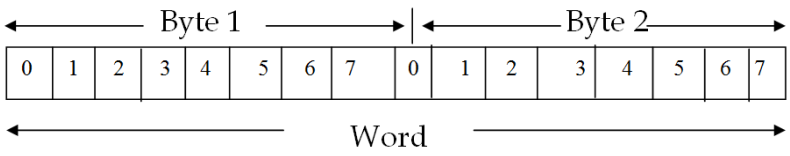
UNIT V

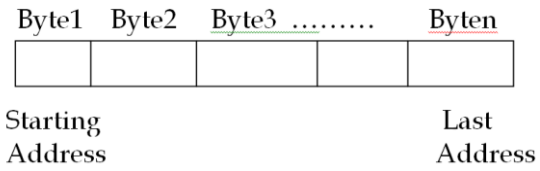
POINTERS:

Data representation in memory:

- Sets of one or more bytes are used to store data objects.
- Each byte in a computer memory is called memory location or memory cell.



- Each memory location is numbered for its reference.
- The number attached to a memory location is called the memory address of that location.
- The memory address is merely a positive number and not an integer.



- A word must have at least 8 bits and 16 bits respectively.

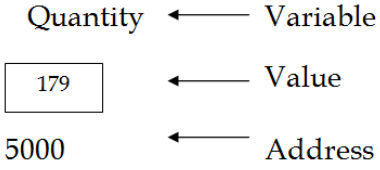
Pointers:

- A pointer value or simply a pointer is a data object that refers to a memory location which is an address.
- It resides in the memory locations, which is represented by the pointers variable.
- A pointer variable may contain the address of another variable or any valid address in the computer memory.
- Pointer variables use two bytes or four bytes of memory locations to hold the addresses.

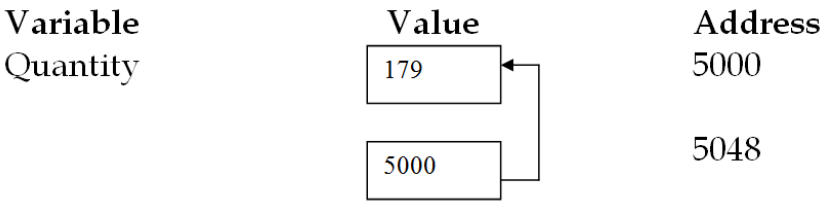
Consider the following statement:

int quantity = 179;

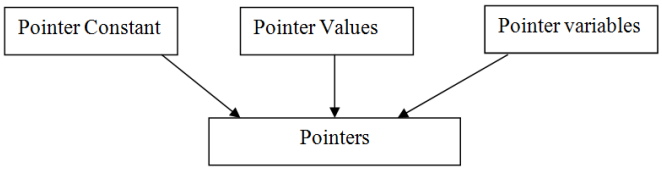
- This statement instructs the system to find a location for the integer variable quantity and put the value 179 in that location.



- To access the value 179 by using either the name quantity or the address 5000. The memory addresses are simply numbers they can assigned to variables like any other variable. Such variable that hold memory addresses are called pointer variables.
- Pointer is a variable; its value is also stored in the memory in another location.



- The value of the variable p is the address of the variable quantity, we may access the value of quantity, we may access the value of quantity by using the value p and therefore we say that the variable p 'points' to the variable quantity. Thus, p gets the name 'pointer'.
- Pointers are built on the three underlying concepts.



Accessing the address of a variable:

- The operator & immediately preceding a variable returns the address of the variable associated with it.

  P = &quantity; Would assign the address 5000 to the variable p.

  & Operator called as address of operator.

- The & Operator can be used only with a simple variable or an array element.

Valid	**Invalid**
&x	&125
int x[10];	int x[10];
&x[0]	&x
&x[i+3]	&(x+y)

/* Example Program */

```
main()
{
Char a;
int x;
```

```
float p,q;
a='A';
x=125;
p=10.25, q=18.76;
printf("%c is stored at address %u \n",a,&a);
printf("%d is stored at address %u\n",x,&x);
printf("%f is stored at address %u\n",p,&p);
printf("%f is stored at address %u \n",q,&q);
}
```

Declaring Pointer variables:

Syntax:

```
Data_type *pt_name;
```

- The asterisk (*) tells that the variable pt_name is a pointer variable.
- Pt_name is a pointer variable.
- Pt_name needs a memory location.
- Pt_name points to a variable of type data_type.

Example:

```
int *p;          /* Integer pointer */
```

- declares the variables p as a pointer variable that points to an integer data type.
- Type int refers to the data type of the variable being pointed by p.
- Different styles to declare pointer variables:

```
int *   p;          /*Style 1*/
int   *p;  /* Style 2*/
int  * p;  /* Style 3*/
```

- Style 2 is becoming popular.
- It is convenient to have multiple declaration

```
int *p,x,*q;
```

- This style matches with the format used for accessing the target values.

```
int *p,x,y;
x=10;
p=&x;
y=*p;
*p=20;
```

Initialization of Pointer Variables:

- The process of assigning the address of a variable to a pointer variable is known as initialization.

- Uninitalized pointers will have some unknown values.

Example:

```
int quantity;
int *p;
p=&quantity;
```

- we can also combine the initialization on with the declaration

  int *p = &quantity; is allowed

- The variable quantity must be declared before the initialization takes place.
- We must ensure that the pointer variable always point to the corresponding type of data.

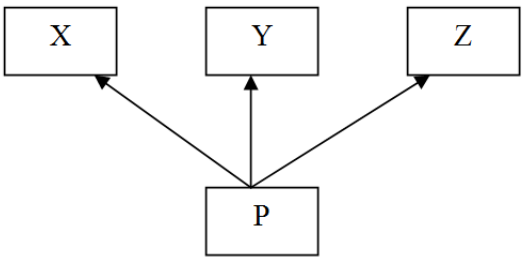
```
float a,b;
int x,*p;
p=&a;
```

- b=*p; will result in erroneous output, because we are trying to assign the address of a float variable to an integer pointer.
- int x, *p = &x; /* Three in one is perfectly valid */
- X is an integer variable and p as a pointer variable and then initializes p to the address of x.
- int *p=&x,x; is not valid.
- The pointer variable with an initial value of NULL or 0.

```
int *p=NULL;
int *p=0;
```

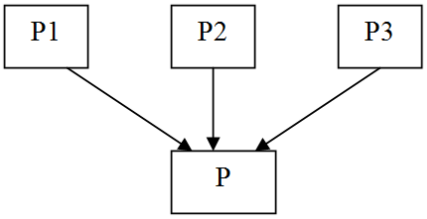
- No other constants value can be assigned the pointer variable.
- Pointers are flexible.
- We can make the same pointer to point to different data variables in different statements.

```
int x,y,z,*p;
P=&x;
…
P=&y;
…
P=&z;
…
```



- We can also use different pointers to point to the same data variable.

```
int x;
int *p1=&x;
int *p2=&x;
int *p3=&x;
```



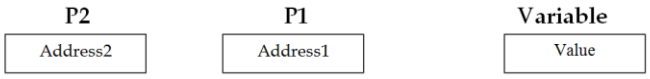
Assigning a variable through its pointer:

To access the value of the variable using the pointer. This is done by using unary operator * also known as indirection operator or dereferencing operator.

```
int quantity ,*p,n;
quantity=179;
p=&quantity;
n=*p;
main()
{
 int x,y;
 int *ptr;
 x=10;
 ptr=&x;
 y=*ptr;
 printf("Value of x is %d \n\n ", x);
 printf(" %d is stored at address %u \n",x,&x);
 printf(" %d is stored at address %u \n",*&x,&x);
 printf(" %d is stored at address %u \n",*ptr,ptr);
 printf(" %d is stored at address %u \n",ptr,&ptr);
 printf(" %d is stored at address %u \n",y,&y);
 *ptr=25;
 Printf("\n Now x = %d\n",x);
}
```

Chain of pointers:

- It is possible to make a pointer to point to another pointer, thus creating a chain of pointers.



- The pointer variable p2 contains the address of the pointer variable p1, which points to the location that contains the desired value. This is known as multiple indirection.
- A variable that is a pointer to a pointer must be declared using additional indirection operators symbols.

  int **p2;

- pointer p2 is not a pointer to an integer, bur rather a pointer to an integer pointer.

Example Program

```
main()
{
  int x,*p1,**p2;
   x=100;
```

```
    p1=&x;
    p2=&p1;
    printf("%d",**p2);
}
```

✍ This code will display the value 100.

POINTER EXPRESSION:

- Pointer variables can be used in expressions.

```
Y=(*p1) * (*p2);
Sum=sum+(*p1);
Z=(5* (-(*p2)))/*p1;
*p2=*p2 + 10;
```

- We may also use short-hand operators with the pointers.

```
P1++;
-p2;
Sum+=*p2;
```

✍ Pointers can also be compared using the relational operators, such as

P1 < p2, p1 <=p2, p1 >= p2, P1 > p2, p1 == p2 and p1 !=p2 are allowed.

✍ We may not use pointers in division and multiplication.

P1/p2 or p1 * p2 or p1/3 are not allowed.

Example:

```
main()
{
int a,b,*p1,*p2,x,y,z;
a=12;
b=4;
p1=&a;
p2=&b;
x=(*p1) * (*p2) – 6;
y=(4 * (-(*p2)))/(*p1) + 10;
printf("Address of a = %u \n",p1);
printf("Address of b = %u \n",p2);
printf("\n");
printf("a=%d ,  b=%d \n",a,b);
printf("x=%d , y=%d \n ", x,y);
*p2=*p2 + 3;
*p1= *p2 – 5;
```

```
Z= *p1 * p2 – 6;
Printf("\n a = %d , b = %d", a,b);
Printf("Z = %d \n",z);
}
```

Pointer Increments and Scaled Factor:

- Pointers can be incremented like

```
P1 = p2+2;
P1=p1 +1;
```

and so on.

```
P1++;
```

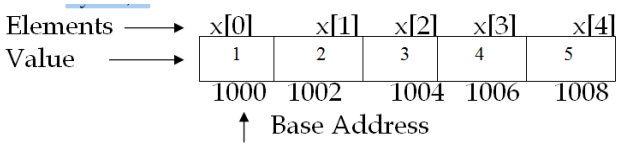
- Will cause the pointer p1 to point to the next value of its type.
- If p1 is an integer pointer with an initial value say 2800, then after the operation p1=p1+1, the value of p1 will be 2802, and not 2801.
- We increment a pointer , its value is increased by the 'length' of the data type. This length called as scale factor.

Pointers and Array:

- When an array is declared, the compiler allocates a base address and sufficient amount of storage to contain all the elements of the array in contiguous memory locations.
- The base address also defines the array name as a constant pointer to the first element.
- We declare x is an array

```
int x[5] = {1,2,3,4,5};
```

- The base address of x is 1000 and that each integer requires two bytes,



- This name x is defined as a constant pointer pointing to the first element, x[0]

```
x=&x[0] = 1000
```

- Pointer variable p is used to point to the array x.

```
P=x;   equivalent to p = &x[0];
```

- We can access every value of x using p++ to move from one element to another.

Address of x[3]=base address +(3 x scale factor of int)

```
= 1000 + (3 x 2) = 1006
p= &x[0] (=1000)
p+1 = &x[1] (=1002)
p+2 = &x[2] (=1004)  …
```

*(p+2) gives the value of x[2].

- It is faster than array indexing.

Program:

```
Main()
{
 int *p,sum;
 int x[5]={5,9,6,3,7};
 i=0;
 p=x;
 printf("Element value address \n\n");
 while(i<5)
{
  printf("x[%d] %d %u \n",i,*p,p);
  sum=sum+(*p);
  i++; p++;
}
Printf("\n sum = %d \n",sum);
Printf("\n &x[0] = %u\n",&x[0]);
Printf("\n P = %u \n",p);
}
```

- similarly we can access two-dimensional array such s
- one-dimensional array values accessed by

  *(x+i) or *(p+i)

- represents the element x[i]

Two-dimensional Array values are accessed by

((a+i)+j) or *(*(p+i)+j)

- p->Pointer to first row
- p+i ->pointer to ith row
- *(p+i) -> pointer to first element in the ith row.
- *(p+i)+j ->pointer to jth element in the ith row.
- *(*(p+i)+j) -> value stored in the cell(i,j)

  (ith row and jth column)

Pointers and character strings:

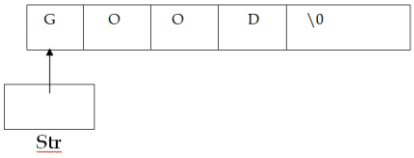
- Strings are treated like character arrays and therefore they are declared and initialized as

  Char str[5]="good";

- The compiler automatically inserts the null character '\0' at the end of the string.

  Char *str ="good";

- Creates a string for the literal and then stores its address in the pointer variable str.
- The pointer str now points to the first character of the string.



- We can use the run-time assignment for giving values to a string pointer.

  Char *string1;

  String1 = "good";

- Is not a string copy, because the variable string1 is a pointer, not a string.
- We can print the content of the string string1 using either printf or puts.

  Printf("%s",string1);

  Puts(string1);

Example Program

```
/*Program to find length of a character string using pointer */
main()
{
 char *name;
 int length;
 char *cptr=name;
 name="DELHI";
 printf("%s\n",name);
 while(*cptr!='\0')
{
 printf("%c is stored at address %u \n",*cptr,cptr);
  cptr++;
}
length=cptr-name;
printf("\n Length of the string = %d \n",length);
}
```

Output:

```
DELHI
D is stored at address 54
E is stored at address 55
L is stored at address 56
H is stored at address 57
```

I is stored at address 58

Length of the string = 5.

Array of Pointers:

Array of strings:

Char name[3][15];

- Name is a table containing three names, each with a maximum length of 15 characters.
- The total storage requirements for the name table are 45 bytes.
- Make a pointer to a string of varying length

Char *name[3]={“India”,“SalemDt”,“Chennai”};

- declares name to be an array of three pointers to characters.
- Each pointer pointing to a particular name.

Name[0] India

Name[1] Salemdt

Name[2] Chennaist

- This declaration allocates only 28 bytes, sufficient to hold all the characters.
- Print all three names as

```
for(i=0;i<=2;i++)
 printf(“%s\n”,name[i]);
```

Pointers as Function Arguments:

- We can pass the address to a function, the parameters receiving the addresses should be pointers.
- The process of calling a function using pointers to pass the addresses of variable is known as “Call by reference”.

Example Program

```
main()
{
 int x;
 x=20;
 change(&x);
 printf(“%d\n”,x);
}
Change(int *p)
{
 *p=*p+10;
}
```

Points to Remember:

- The function parameters are declared as pointers
- The dereferenced pointers are used in the function body.
- When the function is called, the addresses are passed as actual arguments.

Functions Returning pointers:

- Function can return a single value by its name or return multiple values through pointer parameters.
- Function to return a pointer to the calling function.

```
int  *larger(int *,int *)
main()
{
 int a=10;
 int b=20;
 int *p;
 p=larger(&a,&b);
printf(" %d",*p);
}
```

POINTERS TO FUNCTION:

- A function like a variable, has a type and an address location in the memory.
- To declare a pointer to a function, which can then be used as an argument in another function.

Syntax:

```
Type (*fptr)();
```

- fptr is a pointer to a function, which returns type value.
- The parentheses around *fptr are necessary.

```
type *gptr();
```

- would declare gptr as a function returning a pointer to type.
- We can make a function pointer to point to a specific function by simply assigning the name of the function to the pointer.

```
double mul(int,int);
double (*p1)();
p1=mul;
```

- declare p1 as a pointer to a function and mul as a function and then make p1 to point to the function mul.
- To call the function mul,

(*p1)(x,y) is equivalent to mul(x,y)

Example Program

```
#include<math.h>
#define PI  3.1415
Double y(double);
Double cos(double);
Double table(double (*f)(),double,double,double);
Main()
{
  Printf("Table of y(x) = 2*x*x-x+1 \n\n");
Table(y,0.0,2.0,0.5);
Printf("\n Table of cos(x) \n\n");
Table(cos,0.0,PI,0.5);
}
Double table(double (*f)(),double min, double max, double step)
{
 Double a,value;
 for(a=min;a<=max;a+=step)
  {
  Value=(*f)(a);
  Printf("%5.2f %10.4f",a,value);
}
}
Double y(double x)
{
  Return(2*x*x-x+1);
}
```

Pointers and structures:

```
Struct inventory
{
 char name[30];
 int number;
float price;
}product[2],*ptr;
```

- Product as an array of two elements, each of the type struct inventory and ptr is a pointer to data objects of type struct inventory.

```
Ptr=product;
```

- Would assign the address of the zeroth element of product to ptr. The pointer ptr will now point to product[0].

- Its members can be accessed using

  Ptr->name

  Ptr->number

  Ptr->price

- The symbol -> is called the arrow operator (also known as member selection operator) and is made up of minus sign ang greater than sign.
- When the pointer ptr is incremented by one, it is made to point to the next record.

  for(ptr=product;ptr<product+2;ptr++)

  printf("%s%d%f\n",ptr->name,ptr->number,ptr->price);

- (*ptr).number is also use to access the member number. The parentheses around *ptr are necessary because the member operator '.' has a higher precedence than the operator *.

/* Example Program */

```
Struct invent
{
  Char *name[20];
 Int  number;
Float price;
};
main()
{
Struct invent product[3],*ptr;
Printf("Input \n\n");
for(ptr=product;ptr<product+3;ptr++)
  scanf("%s%d%f",ptr->name,&ptr->number,&ptr->price);
printf("\n  Output \n\n");
ptr=product;
while(ptr<product+3)
{
   Printf("%-20s %5d %10.2f\n",ptr->name,ptr->number,ptr->price);
Ptr++;
} } }
```

FILE MANAGEMENT IN C

Introduction:

- Functions such as scanf and printf to read and write data.
- These are console oriented I/O Functions, which always use the terminal as the target place.
- This works fine as long as the data is small.

Disadvantages:

- It take time consuming to handle large volumes of data through terminals.
- The entire data is lost when either the program is terminated or the computer is turned off.
- More flexible approach where data can be stored on the disks and read whenever necessary, without destroying the data.
- A file is a place on the disk where group of related data is stored.

Basic file Operations:

1) Naming a file
2) Opening a file.
3) Reading data from the file.
4) Writing data to a file and
5) Closing a file.

Fopen()	Creates a new file for use opens an existing file for use.
Fclose()	Closes a file which has been opened for use.
Getc()	Reads a character from a file.
Putc()	Writes a character to a file.
Fprintf()	Writes a set of data values to a files
Fscanf()	Reads a set of data values from a files
Getw()	Reads an integer to a file
Putw()	Writes an integer to a file
Fseek()	Sets the position to the desired point in the file.
Ftell()	Gives the current position in the file
Rewind()	Set the position to the beginning of the file

Defining and opening a file:

If we want to store data in a file in the secondary memory, we must specify the

- Filename
- Data structure
- Purpose to the operating system

Filename is a string of characters

It may contains two parts

1) Primary name
2) Extension

Example:

```
Student.c
Text.out
Store     /* Extension is optional */
```

- Data structure of a file is defined as FILE in the library of standard I/O Function definitions.
- All files should be declared as type FILE.
- FILE is a defined data type.
- When we open a file, we must specify what we want to do with the file.

Declaring and opening a file:

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

- Fp is a pointer variable to the data type "FILE".

- FILE is a structure defined in the I/O library.
- The second statement assigns the filename and purpose of opening this file to the FILE type pointer fp.

Mode can be one of the following:

1	**R**	Opening the file for reading only. If the purpose is 'reading' and if it exists, then the file is opened with the current contents safe otherwise an error occurs.
2	**W**	Open the file for writing only. When the mode is 'writing' a file with the specified name is created if the file does not exist. The contents are declared, if the file already exists.
3	**A**	Open the file for appending data to it. When the purpose is 'appending', the file is opened with the current content safe.

Example:

```
FILE     *p1,*p2;
P1=fopen("data","r");
P2=fopen("results","w");
```

- The file data is opened for reading and results is opened for writing.

Additional Modes of operation:

1	**R+**	The existing file is opened to the beginning for both reading and writing.
2	**W+**	Same as w except both for reading and writing.
3	**A+**	Same as a except both for reading and writing

Closing a file:

- A file must be closed as soon as all operations on it have been completed.
- It also prevents any accidental misuse of the file.
- Closing of unwanted files might help open the required files.
- The I/O Library supports a function:

```
fclose(file-pointer);
```

- This would close the file associated with the FILE pointer file-pointer.

```
…
…
FILE   *p1,*p2;
P1=fopen("INPUT","W");
P2=fopen("OUTPUT","r");
…
…
fclose(p1);
fclose(p2);
```

- This program opens two files and closes then after all operations on them are completed.

Input/output operations on files:

- The simplest file I/O functions are getc and putc.
- It is used to handle one character at a time when the file is open.

Putc(c,fp1);

- Writes the character contained in the character variable c to the file associated with FILE pointer fp1.
- Getc() is used to read a character from a file that has been opened in read mode.

C=getc(fp2);

- Would read a character from the file whose file pointer is fp2.

/* Example program */

```
#include <stdio.h>
main()
{
 FILE *f1;
Char c;
Printf("DATA INPUT \n\n");
/* Opening the file Input */
f1=fopen("INPUT","W");
/* Get a character from keyboard */
while((c=getchar()) !=EOF)
/* Writes a character to INPUT */
putc(c,f1);
/*close the file INPUT */
fclose(f1);
printf("\n Data output ");
f1=fopen("Input","r");
/* Read a character from INPUT */
while((c=getc(f1))!=EOF)
/* Display a character on screen */
printf("%c",c);
/*close the file Input */
fclose(f1);
}
```

The getw and putw Functions:

- Getw and putw are integer-oriented functions:
- These functions only used for integer data.

```
Putw(integer,fp);
Getw(fp);
#include<stdio.h>
main()
{
FILE *f1,*f2,*f3;
int number,i;
printf("Contents of data files \n");
```

```
f1=fopen("DATA","W");
for(i=1;i<=30;i++)
{
 scanf("%d",&number);
 if(number==-1) break;
 putw(number,f1);
}
fclose(f1);
f1=fopen("DATA","r");
f2=fopen("ODD","w");
f3=fopem("EVEN","w");
while(number=getw(f1)!=EOF)
{
 if(number%2==0)
   putw(number,f3);
else
 putw(number,f2);
}
fclose(f1);  fclose(f2); fclose(f3);
f2=fopen("ODD","r");
f3=fopen("EVEN","r");
printf("\n\n Contents of ODD file \n\n ");
while(number=getw(f2))!=EOF)
 printf("%4d",number);
printf("\n\n contents of EVEN file \n\n");
while((number=getw(f3))!=EOF)
 printf("%4d",number);
fclose(f2);
fclose(f3);
}
```

fprintf and fscanf Functions:

- These functions can handle a group of mixed data simultaneously.
- It is identical to the familiar printf and scanf functions.
- Except the first argument of these functions is a file pointer which specifies the file to be used.

  fprintf(fp,"control string",list);
- Where fp is a file poiner associated with a file that has been opened for writing.
- The control string contains output specifications for the items in the list.
- The list may include variables, constants and strings.
- Reading of the items in the list from the file specified by fp, according to the specifications contained in the control string.

```
fscanf(f2,"%s%d",item,&quantity);
```

Example Program

```
#include<stdio.h>
main()
{
 FILE *fp;
 int number,quantity,I;
 float price,value;
 char item[10],filename[10];
 printf("Input file name \n");
 scanf("%s", filename);
 fp=fopen(filename,"w");
 printf("Input inventory data \n\n");
 printf("Item name number price quantity \n");
 for(i=1;i<=3;i++)
{
 fscanf(stdin,"%s%d%f%d",item,&number,&price,&quantity);
fprintf(fp,"%s%d%.2f%d",
                item,number,price,quantity);
}
fclose(fp);
fprintf(stdout,"\n\n");
fp=fopen(filename,"r");
printf("Itemname Number, Price, Quantity, Value \n");
for(i=1;i<=3;i++)
{
 fscanf(fp,"%s%d%f%d",item,&number,&price,&quantity);
value=price*quantity;
fprintf(stdout,"%8s%7d%8.2f%8d%11.2f",item,number,price,quantity,
                                                        value);
}
fclose(fp); }
```

Error Handling During I/o Operations:

- Errors occurs during I/O Operations on a file.
- Trying to read beyond the end-of-file mark.
- Device overflow.
- Trying to use a file that has not been opened.
- Opening with invalid filename.
- Attempting to write to a write-protected file.
- Trying to perform an operation on a file, when the file is opened for another type of operation.

- Feof and ferror are library functions that can help us detect I/O errors in the files.
- The feof is used to test for an end of file condition.
- It takes a FILE Pointer as its only argument and returns a non-zero integer value if all of the data from the specified file has been read and returns zero otherwise.

```
if(feof(fp))
        printf("End of data \n");
```

- would display the message "End of data".
- The ferror function reports the status of the file indicated.
- It also takes a FILE pointer as its argument and returns a nonzero integer if an error has been detected, it returns zero otherwise.

```
        If(ferror(fp)!=0)
        Printf("A error has occurred \n");
```

/* Program for error handling */

```
#include<stdio.h>
main()
{
  char *filename;
FILE *fp1,*fp2;
int i,number;
fp1=fopen("TEST","W");
for(i=10;i<=100;i+=10)
   putw(i,fp1);
fclose(fp1);
printf("\n Input Filename \n");
open-file:
  scanf("%s",filename);
if(fp2=fopen(filename,"r"))=NULL)
{
   printf("Cannot open the file \n");
   printf("Type filename again \n\n");
  goto open-file;
}
else
  for(i=1;i<=20;i++)
{
  number=getw(fp2);
  if(feof(fp2))
  {
    printf("\n Ran out of data \n");
  break;
}
```

```
else
  printf("%d\n",number);
}
fclose(fp2);   }
```

Random Access to Files:

- File functions that are useful for reading and writing data sequentially.
- Accessing only a particularly part of a file and not in reading the other parts.
- This can be achieved with the help of the functions fseek, ftell, and rewind available in the I/O Library.
- Ftell takes a file pointer and return a number of type long, that corresponds to the current position.
- This function is useful in saving the current position of a file.

```
n=ftell(fp);
```

- N would give the relative offset (in bytes) of the current position. This means that n bytes have already been read (or written).
- Rewind takes a file pointer and resets the position to the start of the file.
- For example, the statement

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

- would assign 0 to n because the file position has been set to the start of the file by rewind.
- fseek function is used to move the file position to a desired location within the file.
- It takes the following form:

```
fseek(file_ptr,offset,position);
```

- file_ptr is a pointer to the file concerned, offset is a number or variable of type long, and position is an integer number.
- The offset specifies the number of positions to be moved from the location specified by position.
- The position can take one of the following three values:

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

- The offset may be positive, meaning move forwards, or negative, meaning move backwards.

Operations of fseek function:

Statement	**Meaning**
fseek(fp,0L,0);	Go to the beginning.
fseek(fp,0L,1);	Stay at the current position
fseek(fp,0L,2);	Go to the end of the file, past the last character of the file
fseek(fp,m,0)	Move to (m+1)th byte in the file
fseek(fp,m,1);	Go forward by m bytes
fseek(fp,-m,1);	Go backward by m bytes from the current position

fseek(fp,-m,1);	Go backward by m bytes from the end
fseek(fp,-m,2);	Go backward by m bytes from the end

Example Program:

```
# include<stdio.h>
main()
{
        FILE *fp;
        long n;
        char c;
        fp=fopen("RANDOM","w");
        while((c=getchar())!=EOF)
                putc(c,fp);
        printf("No. of characters entered = %d\n",ftell(fp));
        fclose(fp);
        fp=fopen("RANDOM","r");
        n=0L;
        while(feof(fp)==0)
        {
                fseek(fp,n,0); /* Position to (n+1)th character */
                printf("Position of %c is %ld\n",getc(fp),ftell(fp));
                n=n+5L;
}
Putchar('\n');
fseek(fp,1L,2);
do
{
        Putchar(getc(fp));
}
While(!fseek(fp,-2L,1));
fclose(fp);
}
```

Output:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ^Z
No. of characters entered = 26
Position of A is 0
Position of F is 5
Position of K is 10
Position of P is 15
Position of U is 20
Position of Z is 25
Position of          is 30
```

ZYXWVUTSRQPONMLKJIHGFEDCBA

Command Line Arguments:

- ☼ The function main() in c can also pass arguments or parameters like the other functions.
- ☼ It has two arguments argc(for argument count) and argv(for argument vector.

Syntax:

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

Where

- Argc represents the number of arguments.
- Argv is a pointer to an array of string s or pointer to pointer to character.

- ☼ The argument argv is used to pass strings to the programs.
- ☼ The arguments argc and argv are called as program parameters.
- ☼ After successful compilation, the program is executed by using an execution command.

Ex: a.out

- ✍ Actually, the execution command is a command line
- ✍ The values of the program parameters argvare given in the command line.
- ✍ The value of argc is automatically counted by the compiler when main() is invoked.
- ✍ Since the values are available and obtained from the command line, they are also known as command-line arguments.
- ✍ The values are obtained from the command line itself.
- ✍ a.out line plane circle
- ✍ passes the string constants a.out,line plane and circle to the program

In case argc is 4 and

- Argv[0] is a.out
- Arv[1] is pointer to first character 1 in the string constant line.
- Argv[2] string plane
- Argv[3] string circle
- Argv[4] is always null pointer to mark the end of the argv array.

Example Program

```
#include <stdio.h>
main(int argc,char *argv[])
{
FILE *fp;
int i;
char word[15];
```

```
fp=fopen(arv[1],"w");
printf("\n No. of arguments in command line = %d\n\n",argc);
for(i=2;i<argc;i++)
fprintf(fp,"%s",argv[i]);
fclose(fp);
/* Writing contents of the file to screen */
Printf("Contents of %s file \n\n\n",argv[i]);
fp=fopen(argv[1],"r");
for(i=2;i<argc;i++)
{
  Fscanf(fp,"%s",word);
Printf("%s",word);
}
Fclose(fp);
Printf("\n\n");
/* Writing the arguments from memory */
for(i=0;i<argc;i++)
 printf("%*s\n",i*s,argv[i]);
}
```

OUTPUT

```
c>F12_7 TEXT AA      BB      CC      DD      EE      FF
No. of arguments in command line = 9
Contents of Text file
AA      BB      CC      DD      EE      FF
C:\C\F12_7.exe
Text
        AA              BB              CC              DD              EE              FF
```