Intelligent programming

Lecture Four

Call by value and Call by reference

Call by value

This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.

Call by reference

- This method copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.

Call by value

```
package main
import "fmt"
func swap( a, b int ) int {
    var o int
    o= a
    a=b
    b=o
   return o
}
func main( ) {
 var p int = 10
 var q int = 20
  fmt.Printf("p = %d and q = %d", p, q)
 swap(p, q)
   fmt.Printf("\np = %d and q = %d",p, q)
}
```

```
Output
p = 10 and q = 20
p = 10 and q = 20
```

Formal parameters

Actual parameters

Call by reference

```
package main
import "fmt"
func swap( a, b *int) int{
    var o int
    o = *a
    *a = *b
    *b = o
   return o
}
func main() {
 var p int = 10
 var q int = 20
 fmt.Printf("p = %d and q = %d", p, q)
 swap(&p, &q)
   fmt.Printf("\np = %d and q = %d",p, q)
}
```

```
p = 10 and q = 20
p = 20 and q = 10
```

Variadic Functions

- The function that is called with the varying number of arguments is known as variadic function.

- A user is allowed to pass zero or more arguments in the variadic function.

Variadic Functions

```
func  function_name  (para1, para2...type) type {
// code...
}
```

Anonymous function

```
package main

import "fmt"

func main() {

      func () {

     fmt.Println(“FCDS")
  }

(  )

}
```

```
Output

FCDS
```

```
package main

import "fmt"

func main() {
        func (l int, b int) {
                fmt.Println(l * b)
        }(20, 30)
}
```

You are allowed to assign an anonymous function to a variable.
Then the type of the variable is of function type and you can call that variable like a function call as shown in the below example.

```
package main

import "fmt"

func main() {

value := func( ){
     fmt.Println("FCDS")
  }
  value()

}
```

Structure in Go

- A structure or struct in Golang is a user-defined type that allows to group/combine items of different types into a single type.

- Any real-world entity which has some set of properties/fields can be represented as a struct.

- It can be termed as a lightweight class that does not support inheritance but supports composition.

```
type    structure Name    struct {
  member definition;
  member definition;
  ...
  member definition;
}
```

Structure in Go

First define the type of structure

```
type  Address  struct {
      name string
      street string
      city string
      country string
      Pincode int
}
```

```
type Address struct { name, street, city, state string, Pincode int }
```

To Define a structure:

```
var a Address
```

initialize a variable of a struct type using a struct

```
type Address struct { name, street, city, state string, Pincode int }
var a = Address{“Ahmed", “Foad", "Alexandria", “Egypt",
252636}
```

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

- variable_name := structure name {value1, value2...valuen}
- a := Address{“Ahmed", “Foad", "Alexandria", “Egypt", 252636}

- Similar to
- y:=20

```
package main
 import "fmt"
type Books struct {
  title string
  author string
  subject string
  book_id int
}
func main() {
  var Book1 Books
  var Book2 Books
 Book1.title = “Intelligent  Programming"
  Book1.author = " Kumar"
  Book1.subject = "Go Programming "
  Book1.book_id = 634507
 Book2.title = "Telecom Billing"
  Book2.author = "Zara "
  Book2.subject = "Telecom Billing Tutorial"
  Book2.book_id = 6495700
```

```
fmt.Printf( "Book 1 title : %s\n", Book1.title)
  fmt.Printf( "Book 1 author : %s\n", Book1.author)
  fmt.Printf( "Book 1 subject : %s\n", Book1.subject)
  fmt.Printf( "Book 1 book_id : %d\n", Book1.book_id)

  fmt.Printf( "Book 2 title : %s\n", Book2.title)
  fmt.Printf( "Book 2 author : %s\n", Book2.author)
  fmt.Printf( "Book 2 subject : %s\n", Book2.subject)
  fmt.Printf( "Book 2 book_id : %d\n", Book2.book_id)
}
```

Define a car structure that includes name, model, color as strings and weight as float, then define a variable of type car and assign values to it and print name and colour only, then print all values

```
package main
import "fmt"
type Car struct {
    Name, Model, Color string
    WeightInKg        float64
}
func main() {
    c := Car{Name: "Ferrari", Model: "GTC4", Color: "Red", WeightInKg: 1920}
     fmt.Println("Car Name: ", c.Name)
    fmt.Println("Car Color: ", c.Color)
     fmt.Println("Car: ", c)
}
```

Structures as Function Arguments

- You can pass a structure as a function argument in very similar way as you pass any other variable or pointer.

```
type Books struct {
   title string
   author string
   subject string
   book_id int
}
func main() {
   var Book1 Books
   var Book2 Books
    Book1.title = “Intelligent Programming"
   Book1.author = " Kumar"
   Book1.subject = "Go Programming "
   Book1.book_id = 6452307
   Book2.title = "Telecom Billing"
   Book2.author = "Zara Ali"
   Book2.subject = "Telecom Billing Tutorial"
   Book2.book_id = 6495700
   printBook(Book1)
    printBook(Book2)
}
```

```
func  printBook ( book Books ) {

   fmt.Printf( "Book title : %s\n", book.title);

   fmt.Printf( "Book author : %s\n", book.author);

   fmt.Printf( "Book subject : %s\n", book.subject);

   fmt.Printf( "Book book_id : %d\n", book.book_id);

}
```

- Define a structure of a product including a name, id, manufacture year, then input three products and define a function to print data of the product and use function to print products data

Pointers to Structures

You can define pointers to structures in the same way as you define pointer to any other variable

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's name as follows

struct_pointer = &Book1;

To access an member (titel)
struct_pointer.title;

```
type Books struct {
   title string
   author string
   subject string
   book_id int
}
func main() {
   var Book1 Books
    var Book2 Books
 Book1.title = “Intelligent Programming"
   Book1.author = "Kumar"
   Book1.subject = "Go Programming "
   Book1.book_id = 6495407
  Book2.title = "Telecom Billing"
   Book2.author = "Zara Ali"
   Book2.subject = "Telecom Billing Tutorial"
   Book2.book_id = 6495700
     printBook  (&Book1)
      printBook (&Book2)
}
```

```
func printBook( book *Books ) {

   fmt.Printf( "Book title : %s\n", book.title);

   fmt.Printf( "Book author : %s\n", book.author);

   fmt.Printf( "Book subject : %s\n", book.subject);

   fmt.Printf( "Book book_id : %d\n", book.book_id);

}
```

Define an employee structure containing first name and last name as strings and age, salary as int the define a pointer to structure.
Define a pointer of type pointer to employee and assign values to it in one statement.
Using this pointer print the employee first name and age

```
mport "fmt"

type Employee struct {
   firstName, lastName string
   age, salary int
}

func main() {

emp8 := &Employee{“Ahmed", “Sayed", 45, 6000}

    fmt.Println("First Name:", (*emp8).firstName)
   fmt.Println("Age:", (*emp8).age)
}

Output:

First Name: Ahmed
Age: 45
```