Université d'Ottawa
Faculté de génie

École de science
d'informatique
et de génie électrique

uOttawa
L'Université canadienne
Canada's university

University of Ottawa
Faculty of Engineering

School of Electrical
Engineering
and Computer Science

Assignment 1
CSI2120 Programming Paradigms
Winter 2019
Due on February 1st before 11:00 pm in Virtual Campus
6 marks

There are [10 points] in this assignment. The assignment is worth 6% of your final mark.

All code must be submitted in go files. Screenshots, files in a format of a word editor, pdfs, handwritten solutions, etc. will not be marked and receive an automatic 0.

Question 1. Structures, Methods and Interfaces **[4 points]**

Create a simple delivery simulation for a company that is located in Ottawa and ships to Montreal and Toronto.

1. Create a `struct Trip` with the following fields :
   - A `string` for the name of the `destination` for the trip,
   - A `float32` for the `weight` of the load to carry on the trip,
   - A `int` for the `deadline` in hours from for the trip.
2. Create the structures `Truck, Pickup` and `TrainCar` with the following fields (Note that the default values are given in brackets):
   - A `string` for the `vehicle` of it (Truck, Pickup and TrainCar),
   - A `string` for the `name` of it (Truck, Pickup and TrainCar),
   - A `string` for the name of the `destination` (""),
   - A `float32` for the average `speed` (40, 60 and 30),
   - A `float32` for the carrying `capacity` (10, 2 and 30),
   - A `float32` for the `load` the vehicle is assigned to carry (0),
   - In addition, the `PickupTruck` will have a `bool` field `isPrivate` (true) and the `TrainCar` will have an extra `string` field `railway` (CNR).
   - You must use embedded types in the structures minimizing duplication for full marks.
   - Implement the corresponding global functions

     `NewTruck, NewPickUp` and `NewTrainCar` returning a structure of the corresponding type with the above initializations.
3. Create an interface `Transporter` with the following two methods

- `addLoad` with a `Trip` as argument returning a `error` if the transporter has insufficient capacity to carry the weight, has a different destination or cannot make the destination on time. If the current destination is empty, the destination needs to be updated to the trip's destination.
- `print` with no argument and no return, printing the transporter to console (see below for an example)

4. Implement the following global functions
   - `NewTorontoTrip` with arguments weight as `float32` and deadline in hours as `int` returning a pointer to `Trip` with the `destination` field set to `"Toronto"`
   - `NewMontrealTrip` as above but with the `destination` field set to `"Montreal"`
5. Implement methods of the interface `Transporter` for a **pointer** to `Truck, PickUp` and `TrainCar`
6. Supply a main routine that constructs 2 `Truck,` 3 `Pickup` and 1 `TrainCar.` Then go into a loop where you ask a user to create a Trip where the user supplies the weight and the deadline in hours form now. You must only create trips that are on time and by transporters that can carry the weights. You are not asked to be efficient, i.e., you may assign the `Trip` to the first `Vehicle` in the list that can make the `Trip`. A trip as a whole must be assigned to one vehicle, but one vehicle can carry multiple trips if the destinations match and there is enough time. Print the list of trips after the loop.

Example Input/Output:

```
Destination: (t)oronto, (m)ontreal, else exit? Tor
Weight: 8
Deadline (in hours): 12
Destination: (t)oronto, (m)ontreal, else exit? mo
Weight: 8
Deadline (in hours): 20
Error: Other destination
Destination: (t)oronto, (m)ontreal, else exit? M
Weight: 8
Deadline (in hours): 12
Error: Other destination
Error: Out of capacity
Error: Out of capacity
Error: Out of capacity
Error: Out of capacity
Destination: (t)oronto, (m)ontreal, else exit? q
Not going to TO or Montreal, bye!
Trips: [{Toronto 8 12} {Montreal 8 20} {Montreal 8 12}]
Vehicles:
Truck A to Toronto with 8.000000 tons
Truck B to Montreal with 8.000000 tons
Pickup A to with 0.000000 tons (Private: true)
Pickup B to with 0.000000 tons (Private: true)
Pickup C to with 0.000000 tons (Private: true)
TrainCar A to Montreal with 8.000000 tons (CNR)
```

Question 2. Errors, Panic and Recovery **[3 points]**

Design a package to triangulate a convex polygon. Examples:

ToDo: Show a square and an octagon triangulated with a triangle fan

The polygons will be represented by an indexed vertex list where the vertices are ordered mathematically positive (counter-clockwise). The indexed vertex list for the above examples will be as follows:

```
Triangulating:
[{0 0} {1 0} {1 1} {0 1}]

Triangulating:
[{-1 -1} {1 -1} {2 0} {1 1} {-1 1} {-2 0}]
```

The result of the triangulation:

```
Mesh (4 Vertices, 2 Triangles)
Vertices: [{0 0} {1 0} {1 1} {0 1}] Faces: [{0 1 2} {0 2 3}]

Mesh (6 Vertices, 4 Triangles)
Vertices: [{-1 -1} {1 -1} {2 0} {1 1} {-1 1} {-2 0}] Faces: [{0 1 2}
{0 2 3} {0 3 4} {0 4 5}]
```

Your `triangulate` **package** will implement the following function to export (i.e., you will have to use capital names for the functions).
Write a function `Triangulate` that calculates a simple triangulation that simply forms a triangle with the first 3 vertices. Then every new vertex will form another triangle with the two previous vertices.

```
func Triangulate(vertexList []Point) (m *Mesh, err error)
```

The function prototype uses two structures directly and one indirect: Point , TriangleIndex and Mesh. These structures need to be implemented and exported from your `triangulate` package.

```
type Point struct {
     X, Y float32
```

```
}

type TriangleIndex struct {
     v1, v2, v3 int
}

type Mesh struct {
     Vertices []Point
     Faces    []TriangleIndex
}
```

Your function Triangulate will need to check if the points in the vertex list form a convex polygon. You will need to check as you form triangles that the edges (lines) between the previous and current point, and the current and next point with the following magic formula:

```
func isConvex(pA, pB, pC *Point) bool {
     return (pA.Y-pB.Y)*pC.X+(pB.X-pA.X)*pC.Y+pA.X*pB.Y-
pA.Y*pB.X >= 0
}
```

Once you have used above function, you then know if the angle is greater or less than 180 degrees. You can than use math.Acos to calculate the angle between the border segments and correct the angle for being between 0…360 degrees (use a dot product!).

If the polygon is not convex, use a panic with a **pointer** to a `convexError`.

```
type convexError struct {
     VertexIndex int     // vertex index where the error occured
     Angle       float64 // inner angle at the vertex
}
```

If two consecutive vertices are the same, use a panic with a **pointer** to a `repeatedVertexError`.

```
type repeatedVertexError struct {
     Org, Cp int // vertex numbers which were repeated
}
```

You must recover from any panic at the end of your `Triangulate` function and return an error with a specific descriptive string. It is best practice in Go to not let a panic cross package boundaries (see golang.org).

Your function `Triangulate` function or a function called from it will also have to check if the VertexList has less than three points and return an error with a descriptive string to the caller.

Your triangulate package must work correctly with the main function below (note the method m.Display).

```
package main

import (
	"fmt"
	"./triangulate"
)

func main() {
	tests := make([][]triangulate.Point, 4)

	tests[0] = []triangulate.Point{{0, 0}, {1, 0}, {1, 1}, {0, 1}}
	tests[1] = []triangulate.Point{{-1, -1}, {1, -1}, {2, 0}, {1,
1}, {-1, 1}, {-2, 0}}
	tests[2] = []triangulate.Point{{0, 0}, {1, 0}, {1, 1}, {1, 1}}
	tests[3] = []triangulate.Point{{-1, -1}, {2, -1}, {1, 0}, {2,
1}, {-1, 1}, {-2, 0}}

	for _, t := range tests {
		fmt.Println("Triangulating:")
		fmt.Printf("%v :\n", t)
		m, err := triangulate.Triangulate(t)
		if err != nil {
			fmt.Println(err)
			continue
		}
		m.Display()
	}
}
```

<u>Question 3. Concurrency</u> **[3 points]**

Write a program that uses buffered channels to monitor and lock resources. The program will use a “ComputeServer” that will use a maximum of three go routines for the calculation. The program will also use a “DisplayServer” making sure that input and output to the console is completed and interleaved.

Use two global buffered channels as semaphores and two wait groups to wait for all routines to finish before exiting.

```
const (
	NumRoutines = 3
	NumRequests = 1000
)
```

```
// global semaphore monitoring the number of routines
var semRout = make(chan int, NumRoutines)
// global semaphore monitoring console
var semDisp = make(chan int, 1)

// Waitgroups to ensure that main does not exit until all done
var wgRout sync.WaitGroup
var wgDisp sync.WaitGroup
```

Use a structure for the compute tasks:

```
type Task struct {
     a, b float32
     disp chan float32
}
```

Implement the following functions:
`func solve(t *Task)` A function that sleeps for a random time between 1 and 15 seconds, adds the numbers a and b and sends the result on the display channel.
`func handleReq(t *Task)` A function that acts as intermediary between `ComputeServer` and `solve`.
`func ComputeServer()(chan *Task)` A function that uses the channel factory pattern (lambda) and listens for requests on the created channel for tasks. It calls the `handleReq` function.
`func DisplayServer() (chan float32)` A function that uses the channel factory pattern (lambda) and listens for requests on the created channel for results to print to the console.

The draft main routine (to be completed) is given as follows:

```
     func main() {
          dispChan := DisplayServer()
          reqChan := ComputeServer()
          for {
               var a, b float32
               // make sure to use semDisp
               // …
               fmt.Print("Enter two numbers: ")
               fmt.Scanf("%f %f \n", &a, &b)
               fmt.Printf("%f %f \n", a, b)
               if a == 0 && b == 0 {
                    break
               }
               // Create task and send to ComputeServer
               // …
               time.Sleep( 1e9 )
          }
          // Don't exit until all is done
```

```
}
```

Example run:

```
Enter two numbers: 2.4 3
2.400000 3.000000
Enter two numbers: 8.0 1.5
8.000000 1.500000
-------
Result: 5.400000
-------
Enter two numbers: 0 0
0.000000 0.000000
-------
Result: 9.500000
-------
```