*/
14    numbers = append(numbers, 1)
15    printSlice(numbers)
16
17    /* add more than one element at a time*/
18    numbers = append(numbers, 2,3,4)
19    printSlice(numbers)
20
21    /* create a slice numbers1 with double the capacity of earlier slice*/
22    numbers1 := make([]int, len(numbers), (cap(numbers))*2)
23
24    /* copy content of numbers to numbers1 */
25    copy(numbers1,numbers)
26    printSlice(numbers1)
27 }
28
29 func printSlice(x []int){
30     fmt.Printf("len=%d cap=%d slice=%v\n",len(x),cap(x),x)
31 }
```

### Output:

```
C:\Users\nikhil\Desktop\SPL\Project>go run appendAndCopy.go
len=0 cap=0 slice=[]
len=1 cap=1 slice=[0]
len=2 cap=2 slice=[0 1]
len=5 cap=6 slice=[0 1 2 3 4]
len=5 cap=12 slice=[0 1 2 3 4]
C:\Users\nikhil\Desktop\SPL\Project>_
```

#### 4) 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.

```
1  /* declare a variable, by default map will be nil*/
2  var map_variable map[key_data_type]value_data_type
3
4  /* define the map as nil map can not be assigned any value*/
5  map_variable = make(map[key_data_type]value_data_type)
```

##### Example:

Following example illustrates creation and usage of map.

```

1 package main
2
3 import "fmt"
4
5 func main() {
6     var countryCapitalMap map[string]string
7     /* create a map*/
8     countryCapitalMap = make(map[string]string)
9
10    /* insert key-value pairs in the map*/
11    countryCapitalMap["France"] = "Paris"
12    countryCapitalMap["Italy"] = "Rome"
13    countryCapitalMap["Japan"] = "Tokyo"
14    countryCapitalMap["India"] = "New Delhi"
15
16    /* print map using keys*/
17    for country := range countryCapitalMap {
18        fmt.Println("Capital of",country,"is",countryCapitalMap[country])
19    }
20
21    /* test if entry is present in the map or not*/
22    capital, ok := countryCapitalMap["United States"]
23    /* if ok is true, entry is present otherwise entry is absent*/
24    if(ok){
25        fmt.Println("Capital of United States is", capital)
26    }else {
27        fmt.Println("Capital of United States is not present")
28    }
29 }

```

### Output:

```

C:\Users\nikhil\Desktop\SPL\Project>go run map.go
Capital of India is New Delhi
Capital of France is Paris
Capital of Italy is Rome
Capital of Japan is Tokyo
Capital of United States is not present

C:\Users\nikhil\Desktop\SPL\Project>

```

### 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:

```

1 package main
2
3 import "fmt"
4
5 func main() {
6     /* create a map */
7     countryCapitalMap := map[string] string {"France": "Paris", "Italy": "Rome",
8         "Japan": "Tokyo", "India": "New Delhi"}
9
10    fmt.Println("Original map")
11
12    /* print map */
13    for country := range countryCapitalMap {
14        fmt.Println("Capital of", country, "is", countryCapitalMap[country])
15    }
16
17    /* delete an entry */
18    delete(countryCapitalMap, "France");
19    fmt.Println("Entry for France is deleted")
20
21    fmt.Println("Updated map")
22
23    /* print map */
24    for country := range countryCapitalMap {
25        fmt.Println("Capital of", country, "is", countryCapitalMap[country])
26    }
27 }

```

### Output:

```

C:\Users\nikhil\Desktop\SPL\Project>go run mapDelete.go
Original map
Capital of India is New Delhi
Capital of France is Paris
Capital of Italy is Rome
Capital of Japan is Tokyo
Entry for France is deleted
Updated map
Capital of Italy is Rome
Capital of Japan is Tokyo
Capital of India is New Delhi
C:\Users\nikhil\Desktop\SPL\Project>_

```

## The major types of your language

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	<b>Boolean Types</b> They are Boolean types and consists of the two predefined constants: (a) true (b) false
2	<b>Numeric Types</b> They are again arithmetic types and they represent a) integer types or b) floating point values throughout the program.
3	<b>string types:</b> 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	<b>Derived types:</b> 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	<b>uint8</b> Unsigned 8-bit integers (0 to 255)
2	<b>uint16</b> Unsigned 16-bit integers (0 to 65535)
3	<b>uint32</b> Unsigned 32-bit integers (0 to 4294967295)
4	<b>uint64</b> Unsigned 64-bit integers (0 to 18446744073709551615)
5	<b>int8</b> Signed 8-bit integers (-128 to 127)

6	<b>int16</b> Signed 16-bit integers (-32768 to 32767)
7	<b>int32</b> Signed 32-bit integers (-2147483648 to 2147483647)
8	<b>int64</b> Signed 64-bit integers (-9223372036854775808 to 9223372036854775807)

### **Floating Types**

The predefined architecture-independent float types are:

S.N.	Types and Description
1	<b>float32</b> IEEE-754 32-bit floating-point numbers
2	<b>float64</b> IEEE-754 64-bit floating-point numbers
3	<b>complex64</b> Complex numbers with float32 real and imaginary parts
4	<b>complex128</b> 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	<b>Byte</b> same as uint8
2	<b>Rune</b> same as int32
3	<b>Uint</b> 32 or 64 bits
4	<b>Int</b> same size as uint
5	<b>Uintptr</b> an unsigned integer to store the uninterpreted bits of a pointer value

## 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:

**var** **var\_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:

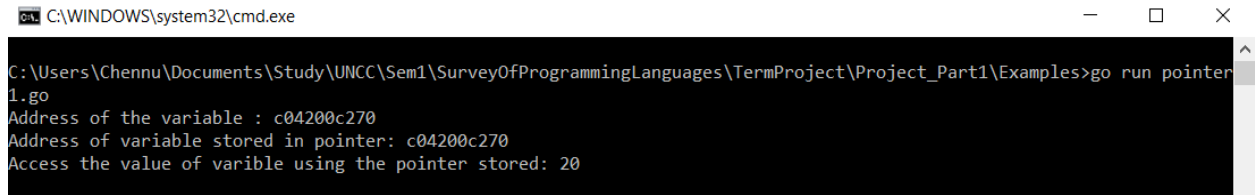
There are few operations, which we frequently perform with pointers:

- 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:

```
1 package main
2 import "fmt"
3 func main() {
4     var a int = 20 //Variable Declaration
5     var ip *int //Pointer Declaration
6
7     ip = &a // Storing address in a pointer variable.
8
9     fmt.Printf("Address of the variable : %x \n", &a)
10    fmt.Printf("Address of variable stored in pointer: %x \n", ip)
11
12    fmt.Printf("Access the value of variable using the pointer stored: %d \n", *ip)
13
14
15 }
```

### Output:



```
C:\WINDOWS\system32\cmd.exe
C:\Users\Chennu\Documents\Study\UNCC\Sem1\SurveyOfProgrammingLanguages\TermProject\Project_Part1\Examples>go run pointer1.go
Address of the variable : c04200c270
Address of variable stored in pointer: c04200c270
Access the value of variable using the pointer stored: 20
```

### Nil Pointers in Go:

Go compiler assigns a nil value to a pointer variable in case you do not 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: `var ptr *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 declare an array of pointers to an integer.

**`var ptr [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:

```
1 package main
2 import "fmt"
3
4 const MAX int = 3
5 func main() {
6     a := []int {10,20,300}
7     var i int
8     var ptr[MAX] *int
9
10    for i=0;i < MAX ; i++ {
11        ptr[i]= &a[i] //Assigning the address to a pointer variable.
12    }
13
14    for i = 0; i < MAX; i++ {
15        fmt.Printf("Value of a[%d] = %d\n", i,*ptr[i] )
16    }
17 }
```

### Output:

```
C:\Users\Chennu\Documents\Study\UNCC\Sem1\SurveyOfProgrammingLanguages\TermProject\Project_Part1\Examples>go run pointer2.go
Value of a[0] = 10
Value of a[1] = 20
Value of a[2] = 300
C:\Users\Chennu\Documents\Study\UNCC\Sem1\SurveyOfProgrammingLanguages\TermProject\Project_Part1\Examples>
```

### 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:

```
1  package main
2  import "fmt"
3  func main() {
4      var a int
5      var ptr *int
6      var pptr **int
7      a = 3000
8
9      /* take the address of var */
10     ptr = &a
11
12     /* take the address of ptr using address of operator & */
13     pptr = &ptr
14
15     /* take the value using pptr */
16     fmt.Printf("Value of a = %d\n", a )
17     fmt.Printf("Value available at *ptr = %d\n", *ptr )
18     fmt.Printf("Value available at **pptr = %d\n", **pptr)
19 }
```

### **Output:**

```
C:\WINDOWS\system32\cmd.exe
C:\Users\Chennu\Documents\Study\UNCC\Sem1\SurveyOfProgrammingLanguages\TermProject\Project_Part1\Examples>go run pointer3.go
Value of a = 3000
Value available at *ptr = 3000
Value available at **pptr = 3000
C:\Users\Chennu\Documents\Study\UNCC\Sem1\SurveyOfProgrammingLanguages\TermProject\Project_Part1\Examples>
```

### **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:

```

1  package main
2  import "fmt"
3  func main() {
4      /* local variable definition */
5      var a int = 100
6      var b int = 200
7
8      fmt.Printf("Before swap, value of a : %d\n", a )
9      fmt.Printf("Before swap, value of b : %d\n", b )
10
11     /* calling a function to swap the values.
12     * &a indicates pointer to a ie. address of variable a and
13     * &b indicates pointer to b ie. address of variable b.
14     */
15     swap(&a, &b);
16
17     fmt.Printf("After swap, value of a : %d\n", a )
18     fmt.Printf("After swap, value of b : %d\n", b )
19 }
20
21 func swap(x *int, y *int) {
22     var temp int
23     temp = *x /* save the value at address x */
24     *x = *y /* put y into x */
25     *y = temp /* put temp into y */
26 }

```

### Output:

C:\WINDOWS\system32\cmd.exe

```

C:\Users\Chennu\Documents\Study\UNCC\Sem1\SurveyOfProgrammingLanguages\TermProject\Project_Part1\Examples>go run pointer4.go
Before swap, value of a : 100
Before swap, value of b : 200
After swap, value of a : 200
After swap, value of b : 100
C:\Users\Chennu\Documents\Study\UNCC\Sem1\SurveyOfProgrammingLanguages\TermProject\Project_Part1\Examples>_

```

## **MAJOR STRENGTHS AND WEAKNESS OF GO LANGUAGE**

Every language has its own merits and demerits. So it'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.

# **EVOLUTION OF LANGUAGE'S WRITABILITY, READABILITY AND RELIABILITY**

## **Readability**

### **Simplicity**

- Go language 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
- 4.

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, OS, 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>. 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.

**Sources:**

<https://gobyexample.com/>

<https://golang.org/>

<http://www.tutorialspoint.com/>