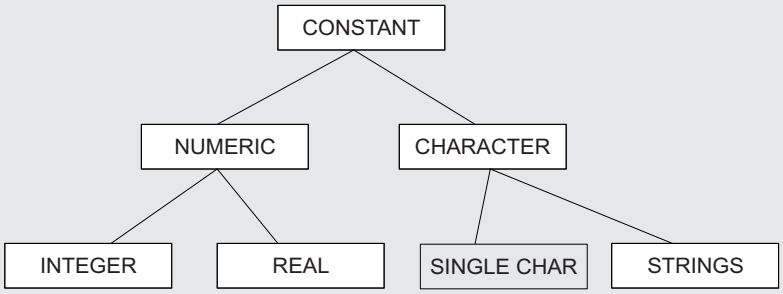
PAPER 3

1. (a) What are constants?

Constants refer to fixed values, which do not change during the execution of a program. There are many types of constants in C as shown below:



1. (b) Name the different data types that C supports and explain them in detail.

- **Integer types**—these are whole numbers without any decimal portion. They can be signed or unsigned. They are also classified as short and long integers depending on their value.
- **Floating point types**—these are real numbers (with decimal portion) which are stored in 32 bits with 6 digits of precision. There are two types of such numbers in C. These are float and double. The data type double offers more precision when compared to float data type.
- **Character type**—A single character is defined as a char data type. They are stored in one byte of internal storage. They can be signed or unsigned. Signed char can have values between 0 and 255 while unsigned have –128 to 127 ranges.
- **User defined data types**—Apart from the above, the programmers can define their own data types using the typedef keyword.

2. (a) Distinguish between getchar and scanf functions for reading strings.

- **getchar**—this function allows us to read a character from the standard input (normally keyboard). Getchar is always used to read only one character at a time and never a string. The general form of getchar function is:

```
char c;
int d;
c = getchar();
d = getchar();
```

Note that we have an int as well as char variable defined for getting the character. The int variable will store the ASCII code of the input character while the char variable will store it as such.

- **scanf**—If you want to input a string value, you need to use getchar in a loop and assemble the string into a character array. Instead, you can use the formatted input function, which is the scanf function. You can use %s to accept a string or %c to accept characters using scanf. The scanf takes the following form:

```
char a[10], b[15];
scanf("%s %15c", a, b);
```

2. (b) Write a program to count the number of words, lines and characters in a text.

```
#include<stdio.h>

main()
{
  char line[81], ctr;
  int i,c,end = 0, characters = 0, words = 0, lines = 0;
  printf("\nKEY IN THE TEXT ");
  printf("GIVE ONE SPACE AFTER EACH WORD.\n");
  printf("WHEN COMPLETED, PRESS 'ENTER'\n\n");
  while (end == 0)
  {
    c = 0;
    while ((ctr = getchar()) != '\n')
      line[c++] = ctr;
    line[c] = '\0';
    if (line[0] == '\0')
      break;
    else
    {
    words++;
    for (i = 0; line[i] != '\0'; i++)
      if (line[i] == ' ' || line[i] == '\t')
      words++;
    }
    lines = lines + 1;
    characters = characters + strlen(line);
  }
  printf("\n");
  printf("\nNumber of lines = %d", lines);
  printf("\nNumber of words = %d", words);
  printf("\nNumber of characters = %d", characters);
}
```

3. (a) When are array of structures used? Declare a variable as array of structure and initialize it.

Consider a simple example. While analyzing the marks obtained by a class of students, we may use a template to describe student name and marks obtained in various subjects and

then declare all the students as structure variables. In such cases, we may declare an array of structures, each element of the array representing a structure variable.

```
struct marks
{
        int marks1;
        int marks2;
        int marks3;
};
main()
{
        static struct marks IIIMCA[2] =
                {{78, 76, 87}, {67, 87, 76}};
}
```

3. (b) Write a C program to calculate student-wise total for three students using an array of structure.

```
struct marks
{
        int sub1;
        int sub2;
        int sub3;
        int total;
};

main()
{
        int i;
        static struct marks student[3] = { {56, 45, 65},
                                           {59, 55, 75},
                                           {45, 65, 75}};
        static struct marks total;
        for (i = 0; i <= 2; i++)
        {
             student[i].total = student[i].sub1 + student[i].sub2 +
student[i].sub3;
             total.sub1 = total.sub1 + student[i].sub1;
             total.sub2 = total.sub2 + student[i].sub2;
             total.sub3 = total.sub3 + student[i].sub3;
             total.total = total.total + student[i].total;
        }
        printf("\nSTUDENT              TOTAL\n\n");
        for (i = 0; i <= 2; i++)
                printf("\nStudent[%d]     %d", i + 1, student[i].total);
        printf("\nSUBJECT              TOTAL\n\n");
        printf("%s %d\n%s %d\n%s %d\n",
                  "Subject 1", total.sub1,
                  "Subject 2", total.sub2,
                  "Subject 3", total.sub3);
        printf(\nGrand Total = %d",total.total);
}
```

4. (a) Write a C program to find the factorial of a given number using pointers.

```
main()
{
        int n, factorial;
        printf("\nEnter a number ");
        scanf("%d", &n);
        factcompute(n, &factorial);
        printf("\nThe factorial of %d is %d", n, factorial);
}
factcompute(a, b)
int a, *b;
{
        int m;
        *b = 1;
        for (m = 1; m <= a; m++)
                *b = *b * m;
}
```

4. (b) Write a C program to arrange the given names in alphabetical order using pointers.

```
#define COUNT 5
main()
{
char *names[5], temp[20];
int m, n;
for (m = 0; m < COUNT; m++)
{
  printf("\nEnter name[%d] : ",m);
  names[m] = (char *) malloc (20);
  scanf("%s",names[m]);
}
for (m = 1; m <= COUNT; m++)
  for (n = 1; n <= COUNT - m; n++)
    if (strcmp(names[n - 1], names[n]) > 0)
    {
      strcpy(temp, names[n - 1]);
      strcpy(names[n - 1], names[n]);
      strcpy(names[n], temp);
      temp[0] = '\0';
    }
printf("\nSorted list is ");
for (m = 0; m < COUNT; m++)
  printf("\n%s", names[m]);
}
```

5. Write a program that will receive a filename and a line of text as command line arguments and write the text to the file.

Program

```
/*****************************************************************************/
/*                    COMMAND LINE ARGUMENTS                                 */
/*****************************************************************************/

#include <stdio.h>
```

```
main(argc, argv)                    /*      main with arguments             */
int argc;                           /*      argument count                  */
char *argv[ ];                      /*      list of arguments               */
{
    FILE        *fp;
    int         i;
    char        word[15];

    fp = fopen(argv[1], "w");       /*  open file with name argv[1]         */
    printf("\nNo. of arguments in Command line = %d\n\n", argc);
    for(i = 2; i < argc; i++)
        fprintf(fp, "%s", argv[i]);     /*  write to file argv[1]           */
    fclose(fp);

/*  Writing content of the file to screen                                   */

    printf("Contents of %s file\n\n", argv[1]);
    fp = fopen(argv[1], "r");
    for(i = 2; i < argc; i++)
    {
        fscanf(fp, "%s", word);
        printf("%s", word);
    }
    fclose(fp);
    printf("\n\n");

/*  Writing the arguments from memory                                       */

    for(i = 0; i < argc; i++)
        printf("%*s\n", i*5,argv[i]);
}
```

Output

```
C>F12_6 TEXT AAAAAA BBBBBB CCCCCC DDDDDD EEEEEE FFFFFF GGGGGG

No. of arguments in Command line = 9

Contents of TEXT file

AAAAAA BBBBBB CCCCCC DDDDDD EEEEEE FFFFFF GGGGGG

C:\C\F12_6.EXE
TEXT
    AAAAAA
          BBBBBB
                CCCCCC
                      DDDDDD
                            EEEEEE
                                  FFFFFF
                                        GGGGGG
```

6. What is stack and write a program to perform operations on a stack using linked list.

A stack is exactly opposite of a queue. Items in a queue exit in the same order in which they had entered the queue. Therefore, a queue is called as a First In First Out (FIFO) structure. In contrast, a stack is called as a Last In First Out (LIFO) structure, because the item that entered the stack last is the first one to get out. To understand this, imagine a pile of books. Usually, we pick up the topmost book from the pile. Similarly, the *topmost* item (i.e. the item which made the entry last) is the first one to be picked up/removed from the stack.

Stacks are extensively used in computer applications. Their most notable use is in system software (such as compilers, operating systems, etc.). For example, when one C function calls

another function, and passes some parameters, these parameters are passed using the stack. In fact, even many compilers store the local variables inside a function on the stack.

In general, writing a value to the stack is called as a *push* operation, whereas reading a value from it is called as a *pop* operation. Like in the case of queues, a pop (i.e. read) operation in the case of a stack is also destructive. That is, once an item is popped from the stack, it is no longer available.

```
/* STACK IMPLEMENTATION USING LINKED LIST */

# include<stdio.h>
# include<malloc.h>

struct link
{
    int info;
    struct link *next;
} *start;


void display(struct link *);
struct link *push(struct link *);
struct link *pop(struct link *);
int main_menu();

void display(struct link *rec)
{
    while(rec != NULL)
    {
        printf(" %d ",rec–>info);
        rec = rec–>next;
    }
}

struct link * push(struct link *rec)
{
    struct link *new_rec;
    printf("\n Input the new value for next location of the stack:");

    new_rec = (struct link *)malloc(sizeof(struct link));
    scanf("%d", &new_rec->info);
    new_rec->next = rec;
    rec = new_rec;
    return(rec);
}

struct  link * pop(struct link *rec)
{
    struct link *temp;
```

```
    if(rec == NULL)
    {
        printf("\n Stack is empty");
    }
    else
    {
        temp = rec–>next;
        free(rec);
        rec = temp;
        printf("\n After pop operation the stack is as follows:\n");
        display(rec);
        if(rec  == NULL)
            printf("\n Stack is empty");
    }
    return(rec);
}

int main_menu ()
{
    int choice;
    do
    {
        printf("\n 1. Push ");
        printf("\n 2. Pop");
        printf("\n 3. Quit");
        printf("\n Input your choice :");
        scanf("%d", &choice);
        if(choice <1 || choice >3)
            printf("\n Incorrect choice-> Please retry");
    } while(choice <1 || choice >3);

    return(choice);
}

/* main function */

void main()
{
    struct link *start ;
    int choice;
    start = NULL;
    do
    {
        choice = main_menu();
        switch(choice)
        {
        case 1:
```

```
            start = push(start);
            printf("\n After push operation stack is as follows:\n");
            display(start);
            break;
        case 2:
            start = pop(start);
            break;
        default :
            printf("\n End of program");
        }
    } while(choice != 3);
}
```

7. Circular lists are usualy set up with so called list header. What is the reason for introducing such a header? Write routines to insert and delete elements for this implementation.

```
typedef struct node
  {
    int data;
    struct node *next,*prev;
  } NODE;
typedef struct
  {
  NODE *mynode;
} LIST;
```

Addition

```
int insert(LIST *mylist,int data,int position)
{
        NODE *newnode,*current_node;
        int i=0;
        newnode = (NODE*)calloc(1,sizeof(NODE));
        if (newnode==NULL)
          return 0;
        newnode->data=data;
        if (position==1)
        {
              newnode->next=mylist->l;
              mylist->l->prev=newnode;
              mylist->l=newnode;
              return 1;
        }
      current_node=mylist->l;
      while (current_node->next!=NULL && i<position-2)
        {
              i++;
              current_node=current_node->next;
        }
      if (position==i+2)
        { if (current_node->next==NULL )
              {
```

```
                        current_node->next=newnode;
                        newnode->prev=current_node;
                    }
                  else
                    {
                        newnode->next=current_node->next;
                        newnode->prev=current_node;
                        current_node->next=newnode;
                        newnode->next->prev=newnode;
                    }
                  return 1;
            }
          else
          return 0;
}
```

Deletion

```
void delete_node(LIST *mylist,int data,int position)
{
       NODE *mytemp1,*mytemp2;
       int i=0;
       if (position==1)
         {
              mytemp1=mylist->l;
              mylist->l=mylist->l->next;
              mylist->l->prev=NULL;
              free(mytemp1);
              return 1;
         }
      mytemp2=mylist->l;
      while (mytemp2->next!=NULL && i<position-2)
         {
              i++;
              mytemp2=mytemp2->next;
         }
      if (i==position-2)
         {
              if (mytemp2->next->next==NULL)
              { mytemp1=mytemp2->next;
                   mytemp2->next=NULL;
                   free(mytemp1);
              }
             else
              {
                   mytemp1=mytemp2->next;
                   mytemp1->next->prev=mytemp2;
                   mytemp2->next=mytemp1->next;
                   free(mytemp1);
              }
       }
}
```

8. (a) Explain the algorithm for exchange sort with a suitable example.

The simplest and the oldest sorting technique is the bubble sort. However this is the most inefficient algorithm. Let us understand this method.

The method takes two elements at a time. It compares these two elements. If the first element is less than the second element, they are left undisturbed. If the first element is greater than the second element, then they are swapped. The procedure continues with the next two elements, goes and ends when all the elements are sorted. Consider the list 74, 39, 35, 97, 84.

Pass 1: (first element is compared with all other elements)—Note that the first element may change during the process.

(1) Compare 74 and 39. Since 74 > 39, they are swapped. The array now changes like 39, 74, 35, 97, 84.
(2) Compare 39 and 35. Since 39 > 35, they are swapped. The array now changes like 35, 39, 74, 97, 84.
(3) Compare 35 and 97. Since 35 < 97, they are undisturbed. The array is now 35, 39, 74, 97, 84.
(4) Compare 35 and 84. Since 35 < 84, they are undisturbed. The array is now 35, 39, 74, 97, 84.

At the end of this pass, the first element will be in the correct place. The second pass begins with the second element and is compared with all other elements. Note that the number of comparisons will be reduced by one since the first element is not compared any more.

Pass 2: (second element is compared)

(1) Compare 39 and 74. Since 39 < 74, they are undisturbed. The array is now 35, 39, 74, 97, 84.
(2) Compare 39 and 97. Since 39 < 97, they are undisturbed. The array is now 35, 39, 74, 97, 84.
(3) Compare 39 and 84. Since 39 < 84, they are undisturbed. The array is now 35, 39, 74, 97, 84.

This pass did not bring any change in the array elements. The next pass starts.

Pass 3: (third element is compared)

(1) Compare 74 and 97. Since 74 < 97, they are undisturbed. The array is now 35, 39, 74, 97, 84.
(2) Compare 74 and 84. Since 74 < 84, they are undisturbed. The array is now 35, 39, 74, 97, 84.

This pass also did not bring any change in the array elements. The next pass starts.

Pass 3: (fourth element is compared)

(1) Compare 97 and 84. Since 97 > 84, they are swapped. The array is now 35, 39, 74, 84, 97.

The sorting ends here. Note that the last number is not compared with any more numbers.

C program for bubble sort

Let us assume that an array named arr[] will hold the array to be sorted and n is the number of elements to be sorted. We will also assume that the sorting order is ascending in nature.

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

#define MAXSIZE 500

void bubblesort(int arr[], int maxsize);

int arr[n],n;

int main()
{
  int i;
  printf("\nHow many arr you want to sort: ");
  scanf("%d",&n);
  printf("\nEnter the values one by one: ");
  for (i = 0; i < n; i++)
  {
    printf ("\nEnter element %i :",i);
    scanf("%d",&arr[i]);
  }
  printf("\nArray before sorting:\n");
  for (i = 0; i < n; i++)
    printf("[%i], ",arr[i]);

  printf ("\n");
  bubblesort(arr, n);

  printf("\nArray after sorting:\n");
  for (i = 0; i < n; i++)
    printf("[%i], ", arr[i]);
}
void bubblesort(int arr[], int n)
{
  int i, j, temp;

for (i = 0; i < n - 1; i++)
  {
  for (j = i; j < n; j++)
  {
  if (arr[i] > arr[j])
    {
          temp = arr[i];
      arr[i] = arr[j];
          arr[j] = temp;
    }
  }
  }
}
```

At the end of the execution, the array arr[] will be sorted in the ascending order. If you need to perform a descending order sorting, replace the > symbol with < symbol in the comparison statement.

Efficiency of bubblesort

Let us now analyze the efficiency of the bubble sort. You can see from the algorithm that each time (each pass), the number of elements scanned for comparison reduces by 1. Thus, we have:

Number of comparisons in the first pass = (n – 1)

Number of comparisons in the second pass = (n – 2)
Number of comparisons in the last pass = 1

Thus, the total number of comparisons at the end of the algorithm would have been:

$$(n - 1) + (n - 2) + \ldots + 2 + 1 = n\,(n - 1) / 2 = n^2 / 2 + O(n) = O(n^2)$$

Hence the order of the bubble sort algorithm is $O(n^2)$.