Laboratory 2

Title of the Laboratory Exercise: Using dynamic memory, Arrays, Vectors and String pattern matching

1. Introduction and Purpose of Experiment

   Dynamic memory allocation is an important concept in C for allocation and release of the allocated memory dynamically which would help in many applications. This experiment includes creating and applying a vector ADT which uses the dynamic memory allocation concept. Use the created vector ADT for string pattern matching.

2. Aim and Objectives

   Aim

   To develop vector ADT and to use vector ADT for string pattern matching also demonstrate the concept of dynamic memory allocation

   Objectives

   At the end of this lab, the student will be able to

   - Illustrate the concept of dynamic memory allocation
   - Create vector ADT
   - Apply vector ADT for string pattern matching application

3. Experimental Procedure

   i. Analyse the problem statement
   ii. Design an algorithm for the given problem statement and develop a flowchart/pseudo-code
   iii. Implement the algorithm in C language iv. Compile the C program
   v. Test the implemented program
   vi. Document the Results vii. Analyse and discuss the outcomes of your experiment

4. Questions

   Design the vector ADT and apply vector ADT for string matching problem. Tabulate the output for various inputs and verify against expected values. Analyse the efficiency of the

algorithm designed. Describe your learning along with the limitations of overall approach if any. Suggest how these can be overcome.

5. Calculations/Computations/Algorithms

1. Using Pointers

```
Step 1: Start
Step 2: Declare Pointers to character, word and sentence
Step 3: Read input into word and sentence
Step 4: Declare variable i
Step 5: Repeat the steps until i = length of sentence
  5.1 if sentence + i equals word
        print found at i
Step 6: Stop
```

2. Using Malloc

```
Step 1: Start
Step 2: Declare Pointers to character, word and sentence
Step 3: Malloc 1000 bytes of memory into word and sentence
Step 4: Read input into word and sentence
Step 5: Declare variable i
Step 6: Repeat the steps until i = length of sentence
  6.1 if sentence + i equals word
        print found at i
Step 7: Stop
```

3. Using Vector ADT

```
Step 1: Start
Step 2: Declare Vector list, Pointer word, sentence
Step 3: Initialize the list
Step 4: Read input into word and sentence
Step 5: Add the sentences into the Vector list
Step 6: Repeat the steps until i = size of list
  6.1 if word is a substring of ith element of list
        print found word at pos of substring
Step 7: Stop
```

6. Presentation of Results

1. Using Pointers

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "input_helper.h"
#include "debug_helper.h"
#include "vector.h"

void check_substring(char*, char*);

int main(int argc, char **argv) {
    char* word;
    printf("Enter the word to be searched : ");
    get_word(&word);
    char* sentence;
    printf("Enter the sentences : \n");
    while (get_line(&sentence) > 0) {
        check_substring(sentence, word);
    }

    return 0;
}

void check_substring(char* sentence, char* word) {
    for (int _i = 0 ; _i < strlen(sentence) ; _i++)
        if (strncmp(sentence + _i, word, strlen(word)) == 0)
            printf("found at : %d\n", _i);
    printf("\n");
}

```

2. Using malloc

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "input_helper.h"
#include "debug_helper.h"

void check_substring(char*, char*);

int main(int argc, char **argv) {

    char* word = malloc(1000 * sizeof *word);
    printf("Enter the word to be searched : ");
    scanf("%[^\n\t]s", word);
    word = realloc(word, strlen(word));

    char* sentence = malloc(1000 * sizeof *sentence);
    printf("Enter the sentence : ");
    scanf("\n%[^\n\t]s", sentence);
    sentence = realloc(sentence, strlen(sentence));

    check_substring(sentence, word);

    return 0;
}

void check_substring(char* sentence, char* word) {
    for (int _i = 0 ; _i < strlen(sentence) ; _i++)
        if (strncmp(sentence + _i, word, strlen(word)) == 0)
            printf("found at : %d\n", _i);
    printf("\n");
}

```

3. Using ADT

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "input_helper.h"
#include "debug_helper.h"
#include "vector.h"

void check_substring(Vector, char*);

int main(int argc, char **argv) {
    Vector mylist;
    init(&mylist, print_string);

    char* word;
    printf("Enter the word to be searched : ");
    get_word(&word);
    printf("Note: Blank line to end adding more lines\n");
    char* sentence;
    printf("Enter the sentences : \n");
    while (get_line(&sentence) > 0) {
        mylist.add(&mylist, new_string(sentence));
        free(sentence);
    }
    check_substring(mylist, word);

    return 0;
}

void check_substring(Vector mylist, char* word) {
    for (int i = 0 ; i < mylist.length ; i++) {
        printf("line[%d] :\n", i);
        char* curr_string = *(mylist.data + i);
        for (int _i = 0 ; _i < strlen(curr_string) ; _i++) {
            if (strncmp(curr_string + _i, word, strlen(word)) == 0) {
                printf("found at : %d\n", _i);
            }
        }
        printf("\n");
    }
}

```

```c
#include <stdio.h>
#include <string.h>
#include "vector.h"
#include "debug_helper.h"

void init(Vector *list, void (*print_util)(Vector*)) {
    list -> length = 0;
    list -> data = NULL;
    list -> print = print_util;
    list -> add = add;
    list -> remove = del;
}

void add(Vector *list, void * DATA) {
    list -> data = realloc(list -> data, sizeof *(list -> data) * (list -> length + 1));
    (list -> data)[(list -> length)] = DATA;
    list -> length++;
}

void del(Vector *list, int index) {
    for (int i = index ; i < list -> length - 1 ; i++) {
        (list -> data)[i] = (list -> data)[i+1];
    }
    list -> length --;
    list -> data = realloc(list -> data, sizeof *(list -> data) * (list -> length));
}

/*Print Utils*/

void print_int(Vector *list) {
    for (int i = 0 ; i < list -> length ; i++) {
        di(*(int *)((list -> data)[i]));
    }
}

void print_string(Vector *list) {
    for (int i = 0 ; i < list -> length ; i++) {
        ds((char *)((list -> data)[i]))
    }
}

/* New Data of Specific DataType Methods */

void* new_int(int data) {
    int* new_data = malloc(sizeof *new_data);
    *new_data = data;
    return new_data;
}

void* new_string(char* data) {
    char* new_data = malloc(strlen(data) * sizeof *new_data);
    strcpy(new_data, data);
    return new_data;
}
```

Name: SATYAJIT GHANA **Roll Number**: 17ETCS002159

Outputs:

1. Using Pointers

```
 ./string
Enter the word to be searched : dog
Enter the sentences :
cat and dog can never be friends
found at : 8

dog dog dog cat cat cat
found at : 0
found at : 4
found at : 8
```

2. Using Malloc

```
 ./string
Enter the word to be searched : spam
Enter the sentence : this is a spam email
found at : 10
```

3. Using Vector ADT

```
 ./string
Enter the word to be searched : aba
Note: Blank line to end adding more lines
Enter the sentences :
ababbbabaacfaba
aaaabcbccbababcabbababa
ababaacaba
line[0] :
found at : 0
found at : 6
found at : 12

line[1] :
found at : 10
found at : 18
found at : 20

line[2] :
found at : 0
found at : 2
found at : 7

 ./string
Enter the word to be searched : aba
Note: Blank line to end adding more lines
Enter the sentences :
abababababababa
line[0] :
found at : 0
found at : 2
```

```
found at : 4
found at : 6
found at : 8
found at : 10
found at : 12
```

7. Analysis and Discussions

Pattern searching is an important problem in computer science. When we do search for a string in notepad/word file or browser or database, pattern searching algorithms are used to show the search results.

String pattern matching is a very commonly used algorithm in programming, this program is a substring matching program where the given word/substring is to be checked if it is present in a given sentence and the locations of the word/substring in that sentence is to be logged into the console.

The Algorithm used in this Program is very Naïve and we just check for the word from each index of the sentence, the time complexity hence obtained in $O(n^2)$ which is fairly bad, and as the size of the input increases it's not efficient anymore. For small inputs this works well, to check for string match `strncmp` is used which compares at most `n` characters of word in sentence. Alternatively the KMP algorithm can be used which has a complexity of $O(n)$.

The Vector ADT here created can facilitate any size of list of Strings or character arrays. The memory is dynamically allocated using `malloc` and then we keep track of the length of the vector as soon as an element is added to the vector.

8. Conclusions

Vector ADT and Dynamic Memory allocation can be used to store data, abstract data types are data wrapped with methods, and these methods can act upon the data. Such functions like substring/string matching can be created with these data that can be executed to give the output. Vectors here provide an efficient method to wrap the data and easily accessible.

9. Comments

1. Limitations of Experiments

The algorithm here used has a poor time complexity and consumes a lot of CPU Time, this can be improved for better results. Vector ADT can use a V-Table or a function table to store the methods that work on the data for better performance when the data size is huge.

2. Limitations of Results

The results obtained here are input strings from the KeyBoard, this can be further extended to be taken from file inputs or input arguments in argv, this will make the program more generic and usable.

3. Learning happened

The concept of Dynamic Memory allocation, efficiency of algorithms and the construction of Vector ADT's was learnt

4. Recommendations

A better Algorithm that can be used here is the KMP Algorithm. The KMP matching algorithm uses degenerating property (pattern having same sub-patterns appearing more than once in the pattern) of the pattern and improves the worst case complexity to $O(n)$. The basic idea behind KMP's algorithm is: whenever we detect a mismatch (after some matches), we already know some of the characters in the text of the next window. We take advantage of this information to avoid matching the characters that we know will anyway match.