Chapter 2

Objective

- Control Structures
- The break Keyword in Golang
- The continue Keyword in Golang
- Golang Variable Declaration Scopes
- Scopes and Shadowing Declaration in Golang
- Overview of Go Functions
- Function Returning Multiple Values
- Empty return Statement
- Anonymous Functions
- Call Anonymous Function Directly
- Variadic Functions
- Pointers in Golang
- Declaring Pointers
- Pointer To a Pointer
- Comparing Pointers in golang
- Function parameters as Pass by value
- Function Parameters as Pointers
- Returning Pointer Content from a Function
- Arrays in Go
- Golang Multidimensional Arrays
- Pass Array Pointer as Function Argument
- Pointer Arithmetic
- The new() Function In golang

This page is left blank intentionally

Unit 1

Control Structures

- The goto Keyword in Go
- The goto keyword reroutes the control flow to a previously defined label within the body of same code block
- The label name is case sensitive.
- Not a good idea to use goto keyword when developing an application.

```
 1 package main  // Example 2-1
 2
 3 import "fmt"
 4
 5 func main() {
 6    i := 0
 7
 8 Here:   // label ends with ":"
 9    fmt.Println(i)
10    i++
11    goto Here   // jump to label "Here"
12 }
13
```

The for Loop in Golang

- Golang has only one looping construct, the for loop.
- The for loop is the most powerful control logic in Golang.
- It can read data in loops and iterative operations, just like while.
- The for loop unifies for and while and there is no do-while loop.

- There are three forms, only one of which has semicolons.

```
// Like a C while
for condition { }

// Like a C for(;;)
for { }

// Like a C for
for expression1; expression2; expression3 { }
```

expression1 and expression3 are variable definitions or return values from functions
expression2 is a conditional statement.
expression1 will be executed once before looping
expression3 will be executed after each loop.

```
 1 package main  // Example 2-2
 2
 3 import "fmt"
 4
 5 func main() {
 6    sum := 0;
 7
 8    // Easy to declare the index variable right in the loop.
 9    for index:=0; index < 10 ; index++ {
10       sum += index
11    }
12
13    fmt.Println("sum is equal to ", sum)
14 }
15
```

- Omit expression1 and expression3 if they are not necessary and leave the semicolon.

```
sum := 0
for ; sum < 10;  {
    sum += sum
}
```

- Or remove the semicolon and **for loop** become identical to **while loop** in C.

```
sum := 0
for sum < 10 {
    sum += sum
}
```

The break Keyword in Golang

- The break keyword allows the termination of a loop.
- The break jumps out of the loop and continue skips the current loop and starts the next iteration.

```
 1 package main  // Example 2-3
 2
 3 import "fmt"
 4
 5 func main() {
 6
 7    num := 0
 8
 9   for { //since there are no checks, this is an infinite loop
10       if num >= 3 {
11         break // break out of for loop when condition is met
12       }
13
14       fmt.Println("Value of num is:", num)
15       num++;
16     }
17
18     fmt.Println("A statement just after for loop.")
19 }
20
```

The continue Keyword in Golang

- The continue keyword allows to go back to the beginning of the for loop.
- Checks and increments part of for loop are executed and therefore the next iteration of the loop is executed.

```
 1 package main  // Example 2-4
 2
 3 import "fmt"
 4
 5 func main() {
 6
 7 // a continue keyword within for loop block will bring
 8 // back execution to the beginning of the loop.
 9 // Checks and increments in the for loop will be executed.
10
11     for num := 0; num < 7 ; num++ {
12         if num%2 == 0 {  // if it is an even number,
13           continue    // go back to beginning of the for loop
14         }
15
16       // execution will reach here only when num%2 is not 0,
17        // and therefore it is odd
18        fmt.Println("Odd:", num)
19     }
20 }
```

Unit 2

Golang Variable Declaration Scopes

- In Golang the scope of a declaration is bound to the closest pair of curly braces, { and }.
- In Golang the variables are local to their scope.
- There are several scopes in a Golang program:
  - block scope, within curly braces, { and }
  - function scope, within function body
  - file scope, within golang source file
  - package scope, un-exported variable within the package
  - universe scope, globally declared exported variables.

Declare days to be 100 inside main, and inside for loop declare another days variable.

```
 1 package main  // Example 2-5
 2 import "fmt"
 3
 4 func main() {
 5    days := 100
 6
 7    for count := 0; count < 5; count++ {
 8       days := count
 9       fmt.Println(days)
10    }
11
12    fmt.Println(days)
13 }
14
```

Output:
0
1
2
3
4
100

Scopes and Shadowing Declaration in Golang

- The golang allow a variable to shadow another variable in the same function, even if they are in different block scopes

- Accidental Variable Shadowing within block Scope
- It's easy to do by accident and hard to catch.

```
 1 package main  // Example 2-6
 2 import "fmt"
 3
 4 func main() {
 5
 6    days := 0
 7
 8    for count := 0; count < 5; count++ {
 9       fmt.Println("Parent days scope is visible => ", days)
10
11       // Shadowing days (local to the for loop)
12       days := days + 1
13       fmt.Println("Shadowing Counter: ", days)
14     fmt.Println("next iteration -- local days out of scope")
15    }
16 }
17
```

```
Output:
Parent days scope is visible =>  0
Shadowing Counter:  1
next iteration -- local days out of scope
Parent days scope is visible =>  0
Shadowing Counter:  1
next iteration -- local days out of scope
Parent days scope is visible =>  0
Shadowing Counter:  1
next iteration -- local days out of scope
Parent days scope is visible =>  0
Shadowing Counter:  1
next iteration -- local days out of scope
Parent days scope is visible =>  0
Shadowing Counter:  1
next iteration -- local days out of scope
```

Unit 3

Overview of Golang Functions

- Functions divide source code into small reusable pieces of code.
- Use keyword **func** to define a function Name, **signature**, **return** values, named results
- A function can take zero or more typed arguments.
- The argument type comes after the variable name and arguments are separated by , (comma).
- Functions can be defined to return any number of values that are always typed.
- If there is only one return value the name can be omitted and no need to have parenthesis for the return values.
- If the function has return values, function must have return statement somewhere in the body.

```
// a simple function
func functionName() {
}

// function with parameters (types go after identifiers)
func functionName(param1 string, param2 int) {
}

// multiple parameters of the same type
func functionName(param1, param2 int) {
}

// return type declaration, no parameters are passed
func functionName() int { // return only one type (int)
    return 42
}

// return multiple values at once
func returnMulti() (int, string) { // return two different types
    return 42, "Jack"
}
var howOld, firstName = returnMulti()

// Return multiple named results simply by return
func returnMulti2() (age int, name string) {
    age = 42
    name = "Jack"
    return  // age and name will be returned
}

var howOld, firsName = returnMulti2()
```

- In golang, type of the function parameter goes after the parameter name
- The function return type is declared after the parameter list

```
 1 package main  // Example 2-7
 2 import "fmt"
 3
 4 func add(num1 int, num2 int) int {
 5     return num1 + num2
 6 }
 7
 8 func main() {
 9     fmt.Println("Total: ", add(42, 13))
10 }
```

```
Output:
    Total:  55
```

- If function arguments are same type, no need to repeat type name.

```
 1 package main  // Example 2-8
 2 import "fmt"
 3
 4 func add(num1, num2 int) int {
 5     return num1 + num2
 6 }
 7
 8 func main() {
 9     fmt.Println("Total: ", add(42, 13))
10 }
11
```

```
Output:
    Total:  55
```

Function Returning Multiple Values

```
 1 package main  // Example 2-9
 2 import "fmt"
 3
 4 func location(city string) (string, string) {
 5    var region string
 6    var continent string
 7
 8    switch city {
 9    case "Los Angeles", "LA", "Santa Monica":
10       region, continent = "California", "North America"
11
12    case "New York", "NYC":
13       region, continent = "New York", "North America"
14
15    default:
16       region, continent = "Unknown", "Unknown"
17
18    }
19
20    return region, continent
21 }
22
23 func main() {
24    region, continent := location("Santa Monica")
25
26    fmt.Printf("Matt lives in %s, %s\n", region, continent)
27 }
28
```

```
Output:
   Matt lives in California, North America
```

Empty return Statement

- Return statement without arguments returns the current values of the variables that are return.
- If variable names are in the return list, return statement can be left empty.

```
 1 package main  // Example 2-10
 2 import "fmt"
 3
 4 func location(name, city string) (region, continent string)
{
 5
 6    switch city {
 7    case "New York", "LA", "Chicago":
 8       continent = "North America"
 9
10    default:
11       continent = "Unknown"
12    }
13   return // region will be empty, it is not assigned to string value
14 }
15
16 func main() {
17    region, continent := location("Matt", "LA")
18    fmt.Printf("%s lives in %s\n", region, continent)
19    if region == "" {
20       fmt.Println("Region is Empty")
21    }
22 }
23
Output:
     lives in North America
   Region is Empty
```

Anonymous Functions

- The Anonymous functions are also called first class functions.
- Anonymous functions can be assigned to variable, passed as arguments to other functions and returned from other functions.
- Anonymous functions are useful to create inline functions.

- Assign a function to a variable name

```
 1 package main  // Example 2-11
 2 import "fmt"
 3
 4 func main() {
 5     add := func(num1, num2 int) int { // Anonymous Function
 6         return num1 + num2
 7     }
 8
 9     // use the variable name (add) to call the function
10     fmt.Println("Total: ", add(3, 4))
11 }
12
13
Output:
    Total:  7
```

Call Anonymous Function Directly

- Call anonymous function without assigning to a variable name

```
 1 // Direct call of anonymous function, not assign to variable
 2 package main  // Example 2-12
 3 import "fmt"
 4
 5 func main() {
 6
 7     func() {
 8         fmt.Println("Enjoy golang programming!!!")
 9     }()
10    // No prameters are passed to anonymous function
11 }
12
```

```
Output:
  Enjoy golang programming!!!
```

- Pass arguments to anonymous function just like any other function

```
 1 // Pass arguments to anonymous functions
 2 package main  // Example 2-13
 3 import "fmt"
 4
 5 func main() {
 6    func(word string) {
 7       fmt.Println("Moon", word)
 8    }("Light")
 9 }
10
```

```
Output:
    Moon Light
```

Variadic Functions

- Variadic functions can be called with any number of trailing arguments.
- For example, fmt.Println is a common variadic function.

```
 1 package main  // Example 2-14
 2 import "fmt"
 3
 4 func sum(args ...int) { // take an arbitrary number of ints as arguments.
 5     fmt.Print(args, " ")
 6     total := 0
 7
 8     for _, num := range args {
 9        total += num
10      }
11
12     fmt.Println("Total: ", total)
13 }
14
15 func main() {
16     // Variadic function call with individual arguments.
17     sum(1, 2)
18     sum(1, 2, 3)
19 }
20
```

```
Output:
[1 2] Total:  3
[1 2 3] Total:  6
```

Unit 4

Pointers in Golang

- A pointer is a variable which stores the memory address of another variable.

- variable **days** has value **10** and is stored at **memory address 0x0001**.
- The variable **daysPtr** holds the address of **days**. Now **daysPtr** is said to point to **days**.

Declaring Pointers

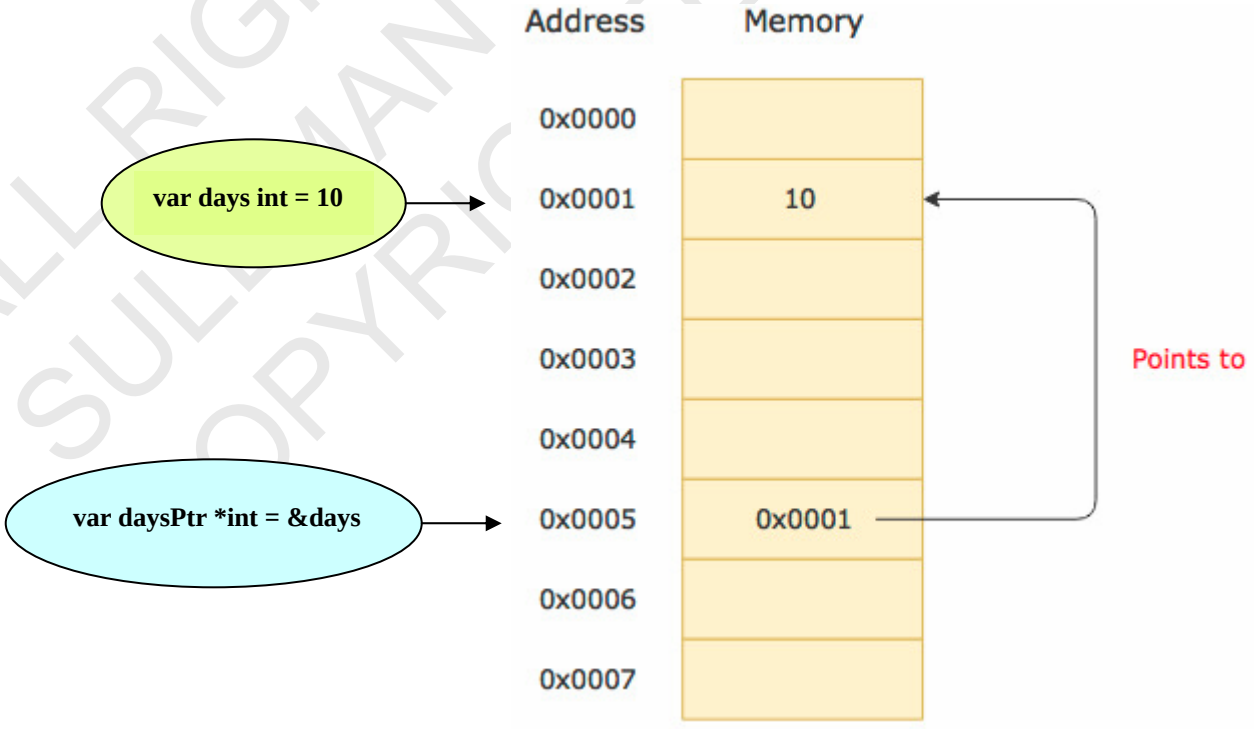
- A pointer of type int, **daysPtr**, is declared using the following syntax

```
var daysPtr *int
```

- The above pointer can only store the memory address of type int variables.
- The **daysPtr** is un-initialized pointer; therefore it has **zero value** which holds **nil**.
- This syntax for initializing pointer with the address of variable is:

```
var days = 10
var daysPtr *int = &days
```

- ***daysPtr** is the type of the pointer variable which points to a value of same type as **days**.



```
 1 package main  // Example 2-15
 2 import "fmt"
 3
 4 func main() {
 5
 6     days := 25
 7     var daysPtr *int  // daysPtr is nil
 8
 9    // daysPtr is not holding any address
10    if daysPtr == nil {
11       fmt.Println("daysPtr is", daysPtr)
12       daysPtr = &days
13
14       // daysPtr holds address of days
15       fmt.Println("address of days is", daysPtr)
16
17       // address of daysPtr
18       fmt.Println("address of daysPtr is", &daysPtr)
19
20     // Dereferencing a pointer, *daysPtr is the syntax to dereference
21       // accessing the value of variable days
22       fmt.Println("content of daysPtr is", *daysPtr)
23    }
24
```

```
25    var newdaysPtr *int = &days // get the address of days
26
27    // %T get data type
28    fmt.Printf("Type of newdaysPtr is %T\n", newdaysPtr)
29    fmt.Println("address of days is", newdaysPtr)
30
31    // address of newdaysPtr
32    fmt.Println("address of newdaysPtr is", &newdaysPtr)
33
34    // compiler creates mydayPtr as a pointer variable
35    mydaysPtr := &days
36
37    if mydaysPtr != nil {
38       fmt.Println("address of days is", mydaysPtr)
39
40       // address of mydaysPtr
41       fmt.Println("address of mydaysPtr is", &mydaysPtr)
42       fmt.Println("value of days is", *mydaysPtr)
43
44     *mydaysPtr++ // changing the content of pointer mydaysPtr
45      fmt.Println("new value of days is", days)
46      fmt.Printf("Type of mydaysPtr is %T \n", mydaysPtr)
47    }
48 }
49
50 /*
51 Output:
52
53 daysPtr is <nil>
54 address of days is 0xc00001a0b8
55 address of daysPtr is 0xc00000e028
56 content of daysPtr is 25
57 Type of newdaysPtr is *int
58 address of days is 0xc00001a0b8
59 address of newdaysPtr is 0xc00000e038
60 address of days is 0xc00001a0b8
61 address of mydaysPtr is 0xc00000e040
62 value of days is 25
63 new value of days is 26
64 Type of mydaysPtr is *int
65
66 */
```

Pointer To a Pointer

- A pointer can point to a variable of any type.
- It can point to another pointer as well.

```
 1 package main  // Example 2-16
 2 import "fmt"
 3
 4 func main() {
 5      var days = 7
 6      var daysPtr = &days
 7      var ptr2ptr = &daysPtr
 8
 9      fmt.Println("days = ", days)
10      fmt.Println("address of days = ", &days)
11
12      fmt.Println("daysPtr = ", daysPtr)
13      fmt.Println("address of daysPtr = ", &daysPtr)
14
15      fmt.Println("ptr2ptr = ", ptr2ptr)
16
17      // Dereferencing a pointer to pointer
18      fmt.Println("*ptr2ptr = ", *ptr2ptr)
19      fmt.Println("**ptr2ptr = ", **ptr2ptr)
20 }
```

Output:

```
days =  7
address of days =  0xc4200120a8
daysPtr =  0xc4200120a8
address of daysPtr =  0xc42000c028
ptr2ptr =  0xc42000c028
*ptr2ptr =  0xc4200120a8
**ptr2ptr =  7
```

Comparing Pointers in golang

- The golang support comparing of two pointers of the same type for equality using == operator.

```
  1 package main main  // Example 2-17
  2 import "fmt"
  3
  4 func main() {
  5    var num = 75
  6    var ptr1 = &num
  7    var ptr2 = &num
  8
  9    if ptr1 == ptr2 {
 10      fmt.Println("ptr1 and ptr2 point to the same variable.")
 11    }
}
```

Unit 5

Function Parameters as Pass by Value

- Pass by value gets the copy of a parameter and therefore change will not affect the original parameter.
- When function call is returned back to caller, the value of parameter will not change.

```
 1 package main  // Example 2-18
 2 import "fmt"
 3
 4 // function to add 1 to num
 5 func addOne(num int) int {
 6    num = num + 1 // value of num is changed
 7    return num  // return new value of num
 8 }
 9
10 func main() {
11    days := 3
12
13    fmt.Println("days = ", days) // should print "days = 3"
14
15   // call addOne(days), the original value of days will not change
16    newDay := addOne(days)
17
18    fmt.Println("days+1 = ", newDay) // should print "days+1 = 4"
19    fmt.Println("days = ", days)    // should print "days = 3"
20 }
21
Output:
    days =  3
    days+1 =  4
    days =  3
```

Function Parameter as Pointers

- Function parameter must know the memory address of the passed parameter (variable)
- Pass variable address using & (emp percent) and declare function parameter as pointer type (*int)
- Low cost by passing memory addresses (8 bytes)

```
 1 package main  // Example 2-19
 2
 3 import "fmt"
 4
 5 func modify(ptr *int) {
 6     *ptr = 55
 7 }
 8
 9 func main() {
10     num := 58
11
12     fmt.Println("value of num before function call is",num)
13
14     ptrNum := &num
15     modify(ptrNum)
16     fmt.Println("value of num after function call is", num)
17 }
18
Output:
    value of num before function call is 58
    value of num after function call is 55
```

Returning Pointer Content from a Function

- Use return statement to return pointer variable's content

```
 1 package main  // Example 2-20
 2 import "fmt"
 3
 4 // function to add 1 to num
 5 func addOne(num *int) int {
 6    *num = *num + 1 // we changed value of num
 7    return *num   // return new value of num
 8 }
 9
10 func main() {
11    days := 3
12
13    fmt.Println("days = ", days) // should print "days = 3"
14
15    // call addOne(&days) pass memory address of days
16    newDay := addOne(&days)
17
18    fmt.Println("days+1 = ", newDay) // should print "days+1
= 4"
19   fmt.Println("days = ", days)   // should print "days = 4"
20 }
21

Output:
    days =  3
    days+1 =  4
    days =  4
```

Unit 6

Arrays in Go

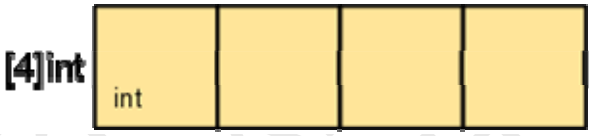
- An ***array*** is a numbered sequence of elements of a specific length.
- An array type definition specifies a length and an element type.
- The type **`[4]int`** represents an array of four integers.
- An array's size is fixed; its length is part of its type ([4]int and [5]int are distinct, **incompatible** types). Arrays can be indexed in the usual way
- The type of elements and length are both part of the array's type.
- By default an array is zero-valued, which for ints means 0s.
- The expression numArray[n] accesses the **nth** element, starting from zero.

```
var numArray [4]int  // zero-valued
numArray[0] = 1
index := numArray[0] // save 0th element value into index
// index == 1
```

- Arrays do not need to be initialized explicitly; the zero value of an array is a ready-to-use array whose elements are themselves zeroed:

```
// numArray[2] == 0, the zero value of the int type
```

- The in-memory representation of **[4]int** is just four integer values laid out sequentially:



- An array variable **"numArray"** denotes the entire array
- Array variable is **not a pointer** to the first array element (as would be the case in C).
- When **passing** or assigning array, the copy of array contents will be made.
- To avoid the copy, pass a pointer to an array

- An array literal can be specified like so:

```
strArray := [2]string{"Penn", "Teller"}
```

- Or allow compiler to **count** the array elements

```
strArray := [...]string{"Penn", "Teller"}
```

- In both cases, the **type** of **strArray** is **[2]string**.

```
// declare an int array with length 10.
// Array length is part of the type!

var numArray [10]int

numArray[3] = 42     // set 4th element

i := numArray[3]     // read 4th element

// declare and initialize numArray at same time
var numArray = [2]int{1, 2}
numArray := [2]int{1, 2} //shorthand array creation & initialize

// elipsis -> Compiler figures out array length
numArray := [...]int{1, 2}
```

```
 1 package main  // Example 2-21
 2 import "fmt"
 3
 4 func main() {
 5
 6    // Create an array that will hold exactly 5 ints.
 7    var numArray[5]int // All 5 array elements will have zero
 8    fmt.Println("show array elements:", numArray)
 9
10   // Set numArray value at an index using the array[index] = value syntax
11    numArray[4] = 100
12    fmt.Println("show updated array elements:", numArray)
13
14    // get a value with array[index].
15    fmt.Println("show 5th element value:", numArray[4])
16
17    // The built-in len returns the length of an array.
18    fmt.Println("show numArray length:", len(numArray))
19
20    // Use this syntax to declare and initialize an array in one line.
21    intArray := [5]int{1, 2, 3, 4, 5}
22    fmt.Println("show intArray:", intArray)
23 }
24
```

Output:

```
show array elements: [0 0 0 0 0]
show updated array elements: [0 0 0 0 100]
show 5th element value: 100
show numArray length: 5
show intArray: [1 2 3 4 5]
```

Golang Multidimensional Arrays

- Following is the general form of a multidimensional array declaration

```
var variable_name [SIZE1][SIZE2]...[SIZEN] variable_type

var twoDim [5][3] int (5 rows and 3 column)
```

Initializing Two-Dimensional Arrays

- Multidimensional arrays may be initialized by specifying bracketed values for each row.
- Following is an array with 3 rows and each row has 4 columns.

```
numArray = [3][4]int{
   {0, 1, 2, 3} ,   /*  initializers for row indexed by 0 */
   {4, 5, 6, 7} ,   /*  initializers for row indexed by 1 */
   {8, 9, 10, 11}   /*  initializers for row indexed by 2 */
}
```

Use subscripts to access element from two dimensional arrays
row index and column index of the array

```
int val = numArray[2][3] // take the 4th element from the 3rd row of the array.
```

```
 1 package main  // Example 2-22
 2 import "fmt"
 3
 4 func main() {
 5    /* an array with 3 rows and 2 columns*/
 6    var twoDimArray = [3][2]int{ {0,0}, {1,2}, {2,4}}
 7    var row, col int
 8
 9    /* output each array element's value */
10    for  row = 0; row < 3; row++ {
11       for col = 0; col < 2; col++ {
12          fmt.Printf("twoDimArray[%d][%d] = %d\n",
13                     row,col, twoDimArray[row][col] )
14       }
15    }
16 }
```

```
Output:
    twoDimArray[0][0] = 0
    twoDimArray[0][1] = 0
    twoDimArray[1][0] = 1
    twoDimArray[1][1] = 2
    twoDimArray[2][0] = 2
    twoDimArray[2][1] = 4
```

Pass Array Pointer as Function Argument

- Changes made to the array inside the function should be visible to the caller
- Pass a pointer to an array as an argument to the function.

```
 1 package main  // Example 2-23
 2 import "fmt"
 3
 4 func modify(arr *[3]int) {
 5     (*arr)[0] = 90  // (*arr)[0] same as arry[0]
 6 }
 7
 8 func main() {
 9     numArray := [3]int{89, 90, 91}
10     fmt.Println("Before function call: ", numArray)
11     modify(&numArray) // pass array address as function parameter
12     fmt.Println("After function call: ", numArray)
13 }
14
```

```
Output:
    Before function call:  [89 90 91]
    After function call:  [90 90 91]
```

Pointer Arithmetic

- Go does not support pointer arithmetic which is present in other languages such as C.
- C language allow pointer to move to the next/previous memory address by using increment/decrement operator.
- In golang following code will throw compilation error

```
1 package main  // Example 2-24
2
3 func main() {
4     numArray := [...]int{109, 110, 111}
5     ArrayPtr := &numArray[0]
6     ArrayPtr++
7 }
8
```

```
Output:

    # command-line-arguments
    ./ex2-21.go:6:13: invalid operation: ArrayPtr++ (non-numeric type *int)
```

Unit 7

The new() Function In golang

- Creating a pointer variable using the built-in new() function
- The new() function takes a type as an argument
- It allocates enough memory to accommodate a value of that type
- The new function returns a pointer to the allocated memory
- When new function is used compiler will always allocate memory from heap.
- If new function is **not** used for memory allocation, compiler may decide to allocate memory from heap or from stack.
- To see where memory is allocated use **"-m"** switch while compiling golang code:
  - **go run -gcflags -m app.go**

```
 1 package main  // Example 2-25
 2 import "fmt"
 3
 4 func main() {
 5    Ptr1 := new(int)  // new allocates int type and allocated
 6                      // memory address is stored in Ptr1
 7
 8    *Ptr1 = 10  // dereference Ptr1 and assign value 10 to it
 9
10    var Ptr2 *int // Ptr2 is a pointer, its zero value is nil
11
12    // Ptr1 will have address of allocated memory and Ptr2 has nil
13    fmt.Println(Ptr1, Ptr2)
14
15    Ptr2 = Ptr1  // Ptr2 gets the same address as Ptr1
16
17    // Ptr1 and Ptr2 will be holding same addresses
18    fmt.Println(Ptr1, Ptr2)
19    fmt.Println(Ptr1, *Ptr1)
20 }
21
```

```
Output:
    0xc00001a0b8 <nil>
    0xc00001a0b8 0xc00001a0b8
    0xc00001a0b8 10
```