Graphic Era
HILL UNIVERSITY
Established by an Act of the State Legislature of Uttarakhand (Adhiniyam Sankhya 12 of 2011)

Programming for problem solving

TCS 201

Theory Notes –Unit 1 to Unit 3
Subject Code-TCS201
Last updated on- 06-Macrh-19

UNIT 1	**STRINGS, STRING & POINTERS**
UNIT 2	**POINTERS , STRUCTURES, UNIONS**
UNIT 3	**FILES**

UNIT 1 PART A-STRINGS

The way a group of integers can be stored in an integer array, similarly a group of characters can be stored in a character array. Character arrays are often called 'strings'. Character arrays or strings are the data types used by programming languages to manipulate text such as words and sentences. A string is a one-dimensional array of characters terminated by a null ('\0').

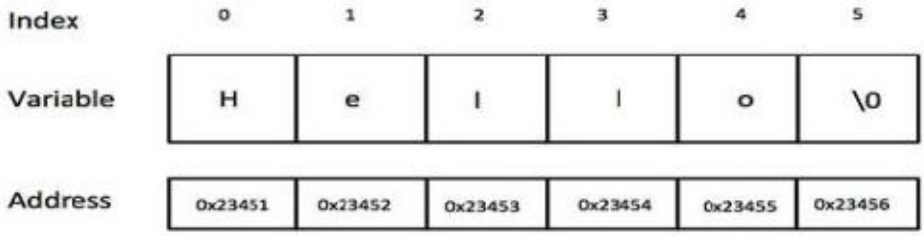
The following declaration and initialization create a string consisting of the word "Hello". To hold the null character at the end of the array, the size of the character array containing the string is one more than the number of characters in the word "Hello."

char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};

If you follow the rule of array initialization then you can write the above statement as follows:

char greeting[] = "Hello";

Following is the memory presentation of the above defined string in C/C++



Actually, you do not place the null character at the end of a string constant. The C compiler automatically places the '\0' at the end of the string when it initializes the array. Let us try to print the above mentioned string –

```
#include<stdio.h>
int main () {
char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
printf("Greeting message: %s\n", greeting );
return 0; }
```

When the above code is compiled and executed, it produces the following result –

Greeting message: Hello

C supports a wide range of functions that manipulate null-terminated strings –

Sr.No.	Function & Purpose
1	**strcpy(s1, s2);** **Copies string s2 into string s1.**
2	**strcat(s1, s2);** **Concatenates string s2 onto the end of string s1.**
3	**strlen(s1);** **Returns the length of string s1.**
4	**strcmp(s1, s2);** **Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2.**
5	**strchr(s1, ch);** **Returns a pointer to the first occurrence of character ch in string s1.**
6	**strstr(s1, s2);** **Returns a pointer to the first occurrence of string s2 in string s1.**

UNIT-2 PART A-POINTERS

Address operator, Indirection operator, Pointer arithmetic, Scale factor, multiple indirection

What are Pointers?

A pointer is a variable whose value is the address of another variable, i.e., direct address of the memory location. Like any variable or constant, you must declare a pointer before using it to store any variable address. The general form of a pointer variable declaration is −

type *var_name;

Here, type is the pointer's base type; it must be a valid C data type and var_name is the name of the pointer variable. The asterisk * used to declare a pointer is the same asterisk used for multiplication. However, in this statement the asterisk is being used to designate a variable as a pointer. Take a look at some of the valid pointer declarations −

```
int    *ip;    /* pointer to an integer */
float  *fp;    /* pointer to a float */
char   *ch     /* pointer to a character */
```

Indirection and Address-of Operators

The indirection operator (*) accesses a value indirectly, through a pointer. The operand must be a pointer value. The result of the operation is the value addressed by the operand; that is, the value at the address to which its operand points. The type of the result is the type that the operand addresses.

The address-of operator (&) gives the address of its operand.

As we know, every variable is a memory location and every memory location has its address defined which can be accessed using ampersand (&) operator, which denotes an address in memory. Consider the following example, which prints the address of the variables defined −

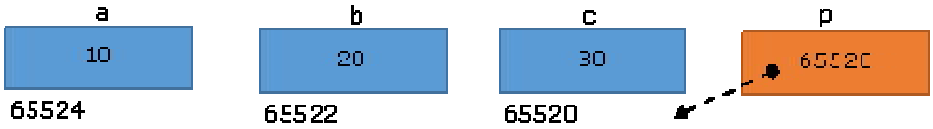
```
#include <stdio.h>
int main () {
   int  var1;
   char var2[10];
   printf("Address of var1 variable: %x\n", &var1  );
   printf("Address of var2 variable: %x\n", &var2  );
   return 0;
}
```

```
OUTPUT:
Address of var1 variable: bff5a400
Address of var2 variable: bff5a3f6
```

Pointer arithmetic – Scale factor

In case of primitive types adding 1 to the variable results incrementing of its value by 1 and subtracting 1 to the variable results decrementing of its value by 1.

Since pointer is not a normal variable adding 1 to the pointer results next variable's address. Subtracting 1 to the pointer results previous variables address.




```
short a=10,b=20,c=30;
short *p=&c;
```

Here p is the pointer to the variable c, holds the address of c 65520. Now p+1 gives the address of next variable b 65522 and p+2 gives the address of next variable a 65524.

```
#include<stdio.h>
int main()
{
        short a=10,b=20,c=30;
```

```
        short *p=&c;
        printf("%u ",p);
        printf("%u ",p+1);
        printf("%u ",p+2);
        return 0;
    }
```

Output: 65520 65522 65524

Scale factor

If we conclude the above observation, short pointer increments by 2 on adding 1. In same manner float pointer increments by 4 on adding 1 and char pointer increments by 1 on adding 1 that is pointer is incrementing by its **type size** on adding 1

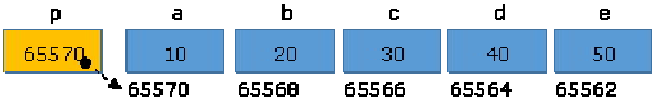
Scale factor is the quantity of increment in the value of a pointer on adding 1.

Arithmetic operations on pointers:

- Operations like +, -, ++ and –- can be performed on pointers
- Can't perform *, / operations
- Arithmetic operations on pointers can only performed with integers but, not with any other types

Let us take a series of short variables and assign the address of first variable to the pointer. Now the pointer is capable to access all the variables using pointer arithmetic

```
short a=10,b=20,c=30,d=40,e=50;
short *p=&a;
```




```
*(p+0)      Value at 65570      10
*(p-1)      Value at 65568      20
*(p-2)      Value at 65566      30
*(p-3)      Value at 65564      40
*(p-4)      Value at 65566      50

            (Or)

for(i=0;i<5;i++)
        *(p-i);
```

Here all the short type of variables are declared in a sequence. The memory allocation of which are allocated in consecutive memory allocations like an array

"p" is the pointer to the first variable.

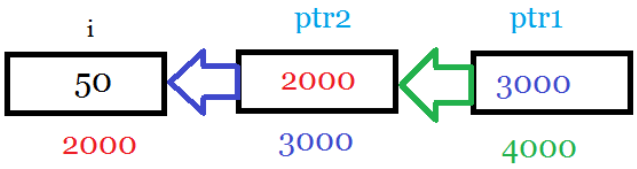
Because scale factor of short type is 2 by decrementing the value of "p" by 1 we can access all the variables one by one.

Multiple indirections:

```
void main()
{
  int i =50;
  int **ptr1; int *ptr2;
  ptr2 = &i;ptr1 = &ptr2;
  printf("\nThe value of **ptr1 : %d",**ptr1);
  printf("\nThe value of *ptr2  : %d",*ptr2);
}
```

The value of **ptr1 : 50

The value of *ptr2 : 50



*ptr1 will tell content at address 3000, so we'll get 2000 address, but using **ptr1 we can get the value of I i.e. 50, this time we are de-referencing twice.

If we want to store address of ptr1 then we require ***ptr pointer variable.

Advantages and disadvantages of pointers in c

Benefits (use) of pointers in c:

- Pointers provide direct access to memory.
- Pointers provide a way to return more than one value to the functions.
- Reduces the storage space and complexity of the program.
- Reduces the execution time of the program.
- Provides an alternate way to access array elements.
- Addresses of objects can be extracted using pointers

Drawbacks of pointers in c:

- Uninitialized pointers might cause segmentation fault.
- Pointers are slower than normal variables.
- If pointers are updated with incorrect values, it might lead to memory corruption.

Dynamic Memory Allocation

As we know, we have to declare the size of an array before we use it. Hence, the array you declared may be insufficient or more than required to hold data. To solve this issue, we can allocate memory dynamically.
Dynamic memory management refers to manual memory management. This allows us to obtain more memory when required and release it when not necessary.
Although C inherently does not have any technique to allocate memory dynamically, there are 4 library functions defined under <stdlib.h> for dynamic memory allocation.

C malloc()

The name malloc stands for "memory allocation".
The function malloc() reserves a block of memory of specified size and return a pointer of type void which can be casted into pointer of any form.

Syntax of malloc()

ptr = (cast-type*) malloc(byte-size)
Here, ptr is pointer of cast-type. The malloc() function returns a pointer to an area of memory with size of byte size. If the space is insufficient, allocation fails and returns NULL pointer.
ptr = (int*) malloc(100 * sizeof(int));
This statement will allocate either 200 or 400 according to size of int 2 or 4 bytes respectively and the pointer points to the address of first byte of memory.

C calloc()

The name calloc stands for "contiguous allocation".
The only difference between malloc() and calloc() is that, malloc() allocates single block of memory whereas calloc() allocates multiple blocks of memory each of same size and s**ets all bytes to zero.**

Syntax of calloc()

ptr = (cast-type*)calloc(n, element-size);
This statement will allocate contiguous space in memory for an array of n elements. For example:
ptr = (float*) calloc(25, sizeof(float));
This statement allocates contiguous space in memory for an array of 25 elements each of size of float, i.e, 4 bytes.

C free()

Dynamically allocated memory created with either calloc() or malloc() doesn't get freed on its own. You must explicitly use free() to release the space.

syntax of free()

free(ptr);

This statement frees the space allocated in the memory pointed by ptr.

C realloc()

If the previously allocated memory is insufficient or more than required, you can change the previously allocated memory size using realloc().

Syntax of realloc()

ptr = realloc(ptr, newsize);

Here, ptr is reallocated with size of newsize.

What is Memory Leak? How can we avoid?

Memory leak occurs when programmers create a memory in heap and forget to delete it.

```
/* Function with memory leak */
#include <stdlib.h>
 void f()
{
  int *ptr = (int *) malloc(sizeof(int));
  /* Do some work */
  return; /* Return without freeing ptr*/
}
```

To avoid memory leaks, memory allocated on heap should always be freed when no longer needed.

```
/* Function without memory leak */
#include <stdlib.h>;
 void f()
{
  int *ptr = (int *) malloc(sizeof(int));
  /* Do some work */
  free(ptr);
  return;
}
```

Dangling pointer

A pointer pointing to a memory location that has been deleted (or freed) is called dangling pointer. There are **three** different ways where Pointer acts as dangling pointer

```
// Deallocating a memory pointed by ptr causes
// dangling pointer
#include <stdlib.h>
#include <stdio.h>
int main()
{
   int *ptr = (int *)malloc(sizeof(int));

   // After below free call, ptr becomes a dangling pointer
   free(ptr);

   // No more a dangling pointer
   ptr = NULL;
}
```

NULL Pointer

NULL Pointer is a pointer which is pointing to nothing. In case, if we don't have address to be assigned to a pointer, then we can simply use NULL.

```
#include <stdio.h>
int main()
{
   // Null Pointer
   int *ptr = NULL;
       printf("The value of ptr is %u", ptr);
   return 0;
}
```

void pointer in C

A void pointer is a pointer that has no associated data type with it. A void pointer can hold address of any type and can be typcasted to any type.

UNIT 2 PART B-STRUCTURES, UNIONS

What is a Structure?

Structure is the collection of variables of different types under a single name for better handling. For example: You want to store the information about person about his/her name, citizenship number and salary. You can create this information separately but, better approach will be collection of this information under single name because all these information are related to person.

Syntax of structure	Example	Structure variable declaration	Another way of creating structure variable is:

struct structure_name
{
data_type member1;
data_type member2;
.
.
data_type memeber;
};
	struct person
	{
	char name[50];
	int cit_no;
	float salary;
	};
		struct person
		{
		char name[50];
		int cit_no;
		float salary;
		};
		Inside main function:
		struct person p1, p2, p[20];
			struct person
			{
			char name[50];
			int cit_no;
			float salary;
			}p1 ,p2 ,p[20];

Different ways of initialising structure variables
struct info
{
int sno;
char name[20];
char add[20];
}stu2,stu3;
	struct info stu1={1,"ajay","dehradun"};
	
	stu2.sno=2;
	strcpy(stu2.name,"rashmi");
	strcpy(stu2.add,"delhi");
	
	scanf("%d%s%s",&stu3.sno,stu3.name,stu3.add);

What is Union?

Unions are quite similar to the structures in C. Union is also a derived type as structure. Union can be defined in same manner as structures just the keyword used in defining union in union where keyword used in defining structure was struct.

```
union car
{
  char name[50];
  int price;
};
```

Difference between union and structure

SN	C Structure	C Union
1	Structure allocates storage space for all its members separately.	Union allocates one common storage space for all its members. Union finds that which of its member needs high storage space over other members and allocates that much space
2	Structure occupies higher memory space.	Union occupies lower memory space over structure.
3	We can access all members of structure at a time.	We can access only one member of union at a time.

4	struct student
	{
	int mark;
	char name[6];
	double average;
	};
		union student
		{
		int mark;
		char name[6];
		double average;
		};

5	For above structure, memory allocation will be like below. int mark – 2B char name[6] – 6B double average – 8B Total memory allocation = 2+6+8 = 16 Bytes	For above union, only 8 bytes of memory will be allocated since double data type will occupy maximum space of memory over other data types. Total memory allocation = 8 Bytes

Typedef

typedef is a keyword used in C language to assign alternative names to existing types. Its mostly used with user defined data types, when names of data types get slightly complicated. Following is the general syntax for using typedef,

typedef *existing_name* *alias_name;*

Example: typedef unsigned long ulong;

The above statement defines a term ulong for an unsigned long type. Now this ulong identifier can be used to define unsigned long type variables. ulong i, j ;

Application of typedef

typedef can be used to give a name to user defined data type as well. Let's see its use with structures.

typedef struct
{
type member1;
type member2;
type member3;
} type_name ;
	typedef struct employee
	{
	char name[50];
	int salary;
	} emp ;
		int main( )
		{**emp e1;**
		printf("\nEnter Employee record\n");
		printf("\nEmployee name\t");
		scanf("%s",e1.name);
		printf("\nEnter Employee salary \t");
		scanf("%d",&e1.salary);
		printf("\nstudent name is %s",e1.name);
		printf("\nroll is %d",e1.salary);
		return 0;
		}

type_name t1, t2 ;	emp e1, e2;	

Bit Fields in C

Bit Field Declaration

The declaration of a bit-field has the following form inside a structure −

struct structure_name{ type [member_name] : width ; };

In C, we can specify size (in bits) of structure and union members. The idea is to use memory efficiently when we know that the value of a field or group of fields will never exceed a limit or is within a small range.For example, consider the following declaration of date without use of bit fields.

```
#include <stdio.h>
// A simple representation of date
struct date
{
```

```
  unsigned int d;
  unsigned int m;
  unsigned int y;
};

int main()
{    struct date dt = {31, 12, 2014};
  printf("Size of date is %d bytes", sizeof(struct date));
  printf("    Date is %d/%d/%d", dt.d, dt.m, dt.y);
}
```

Output:
Size of date is 12 bytes Date is 31/12/2014

The above representation of 'date' takes 12 bytes on a compiler where an unsigned int takes 4 bytes. Since we know that the value of d is always from 1 to 31, value of m is from 1 to 12, we can optimize the space using bit fields.

```
#include <stdio.h>
// A space optimized representation of date
struct date
{
  unsigned int d: 5;    // d has value between 1 and 31, so 5 bits are sufficient
  unsigned int m: 4; // m has value between 1 and 12, so 4 bits are sufficient
  unsigned int y :14; // y has value between 1 and 9999, so 14 bits are sufficient
};

int main()
{
  printf("Size of date is %d bytes\n", sizeof(struct date));
  struct date dt = {31, 12, 2014};
  printf("Date is %d/%d/%d", dt.d, dt.m, dt.y);
  return 0;
}
```

Output:
Size of date is 4 bytes Date is 31/12/2014

Enum: User defined Data Type in C

An enumeration is a user-defined data type consists of integral constants and each integral constant is give a name. Keyword enum is used to define enumerated data type.

enum enum_name {value1, value2,.... Value n};

Here, enum_name is the name of enumerated data type or tag. And value1, value2,....,valueN are values of type enum_name.

By default, value1 will be equal to 0, value2 will be 1 and so on but, the programmer can change the default value.

Example:

```
enum month {JAN,FEB,MAR,APR,MAY,JUN,JUL,AUG,SEP,OCT, NOV,DEC};
enum month var_month;
```

1. First Line Creates **"User Defined Data Type"** called **month**.
2. It has 12 values as given in the **pair of braces**.
3. In the second line "var_month" variable is declared of type "month" which can be initialized with any "data value amongst 12 values".

```
int main()
{
int i;
```

```
enum month {JAN,FEB,MAR,APR,MAY,JUN,JUL,AUG,SEP,OCT,DEC};
clrscr();
  for(i=JAN;i<=DEC;i++)
     printf("%d ",i);
return 0;
}
```

Output-0 1 2 3 4 5 6 7 8 9 10 11

```
int main()
{    enum month {JAN,FEB,MAR,APR,MAY=7,JUN,JUL,AUG=35,SEP,OCT,DEC};
     printf("%d ",APR);
     printf("%d ",MAY);
     printf("%d ",JUN);
     printf("%d ",SEP);
return 0;
}
```

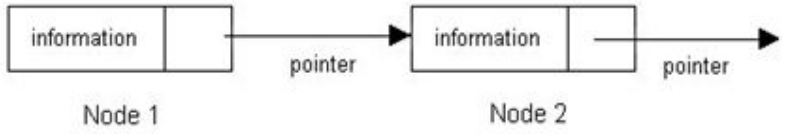
Output-3 7 8 36

Self Referential Data Structure in C - create a singly linked list

A self referential data structure is essentially a structure definition which includes at least one member that is a pointer to the structure of its own kind. A chain of such structures can thus be expressed as follows.

```
struct name {
                member 1;
                member 2;
                . . .
                struct name *pointer;
};
```

The figure is a simplified illustration of nodes that collectively form a chain of structures or linked list.



Unlike a *static data structure* such as array where the number of elements that can be inserted in the array is limited by the size of the array, a self-referential structure can dynamically be expanded or contracted.

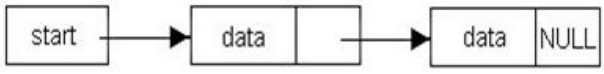
Linear (Singly) Linked List

A linear linked list is a chain of structures where each node points to the next node to create a list. To keep track of the starting node's address a dedicated pointer (referred as *start pointer*) is used. The end of the list is indicated by a *NULL*pointer. In order to create a linked list of integers, we define each of its element (referred as *node*) using the following declaration.

```
struct node_type {
        int data;
        struct node_type *next;
};
struct node_type *start = NULL;
```

Note: The second member points to a node of same type.

A linear linked list illustration:



UNIT 3-FILES

In C programming, file is a place on disk where a group of related data is stored.

Why files are needed?

When the program is terminated, the entire data is lost in C programming. If you want to keep large volume of data, it is time consuming to enter the entire data. But, if file is created, this information can be accessed using few commands.

There are large numbers of functions to handle file I/O in C language. High level file I/O functions can be categorized as:

1. Text file-A text file can be a stream of characters that a computer can process. A text file is usually opened for only one kind of operation (reading, writing, or appending) at any given time. Text files only process characters; they can only read or write data one character at a time. (In C Programming Language, Functions are provided that deal with lines of text, but these still essentially process data one character at a time.)
2. Binary file-A binary file is no different to a text file. It is a collection of bytes. In C Programming Language a byte and a character are equivalent. There are two essential differences:
   - No special processing of the data occurs and each byte of data is transferred to or from the disk unprocessed.
   - C Programming Language places no constructs on the file, and it may be read from, or written to, in any manner chosen by the programmer.

File Operations

1. Creating a new file
2. Opening an existing file
3. Reading from and writing information to a file
4. Closing a file

Standard I/O functions

Reading/Writing characters/integers/strings from/to file

getc( ) and putc( )

getc is a macro that gets one character from a file, putc is a macro that write a character to a file.

```
char ch;                    putc(ch,fp1)
ch=getc(fp1);
```

fgetc( ), fputc( ) (same as getc, putc)

fgetc gets a character from a file. fputc writes a character to a file

```
char ch;                    fputc(ch,fp1)
ch=fgetc(fp1);
```

getw( ), putw( )

getw gets an integer from file. putw writes an integer on a file.

```
int ch;                     putw(ch,fp1)
ch=getw(fp1);
```

Because EOF is a legitimate value for getw to return, use feof to detect end-of-file or ferror to detect error.

fgets( ), fputs( )

fgets gets a string from a file. fputs writes s a string to a file.

```
Char str[50];               fputs(str,fp1)
fgets(str, 50,fp1);
```

fgets reads characters from stream into the string s. It stops when it reads either n-1 Characters or a newline character, whichever comes first. On end-of-file or error, fgets returns null. On error, fputs returns EOF.

Reading/Writing characters/strings from/to console(stdin/stdout)

getchar( ), putchar ( )

getchar is a macro that gets a character from stdin(user).

putchar is a macro that outputs a character on stdout(user).

```
char ch;                    putchar(ch)
ch=getchar()
```

fgetchar( ), fputchar( )

fgetchar gets a character from stdin
fputchar outputs a character to stdout
char ch; fputchar(ch)
ch=fgetchar()

getch and getche
getch gets a character from console but does not echo (print) to the screen
getche gets a character from console, and echoes(print) to the screen
char ch;
ch= getch();
ch= getche();
Both functions return the character read from the keyboard.

gets, puts
gets gets a string from stdin
puts outputs a string to stdout (and appends a newline character)
char str[50]; puts(str)
gets(str)

Working with file

While working with file, you need to declare a pointer of type FILE. This declaration is needed for communication between file and program.
FILE *ptr;

Opening a file fopen()

The fopen() function is used to open a file and associates an I/O stream with it. This function takes two arguments. The first argument is a **pointer to a string** containing name of the file to be opened while the second argument is the **mode** in which the file is to be opened. The mode can be:

- 'r' : Open text file for reading.
- 'r+' : Open for reading and writing
- 'w' : Truncate file to zero length or create text file for writing
- 'w+': Open for reading and writing. The file is created if it does not exist, otherwise it is truncated.
- 'a' : Open for appending (writing at end of file). The file is created if it does not exist. The stream is positioned at the end of the file.
- 'a+': Open for reading and appending (writing at end of file). The file is created if it does not exist. The initial file position for reading is at the beginning of the file, but output is always appended to the end of the file.

The fopen() function returns a FILE stream pointer on success while it returns NULL in case of a failure.

Closing a file fclose()

The C library function **int fclose(FILE *stream)** closes the stream. All buffers are flushed.
Following is the declaration for fclose() function:

int fclose(FILE *stream)

Where **stream** − this is the pointer to a FILE object that specifies the stream to be closed.
This method returns zero if the stream is successfully closed. On failure, EOF is returned.

End of file Function feof()

The C library function **int feof(FILE *stream)** tests the end-of-file indicator for the given stream.

```
val=getw(fp1);
while(!feof(fp1))  //this will check for real end of file
{
   printf("\n%d",val);
```

```
        val=getw(fp1);
    }
```

Note: while(val!=EOF) //this will stop reading as it get val=-1 even before real EOF

ferror()

The C library function **int ferror(FILE *stream)** tests the error indicator for the given stream.
The ferror function returns a nonzero value if the error indicator is set. Otherwise, it returns zero.

```
fp = fopen("file.txt", "w");
  c = fgetc(fp);
  if( ferror(fp) )
  {
    printf("Error in reading from file : file.txt\n");
  }
```

Note: feof() and **ferror()** return non-zero (true) if the file has reached EOF or there has been an error, respectively.

Some more I/O functions for reading /writing

fgets() and fputs()

These are useful for reading and writing entire lines of data to/from a file.

```
fgets(buf, MAX, fp);
```

Here, buf is the name of a char array, MAX is the maximum size of the string, and fp is the pointer-to-FILE. Like gets(), fgets() returns the value NULL when it encounters EOF.

```
fputs(buf, fp);
```

char srt[50];

while(**fgets(str,50,fp1**)!=NULL)//reading from file fp1
{
puts(str); //printing to console (user)
}
	while(gets(str)!=NULL) //reading from user
	{
	fputs(str,fp1);// writing to fp1
	fputs("\n",fp1);
	}

More details about fgets and fputs:

fgets reads characters from stream into the string s. It stops when it reads either n – 1 characters or a newline character, whichever comes first.

fgets retains the newline character at the end of s and appends a null byte to s to mark the end of the string.

fputs copies the null-terminated string s to the given output stream. It does not append a newline character, and the terminating null character is not copied.

Return Value:

On success,
- fgets returns the string pointed to by s.
- fputs returns the last character written.

On end-of-file or error, fgets returns null.
On error, fputs returns EOF.

fprintf() and f scanf()

These perform the same function as *printf* and *scanf*, but work on files. Consider,
fscanf() - This function is used to read data from a file.
Syntax: fscanf(filepointer, "format specifier", list);
Ex: fscanf(fp, "%d%d%d", &a,&b,&c);

fprintf() - this function is used to write data in to a file.
Syntax: fprintf(fileprinter, "format specifier", list);
Ex: fprintf(fp, "%d\t%d\t%d\t", a, b, c);

```
printf("enter the name and age\n");
scanf("%s%d",s1.name,&s1.age);
fprintf(fp1,"%s %d\n",s1.name,s1.age);
```

```
while((fscanf(fp1,"%s%d",s1.name,&s1.age))!=EOF)
        {
           printf("name=%s age=%d\n",s1.name,s1.age);
        }
```

fread() and fwrite()
fread and fwrite are used for block reading and block writing.

size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);

fread() reads *nmemb* objects each of *size* bytes and places them into the array pointed by *ptr*
on success fread() returns the number of bytes successfully read.

size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);

```
typedef struct student
{
        char name[20];
        int age;
}stu;
stu s1[10];
```

```
for(i=0;i<n;i++)
{
        printf("enter the name and age\n");
        scanf("%s%d",s1[i].name,&s1[i].age);
}
fwrite(s1,sizeof(stu),n,fp1);
//or    fwrite(s1,sizeof(s1),1,fp1);
```

```
//reading all n records at once from file
fread(s1, sizeof(stu), n, fp1);

//reading 1 record at a time from file
while(fread(&s2, sizeof(s2), 1, fp1)==1)
{
        printf("\n%s ,%d",s2.name,s2.age);
}
```

Random access to files

There are various file handling functions that are useful for reading and writing data sequentially. To access a particular part of file and not reading the other parts, we can use some random access function available in the I/O library:

ftell ,rewind, and Fseek

ftell()
n=ftell(fp) //Here fp is file pointer and n is a variable of type long.
fp takes a file pointer and return a number of type long, that corresponds to the current position.
n will tell that n bytes have already been read (or written)

rewind()
rewind takes a file pointer and resets the position to start of the file.

```
rewind(fp)
n=ftell(fp)
```

would assign 0 to n because the file position has been set to start of the file by rewind()
we can use this function to read file more than once without reopening the file.
Whenever file is opened for reading/writing , a rewind is done implicitly

fseek()

fseek() is used to move file position to a desired location within the file.

fseek(file_ptr, offset, position)

Value	Meaning
0	Beginning of file
1	Current position
2	End of file

Offset specify the number of bytes to be moved from the position.
Offset can be +ve – Moving forwards
Offset can be -ve – Moving backwards
Examples:
fseek(fp,0,0)→ same as rewind(fp)→ Go to beginning
fseek(fp,0,2)→end of the file
fseek(fp,m,0)→move to (m+1)th byte in the file
fseek(fp,m,1)→Go forwards by m bytes from current position
fseek(fp,-m,1)→Go backwards by m bytes from current position

on success fseek return a zero, on error it returns -1