Assignment - 4

(1) Call by value:

In call by value parameters passing method, the copy of actual parameters values are copied to formal parameters and these formal parameters are used in called function.

Ex:
```
#include<stdio.h>
void main() {
    int num1, num2;
     void swap(int, int);
       num1 = 10;
       num2 = 20;
  printf("Before swap : num1 = %d , num2 = %d", num1,
                                             num2);
  swap(num1,num2);
  printf("After swap: num1=%d , num2 =%d", num1,
                                             num2);
 void swap (int a, int b) {
    int temp;
     temp = a;
     a = b;
      b = temp;
    }
```

* call by reference:

In call by reference parameters passing method, the memory location address of the actual parameters is copied to formal parameters. This address is used to access the memory locations of the actual parameters in called function.

Ex:
```
#include<stdio.h>
void main() {
 int num1, num2;
 void swap (int*,int*);
    num1=20;
    num2= 20;
printf("Before swap: num1= %d, num2= %d", num1,
                                        num2);
swap(&num1, &num2);
 printf("After swap: num1= %d, num2= %d, num1,
                                        num2);
}
 void swap (int*a, int*b){
   int temp;
    temp = *a;
     *a = *b;
     *b = temp;
   }
```

```
2
Q). # include <stdio.h>
  void main() {
int  a[10][10], b[10][10] , c[10][10] ,k,i, j, m, n, p, q;
  printf ("rows & columns for A :\n");
   scanf ("%d%d" &m , &n);
  printf (" rows & columns for B: \n" );
   scanf ("%d%d", &p, &q);
 if  (n== p) {
  printf ("Input A -\n" );
  for (i=0; i<m; i++) {
   for (j=0; j<n; j++) {
      scanf ("%d" ,&a[i][j]);
    }
 }
  printf("Input B - \n");
  for (i=0 ; i<p; i++) {
   for( j=0 ; j<q ; j++) {
      scanf("%d", & b[i][j]);
   }
 }
printf (" resultant matrix : \n");
 for (i=0; i<m; i++) {
  for (j=0; j<q ;j++) {
     c[i][j] = 0;
     for (k=0; k<p ; k++) {
     c[i][j] += a[i][k]*b[k][j];
```

```
      }
        printf ("\n");
      }
}     for (i=0; i<m; i++) {
        for (j=0; j<q; j++) {
          printf ("%d ", c[i][j]);
        }     printf("\n");
      }
      else
        printf ("Matrices can't be multiplied!");

   }
```

Output

```
rows and columns for A:
  2  3
rows and columns for B:
  3  2
Input A:
 2  -3  4  5 3   3  5.
Input B :
3  3  5  0  -3  4
resultant matrix:
   -21    22
   159    149.
```

```
(3)  # include<stdio.h>
  int fibonacci(int num)  {
     if (num==0)
     {   return 0;
     }
    else if (num==1)  {
       return 1;
     }
     else
     {   return fibonacci(num-1) + fibonacci(num-2);
     }
 }
 int main()
 {  int  num;
   printf("enter the numbers :");
    scanf("%d", &num);
    for (int i=0; i<num;  i++)
    {
       printf("%d", fibonacci(i));
    }
      return 0;
 }
```
output:

(u) string length:

The string lenth function, strlen returns the length of a string. If string is empty it returns zero.

Syntax: int strlen (const char* string);

Ex:
```
#include <stdio.h>
# include< string.h>
int main ( )
{
    char str [1000];
    int i;
    printf ("enter the string :");
    scanf ("%s", str);
    for (i=0; str[i] !='\0"; ++i) {
        printf ("length of string : %d", i); }
        return 0;
}
```

Out put:
```
enter the string : Geeks
length of string : 5.
```

* string copy:

The string copy function strcpy, copies the contents of the string including null charater, to the string.

char* strcpy (char* tostr, const char* fromstr);

Ex:
```
#include <stdio.h>
# include <string.h>
int main ( ) {
    char str1[20] = "C programming" ;
    char str2[20] ;
    strcpy (str2 , str1);
    puts (str2);
    return 0;
    }
```

Output: C programming.

* String compare:

The string compare function strcmp, compares the two strings weather those string same or not.

int strcmp (const char* str1 , const char*str2);

Ex:
```
#include <stdio.h>
#include< string.h>
int main () {
char str1[] = "abcd", str2[]= "abCd", str3[] = "abcd"
int result ;
result = strcmp (str1 , str2);
printf ("%d\n", result);
```

Output: 0.

* String reverse:

The string reverse ~~strrove~~ function strrev, reverse the string given by the user.

Ex:
```
#include<stdio.h>
#include< string.h>
int main() {
  char s[100];
  printf("enter a string:");
  gets(s);
  strrev(s);
  printf (" reversed string :");
```

(5)
```
#include<stdio.h>
void main()
{
  char a[5][20];
  int i,j;
  printf("enter the names:");
  for(i=0; i<5; i++){
     scanf("%s", &a[i]);
  printf("sorted list of names:");
  for(i=65; i<122; i++)
   {
     for(j=0; j<5; j++)
     {
       if(a[j][0]==i)
         printf("%\n%s", a[j]);
     }
   }
 }
```

Output:

```
enter the names: Computer
                 Bees
                 Eclipse
                 Apple
                 Zoo.

sorted list of names: Apple
Bees
computer
Eclipse
Zoo.
```

(G) Function:

A function is basically a block of statements that performs a particular task.

user defined function:

user defined functions that are functions that you use to organise your code in the body of a policy.

Once you define a function, you can call it in the same way as the built-in-action and parser function

Structure of user defined function:

* Function declaration:

It is also known as a function prototype. If the function definition is mentioned before the main function.

* Function call:

Once defining a function definition, there must be a function call in the program. A function can be called n number of times.

* Function definition:

The body of function definition

consists of a set of statements to perform a specific operation or task.

Ex:
```
# include <stdio.h>
void sum(); // function declaration
void main() {
  sum(); // function call
}
void sum() // function definition
{
  int a=5, b=10, c;
  c = a+b;
  printf("addition of %d and %d is %d",
                                 a,b,c);
}
```

output:
```
addition of 5 and 10 is 15.
```

A function can be called either with arguments or without arguments. These functions may or may not return values to the calling functions.

(i) Function with no arguments and no return values:

when a function has no arguments,

it does not receive any data from the calling function.

syntax:
```
void function();
function();
void function() {
    statements;
}
```

* Function with arguments but no return value:

when function has arguments, it receives any data from the calling function but it returns no return value.

syntax:
```
void function(int);
function(x);
void function(int x) {
    statements;
}
```

* Function with no arguments but returns a value:

That function that may not take any arguments but returns a value to the calling function.

syntax: int function( );

```
function( );
int function( ) {
    statements;
    return x;
}
```

* functions with arguments and returns value:

The functions has arguments, it receives any data from the calling function and returns a value.

syntax: int function (int);

```
function (x);
int function(int x) {
    statements;
    return x;
}
```

(7)
```
#include <stdio.h>
main( )
{
  int a[10], n, i;
   float avg, sum=0;
   printf("enter array size : ");
    scanf("%d", &n);
    printf("enter array elements : ");
    for (i=0; i<n; i++) {
        scanf("%d", &a[i]);
        { sum+= a[i];
        }
    printf("sum=%d", sum);
    avg = (float) sum/n;
    printf("Average= %f",avg);

}
```

(8) Self referential structures:

Self referential structures are those structures that have one or more pointers which point to the same type of structure, as their member.

Syntax:
```
struct node {
    int data1;
    char data2;
    struct node *link;
};
```

* The point to consider is that the pointer should be initialized properly before accessing, as by default it contains garbage value
* It plays a very important role in creating other data structures.
* It reduces the complexity of the program

* By using this, we can easily implement these data structures efficiently.

Nested structure:

Nested structure is a structure within the structure. One structure can be declared inside another structure in the same way structure members are declared inside a structure.

* It can be used to represent a wide variety of data.
* These are often used in programing to store data such as the pointers on a 2D grid.
* These are often used to represent hierarchical data, such as the contents of the file system.
* Nested structure can be used to create complex data types that are easy to understand & use.

(9) Dynamic memory allocation enables the programmer to allocate memory at run-time.

Malloc( ):

malloc stands for memory allocation.

The function reverse a block of memory of the specified number of bytes.

Syntax:

ptr = (cast-type*) malloc (byte size)

* callo c()

calloc() stands for "contiguous allocation" method in C is used to dynamically allocate the specified no. of blocks of memory of the specified type.

Syntax:

ptr = (cast-type*) calloc (n, element-size);

If space is insufficient, allocation fails and returns a null pointer

* realloc()

realloc() stands for re-allocation. If the dynamically allocated memory is insufficient or more than required, you can change the size of previous allocated memory using this function.

syntax:

ptr = realloc(ptr, x);

Ex:
```
#include <stdio.h>
#include<stdlib.h>
int main()
{
   int * ptr ;
   int n,i;
    n=5;
   printf("enter no.of elements:%d",n);
   ptr=(int*) calloc(n,sizeof(int));
   if (ptr==NULL) {
      printf("memory not allocated");
       exit(0);
   }
   else {
   printf("memory is allocated");
    for (i=0; i<n; i++) {
          ptr[i]= i+1
    }
    printf("elements of array are:\n");
      for (i=0; i<n; i++) {
          printf("%d", ptr[i]);
      }
     n=10;
     printf("enter new size of array:")%d",n);
      ptr= realloc(ptr,n*sizeof(int));
     printf("memory is reallocated");
    for (i=5; i<n; ++i) {
        ptr[i]= i+1 ;
    }
  printf("elements of array are:");
    for(i=0; i<n; ++i) {
        printf("%d", ptr[i]);
    }
    free(ptr);
  }
  return 0;
}
```

Output:

```
enter no.of elements :5
Memory is allocated.
elements of array are: 1,2,3,4,5
enter new size of array: 10
memory is reallocated.
The elements of array are: 1,2,3,4,5,6,7,8,9,10
```

(10) storage classes:

These are used to describe the features of a variable. These features basically include the scope, visibility and life-time which helps us to trace the existance of a particular variable during the runtime of a program.

C uses 4 storage classes:

* auto:
This is the default storage for all the variables declared inside a function or a block. Hence, the keyword auto is rarely used while writing programs. Auto variables can be only accessed within the block they have been declared and not outside them.

* Extern:
This class simply tells us that the variable is defined else where and not within the same block where it is used. Basically, the value is assigned to it in a different block and this can be changed in a different block as well.

* static:
It is used to declare static variable which are popularly used while writing programs in C. These have the property of preserving their value even they are out of scope.

* Register:

This class declares register variables that have the same functionality as that of the auto variables. The only difference is that the ~~com~~ compiler tries to store these variables in the register of the microprocessor if a free registration is available

(12) Command line argument:

These are given after the name of the program in command - line shell of operating systems. Command line arguments are passed to the main() method.

Syntax:

```
int main (int argc, char *argv[])
```

argc counts the number of arguments on the command line and argv[] is a pointer array, which holds pointers of the type char which points to the arguments.

\#Ex:

```
#include<stdio.h>
int main(int argc, char* argv[]) {
    int i;
    if (argc >= 2)
{
 printf("the arguments are:");
 for(i=1; i<argc; i++) {
     printf("%s\t", argv[i]);
 }
}
else {
    printf("argument list is empty.");
}
 return 0;
}
```

(13) Output:

```
Argument list is empty.
```

(13) Pointer:

It is a variable that stores the memory address of another as its value. Pointer is ~~wit~~ created with the * operator.

There are few operatorsions that are allowed to perform on pointers.

* Increment | Decrement:

→ Increment:

when pointer is incremented, it actually increments by the number equals to the size of the datatype for which it is a pointer.

→ Decrement:

when pointer is decremented, it actually decrements by the number equal to the size of the data type for which it is a pointer.

Ex:
```
#include<stdio.h>
  int main () {
     int  a = 22;
     int *p = a;
     printf ("p= %u", p);
     p++;
     printf ("p++ = %u \n"}, p);
     p--;
     printf("p-- = %u\n", p);
     }
```

output:
```
p= 1441900792
p++ = 1441900796
p-- = 1441900792.
```

* Addition:

when a pointer is added with a value, the first multiplied by the size of datatype

then added to pointer.
and
Ex:
```
# include <stdio.h>
int main() {
  int n=4;
  int *ptr1, *ptr2;
  ptr1 = &n;
  ptr2 = &n;
  ptr2 = ptr2+3;
  printf(" Pointer ptr2 : %p", ptr2);
}
```
output:
```
Pointer ptr2 = 0x7ffca373daa8
```

* subtraction:

When a pointer is subrateed with a value, the value first is multiplied by the size of the data type and then subtracted from pointer.

Ex:
```
#include<stdio.h>
int main() {
   int n=4;
   int *ptr1, *ptr2;
   ptr1 = &n;
   ptr2 = &n;
   ptr2 = ptr2 - 3;
   printf ("Pointer ptr2 : %p", ptr2);
   return 0;
}
```

Output: Pointer ptr2 : 0x7ffd718ffeb0

(14) File:

It is collection of data stored in the secondary memory. It is used to storing informations that can be processed by programs. File mode is categorized into 4 type of modes.

* Create mode:

Opens the specified file and positions it to the beginning. To create a new file, open the file in create mode, user can't read, position a file opened with create mode.

* Read mode:

This mode opens a file for the reading of data. Read mode opens a file to the beginning.

* Update mode:

This mode allows both reading & writing of data. Uptade mode ~~file~~ opens a file to the beginning.

* Append mode:

This allows writing data to the end of

a file. User can't read, position or rewind a file opened ~~to~~ with append mode.

(15)
```
#include<stdio.h>
 #include<stdlib.h>
int main()
{
  FILE *fptr1, *fptr2;
  char filename[100], c;
  printf("enter filename to open :\n");
     scanf("%s", filename);
     fptr1= fopen (filename,"r");
      if(fptr1== NULL) {
        printf("can't open file %s\n",filename);
        exit(0);
      }
     printf("enter file name to open for
           writing \n");
     scanf("%s", file name);
      fptr2= fopen (filename,"w");
      if(fptr2 == NULL) {
        printf("can't open file %s\n", file name);
        exit(0);
      }
     c= fgetc(fptr1);
     while (c!= EOF) {
      fputc(c, fptr2);
```

```
        c = fgetc(fptr1);
    }
    printf("\n contentas copied to %s",
                            filename);
        fcolose (fptr1);
        fclose (fptr2);
        return 0;
    }
```

output:

```
enter filename to open:
a.txt
enter filename to open for writing
b.txt
contents copied to b.txt:
```

(16) 'fscanf()':

This function is used to read formatted input form a file. It works similarly to the scanf() function, but it takes an additional file pointer as the first argument.

```
FILE*fp;
int i;
char str[100];
float f;
fp = fopen("data.txt,"r");
fscanf (fp"%d %s %f", &i, str, &f);
printf("Read = %d %s%f", i, str, f);
fclose (fp);
```

fprintf():

This function is used to write formatted output to a file. It works similarly to the printf() function. but it takes an additional file pointers as the first argument.

```
FILE *fp;
int i=42;
char str[]="Hello world";
float f=3.14;
fp=fopen("data.txt","w");
fprintf("fp,"%d%.4s%f", i, str, f);
fclose(fp)
```

'fgets()':

This function is used to read a line of text from a file. It takes a file pointer, a buffer to store the read text and the max number of characters to read as arguments.

```
FILE *fp;
char line[100];
fp = fopen("data.txt","r");
fgets(line, sizeof(line)},fp);
printf("Read: %s", line);
fclose(fp);
```

'fwrite()':

This function is used to write binary data to a file. It takes a pointer to the data, the size of each element, the number of elements and a file pointer as arguments.

```
FILE *fp;
int data[] = {1,2,3,4,5};
fp = fopen("data.bin", "wb");
fwrite(data, sizeof(int), sizeof(data)/sizeof(int),
                                          fp);

fclose(fp);
```

(4)
```
#include<stdio.h>
int main()
{
  FILE *source, *target;
  source = fopen("source.txt","r")
  if (source == NULL) {
    printf("could not open source file.\n");
    return 1;
  }
  target = fopen("target.txt","w");
  if (target == NULL) {
    printf("could not open target file.\n");
    fclose(source);
    return1;
  }
    char ch;
    while((ch=fgetc(source))!=EOF)
    {
      fputc(ch,target);
    }
printf("File copied successfully");

fclose(source);
fclose(target);
return 0;
}.
```

(18)

fopen:

This function is used to open a file. It takes the name of the file and the mode in which the file should be opened as arguments.

syntax:

```
FILE *fopen(const char* filename, const char*mode);
```

Ex:

```
FILE *fp;
fp = fopen("example.txt","r");
```

fclose:

This function is used to close an open file. It takes a file pointer as an argument.

syntax:

```
int fclose(FILE* fp);
```

Ex:

```
fclose(fp);
```

fgetc:

This function is used to read a single character from a file. It takes file pointer as argument and return the character read as int.

Syntax:
```
    int fgetc(FILE* fp);
```

Ex:
```
    int ch;
    ch = fgetc(fp);
```

fputc: This function is used to write a single character to a file. It takes an int and a file pointer as arguments.

Syntax:
```
    int fputc(int c, FILE* fp);
```

Ex:
```
    fputc('A', fp);
```

fread:
This function is used to read binary data from a file. It takes a pointer to the buffer.

Syntax:
```
    size_t fread(void *ptr, size_t size, size_t count,
                                        FILE *fp);
```

Ex:
```
    int data[100];

    fread(data, size of (int), 100, fp);
```

fwrite:
This function is used to write binary data to a file. It takes a pointer to the

data of a file.

syntax:

```
size_t fwrite(const void* ptr, size_t size, size_t count,
                                     FILE* fp);
```

Ex:

```
int data[100] = {1,2,3,4,5}
fwrite (data, sizeof(int), 100, fp);
```

fprintf:

This function is used to write formatted output to a file

syntax:

```
int fprintf(FILE* fp, const char * format, ...);
```

Ex:

```
FILE *fp;
int i = 42;
float f = 3.14;
char str[] = "Helloworld";
fp = fopen("example.txt", "w");
fprintf(fp, " Integers: %d, float: %.f, string: %s", i, f,
                                              str);
fclose(fp);
```

(ii)

```
#include<stdio.h>
#include<string.h>
#define MAX_BOOKS 10
```

```
struct book {
int access-no;
char author [50];
char title [100];
int year;
float price;
};
 int main() {
struct book library [MAX-BOOKS];
int i, n;
printf("enter the no. of books:");
scanf("%d", &n);
for (i=0; i<n; i++) {
 printf("enter details %d :\n", i+1);
 printf("Access number : ");
 scanf("%d", &library [i].access-no ~~author~~);
 printf("Authors : ");
 scanf ("%s", library [i].author);
 printf (" title :");
 scanf ("%s", library [i]. title);
 printf("Year of publication:");
 scanf("%d", &library[i]. year);
 printf("price:");
 scanf ("%f", &library [i], price);
}
 printf("\n library catalogue :\n");
for (i=0; i<n ; i++) {
printf ("Access number : %d\n", library [i].access_no);
printf ("Authors : %s \n", library [i]. author);
printf ("Title : %s\n", library[i]. title);
printf ("Year of publication : %d\n", library[i].year);
printf (" Price : %.2f \n", library[i]. price);

} return 0;

}.
```