Words of Concern

Point to be As Allan Bloom has said "Education is the movement from darkness to light". Through this handbook, I have tried to illuminate what might otherwise appear as black boxes to some. In doing so, I have used references from several other authors to synthesize or simplify or elaborate information. This is not possible without omitting details that I deem trivial while dilating the data that I consider relevant to topic. Every effort has been made to avoid errors. Inspite of this, some errors might have crept in. Any errors or discrepancies noted maybe brought to my notice which I shall rectify in my next revision.

Thanks,
Raghu Gurumurthy

Bibliography

Langsam, Augenstein and Tenenbaum, Data structures Using C and C++, Prentice Hall of India, 2nd Edition.
Kamthane :Introducation to Data structures in C Pearson Education.
Weiss Data structures and Algorithm Analysis in C II Edition , Pearson Education.
LipschutzSchaum's outline series Data structures Tata MgGraw-Hill.

Syllabus

BCA Syllabus CBCS-2016-17

BCA 2nd Semester		Data Structures Using C	
Subject Code :	16BB24	Total Teaching Hours :	52
IA Marks :	20	Teaching Hours/Week :	05
Exam Marks :	80	Examination Hours :	03
Credits:	5		

1. A Unit I:. Structure and union: Structure definition, giving values to members, structure initialization, comparison of structure variables, arrays of structure, self-referential structures, union. Pointers: Understanding pointers, accessing the address of variables, Declaring and initializing pointers, accessing a variable through its Pointer. Dynamic memory allocation: Meaning of static and dynamic memory allocation. Memory allocation functions: malloc(), calloc(), free() and realloc(). Files: Introduction, definition, Basic file operations: Naming a file, opening a file, Reading data from file, writing data to a file and closing a file, Input/Output operations on files, Error Handling in files, Random Access to files. 12 Hrs

2. Unit II: : Introduction to Data Structures & Stack: Definition, Applications, Classification of data structures: primitive and non-primitive, Operations on data structures Definition, Array Implementation of stack(using structure) and operations on stack, Applications of stacks, Infix, prefix and postfix notations, Conversion of an arithmetic expression from Infix to postfix. 12 Hrs

3. Unit III: Queue and Recursion: Definition, Types of queue: Simple queue, circular queue, double ended queue, priority queue, Array Implementations of queue (using structure) and operations on all types of queues. Definition, Recursion in C, Writing Recursive programs – Binomial coefficient, Fibonacci, GCD, towers of Hanoi. 12 Hrs

4. Unit IV: Linked list: Definition, components of linked list, Representation of linked list, Advantages and disadvantages of linked list, Types of linked list: singly linked list, doubly linked list, Circular list and circular doubly linked list, operations on all types of linked lists: Creation, insertion, deletion, search and display. 12Hrs

5. Unit V Tree: Definition: Tree, Binary tree, complete binary tree, Binary search tree, Tree terminology: root, Node, Degree of a node, ancestors of a node, Binary tree, Array representation of tree, Creation of Binary tree, Traversal of Binary tree: Preorder, In order and post order. 12Hrs

Unit I

Structures in C

Definition 1:

Structures are collection of variables of same or **different** data types under a single name.

Definition 2:

Structures are group of items in which each items is identified by its own identifies, each of which is known as **member** of the structure.

Keyword **struct** is used for creating a structure.

A Structure is a derived data type.

Defining a Structure

To define a structure, we must use the struct statement. The struct statement defines a new data type, with more than one member. The format of the struct statement is as follows:

```
struct [structure tag] {
   member definition;
   member definition;
   ...
   member definition;
} [one or more structure variables];
```

Note :

- [structure tag] is **Optional**.
- Don't forget the **semicolon** }; in the ending line.

Structure variable declaration

When a structure is defined, it creates a user-**defined type** but, no storage or memory is allocated, but for structure variables memory gets created.

Creation of variable can be done as below:

```
struct Person
{
    char name[50];
    int citNo;
    float salary;
};

int main()
{
    struct person person1, person2;
    return 0;
```

```
}
```

Another way of creating a structure variable is:

```
struct Person
{
    char name[50];
    int citNo;
    float salary;
} person1, person2;
```

Size of the Structure

The amount of memory required to store a structure variable is the **sum of memory** size of all members.

By considering above Structure Person

char name[50] : Size of name is 50 byes(1 * 50 = 50 bytes).
int citNo : Size of citNo is 2 byes(1 * 2 = 2 bytes).
float salary: Size of salary is 4 byes(1 * 4 = 4 bytes).

Size of the structure variable person1 is **56 bytes**.

C program to find the size of the structure

```
#include<stdio.h>
#include<conio.h>
struct Person
{
        char name[50];
        int citNo;
        float salary;
}person1;
void main()
{
        clrscr();
        printf("\nSize of person1 is %d\n", sizeof(person1));
        getch();
}
```


OUTPUT:

```
Size of person1 is 56
```

Accessing Structure Members

Structure variables can be accessed by using Member **Operator(.)(Dot)** as below

```
structure_variable_name.member_name
```

Suppose, we want to access salary for variable person2. Then, it can be accessed as:

person2.salary

Initializing Structure Members

Initialization of the structure members can be done by **dot "."** Operator

C program to demonstrate initialization and accessing of structure members

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

struct Person{
                char adharNo[20];
                char name[20];
                short int age;
                }_person1; //Declaring _person1
void main()
{
        clrscr();
        struct Person _person2; //Declaring _person2

        //Initializing _person1
        strcpy(_person1.adharNo, "adhar1");
        strcpy(_person1.name,"Raghu");
        _person1.age = 29;

        //Initializing _person2 by reading values from keyboard
        printf("Enter the Person details(Addhar no,Name,Age_) of person2\n");
        scanf("%s",_person2.adharNo);
        scanf("%s",_person2.name);
        scanf("%d",&_person2.age);

        //Accessing _person1
        printf("The preintialized person1 details are as below\n");
        printf("AdharNo: %s\n", _person1.adharNo);
        printf("Name: %s\n", _person1.name);
        printf("age: %d\n", _person1.age);
        printf("Size of struct %d\n",sizeof(_person1));

        //Accessing _person2
        printf("\nThe entered person2 details are as below\n");
        printf("AdharNo: %s\n", _person2.adharNo);
        printf("Name: %s\n", _person2.name);
        printf("age: %d\n", _person2.age);
        printf("Size of struct %d\n\n",sizeof(_person2));
        getch();
}
```

Comparing Structure Variables

Structure variables can be comparing as like comparing other variables, but need to access structure variable thorough dot (.) operator.

C program to demonstrate initialization and comparing of structure variables

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

struct Person{
		char adharNo[20];
		char name[20];
		short int age;
		}_person1; //Declaring _person1
void main()
{
	clrscr();
	struct Person _person2; //Declaring _person2

	//Initializing _person1
	strcpy(_person1.adharNo, "adhar1");
	strcpy(_person1.name,"Raghu");
	_person1.age = 29;

	//Initializing _person2 by reading values from keyboard
	printf("Enter the Person details(Addhar no,Name,Age_) of person2\n");
	scanf("%s",_person2.adharNo);
	scanf("%s",_person2.name);
	scanf("%d",&_person2.age);

	if(_person1.age == _person2.age)
	{
		printf("Person1 and Person2 age's are same\n");
	}
	else
	{
		printf("Person1 and Person2 age's are different\n");
	}

	if(strcmp(_person1.name,_person2.name) == 0)
	{
		printf("_person1's name is same as _person2's name\n");
	}
	else
	{
		printf("_person1's name is not same as _person2's name\n");
	}
	getch();
}
```

Array of structures

As like array of primitive data types, we can create array of structures. Elements of the structure need to be accesses using array index which starts from zero(0).

C program to demonstrate array of structure

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

struct Person{
                char adharNo[20];
                char name[20];
                short int age;
                }
void main()
{
        clrscr();
        struct Person _arrayOfPersons[3]; // Declaration the array of structure of type
Person.

        //Initializing _arrayOfPersons[0]
        strcpy(_arrayOfPersons[0].adharNo, "adhar3");
        strcpy(_arrayOfPersons[0].name,"Mitun");
        _arrayOfPersons[0].age = 27;

        printf("\nThe entered _arrayOfPersons[0] details are as below\n");
        printf("AdharNo: %s\n", _arrayOfPersons[0].adharNo);
        printf("Name: %s\n", _arrayOfPersons[0].name);
        printf("age: %d\n", _arrayOfPersons[0].age);
        printf("Size of struct %d\n",sizeof(_arrayOfPersons[0]));
        getch();
}
```

Unions:

Definition:

Unions are derived data types like structure, Creation and accessing union variables is similar to the accessing and creating of structures.

The difference between structures and unions is discussed as below.

1. Difference in memory allocation

```
#include <stdio.h>
#include <conio.h>
union unionJob
{
   //defining a union
   char name[32];
   float salary;
   int workerNo;
} uJob;

struct structJob
{
   char name[32];
   float salary;
   int workerNo;
} sJob;

void main()
{
    clrscr();
   printf("size of union = %d", sizeof(uJob));
   printf("\nsize of structure = %d", sizeof(sJob));
   getch();
}
```


OUTPUT:

```
Size of union = 32
Size of structure = 38
```

The memory required to store a union variable is the memory required for the **largest element** of an union.

2. Difference in member accessing

In the case of structure, all of its members can be accessed at any time.

But, in the case of union, only one of its members can be accessed at a time and all other members will contain garbage values.

```
#include<stdio.h>
#include<conio.h>
union Person{
                int id;
                char name[15]; // Big member of length 15 bytes
                };
void unionDemo1();
void unionDemo2();
void main()
{
        clrscr();
        unionDemo1();
        unionDemo2();
        getch();
}
void unionDemo1()
{
        union Person p1;
        p1.id=23;
        printf("Person p1 details\n");
        printf("%d\n",p1.id);
        printf("Size of union p1 is %d\n", sizeof(p1));
}
void unionDemo2()
{
        union Person p2;
        p2.id=24;
        strcpy(p2.name,"RaghuGurumurthy");
        printf("\nPerson p2 detials\n");
        printf("Id:%d",p2.id);
        printf("Name:%s",p2.name);
}
```

OUTPUT:

```
Person p1 details
Id:23
Size of union p1 is 15

Person p2 detials
Id:24914
Name:RaghuGurumurthy
```

Note:
We can getting garbage value for p2.id, since we can access only one value of union at a time.

Pointers:

Definition 1:

A pointer is a variable itself, which stores the address of another variable of some specific data type. The contents of pointer are address of another variable of type int, float, char etc.

Definition 2:

A pointer is a special type of variable; the pointer contains the address of another variable.

Normal variables meant for storing value, where as pointer variables for storing address of other variables.

Syntax:

data_type *pointer_variable_name;

data_type : Data type of the pointer like int,float.
pointer_variable_name : Name of the pointer variable.
"*": is the notation for declaring pointer variables.

Example :

int *p;
float *f;

Declaration and Initialization of pointers:

Pointers declaring are done as follows.

int *ptr;

Pointers initialization is done as follows.

ptr = &a;

&a gives the address of the variable a
ptr is the pointer variable
Address of the variable a is assigned to point ptr.

Reference operator (&) and Dereference operator (*)

Reference operator (&) is used to get the address of the variable

Dereference operator (*) operator is used to get the value of the variable from the address.

Accessing a variable through pointers

The value of the variable is accessed through **dereference** operator (*) as below:

printf("The value of the variable a is : %d\n", ***ptr**);

The address of the variable is accessed as below.

printf("The value of the variable a is : %u\n", **ptr**);

Note : For value need to put * (*ptr)
For address need to remove * (ptr)

Program to explain the concept of referencing and dereferencing operator

```
#include<stdio.h>
#include<conio.h>
void main()
{
        int a = 10;
        int *ptr;
        clrscr();
        ptr = &a;
        printf("\n *********Pointer Demo*********\n");
        printf("\nAccessing address and value of variable through normal variable\n");
        printf("The address of variable a is : %u\n", &a);
        printf("The value of the variable a is : %d\n", a);
        printf("\nAccessing address and value of variable through normal pointer variable\n");
        printf("The address of variable a is : %u\n", ptr);
        printf("The address of variable a is : %p\n", ptr);
        printf("The value of the variable a is : %d\n", *ptr);
        getch();
}
```

OUTPUT:

```
**********Pointer Demo**********

Accessing address and value of variable through normal variable
The address of variable a is : 65524
The value of the variable a is : 10

Accessing address and value of variable through normal pointer variable
The address of variable a is : 65524
The address of variable a is : FFF4
The value of the variable a is : 10
```

Note :

Address of a ie FFF4 is in hexadecimal notation, since we used %p as data type notation.
Address of a ie 65524 is in decimal notation. Since we used %u as data type notation.

Self Referential Structure:

Definition 1:

A self referential structure is essentially a structure definition which includes at least one member that is a pointer to the structure of its own kind.

Definition 2:

In structures, members can be of any data type. If we include a member in the structure, which is a pointer to the same structure type, such a structure is called self-referential structure.

Syntax:

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

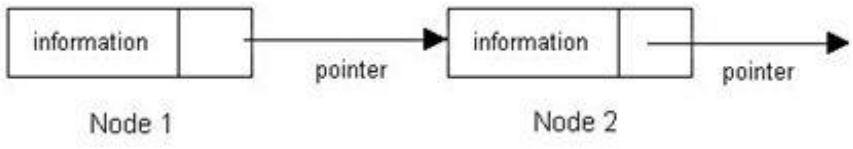
member1: Structure member and its type may int or float or char.
member2: Structure member and its type may int or float or char.
*pointer : Structure member and its type is struct name.

Example:

```
struct Person
{
        char name[10];
        struct Person *nextPerson;
}Person1,Person2;
```

Self referential structures are useful to implement advance data structures like linked list, binary tree etc....

The following figure shows how self referential structures points to the other structure which of same structure type.



Program to explain the concept of self referential structure

```
#include<stdio.h>
#include<conio.h>
struct Person
{
        char name[10];
        struct Person *nextPerson;
}Person1,Person2;

void main()
{     clrscr();
        strcpy(Person1.name,"Kishore");
        Person1.nextPerson = &Person2;
        strcpy(Person2.name,"Gautham");
        Person2.nextPerson = NULL;

        printf("Address of Person2 :%u",Person1.nextPerson);
        printf("Value of Person2 :%d",Person1.nextPerson.name);
        getch();
}
```

OUTPUT:

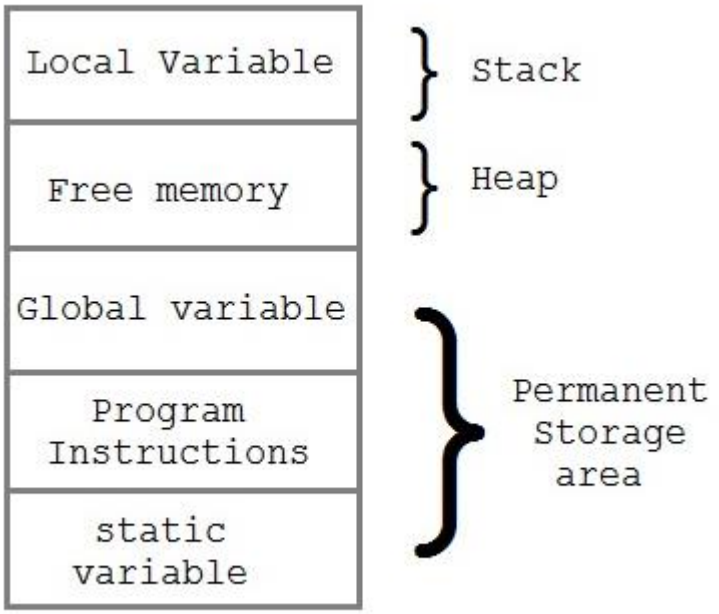
```
Address of Person2 :890
```

Note : Person1 points to the Person2, so we can able to access the address of Person2 from Person1.

Dynamic memory allocation:

Memory Allocation Process:

The memory allocation process includes allocation of memory at different areas of memory. Below diagram shows different types of areas in memory along with different components of the program which stores in the specific areas of memory.



- Local Variables: Stores in the stack area of the memory.
- Free memory meant for storing dynamic memory allocation and which belongs to heap area of memory.
- Global variable, Program instructions and static variable are gets stored in permanent storage area of memory.

What is dynamic memory allocation..?

Definition 1:

The process of allocating memory at runtime is known as dynamic memory allocation. Library routines known as "memory management functions" are used for allocating and freeing memory during execution of a program.

Definition 2:

Dynamic memory allocation allows your program to obtain more memory space while running or execution time of program, or to release it if it's not required. Dynamic memory allocation allows you to manually handle memory space for your program.

The dynamic memory allocation and memory deallocation is carried out by library functions which are available in library called stdlib.h. They are as below

1. 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)

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.

Example of malloc()

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.

Program to explain the concept of dynamic allocation of memory using malloc

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

void memoryAllocationForAVariableUsingMalloc();
void memoryAllocationForAArrayUsingMalloc();
void main()
{
        clrscr();
        printf("Memory allocation for single variable using malloc\n");
        memoryAllocationForAVariableUsingMalloc();
        printf("Memory allocation for an array using malloc\n");
        memoryAllocationForAArrayUsingMalloc();
        getch();
}

void memoryAllocationForAVariableUsingMalloc()
{
        float* ptrFloat;
        ptrFloat = (float*) malloc(1 * sizeof(float));  //memory allocated using malloc
        if(ptrFloat == NULL)
        {
                printf("Error! memory not allocated.");
                exit(0);
        }
```

```
        else
        {
                printf("Enter float number: ");
                scanf("%f", ptrFloat);
                printf("Entered no:%f and its location :%d\n",*ptrFloat,ptrFloat);
                free(ptrFloat);
        }
}
void memoryAllocationForAArrayUsingMalloc()
{
        int num, i, *ptr;
        printf("Enter number of elements:\n ");
        scanf("%d", &num);
        ptr = (int*) malloc(num * sizeof(int));  //memory allocated using malloc
        if(ptr == NULL)
        {
                printf("Error! memory not allocated.");
                exit(0);
        }
        printf("Enter elements of array:\n ");
        for(i = 0; i < num; i++)
        {
                scanf("%d", ptr + i);
        }
        for(i=0; i< num; i++)
        {
                printf("Element %d stored in %d\n",*(ptr + i),(ptr + i));
        }
        free(ptr);
}
```

OUTPUT :

```
Memory allocation for single variable using malloc
Enter float number: 23.3
Entered no:23.299999 and its location :2782
Memory allocation for an array using malloc
Enter number of elements:
 2
Enter elements of array:
 1
2
Element 1 stored in 2782
Element 2 stored in 2784
```

2. calloc()

The name calloc stands for "contiguous allocation".

The 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 sets all bytes to zero.

Syntax of calloc()

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

ptr is pointer of cast-type.

n number of memory blocks required.

This statement will allocate contiguous space in memory for an array of n elements.

Example of calloc()

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.

Program to explain the concept of dynamic allocation of memory using calloc

```
#include <stdio.h>
#include <stdlib.h>
void memoryAllocationForAArrayUsingCalloc();
void main()
{
        clrscr();
        printf("Memory allocation for a array using calloc\n");
        memoryAllocationForAArrayUsingCalloc();
        getch();
}
void memoryAllocationForAArrayUsingCalloc()
{
   int num, i, *ptr;
   printf("Enter number of elements: ");
   scanf("%d", &num);
   ptr = (int*) calloc(num, sizeof(int));
   if(ptr == NULL)
   {
        printf("Error! memory not allocated.");
        exit(0);
   }
   printf("Enter elements of array: ");
   for(i = 0; i < num; ++i)
   {
```

```
        scanf("%d", ptr + i);
    }
    for(i=0; i< num; i++)
    {
        printf("Element %d stored in %d\n",*(ptr + i),(ptr + i));
     }
    free(ptr);
}
```

OUTPUT:

```
Memory allocation for a array using calloc
Enter number of elements: 2
Enter elements of array: 1
2
Element 1 stored in 1982
Element 2 stored in 1984
```

3. realloc()

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

Syntax of realloc()

ptr = realloc(ptr, new_size)

ptr is pointer of cast-type.

new_size number of new memory blocks required.

This statement will reallocate space in memory for an array of new_size elements.

Example of realloc()

ptr = realloc(ptr ,5);

This statement allocates additional space in memory for an array of 5 elements each of size of int, i.e, 2 bytes. If ptr is of type int then (5 * 2 = 10 bytes) or if ptr is of type float (5* 4 = 20 bytes) memory will allocate.

Program to explain the concept of dynamic allocation of memory using realloc

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

void memoryReAllocationUsingRealloc();
void main()
{
        clrscr();
        printf("Memory Reallocation for an array using realloc\n");
        memoryReAllocationUsingRealloc();
        getch();
}

void memoryReAllocationUsingRealloc()
{
   int *ptr, i , n1, n2;
   printf("Enter size of array: ");
   scanf("%d", &n1);
   ptr = (int*) malloc(n1 * sizeof(int));
   printf("Address of previously allocated memory: ");
   for(i = 0; i < n1; ++i)
        printf("%u\t",ptr + i);
   printf("\nEnter new size of array: ");
   scanf("%d", &n2);
   ptr = realloc(ptr, n2);
   printf("Address of the old allocated memory + newly allocated memory: ");
   for(i = 0; i < (n1 + n2); ++i)
        printf("%u\t", ptr + i);
        free(ptr);
}
```

OUTPUT:

```
Memory Reallocation for an array using realloc
Enter size of array: 2
Address of previously allocated memory: 2040    2042
Enter new size of array: 2
Address of the old allocated memory + newly allocated memory: 2040      2042
2044    2046
```

4. free()

   Dynamically allocated memory created with either calloc() or malloc() doesn't get freed on its own. we 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.

Files:

Introduction:

Why files are needed?

- When a program is terminated, the entire data is lost. Storing in a file will preserve your data even if the program terminates.
- If we have to enter a large number of data, it will take a lot of time to enter them all. However, if we have a file containing all the data, you can easily access the contents of the file using few commands in C.
- We can easily move your data from one computer to another without any changes.

Definition 1:

A file is a place on the disk where a group of related data is stored. Files provides the way to store the data permanently, however we can delete the contents of the file or delete whole file itself, when it is not needed.

Definition 2:

A file represents a sequence of bytes on the disk where a group of related data is stored. File is created for permanent storage of data. It is a readymade storage structure.

C programming is capable of handling files by performing four major operations on the files.

- Creating a new file.
- Opening an existing file.
- Closing a file.
- Reading from and writing information to a file.

C standard I/O library (stdio.h) provides the below functions to operate with files.

Function	Description
fopen()	Create a new file or open a existing file.
fclose()	Closes a file.
getc()	Reads a character from a file.
putc()	Writes a character to a file.
fscanf()	Reads a set of data from a file.
fprintf()	Writes a set of data to a file.
getw()	Reads a integer from a file.
putw()	Writes a integer to a file.

fseek()	Set the position to desire point.
ftell()	Gives current position in the file.
rewind()	Set the position to the begining point.

Basic File Operations.

To operate with files, we must specify below things about files.

1. Filename.
2. Data Structure.
3. Purpose.

Filename: Name of the files, need to be given as strings in double quotes.

Data structure of file defined as FILE, File should be declared as type FILE and FILE is a defined data type.

Purpose: When we open a file we need to specify the purpose ie why we are opening a file whether we need to do read or write or append the file.

Basic file operations:

Naming a file:

Name of the file need to be given in double quotes since name of the file is string.File name come with two parts ie a primary name and an optional period with extension.

Ex:
Demo.txt
LAB1.c

Here Demo and LAB1 are primary names and .txt and .c are the extensions of the file.

```
FILE *fp;
fp = fopen("filename","mode");

filename : needs to be given in quoted strings.

Ex:
fp = fopen("demo.txt",r);
```

Opening a file (May be for creation or editing previously create file).

Below is the general format for declaring and opening a file.

Syntax:

```
FILE *fp;
fp = fopen(“filename”,”mode”);
```

FILE *fp : Declaring of pointer variable *fp of data type FILE.
fopen() : Opens the file with provided filename and mode(read,write,append).

Ex:

```
FILE *fp :
fp = fopen("demo.txt", "w");
```

Modes of files:

Modes specifies the type of the operations what we need to do on the file, which may be read, write or append.

r (Read mode): open the file for reading only. If file exist then file opens, else error occurs.

w (Write mode): Open the file for writing only. A file is created if file does not exist, else contents of the already existed file deleted.

a (Append mode): open the file for appending or adding data to it. A file is created if file does not exist, else contents of the already existed file remains safe.

Reading data from file:

To read the data from a file, we need to open a file in reading mode(r). While opening file for read, if file exist then file opens, else error occurs.

Ex:

```
FILE *fp;
fp = fopen(“demo.txt”,”r”);
```

demo.txt : Name of the file, which need to be open.
r : Mode of file for reading i.e. r.

Writing data into file:

To write the data into file, we need to open a file in writing mode (w). While opening file for writing, a file is created, if file does not exist. Else contents of the already existed file get deleted.

Ex:

```
FILE *fp;
fp = fopen("demo.txt","w");
```

demo.txt : Name of the file, which need to be open.
w : Mode of file for writing i.e. w.

Closing a File

The file should be closed after reading/writing. Since closing results in flushing (clearing) all data related to the files.

Closing a file is performed using library function fclose().

Syntax :

```
fclose(file_pointer)
```

Ex:

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

fp : File pointer.

Program to write into file and read from file.

```
#include<stdio.h>
#include<conio.h>
void main()
{
 FILE *fp;
 char ch;
 clrscr();
 fp = fopen("demo.txt", "w"); // Opening file in write mode.
 printf("Enter text in to the file\n");
 while( (ch = getchar()) != EOF) //Looping until  reading EOF(End of file)
 {
    putc(ch,fp);
 }
 fclose(fp);// Closing the file.
 fp = fopen("demo.txt", "r"); //Opening file in read mode
 printf("*****Text in the file*****\n");
 while((ch = getc(fp))!= EOF) //Looping until EOF
    printf("%c",ch);
```

```
 fclose(fp);//Closing the file after reading.
 getch();
}
```

OUTPUT:

EOF is defined in stdio.h. On Linux ctrl + d signals EOF and in windows ctrl + z signals EOF.

Input/Output operations.

Input and output functions are classified on the basis of what kind of data the functions will operate.

getc() and putc() : Handles one character at a time.

getc() : Used to read a character at a time takes one argument ie file pointer.

Systax : getc(fp)
Here fp:filepointer.

Ex :

```
FILE *fp;
char c;
fp = open("demo.txt","r")
while((c=getc(fp)) != EOF)
```

putc() : Used to write a character at a time takes two argument ie character which needs to write into file and file pointer.

Systax: putc(c,fp)

Here c : Character needs to write in to file.
fp: File pointer.

Ex:

```
FILE *fp;
char c;
fp=open("demo.txt", "w")
while((c=getchar()) != EOF)
putc(c,fp);
```

getw() and putw() : Integer oriented functions used to read and write integer data respectively. These functions are useful when we dealing with only integer data.

The general form is as below.
putw(integer, fp);
getw(fp);

frpintf() and fsacnf() : These functions handle a group of mixed data simultaneously.

fprintf() and fscanf() performs I/O operations that are identical to the printf() and scanf() functions, except the course they work on files.

fprintf() : Used to write the mixed data in to the file. And the file needs to be open in write mode.

Syntax:

fprintf(fp,"control string", list);

Here : fp: File pointer
control string : Output specifications for the items in the list.
list: May contain variables, constants and strings

Ex: fprintf(fp,"%d %s %d %d\n",rollNo,name,mark1,mark2)

fscanf() : Used to read the mixed data from the file. And the file needs to be open in read mode.

Syntax:

fscanf(fp,"control string", list);

Here : fp: File pointer
control string : Output specifications for the items in the list.
list: May contain variables, constants and strings

Ex fscanf(fp,"%d %s %d %d",&rollNo,name,&mark1,&mark2);

Error handling during I/O operations.

There is chance of errors during I/O operations, the situations are as below.

- Try to read behind the EOF.
- Device Overflow.
- Try to read the file which is not opened.
- Opening file with invalid filename.
- Attempting to write into file, when file is write protected.

To handle above mentioned errors c comes with functions like feof() and ferror()

feof(): feof function is used to test for end of file condition. It takes a file pointer as argument and retrun non zero integer value to specify the reading is complete(up to EOF) and return zero if the file reading is not completed.

Note:
Arguments: No of argument 1(i.e file pointer)
Return type: Integer.
Possible return value : Non zero(>0) if file reached EOF.
Zero if the file not reached EOF.

Syntax:

feof(fp);

Here: fp: File pointer

Ex:

```
if(feof(fp))
printf("End of file data\n");
else
printf("End of file is not reached, might be chance of error\n");
```

ferror(): ferror function reports the status of file indicated.It takes a argument i.e file pointer and returns non zero if file operation end with a error and returns zero if the file operations successfully happened.

Note:
Arguments: No of argument 1(i.e file pointer)
Return type: Integer.
Possible return value : Zero(0) if the file operations not results in Error.
Non zero(>0) if the file operations results in Error.

Syntax:

feof(fp);

Here: fp: File pointer

Ex:

```
if(ferror(fp) != 0)
printf("Error occurred while file operation\n");
else
printf("File operations happened successfully\n");
```

Random access to files

Random access to files is all about accessing only required part of the file by neglecting other parts of file. Random accessing is achieved by below functions in the files.

ftell(): ftell function useful when we need to save the current position of the file. it takes file pointer as an argument and returns long value that corspondance to the current position of the file.

Note:
Arguments: No of argument 1(i.e file pointer)
Return type: Long.
Possible return value: Long value which gives the address of the current position in bytes.

Syntax:

ftell(fp);

Here: fp: File pointer

Ex:

n=ftell(fp)
n is assigned with current position of the file.

rewind(): rewind function takes the file pointer as argument and resets the position to the starting of the file.

Note:
Arguments: No of argument 1(i.e file pointer)
Return type: Not returning anything just do resetting the position of the file to start

Syntax:

rewind(fp);

Here: fp: File pointer

Ex:

rewind(fp);
n = ftell(fp);
n is assigned with zero(0) because file has been set to the start of the file by rewind.

rewind() helps us to read the file from begning without open and close the file the file by fopen() and fclose() respectively .

point to remember : Whenever we open the file for reading or writing, a rewind is done implicitly.

fseek(): fseek function used to move the file position to a desired location within the file. It takes the following form.

Note:
Arguments: No of argument 3(file pointer,offset,position)
Return type: Not returning anything just move the file to the desired location.

Syntax:

fseek(fp,offset,position);

Here: fp: File pointer
offset : is a number or variable of type long and may be positive(move forward) or negative(move backward).
position : is an integer number

The position can take one of the following three values

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

Ex:

fseek(fp,0L,0) Go to beginning, similar to rewind
fseek(fp,m,1) Go forward by m bytes.
fseek(fp,m,2) Go backward by m bytes.

Sign Up And Download Full Notes