UNIT - V
POINTERS, STRUCTURES & UNIONS, FILES

Pointers: Concept of a Pointer, Initialization of pointer variables, pointers as function arguments, address arithmetic, pointers to pointers, Pointers and arrays, Array of Pointers, Dynamic memory management functions, parameter passing by address.
Structures and Unions: Structures declaration, Initialization of structures, accessing structures,unions
Files: Declaring, Opening, and Closing File Streams, Reading from and Writing toText Files.

ADDRESS IN C:

If you have a variable 'var' in your program, '&var' will give you its address in the memory, where & is commonly called the reference operator.
This notation is seen in the scanf function. It is used in the function to store the user inputted value in the address of var.

```
scanf("%d", &var);
```

Program:

```
#include <stdio.h>
int main()
{
  int var = 5;
  printf("Value: %d\n", var);
  printf("Address: %u", &var);  //Notice, the ampersand(&) before var.
  return 0;
}
```

Output:

```
Value: 5
Address: 2686778
```

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.

Declaration:

Syntax: data_type *var-name;

Here, type is the pointer's base type; it must be a valid C data type and var-name is the nameof the pointer variable.

Examples:

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

Initialization of pointer variable:

The initialization of a pointer variable is similar to other variable initialization, but in the pointer variable we assign the address instead of value.

```
int  *ptr;
int  var=10;
ptr = &var;
```

In the above example we have a pointer variable (*ptr) and a simple integer variable (var). We have to assign the pointer variable to the address of 'var'. It means that the pointer variable '*ptr' now has the address of the variable 'var'.

REFERENCE OPERATOR (&) AND DEREFERENCE OPERATOR (*):

&is called reference operator which gives the address of a variable. Likewise, there is another operator that gets you the value from the address, it is called a dereference operator (*).**Note:** The * sign when declaring a pointer is not a dereference operator.

Program:

```
#include <stdio.h>
int main(){
   int *pc;
   int c;
   c=22;
   pc=&c;
   printf("Address of pointer pc: %u\n",pc);
   printf("Content of pointer pc: %d\n",*pc);
   printf("Address of c: %u\n",&c);
   printf("Value of c: %d\n",c);
   return 0;
}
```

Output:

```
Address of pointer pc: 2686784
Content of pointer pc: 22
Address of c: 2686784
Value of c: 2
```

NULL POINTERS: It is always a good practice to assign a NULL value to a pointer variable in case you do not have an exact address to be assigned. This is done at the time of variable declaration. A pointer that is assigned NULL is called a nullpointer.

```
#include <stdio.h>
int main () {
   int  *ptr = NULL;
   printf("The value of ptr is : %d\n", ptr );
   return 0;
}
```

Output:
The value of ptr is 0

GENERIC POINTERS: It is a pointer variable that has void as its data type. It can be used to point to variables of any type.

```
void *ptr;
```

In C, since we cannot have a variable of type void, the void pointer will therefore not point to any data and thus cannot be dereferenced. We need to typecast a void pointer to another kind of pointer before using it. It is often used when we want a pointer to point to data of different types at different time.

```
#include <stdio.h>
int main () {
int x=10; void *ptr;
   ptr = (int *)&x;
   printf("generic pointer points to the integer value %d", *ptr);
   return 0;
}
```

Output:
Generic pointer points to the integer value 10

POINTERS AS FUNCTION ARGUMENTS:
C programming allows passing a pointer to a function. To do so, simply declare the function parameter as a pointer type and send the addresses of the variables in the function call.
Program: to swap two numbers using pointers and function.

```
#include <stdio.h>
void swap(int *n1, int *n2);
int main()
{
```

```
    int num1 = 5, num2 = 10;
    swap( &num1, &num2); // address of num1 and num2 is passed to the swap function
    printf("Number1 = %d\n", num1);
    printf("Number2 = %d", num2);
    return 0;
}
void swap(int * n1, int * n2)
{
    // pointer n1 and n2 points to the address of num1 and num2 respectively
    int temp;
    temp = *n1;
    *n1 = *n2;
    *n2 = temp;
}
```

Output:

Number1 = 10
Number2 = 5

- The address of memory location num1 and num2 are passed to the function swap and the pointers *n1 and *n2 accept those values.
- So, now the pointer n1 and n2 points to the address of num1 and num2 respectively.
- When, the value of pointers are changed, the value in the pointed memory location also changes correspondingly.
- Hence, changes made to *n1 and *n2 are reflected in num1 and num2 in the main function.
- This technique is known as Call by Reference in C programming.

DANGLING POINTERS:

- Dangling pointers arise when an object is deleted or de-allocated, without modifying the value of the pointer, so that the pointer still points to the memory location of the de-allocated memory.
- In short pointer pointing to non-existing memory location is called dangling pointer.

Examples:

Way 1 : Using free or de-allocating memory

```
#include<stdlib.h>
{
    char *ptr = malloc(Constant_Value);
    .....
    .....
    free (ptr);      /* ptr now becomes a dangling pointer */
}
```

We have declared the character pointer in the first step. After execution of some statements we have de-allocated memory which is allocated previously for the pointer.

As soon as memory is de-allocated for pointer, pointer becomes dangling pointer

How to Ensure that Pointer is no Longer Dangling ?

```
#include<stdlib.h>
{
   char *ptr = malloc(Constant_Value);
   .......
   .......
   .......
   free (ptr); /* ptr now becomes a dangling pointer */
   ptr = NULL  /* ptr is no more dangling pointer */
}
```

After de-allocating memory, initialize pointer to NULL so that pointer will be no longer dangling. Assigning NULL value means pointer is not pointing to any memory location

ADDRESS ARITHMETIC:

Address arithmetic is a method of calculating the address of an object with the help of arithmetic operations on pointers and use of pointers in comparison operations. Address arithmetic is also called pointer arithmetic.

We can perform arithmetic operations on the pointers like addition, subtraction, etc. However, as we know that pointer contains the address, the result of an arithmetic operation performed on the pointer will also be a pointer if the other operand is of type integer.

```
#include<stdio.h>
int main()
{
   int number=50;
   int *p;
   p=&number;
  printf("Address of p variable is %u \n",p);
  p=p+1;
  printf("After increment: Address of p variable is %u \n",p);
  return 0;
 }
```

POINTER TO POINTERS:

A pointer to a pointer is a form of multiple indirection, or a chain of pointers. Normally, a pointer contains the address of a variable. When we define a pointer to a pointer, the first pointer contains the address of the second pointer, which points to the location that contains the actual value as shown below.

Declaration:

```
int **var;      //declares a pointer to pointer of type int
```

When a target value is indirectly pointed to by a pointer to a pointer, accessing that value requires that the asterisk operator be applied twice, as is shown below in the example:

Program:

```
#include <stdio.h>
int main () {
   int var;
   int  *ptr;
   int **pptr;
   var = 3000;
   ptr = &var;  /* take the address of var */
   pptr = &ptr; /* take the address of ptr using address of operator & */
   printf("Value of var = %d\n", var ); /* take the value using pptr */
   printf("Value available at *ptr = %d\n", *ptr );
   printf("Value available at **pptr = %d\n", **pptr);
   return 0;
}
```

Output:

Value of var = 3000
Value available at *ptr = 3000
Value available at **pptr = 3000

POINTERS AND ARRAYS:

When an array is declared, compiler allocates sufficient amount of memory to contain all the elements of the array. Base address i.e address of the first element of the array is also allocated by the compiler.

Suppose we declare an array arr,
int arr[5] = { 1, 2, 3, 4, 5 };
Assuming that the base address of arr is 1000 and each integer requires two bytes.
Here variable arr will give the base address, which is a constant pointer pointing to the first element of the array, arr[0]. Hence arr contains the address of arr[0] i.e 1000. In short, arr has two purpose - it is the name of the array and it acts as a pointer pointing towards the first element in the array.

arr is equal to &arr[0] by default

We can also declare a pointer of type int to point to the array arr.

```
int *p;
p = arr;
// or,
p = &arr[0];    //both the statements are equivalent.
```

Pointer to Array:

We can use a pointer to point to an array, and then we can use that pointer to access the array elements. Lets have an example,

```
#include <stdio.h>
int main()
{
   int i;
   int a[5] = {1, 2, 3, 4, 5};
   int *p = a;     // same as int*p = &a[0]
   for (i = 0; i < 5; i++)
   {
      printf("%d", *p);
      p++;
   }
   return 0;
}
```

In the above program, the pointer *p will print all the values stored in the array one by one. We can also use the Base address (a in above case) to act as a pointer and print all the values.
The generalized form for using pointer with an array,
*(a+i)
is same as:
a[i]

ARRAY OF POINTERS:

Just like we can declare an array of int, float or char etc, we can also declare an array of pointers, here is the syntax to do the same.

Syntax:

datatype *array_name[size];

Example:

int *arrop[5];

Here arrop is an array of 5 integer pointers. It means that this array can hold the address of 5 integer variables, or in other words, you can assign 5 pointer variables of type pointer to int to the elements of this array.

Program:

```
#include<stdio.h>
int main()
{
   int a[5]={10,20,30,40,50};
   int *b[5];
   int i;
   for(i=0;i<5;i++)
  {
    b[i]=&a[i];
    printf("Elements are :%d", *b[i]);
  }
```

DYNAMIC MEMORY MANAGEMENT:

Dynamic memory management refers to the process of allocating memory to the variables during execution of the program ot at run time. This allows you to obtain more memory when required and release it when not necessary.

There are 4 library functions defined under <stdlib.h> for dynamic memory allocation.

Function	Use of Function
malloc()	Allocates requested size of bytes and returns a pointer to the first byte of allocated space
calloc()	Allocates space for an array of elements, initializes them to zero and then returns a pointer to memory
free()	deallocate the previously allocated space
realloc()	Change the size of previously allocated space

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:

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.

Example:

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 find sum of n elements entered by user using malloc() and free()

```
#include  <stdio.h>
#include <stdlib.h>
int main()
{
   int  *ptr, n,i;
   printf("Enter number of elements: ");
   scanf("%d", &n);
```

```
    ptr = (int*) malloc(n * sizeof(int));
    if(ptr == NULL)
    {
        printf("Memory not allocated.");
        exit(0);
    }
    printf("Memory allocated ");
    for(i = 0; i < n; ++i)
    {
      ptr[i]=i+1;
      printf(" %d", ptr[i]);
    }
    free(ptr);
    return 0;
}
```

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 sets all bytes to zero.

Syntax:

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

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

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.

Program: to find sum of n elements entered by user using calloc() and free

```
#include  <stdio.h>
#include <stdlib.h>
int main()
{
    int  *ptr, n,i;
    printf("Enter number of elements: ");
    scanf("%d", &n);
```

```
    ptr = (int*) calloc(n, sizeof(int));
    if(ptr == NULL)
    {
        printf("Memory not allocated.");
        exit(0);
    }
    printf("Memory allocated ");
    for(i = 0; i < n; ++i)
    {
      ptr[i]=i+1;
      printf(" %d", ptr[i]);
    }
    free(ptr);
    return 0;
}
```

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:

free(ptr);

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

realloc()

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

Syntax:

ptr = realloc(ptr, newsize);

Here, ptr is reallocated with size of newsize.

Program:

```
#include  <stdio.h>
#include <stdlib.h>
int main()
{
    int  *ptr, n,i;
    printf("Enter number of elements: ");
    scanf("%d", &n);
```

```
    ptr = (int*) calloc(n, sizeof(int));
    if(ptr == NULL)
    {
        printf("Memory not allocated.");
        exit(0);
    }
    printf("Memory allocated ");
    for(i = 0; i < n; ++i)
    {
      ptr[i]=i+1;
      printf(" %d", ptr[i]);
    }
    printf("Enter number of elements: ");
    scanf("%d", &n);
    ptr = (int*) realloc(ptr, n*sizeof(int));
    for(i = 5; i < n; ++i)
    {
      ptr[i]=i+1;
      printf(" %d", ptr[i]);
    }
    free(ptr);
    return 0;
}
```

PARAMETER PASSING BY ADDRESS:

In this approach, the addresses of actual arguments are passed to the function call and the formal arguments will receive the address. Inside the function, the address is used to accessthe actual argument used in the call. It means the changes made to the parameter affect the passed argument.

Note: Actual arguments are address of the ordinary variable, pointer variable or array name. Formal arguments should be a pointer variable or array. This approach is of practical importance while passing arrays to functions and returning back more than one value to the calling function. Passing arrays to functions is call by reference by default.

```
#include <stdio.h>
void swap(int *x, int *y);
int main ()
{
   int a = 100;
   int b = 200;
   printf("Before swap, value of a : %d\n", a );
   printf("Before swap, value of b : %d\n", b );
   swap(&a, &b);
   printf("After swap, value of a : %d\n", a );
   printf("After swap, value of b : %d\n", b );
   return 0;
}

void swap(int *x, int *y)
{
   int temp;
   temp = *x;    /* save the value at address x */
   *x = *y;      /* put y into x */
   *y = temp;    /* put temp into y */
   }
```

Output:
Before swap, value of a :100
Before swap, value of b :200
After swap, value of a :200
After swap, value of b :100

STRUCTURES AND UNIONS

STRUCTURES:
A structure is a user defined data type in C. A structure creates a data type that can be used to group items of possibly different types into a single type

Example: Suppose you want to keep track of your books in a library. You might want to track the following attributes about each book

Title
Author
Subject
Book ID

DEFINING A STRUCTURE:

To define a structure, you must use the struct statement. The struct statement defines a new data type, with more than one member.

Syntax:

```
struct [structure tag]
{
  member definition;
   member definition;
   ...
   member definition;
 };
```

The structure tag is optional and each member definition is a normal variable definition, such as int i; or float f; or any other valid variable definition. At the end of the structure's definition, before the final semicolon, you can specify one or more structure variables but it is optional.

Example: to declare the Employee structure

```
struct Employee
 {
     int eno;
     char name[20];
     float salary;
  };
```

ACCESSING STRUCTURE MEMBERS:

To access any member of a structure, we use the member access operator (.). The member access operator is coded as a period between the structure variable name and the structure member that we wish to access.

Program:

```
#include<stdio.h>
#include<conio.h>
struct Employee
{
  int eno;
  char ename[20];
  float esalary;
};
void main()
{
  struct Employee e={101,"Ravi",20000};
  printf("Emp no: %d",e.eno);
  printf("Emp name:%s",e.ename);
  printf("Emp salary:%f",e.esalary);
  getch();
}
```

ARRAY OF STRUCTURES:

An array of structures is nothing but collection of structures. This is also called as structure array in C.

This program is used to store and access "id, name and percentage" for 3 students. Structure array is used in this program to store and display records for many students. You can store "n" number of students record by declaring structure variable as 'struct student record[n]", where n can be 1000 or 5000 etc.

Example:

```
 #include<stdio.h>
#include<conio.h>
struct Employee
{
  int eno;
  char ename[20];
  float esalary;
};
void main()
{
  struct Employee e[5];
  int i;
  for(i=0;i<5;i++)
 {
  printf("Enter Employee details");
  scanf("Emp no: %d",&e[i].eno);
  scanf("Emp name:%s",e[i].ename);
  scanf("Emp salary:%f",&e[i].esalary);
 }
 for(i=0;i<5;i++)
 {
  printf("Enter Employee details");
  printf("Emp no: %d",e[i].eno);
  printf("Emp name:%s",e[i].ename);
  printf("Emp salary:%f",e[i].esalary);
}
getch();
}
```

POINTER TO STRUCUTRE:

```
#include<stdio.h>
#include<conio.h>
struct Employee
{
  int eno;
  char ename[20];
  float esalary;
};
```

```
void main()
{
  struct Employee *p,e={101,"Ravi",20000};
  p=&e;
  printf("Emp no: %d",p->eno);
  printf("Emp name:%s",p->ename);
  printf("Emp salary:%f",p->esalary);
  getch();
}
```

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.
- In other words, structures pointing to the same type of structures are self-referential in nature.
- A self-referential structure is used to create data structures like linked lists, stacks, etc.

Syntax:

```
struct struct_name
{
datatype datatypename;
struct_name *pointer_name;
};
```

Self-referential Structure with Single Link:

These structures can have only one self-pointer as their member. The following example will show us how to connect the objects of a self-referential structure with the single link and accessthe corresponding data members. The connection formed is shown in the following figure.

ob1 ob2

Program:

```
#include <stdio.h>
struct node {
   int data1;
   char data2;
   struct node* link;
};

int main()
{
```

```
    struct node ob1; // Node1
    // Intialization
    ob1.link = NULL;
    ob1.data1 = 10;
    ob1.data2 = 20;

    struct node ob2; // Node2
    // Initialization
    ob2.link = NULL;
    ob2.data1 = 30;
    ob2.data2 = 40;

    // Linking ob1 and ob2
    ob1.link = &ob2;

    // Accessing data members of ob2 using ob1
    printf("%d",  ob1.link->data1);
    printf("\n%d", ob1.link->data2);
    return 0;
}
```

Output:

30
40

UNIONS:

A union is a special data type available in C that allows to store different data types in the same memory location. You can define a union with many members, but only one member can contain a value at any given time. Unions provide an efficient way of using the same memory location for multiple-purpose.

Defining a Union:

To define a union, you must use the union statement in the same way as you did while defininga structure. The union statement defines a new data type with more than one member for your program.

Syntax:

```
union [union tag]
{
  member definition;
  member definition;
  …….
  member definition;
};
```

The union tag is optional and each member definition is a normal variable definition, such as int i; or float f; or any other valid variable definition. At the end of the union's definition, before the final semicolon, you can specify one or more union variables but it is optional.

Example:

```
union  Employee
 {
     int eno;
     char name[20];
     float salary;
  };
```

Accessing Union Members:

To access any member of a union, we use the member access operator (.). The member access operator is coded as a period between the union variable name and the union member that we wish to access.

Program:

```
#include<stdio.h>
#include<conio.h>
union Employee
{
  int eno;
  char ename[20];
  float esalary;
};
void main()
{
  union  Employee e={101,"Ravi",20000};
  printf("Emp no: %d",e.eno);
  printf("Emp name:%s",e.ename);
  printf("Emp salary:%f",e.esalary);
  getch();
}
```

FILES

In C programming, file is a place on your physical disk where information is stored.

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 you have to enter a large number of data, it will take a lot of time to enter them all. However, if you have a file containing all the data, you can easily access the contents of the file using few commands in C.
- You can easily move your data from one computer to another without any changes.

TYPES OF FILES:

When dealing with files, there are two types of files you should know about:

- Text files
- Binary files

Text files:

- Text files are the normal .txt files that you can easily create using Notepad or any simple text editors.
- When you open those files, you'll see all the contents within the file as plain text. You can easily edit or delete the contents.
- They take minimum effort to maintain, are easily readable, and provide least security and takes bigger storage space.

Binary files:

- Binary files are mostly the .bin files in your computer.
- Instead of storing data in plain text, they store it in the binary form (0's and 1's).
- They can hold higher amount of data, are not readable easily and provides a better security than text files.

C provides a number of functions that helps to perform basic file operations. Following are the functions,

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()	s et the position to the begining point

FILE OPERATIONS:

In C, you can perform four major operations on the file, either text or binary:

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

DECLARATION:

When working with files, you need to declare a pointer of type file. This declaration is neededfor communication between the file and program.

```
FILE *fptr;
```

OPENING A FILE:

Opening a file is performed using the library function in the "stdio.h" header file: fopen().

Syntax:

```
fptr=fopen(“filename/path”, “mode”);
```

Examples:

```
fopen("E:\\cprogram\\newprogram.txt","w");
fopen("file1.txt","w");
```

- Let's suppose the file newprogram.txt doesn't exist in the location E:\cprogram. The first function creates a new file named newprogram.txt and opens it for writing as per the mode 'w'.The writing mode allows you to create and edit (overwrite) the contents of the file.

CLOSING A FILE:

The file (both text and binary) should be closed after reading/writing.Closing a file is performed using library function fclose().

Syntax:

```
fclose(fptr); //fptr is the file pointer associated with file to be closed.
```

READING AND WRITING TO A TEXT FILE:

For reading and writing to a text file, we use the functions fprintf() and fscanf().They are just the file versions of printf() and scanf(). The only difference is that, fprint and fscanf expects a pointer to the structure FILE.

Example1:

```
 #include<stdio.h>
#include<conio.h>
void main()
{
 FILE *fp;
 char ch;
 fp=fopen("File1.txt","w");
 printf("Enter data");
 while((ch=getchar())!=EOF)
 {
   fputc(ch,fp);
 }
 fclose(fp);
  getch();
}
```

Example2:

```
#include<stdio.h>
#include<conio.h>
void main()
{
 FILE *fp;
 char ch;
 fp=fopen("File1.txt","r");
while((ch=fgetc(fp))!=EOF)
 {
   printf("%c",ch);
 }
 fclose(fp);
  getch();
}
```

Example3:

```
#include<stdio.h>
#include<conio.h>
void main()
{
 FILE *fp;
 char ch;
 fp=fopen("File1.txt","a");
 printf("Enter data");
 while((ch=getchar())!=EOF)
 {
   fputc(ch,fp);
 }
 fclose(fp);
  getch();
}
```

Example 4: To print number of characters and Lines in a file

```
#include<stdio.h>
#include<conio.h>
void main()
{
 FILE *fp;
 char ch;
 int lines=0,characters=0;
 fp=fopen("File1.txt","r");
while((ch=fgetc(fp))!=EOF)
 {
   printf("%c",ch);
   characters++;
   if(ch=='\n')
   {
      lines++;
```

```
   }
 }
printf("Numbers of characters : %d", characters);

printf("Numbers of Lines : %d", Lines);
 fclose(fp);
  getch();
}
```

feof():

Check end-of-file indicator Checks whether the end-of-File indicator associated with stream is set, returning a value different from zero if it is.This indicator is generally set by a previous operation on the stream that attempted to read at or past the end-of-file.

Program:to open a file and to print it contents on screen.

```
#include  <stdio.h>
#include <conio.h>
void main()
{
          FILE *fp;
          char s;
          clrscr();
          fp=fopen("File1.txt","r");
     while(1)
     {
         ch=fgetc(fp);
         if(feof(fp))
         {
            break;
         }
         printf("%c",ch);
      }
      fclose(fp);
      getch();
}
```

Program:to copy files

```
#include<stdio.h>
void main()
{
FILE *fp1,*fp2;
char ch;
fp1=fopen("File1.txt","r");
fp2=fopen("File2.txt", "w");
while((ch = fgetc(fp1))!= EOF)
{
    fputc(ch,fp2);
}
printf("File copied successfully");
fclose(fp2);
fclose(fp1);
getch();
}
```

Program:to merge two files and store the contents in another file

```
#include <stdio.h>
void main()
{
FILE *fp1, *fp2, *fp;
char ch;
clrscr();
fp1=fopen(sample1.txt,"r");
fp2=fopen(sample2.txt,"r");
fp=fopen(mergefile.txt,"w");
while((ch=fgetc(fp1))!=EOF)
{
   fputc(ch,fp);
}
while((ch=fgetc(fp2))!=EOF)
{
fputc(ch,fp);
}
printf("Two files were merged successfully.\n");
fclose(fp1);
fclose(fp2);
fclose(fp);
getch();
}
```

Random Access to File

There is no need to read each record sequentially, if we want to access a particular record.C supports these functions for random access file processing.

1. fseek()
2. ftell()
3. rewind()

1. fseek(): This function is used for seeking the pointer position in the file at the specified byte.

Syntax: fseek(file pointer, offset, pointer position);

Where,

file pointer ---- It is the pointer which points to the file.

offset ---- It is positive or negative.This is the number of bytes which are skipped backward (if negative) or forward(if positive) from the current position.This is attached with L because this is a long integer.

Pointer position:

This sets the pointer position in the file.

Value	pointer position
0	Beginning of file.
1	Current position
2	End of file

Example:

```
 #include<stdio.h>
#include<conio.h>
void main()
{
    FILE *fp;
    char ch;
    clrscr();
    fp=fopen("file1.c", "r");
      printf("Enter value of n  characters");
      scanf("%d",&n);
      fseek(fp,-n,2);
      while((ch=fgetc(fp))!=EOF)
      {
          printf("%c\t",ch);
        }
 fclose(fp);
 getch();
}
```

ftell():

This function returns the value of the current pointer position in the file.The value is count from the beginning of the file.

Syntax: ftell(fptr);

Where fptr is a file pointer.

rewind():

This function is used to move the file pointer to the beginning of the given file.

Syntax: rewind(fptr); // Where fptr is a file pointer.

Example:

```
 #include<stdio.h>
#include<conio.h>
void main()
{
    FILE *fp;
    char ch;
    int n;
    fp=fopen("file1.c", "r");
    n=ftell(fp);
    printf("Current position: %d",n);
```

```
 rewind(fp);
printf("Current position: %d",ftell(fp));
 fclose(fp);
 getch();
}
```

typedef in C

The **typedef** is a keyword used in C programming to provide some meaningful names to the already existing variable in the C program. It behaves similarly as we define the alias for the commands. In short, we can say that this keyword is used to redefine the name of an already existing variable.

Syntax

```
typedef <existing_name> <alias_name>
```

In the above syntax, '**existing_name**' is the name of an already existing variable while '**alias name**' is another name given to the existing variable.

Example:

```
 void main()
{
 typedef int cse;
 cse a=5,b=6,c;
 c=a+b;
 printf("%d",c);
getch();
}
```

Enumeration (or enum) in C:

Enumeration (or enum) is a user defined data type in C. It is mainly used to assign names to integral constants, the names make a program easy to read and maintain.

Example:

```
#include<stdio.h>
enum week{Mon, Tue, Wed, Thur, Fri, Sat, Sun};
int main()
{
   enum week day;
   day = Wed;
   printf("%d",day);
   return 0;
 }
```