**Structures, Pointers and Linked List**

1. Define a structure for **Employee** with the following fields.
   1. First name – string
   2. Last name – string
   3. ID – integer
   4. Designation – string
   5. Salary – number
2. Define the **main** function which declares a variable of type Employee.
   1. Read the values of the fields from keyboard

Sample input: Abhijith Kataria 18236 Manager 50000

* 1. Print only the first name and ID.

Sample output: Abhijit 18236

1. Declare an **array** of n Employee(s).
   1. Read the values and populate the fields of n employees

Sample input:

5

Shilpa Mahuli 26314 Worker 25000

Abhijith Kataria 18236 Manager 50000

Jitendra Chauhan 16331 Pune 10000

Mohit Mahajan 38745 Worker 20000

Sulu Kapoor 17331 Director 100000

* 1. Print the first name and designation of all employees who earns less than 40000.

Jitendra Chauhan 16331 Pune 10000

Mohit Mahajan 38745 Worker 20000

Shilpa Mahuli 26314 Worker 25000

1. Write a basic program with pointers as directed below.
   1. Declare a pointer to an integer variable ptr.

int \* ptr;

* 1. Use malloc to dynamically allocate memory for ptr.

ptr = (int \*) malloc( sizeof(int) );

* 1. Assign an integer value to the memory pointed to by ptr.

\*ptr = 10;

* 1. Print the value pointed to by ptr to the terminal.

printf(“%d\n”, \*ptr);

* 1. Free the ptr memory.

free(ptr);

1. Perils of pointers: Variations of 4 to simulate the problems due to mishandling of pointers.
   1. **A case of null pointer**: Access value without allocating the memory
      1. Perform steps a and c (i.e. without b).
      2. Note down the error.
   2. **Another case of null pointer**: Access value after freeing the memory
      1. Perform steps a, b, c, e and then d.
      2. Note down the error.
   3. **A case of memory leak**: Allocating memory without freeing.
      1. In an infinite loop, do steps a, b, c and d (i.e. without e).
      2. Run the program for as long as it can.
      3. The program + system will crash after some time. Restart your computer.
   4. **A case of not allowing memory leak**: Allocating memory with freeing
      1. In an infinite loop, do steps a, b, c, d and e.
      2. The program should run forever without crashing.
      3. Press CTRL+C to stop execution.
   5. **A case of lost pointer and memory leak**: Re-assigning a pointer to another location

int \* p = (int \*) malloc( sizeof(int) ); // p points to a location\_1 in memory

\*p = 5;

int \* q = (int \*) malloc( sizeof(int) ); // p points to a location\_2 in memory

\*q = 10;

p = q; // p is reassigned and both p and q point to location\_2

* + 1. The access to location\_1 is lost. It is impossible to retrieve the value 5.
    2. It can’t be freed either since the pointer is lost. This leads to memory leak.
  1. **Another case of lost pointer**: Re-assigning a pointer to a new memory

int \* p = (int \*) malloc( sizeof(int) ); // p points to a location\_1 in memory

\*p = 5;

p = (int \*) malloc( sizeof(int) ); // p reassigned to a new location (location\_2)

\*p = 10;

* + 1. The access to location\_1 is lost. It is impossible to retrieve the value 5.
    2. It can’t be freed either since the pointer is lost. This leads to memory leak.
  1. **A case of dangling pointer**: Freeing up a pointer when another is accessing the same location.

int \* p = (int \*) malloc( sizeof(int) ); // p points to a location\_1 in memory

int \* q = p; // Both p and q not point to location\_1

free(p); // p frees location\_1. The runtime system claims it.

\*q = 10; // Can’t access location\_1 since it is no more program memory

* + 1. In short, q points to a location which does not legally belong to the program
    2. Note down the error
  1. **A case of messing up with pointer**: incorrect type casting leads

long \* ptr = (long \*) malloc( sizeof(int) ); // 4 bytes allocated

\*ptr = 10; // This will access 8 bytes of memory

* + 1. Out of 8 bytes, only first 4 can be legally accessed.
    2. The runtime system will report an error when trying to assign value 10.
    3. Note down the error

1. Declare a pointer to 10 integers
   1. int \* p;
   2. p = (int \*) malloc( sizeof(int) \* 10 );
   3. In a loop, assign values to 10 integers

\*(p+i) = i; OR scanf(“%d”, (p+i));

* 1. Print the values of all integers in a loop

1. Implement a linked list with insert, search, insertat and deleteat functions.
   1. Define a structure Node with integer data and pointer to next Node.
   2. Create a head node.
   3. First insert should be directly done on head node.
      1. Check if head is null.
      2. If so, malloc space for a node.
      3. Assign head->data.
      4. Test it out.
      5. An alternative is to define a separate function insertfirst which will create the head node and return the address so that head can now point to the new address.

struct Node insertfirst(struct Node \*head, int d) {

head = (struct Node \*)malloc( sizeof(struct Node) );

head->data = d;

return head;

}

* 1. Implement **insert** function to insert a new node at the end of the list.
     1. Pass the head and integer value as arguments
     2. Test the working of insert by calling it with multiple values
  2. Implement a **print** function which prints the data from head till end.
     1. Pass the head as the argument
  3. Implement **search** function which returns the position of an integer data.
     1. Pass the head and the integer value searched for as arguments
     2. Test the working of search by calling it with multiple values
  4. Implement **insertat** function which inserts a new node at position i.
     1. Pass the head, integer value and position to insert as arguments
     2. Test the working of insert by calling it with multiple values
  5. Implement **deleteat** function which deletes a node at position i.
     1. Pass the head and position to delete as arguments
     2. Test the working of delete by calling it a multiple values
  6. You can build other functions by using the above functions.
     1. Insertbefore – search & insertat the position returned by search
     2. Insertafter – search & insertat 1+position returned by search
     3. Push – insertat position 0
     4. Pop – deleteat position 0
     5. Enqueue – insert at the end
     6. Dequeue – deleteat postion 0

Note:

* During implementation, some nitty-gritties will crop up. Seek help if you are stuck for a while.
* Incorrect use of pointers can lead to unpredictable results. It can vary from system to system. It is better to make mistakes deliberately (as given in question 5) so that you remember.
* In case you find any mistakes in this document, please be kind enough to point them out. You can email to [swaminathanj@am.amrita.edu](mailto:swaminathanj@am.amrita.edu).
* The next page contains implementation of insert and print functions. In case you have tried long enough and can’t get your program to work, you may refer it. Don’t cut-n-paste.

#include <stdio.h>

#include <stdlib.h>

struct Node {

    int data;

    struct Node \* next;

};

void print(struct Node \* p) {

    while (p != NULL) {

        printf("%d ", p->data);

        p = p->next;

    }

}

void insert(struct Node \* p, int x) {

    while (p->next != NULL)

        p = p->next;

    struct Node \* temp = (struct Node \*)malloc(sizeof(struct Node));

    temp->data = x;

    p->next = temp;

}

int main() {

    int n; // number of elements

    scanf("%d", &n);

    int x; // read the first element

    scanf("%d", &x);

    // This is first insert. Head is created and assigned directly.

    struct Node \* head = (struct Node \*)malloc( sizeof(struct Node));

    head->data = x;

    // From now on, you can insert by calling insert function

    for (int i=1; i<n; i++) {  // note i starts from 1

        scanf("%d", &x);

        insert(head,x);

    }

    print(head);

    return 0;

}