University of Zurich

Department of Informatics
Database Technology Group
Prof. Dr. Anton Dignös

Informatics II
Exercise 6

Mar 29, 2021

Goals:

- Practise pointers of primitive types: int, double, char.
- Practise pointers of arrays.
- Study the linked list data structure and implement it in C.
- Practise linked list with a coding task .

Pointers

Task 1. Pointers, values, and addresses.

a) Write a short C program that declares and initializes three variables: double d , int i , and char ch. Next declare and initialize a pointer to each of the three variables: pointer p_d for d, p_i for i, p_ch for ch. Print the following information of d, i, ch, p_d, p_i, and p_ch:

(a) their values

(b) their addresses

(c) memory sizes(in bytes)

Use the "%p" formatting specifier to print addresses in hexadecimal. You should see addresses that look something like this: "0xbfe55918". The initial characters "0x" tell you that hexadecimal notation is being used; the remainder of the digits give the address itself.

Use "%f" to print a floating value. Use the *sizeof* operator to determine the memory size allocated for each variable, then use "%lu" to print it .

```c
#include <stdio.h>

int main()
{
  char ch = 'a';
  int i = 1;
  double d = 1.2;
  char* p_ch = &ch;
  int* p_i = &i;
  double* p_d = &d;

  printf("The address of ch is %p \n", &ch);
  printf("The address of i is %p \n", &i);
  printf("The address of d is %p \n", &d);
  printf("The␣address␣of␣p_ch␣is␣%p␣\n", &p_ch);
  printf("The␣address␣of␣p_i␣is␣%p␣\n", &p_i);
  printf("The␣address␣of␣p_d␣is␣%p␣\n", &p_d);

  printf("The␣value␣of␣ch␣is␣%c␣\n", ch);
  printf("The␣value␣of␣i␣is␣%d␣\n", i);
  printf("The␣value␣of␣d␣is␣%f␣\n", d);
  printf("The␣value␣of␣p_ch␣is␣%p␣\n", p_ch);
  printf("The␣value␣of␣p_i␣is␣%p␣\n", p_i);
  printf("The␣value␣of␣p_d␣is␣%p␣\n", p_d);

  printf("The␣size␣of␣ch␣is␣%lu␣bytes␣\n", sizeof(ch));
  printf("The␣size␣of␣i␣is␣%lu␣bytes␣\n", sizeof(i));
  printf("The␣size␣of␣d␣is␣%lu␣bytes␣\n", sizeof(d));
  printf("The␣size␣of␣p_ch␣is␣%lu␣bytes␣\n", sizeof(p_ch));
  printf("The␣size␣of␣p_i␣is␣%lu␣bytes␣\n", sizeof(p_i));
  printf("The␣size␣of␣p_d␣is␣%lu␣bytes␣\n", sizeof(p_d));
}
```

b) Check the two C functions, *swap_nums* seems to work, but *swap_pointers* does not. Fix it.

```c
#include <stdio.h>

void swap_nums(int *x, int *y)
{
  int tmp;
  tmp = *x;
  *x = *y;
  *y = tmp;
}

void swap_pointers(char *x, char *y)
{
  char *tmp;
  tmp = x;
  x = y;
  y = tmp;
}

int main()
{
  int a,b;
  char *s1,*s2;
  a = 3; b=4;
  swap_nums(&a,&b);
  printf("a␣is␣%d\n", a);
  printf("b␣is␣%d\n", b);
  s1 = "I␣should␣print␣second";
  s2 = "I␣should␣print␣first";
  swap_pointers(s1,s2);
  printf("s1␣is␣%s\n", s1);
  printf("s2␣is␣%s\n", s2);
  return 0;
}
```

```c
#include <stdio.h>
```

```

void swap_nums(int *x, int *y)
{
  int tmp;
  tmp = *x;
  *x = *y;
  *y = tmp;
}

void swap_pointers(char **x, char **y)
{
  char *tmp;
  tmp = *x;
  *x = *y;
  *y = tmp;
}

int main()
{
  int a,b;
  char *s1,*s2;
  a = 3; b=4;
  swap_nums(&a,&b);
  printf("a is %d\n", a);
  printf("b is %d\n", b);
  s1 = "I should print second";
  s2 = "I should print first";
  swap_pointers(&s1,&s2);
  printf("s1 is %s\n", s1);
  printf("s2 is %s\n", s2);
  return 0;
}
```

Task 2. Pointers are widely used to access strings and arrays. In this exercise, we use pointers instead of the [] operator to manipulate strings and arrays.

a) Write a program in C to calculate the length of the string **using pointers**.

```
#include <stdio.h>
int calculateLength(char*);

int main()
{
  char str1[100];
  int l;
    printf("\n\n Pointer : Calculate the length of the string :\n");
    printf("--------------------------------------------------\n");

  printf(" Input a string : ");
  fgets(str1, sizeof str1, stdin);

  l = calculateLength(str1);
  printf(" The length of the given string %s is : %d ", str1, l-1);
  printf("\n\n");

}
```

```

int calculateLength(char∗ ch) // ch = base address of array str1 ( &str1[0] )
{
  int ctr = 0;
  while (∗ch != ’\0’)
  {
    ctr++;
    ch++;
  }
  return ctr;
}
```

b) Write a program in C to print a string in reverse **using pointers**.

```
#include <stdio.h>
int main()
{
  char str1[50];
  char revstr[50];
  char ∗stptr = str1;
  char ∗rvptr = revstr;
  int i=−1;
    printf(”\n\n␣Pointer␣:␣Print␣a␣string␣in␣reverse␣order␣:\n”);
    printf(”−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−\n”);
  printf(”␣Input␣a␣string␣:␣”);
  scanf(”%s”,str1);
  while(∗stptr)
  {
    stptr++;
    i++;
  }
  while(i≥0)
  {
    stptr−−;
    ∗rvptr = ∗stptr;
    rvptr++;
    −−i;
  }
  ∗rvptr=’\0’;
  printf(”␣Reverse␣of␣the␣string␣is␣:␣%s\n\n”,revstr);
  return 0;
}
```

c) Write a program in C to print the elements of an array in reverse order **using pointers**.

```
#include <stdio.h>
int main()
{
  int n, i, arr1[15];
  int ∗pt;
    printf(”\n\n␣Pointer␣:␣Print␣the␣elements␣of␣an␣array␣in␣reverse␣order␣:\n”);
    printf(”−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−

  printf(”␣Input␣the␣number␣of␣elements␣to␣store␣in␣the␣array␣(max␣15)␣:␣”);
  scanf(”%d”,&n);
  pt = &arr1[0]; // pt stores the address of base array arr1
  printf("␣Input␣%d␣number␣of␣elements␣in␣the␣array␣:␣\n",n);
  for(i=0;i<n;i++) {
    printf("␣element␣-␣%d␣:␣",i+1);
    scanf("%d",pt);//accept the address of the value
    pt++;
  }

  pt = &arr1[n - 1];

  printf("\n␣The␣elements␣of␣array␣in␣reverse␣order␣are␣:");

  for (i = n; i > 0; i--)
  {
    printf("\n␣element␣-␣%d␣:␣%d␣␣", i, *pt);
    pt--;
  }
  printf("\n\n");
}
```

d) Write a C program to multiply two matrix **using pointers**.

```
/**
 * C program to multiply two matrix using pointers
 */

#include <stdio.h>

#define ROW 3
#define COL 3


/* Function declarations */
void matrixInput(int mat[][COL]);
void matrixPrint(int mat[][COL]);
void matrixMultiply(int mat1[][COL], int mat2[][COL], int res[][COL]);



int main()
{
    int mat1[ROW][COL];
    int mat2[ROW][COL];
    int product[ROW][COL];


    /*
     * Input elements in matrices.
     */
    printf("Enter␣elements␣in␣first␣matrix␣of␣size␣%dx%d\n", ROW, COL);
    matrixInput(mat1);

    printf("Enter␣elements␣in␣second␣matrix␣of␣size␣%dx%d\n", ROW, COL);
    matrixInput(mat2);


    // Call function to multiply both matrices
    matrixMultiply(mat1, mat2, product);


    // Print product of both matrix
    printf("Product␣of␣both␣matrices␣is␣:␣\n");
    matrixPrint(product);

    return 0;
}



/**
 * Function to input elements in matrix from user.
 *
 * @mat Two-dimensional array to store user input.
 */
void matrixInput(int mat[][COL])
{
    int row, col;

    for (row = 0; row < ROW; row++)
    {
        for (col = 0; col < COL; col++)
        {
            scanf("%d", (*(mat + row) + col));
        }
    }
}




/**
 * Function to print elements in a two-dimensional array.
 *
 * @mat Two-dimensional array to print.
 */
void matrixPrint(int mat[][COL])
{
    int row, col;

    for (row = 0; row < ROW; row++)
    {
        for (col = 0; col < COL; col++)
        {
            printf("%d␣", *(*(mat + row) + col));
        }

        printf("\n");
    }
}




```

```c
/**
 * Function to multiply two matrices.
 *
 * @mat1 First matrix
 * @mat2 Second matrix
 * @res Resultant matrix to store product of both matrices.
 */
void matrixMultiply(int mat1[][COL], int mat2[][COL], int res[][COL])
{
    int row, col, i;
    int sum;


    for (row = 0; row < ROW; row++)
    {
        for (col = 0; col < COL; col++)
        {
            sum = 0;

            /*
             * Find sum of product of each elements of
             * rows of first matrix and columns of second
             * matrix.
             */
            for (i = 0; i < COL; i++)
            {
                sum += (*(*(mat1 + row) + i)) * (*(*(mat2 + i) + col));
            }


            /*
             * Store sum of product of row of first matrix
             * and column of second matrix to resultant matrix.
             */
            *(*(res + row) + col) = sum;
        }
    }
}
```

Linked Lists

Task 3. We have seen how a linked list works from the lectures. Now let us implement the linked list data structure in C.

a) *void createNodeList(int n)* and *void displayList()* that create and display Singly Linked List (of n nodes).

```c
#include <stdio.h>
#include <stdlib.h>

struct node
{
    int num; //Data of the node
    struct node *nextptr; //Address of the node
}*stnode;
```

```

void createNodeList(int n);     //function to create the list
void displayList();     //function to display the list
void insertNode(int num, int pos); //function to insert node at the middle
void deleteNode(int pos); //function to delete a node from middle

void createNodeList(int n)
{
    struct node *fnNode, *tmp;
    int num, i;
    stnode = (struct node *)malloc(sizeof(struct node));
    if(stnode == NULL) //check whether the stnode is NULL and if so no memory allocation
    {
        printf("␣Memory␣can␣not␣be␣allocated.");
    }
    else
    {
            // reads data for the node through keyboard
            printf("␣Input␣data␣for␣node␣1␣:␣");
            scanf("%d", &num);
            stnode-> num = num;
            stnode-> nextptr = NULL; //Links the address field to NULL
            tmp = stnode;
            //Creates n nodes and adds to linked list
            for(i=2; i≤n; i++)
            {
                    fnNode = (struct node *)malloc(sizeof(struct node));
                    if(fnNode == NULL) //check whether the fnnode is NULL and if so no memory allocation
                    {
                            printf("␣Memory␣can␣not␣be␣allocated.");
                            break;
                    }
                    else
                    {
                            printf("␣Input␣data␣for␣node␣%d␣:␣", i);
                            scanf("␣%d", &num);

                            fnNode->num = num; // links the num field of fnNode with num
                            fnNode->nextptr = NULL; // links the address field of fnNode with NULL

                            tmp->nextptr = fnNode; // links previous node i.e. tmp to the fnNode
                            tmp = tmp->nextptr;
                    }
            }
    }
}

void displayList()
{
    struct node *tmp;
    if(stnode == NULL)
    {
        printf("␣No␣data␣found␣in␣the␣empty␣list.");
    }
    else
    {
```

```c
        tmp = stnode;
        while(tmp != NULL)
        {
            printf(" Data = %d\n", tmp->num); // prints the data of current node
            tmp = tmp->nextptr; // advances the position of current node
        }
    }
}
```

b) *void insertNode(int num, int pos)* that inserts a new node at the middle of Singly Linked List.

```c
void insertNode(int num, int pos)
{
    int i;
    struct node *fnNode, *tmp;
    fnNode = (struct node*)malloc(sizeof(struct node));
    if(fnNode == NULL)
    {
        printf(" Memory can not be allocated.");
    }
    else
    {
        fnNode->num = num; //Links the data part
        fnNode->nextptr = NULL;
        tmp = stnode;
        for(i=2; i<=pos-1; i++)
        {
            tmp = tmp->nextptr;

            if(tmp == NULL)
                break;
        }
        if(tmp != NULL)
        {
            fnNode->nextptr = tmp->nextptr; //Links the address part of new node
            tmp->nextptr = fnNode;
        }
        else
        {
            printf(" Insert is not possible to the given position.\n");
        }
    }
}
```

c) *void deleteNode(int pos)* that deletes a node from the middle of Singly Linked List.

```c
void deleteNode(int pos)
{
    int i;
    struct node *toDelMid, *preNode;

    if(stnode == NULL)
    {
        printf(" There are no nodes in the List.");
    }
    else
    {
        toDelMid = stnode;
        preNode = stnode;

        for(i=2; i<=pos; i++)
        {
            preNode = toDelMid;
            toDelMid = toDelMid->nextptr;

            if(toDelMid == NULL)
                break;
        }
        if(toDelMid != NULL)
        {
            if(toDelMid == stnode)
                stnode = stnode->nextptr;

            preNode->nextptr = toDelMid->nextptr;
            toDelMid->nextptr = NULL;
            free(toDelMid);
        }
        else
        {
            printf(" Deletion can not be possible from that position.");
        }
    }
}
```

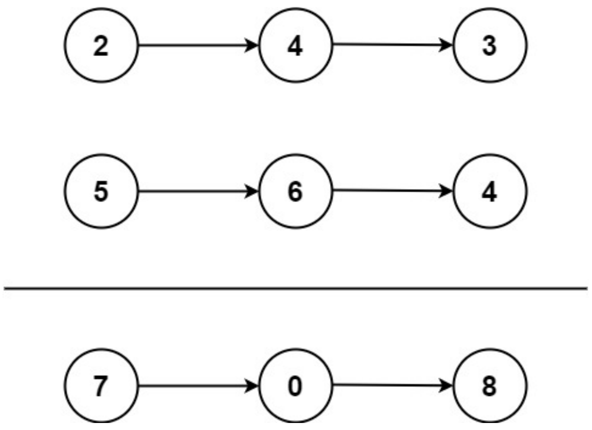
d) *int FindElement(int FindElem)* that searches an existing element in a Singly Linked List.

```
int FindElement(int FindElem)
{
    int ctr=1;
    struct node* ennode = stnode;
    while(ennode->nextptr!=NULL)
    {
        if(ennode->num==FindElem)
            break;
        else
            ctr++;
            ennode=ennode->nextptr;
    }
    return ctr;
}
```

Task 4. You are given two non-empty linked lists representing two non-negative integers. The digits are stored in reverse order, and each of their nodes contains a single digit. Add the two numbers and return the sum as a linked list.

You may assume the two numbers do not contain any leading zero, except the number 0 itself.

Example 1:



Input: $l1 = [2, 4, 3]$, $l2 = [5, 6, 4]$
Output: $[7, 0, 8]$
Explanation: $342 + 465 = 807$.

Example 2:

Input: $l1 = [0], l2 = [0]$
Output: $[0]$

Example 3:

Input: $l1 = [9, 9, 9, 9, 9, 9, 9], l2 = [9, 9, 9, 9]$
Output: $[8, 9, 9, 9, 0, 0, 0, 1]$

Since the two input linked lists store digits in a reverse order, we can add up the same position of the two linked lists directly.

We iterate the two input linked lists at the same time, sum up the digits position by position, and also add the carried value from the previous postion.

If the two input linked lists have different lengths, we can think of the short one have some 0s in the end.

In addition, when we finish the iteration if we still have a carried value $carry > 0$, we need to append a new node in the end that have value $carry$.

```c
struct node* addTwoNumbers(struct node* l1, struct node* l2) {
    struct node *head = NULL, *tail = NULL;
    int carry = 0;
    while (l1 || l2) {
        int n1 = l1 ? l1->num : 0;
        int n2 = l2 ? l2->num : 0;
        int sum = n1 + n2 + carry;
        if (!head) {
            head = tail = malloc(sizeof(struct node));
            tail->num = sum % 10;
            tail->nextptr = NULL;
        } else {
            tail->nextptr = malloc(sizeof(struct node));
            tail->nextptr->num = sum % 10;
            tail = tail->nextptr;
            tail->nextptr = NULL;
        }
        carry = sum / 10;
        if (l1) {
            l1 = l1->nextptr;
        }
        if (l2) {
            l2 = l2->nextptr;
        }
    }
    if (carry > 0) {
        tail->nextptr = malloc(sizeof(struct node));
        tail->nextptr->num = carry;
        tail->nextptr->nextptr = NULL;
    }
    return head;
}
```