Name:- Umer Afzal
Roll no:- L1F23BSCS0635
Teacher:- Asad Kamal
Assignment:- DSA

Question 1:-

```
#include <iostream>
using namespace std;
const int Max = 100;
class PriorityDroneStack{
  private:
    int ids[Max];
    int priorities[Ma];
    int top;
Public:
  priorityDroneStack() : top(-1) {
    for (int i = 0; i < Max; ++i){
        ids[i] = 0;
        priorities [i] = 0;
      }
  }
  bool isEmpty(){
      return top == -1;
  }
```

Scanned with CamScanner

```
bool isfull(){
    return top == Max-1;
}
void Push(int id, int priority){
    if(!isfull()) {
        id[++top]= id;
        priorities[top] = priority;
    }
    else{
        cout<<" stack is full";
    }
}
void dispatchhighestpriority(int &id, int &priority){
    if(isEmpty ()){
        id= -1;
        priority= -1;
        cout<<" No drones is dispatch";
        return;
    }
    int maxIndex = 0;
    for(int i=1 ; i<=top; ++i) {
        if(priorities[i]> priorities[maxIndex]) {
            maxIndex= i;
        }
    }
```

```
id = ids[maxIndex];
priority = priorities[maxIndex];
for (int i = maxIndex; i < top; ++i){
    id[i] = ides [i+1];
    priorities[i] = priorities[i+1];
  }
  top--;
}
void displaystack(){
  if (isEmpty()){
    cout << "stack is empty";
    return;
  }
  cout << "current Drones in stack";
  for (int i = top; i >= 0; --i){
    cout << "Drone ID; " ids[i] << ",priority: "
    << priorities[i] << "\n";
  }
}
};
int main(){
  prioritydronestack stack;
  stack.push(101, 2);
  stack.push(102, 5);
  stack.push(103, 3);
```

Scanned with CamScanner

```
stack.displaystack();
  int id, priority;
 stack.dispatchHighestPriority(id, priority);
 cout << "Dispatched Drone -> ID: " << id << "
 , priority: " << priority << "\n";
   stack.displaystack();
    return 0;
  }
```

Concept:- A stack is a last in first out datastructure means the last item added is the first to be removed. Each drone is represented by an id and a priority. The stack uses two parallel arrays one to store the drone IDs and another to store their priority value. The top variable keeps track the current highest index in the stack, Drones are pushed into the stack using the push(), which add them to the next available slot at the top of the stack if it is not full. The drones with higher priority must be dispatched first, even if they weren't the last added. This is where the stack is modified. The dispatchhighest priority() function searches through the stack to find the drone with highest priority once found it removes that drone by shifting all elements after it one position to the left effecively deleting the drone from the array and preserving the stack order.

Scanned with CamScanner

pseudocode:-

- set top to -1 (this means the stack is currently empty)
- for each Index from 0 to max -1:
    - set ids[i] to 0
    - set priorities[i] to 0
- Return true if top equals -1 (stack is empty)
- otherwise, return false
- Return true if top equal Max-1 (stack is full)
- otherwise return false
- if the stack is not full
- increase top by 1
- Save the drone ID at ids[top]
- Save the priority at priorities[top]
- Else
- Print a message "stack is full".
- if the stack is empty
- set id = -1, priority = -1
- print "No drones to dispatch"
- Exit the function.
- otherwise
- set maxIndex = 0 to track the highest priority
- loop from index 1 to top
- if priorities[i] is greater than priorities[maxindex], update
  - maxIndex = i.
- Store the drone ID and priority at maxindex
- Shift all elements after maxIndex one position to the left (to remove the dispatched drones)
- Decrease top by 1
- if the stack is empty
- Print stack is empty
- otherwise

Scanned with CamScanner

- loop from top dow to 0:
- Print each drone ID and priority

Question 2:-

```
#include<iostream>
#include<cstring>
using namespace std;

class commandBuffer {
 private:
    char** buffer;
    int capacity;
    int size;
  void resize()
  {
    int newcapacity = capacity*2;
    char** newBuffer = new char*[newcapacity];
  for (int i = 0; i<size; ++i){
    newBuffer[i] = buffer[i];
  }
  delete[] buffer;
  buffer = newBuffer;
  capacity = newcapacity;
  }
  void shiftLeft(){
    delete[] buffer[0];
    for (int i = 1; i<size; ++i){
        buffer[i-1] = buffer[i];
      }
      --size;
  }
```

Scanned with CamScanner

```
Public
  commandBuffer(){
    capacity = 4;
     size = 0;
    buffer = newchar* [capacity];
  }
  ~commandBuffer(){
   for (int i = 0; i<size; ++i){
       delete[] buffer;
   }
   void addCommand (const char* command)
   {
     if (size == capacity){
         resize();
     }
    if (size == 10){
       shiftLeft();
     }
    buffer[size] = new char[strlen (command)+1];
    strcpy_s (buffer[size], strlen (command+1, command);
     ++ size;

   }
    void printCommands()const {
     cout << "\nCurrent buffer(" << size << "commands:
    for (int i = 0; i<size; ++i) {
       cout << "-" << buffer[i] << endl;
  ~~int~~ }
   }

  };
```

fun.

```
int main(){
    CommandBuffer cb;
    cb.addCommand("move 10");
    cb.addCommand("Turn left");
    cb.addCommand("move 5");
    cb.addCommand("Turn Right");
    cb.addCommand("Move 20");
    cb.addCommand("Jump");
    cb.addCommand("Turnleft")
    cb.addCommand("Move Backward");
    cb.addCommand("Stop");
    cb.addCommand("Turn Right");
    cb.addCommand("Move forward");
    cb.printCommands();
    return 0;
}
```

Concept :-

CommandBuffer is responsible for storing a list of text commands (Like "move 10" and "turn left") in a dynamically allocated array. The class uses a double pointer to hold the commands as strings. The capacity variable determines the total space allocated initially start from 4. while size track how

Scanned with CamScanner

many commands are currently stored.
When object of this class is created the Constructor initialize the buffer to size 4 sets the current size to 0 and allocates memory for 4 Command strings using new. when the number of Commands reaches the buffer capacity we need more space. The resize() function doubles the capacity of the buffer. It creates a new buffer with double size, copies all existing Commands to the new buffer. To keep memory usage limited to only the 10 most recent commands, this function deletes the oldest command. When size == 10. It then shifts all the remaining commands one position to the left. removing the oldest and making space at the end. first it checks if the buffer is full and rezies it if needed. Then it checks if there are already 10 commands. If yes it removes the oldest using. function print all stored commands in the buffer.

pseudocode:-

- The class is called CommandBuffer.
- It uses a dynamic array of string to store command initially it can store 4 commands but if needed it doubles the size using the resize() fun.
- The constructor setup the array
- The destructor clean up all memory to prevent leaks.

Scanned with CamScanner

- addCommand() is the main function to insert a command.
- if the buffer is full it resizes
- if already holding 10 commands, it removes the oldest.
- it the copies the new command safe and updatesize.
- print commands() shows all stored commands.

Question 3

```
#include<iostream>
#include<fstream>
#include<cstring>
using namespace std;
const int max_lines = 100;
const int max_TAGS = 100;
class TagStack {
char name[max_TAGS][50];
int Line[max_TAGS];
int top;
Public:
TagStack (){
    top = -1;
}
bool isEmpty() {
    return top==-1;
}
void Push(const char* TagName, int lineNum){
if(top <max_TAGS- 1){
    ++top;
    strcpy_s(name[top], TagName);
    line[top]= lineNum;
}
```

Scanned with CamScanner

```
}
void Pop(){
  if(!isEmpty()){
    top--;
  }
}
const char* peekName(){
  if(!isEmpty()){
    return name[top];
  }
  return "";
}
int peekline(){
  if(!isEmpty()){
    return line[top];
  }
  return -1;
}
};
bool isopeningTag(const char* tag){
    return tag[0] == '/';
}
void ~~void~~ extractTagName(const char* fullTag,
  char* tagName){
  if(fullTag[0] == '/'){
    strcpy_s(tagName, 50, fullTag+1);
  }
```

Scanned with CamScanner

```
else{
    strcpy_s (tagName ,50, fullTag);
    }
}
void validateHTMLFromFile (Const char* file name){
 TagStack stack;
 char Line[256];
 Char line[256];
  int lineNum=0;
  if stream file(filename);
  if(!file.is-open())){
   lout<<"Error to open the file";
   return;
  }
 while(file.getline(line, sizeof(line))&&lineNum<
    max_lines) {
     lineNum++
     size_t len= strlen(line);
 for (size_t i=0; i<len; ++i) {
  if (line[i]== '<') {
   char tag [50]= " ";
    size_t j=i+1, k=0;
   while(j<len && line[j]! = '>') {
    if(k < 49) tag[k++] = line[j++];
    else break;
   }
```

```
if (isOpeningTag(tag)) continue;
char tagName[50];
extractTagName(tag, tagName);
if (isOpeningTag(tag)) {
    stack.push(tagName, lineNum);
else {
  if (!stack.isEmpty() && strcmp(stack.peekNam
  (), tagName)==0) {
  }
    else {
        cout<<tagName<<" has no matching
      opening tag (line "<<lineNum<<")\n";
    }
  }
  i=j;
}
}
}
while (!stack.isEmpty()){
 cout<<stack.peekName()<<" opened on line"
  << stack.peekLine()<<" was not closed.\n";
  stack.pop();
}
 file.close();
int main(){
 const char* filename = "filename.txt";
 validateHTMLFromfile(filename);
 return 0;
}
```

Scanned with CamScanner

Concept :-

The Tagstack class is a custom stack data structure designed to store HTML Tags and there line number it use array name and line to hold tag names and the corresponding line number where ~~wi~~ those tags were found. The top variable tracks the current top of the stack. The class provides methods to push new tags onto the stack. pop tags off when closing tag is found and peek method to examine the top tag off. The function isopeningtag is a simple utility that checks if a tag is opening it doesn't begin with forward slash(/) This helps differentiate between tag like and closing tag . The function extractTagName strips the forward slash, from closing tags and extracts tag name. strips the forward slash from closing tags and extract just the tag's name. The validateHTMLformfile function is the core of the program It reads the specified file line by line and searches from HTML tags within each line. when tag is found between < and >.

Scanned with CamScanner

Pseudocode:-

- tandles storage of tage and line number
  - using a stack.
  - store tag names.
  - store line number
    - index of stack top.
  - return true if stack is empty.
  - pushes a tag and line number onto the stack.
  - remove the top element from the stack.
  - return the tag name at the top
    - return the line number at the top
    - A string tag
    - if tag does not start with (/) opening tag.
    - Return true for opening tags, false for closing.
  - Clean the tag name by removing (/) closing tag
    - logic:
      - if tag start with / return string after /
      - otherwise return tag
    - file name contain HTML
      - Open the file.
      - increment line no.
      - Scan character.
      - if < is found.
      - Read character > to extract tag.
      - Clean tag using extractTagName.
      - if it is closing tag.

Scanned with CamScanner

- compare with top stack
- if match pop the stack
- if not match or stack is empty ~~print error~~.
- closes the file after checking.

Scanned with CamScanner