1. What is File management?

- In real life, we want to store data permanently so that later on we can retrieve it and reuse it.
- A file is a collection of bytes stored on a secondary storage device like hard disk, pen drive, and tape.
- There are two kinds of files that programmers deal with text files and binary files.
- Text file are human readable and it is a stream of plain English characters.
- Binary files are not human readable. It is a stream of processed characters and ASCII symbols.

2. Explain File opening mode.

- We want to open file for some purpose like, read file, create new file, append file, read and write file, etc…
- When we open any file for processing, at that time we have to give file opening mode.
- We can do limited operations only based on mode in which file is opened.

  e.g. fp = fopen(“demo.txt”,”r”); //Here file is opened in read only mode.

  C has 6 different file opening modes for text files:

  1) **r** open for reading only.
  2) **w** open for writing (If file exists then it is overwritten)
  3) **a** open for appending (If file does not exist then it creates new file)
  4) **r+** open for reading and writing, start at beginning
  5) **w+** open for reading and writing (overwrite file)
  6) **a+** open for reading and writing, at the end (append if file exists)
- Same modes are also supported for binary files by just adding **b**, e.g. **rb, wb, ab, r+b, w+b, a+b**

3. Write a C program to display file on screen.

```
#include <stdio.h>
void main()
{
        FILE *fp;                   // fp is file pointer. FILE is a structure defined in stdio.h
        char ch ;
        fp = fopen(“prog.c”, “r”);      // Open prog.c file in read only mode.
        ch = getc(fp) ;
        while (ch != EOF)               // EOF = End of File. Read file till end
        {
                putchar(ch);
                ch = getc (fp);
                //Reads single character from file and advances position to next character
        }
        fclose(fp);                     // Close the file so that others can access it.
}
```

4. Write a C program to copy a file.

```
#include <stdio.h>
void main()
{
        FILE *p,*q;
        char ch;
        p = fopen(“Prog.c”,"r");
        q = fopen(“Prognew.c”,"w");
        ch = getc(p);
        while(ch != EOF)
        {
                putc(ch,q);
                ch = getc(p);
        }
        printf("File is copied successfully. ");
        fclose(p);
        fclose(q);
        return 0;
}
```

5. Explain file handing functions with example.

- C provides a set of functions to do operations on file. These functions are known as file handling functions.
- Each function is used for some particular purpose.

1) fopen() (Open file)

- fopen is used to open a file for operation.
- Two arguments should be supplied to fopen function,
- File name or full path of file to be opened
- File opening mode which indicates which type of operations are permitted on file.
- If file is opened successfully, it returns pointer to file else NULL.

Example: fp = fopen(“Prog.c”,”r”); // File name is prog.c and it is opened for reading only.

2) fclose() (Close file)

- Opened files must be closed when operations are over.
- The function fclose() is used to close the file i.e. indicate that we are finished processing this file.
- To close a file, we have to supply file pointer to fclose() function.
- If file is closed successfully then it returns 0 else EOF.

Example: fclose(fp);

3) fprintf() (Write formatted output to file)

- The fprintf() function prints information in the file according to the specified format.
- fprintf() works just like printf(), only difference is we have to pass file pointer to the function.
- It returns the number of characters outputted, or a negative number if an error occurs.

Example: fprintf(fp, ”Sum = %d”, sum);

4) fscanf() (Read formatted data from file)

- The function fscanf() reads data from the given file.
- It works in a manner exactly like scanf(), only difference is we have to pass file pointer to

the function.

- If reading is succeeded then it returns the number of variables that are actually assigned values, or EOF if any error occurred.

Example: fscanf(fp, ”%d”, &sum);

5) **fseek() (Reposition file position indicator)**

- Sets the position indicator associated with the file pointer to a new position defined by adding offset to a reference position specified by origin.
- You can use fseek() to move beyond a file, but not before the beginning.
- fseek() clears the EOF flag associated with that file.
- We have to supply three arguments, file pointer, how many characters, from which location.
- It returns zero upon success, non-zero on failure.

The origin value should have one of the following values

Name	**Explanation**
SEEK_SET	Seek from the start of the file
SEEK_CUR	Seek from the current location
SEEK_END	Seek from the end of the file

Example: fseek(fp,9,SEEK_SET); // Moves file position indicator to 9th position from begging.

6) **ftell() (Get current position in file)**

- It returns the current value of the position indicator of the file.
- For binary streams, the value returned corresponds to the number of bytes from the beginning of the file.

Example: position = ftell (fp);

7) **rewind() (Set position indicator to the beginning)**

- Sets the position indicator associated with file to the beginning of the file.
- A call to rewind is equivalent to: fseek (fp, 0, SEEK_SET);
- On file open for update (read+write), a call to rewind allows to switch between reading and writing.

Example: rewind (fp);

8) **getc() (Get character from file)**

- getc function returns the next character from file or EOF if the end of file is reached.
- After reading a character, it advances position in file by one character.
- getc is equivalent to getchar().
- fgetc is identical to getc.

Example: ch = getc(fp);

9) **putc() (Write character to file)**

- putc writes a character to the file and advances the position indicator.
- After reading a character, it advances position in file by one character.
- If there are no errors, the same character is returned; if error occurs then EOF is returned.

- putc is equivalent to putchar().
- fgetc is identical to putc.

Example: putc(ch, fp);

10) getw() (Get integer from file)

- getw function returns the next int from the file. If error occurs then EOF is returned.

Example: i = getw(fp);

11) putw() (Write integer to file)

- putw function writes integer to file and advances indicator to next position.
- It succeeded then returns same integer otherwise EOF is returned.

Example: putw(I, fp);

12) feof()

- feof() function returns non-zero value only if end of file has reached otherwise it returns 0.

Example : feof(fp)

6. Write a program to count the number of lines, number of tabs, characters and words in a given file.

```
#include <stdio.h>
void main()
{
        FILE *fp;
        int lines=0, tabs=0, characters=0, words = 0;
        char ch, filename[100] ;
        printf(“Enter file name: “);
        gets( filename );        // You can also use scanf(“%s”,filename);
        fp = fopen( filename, “r” );
        if ( fp == NULL )                        // File is not opened successfully then it returns NULL.
        {
                printf(“Cannot open %s for reading \n”, filename );
                exit(1);       /* terminate program */
        }
        ch = getc( fp ) ;
        while (  ch != EOF )
        {
                if ( ch  ==  ‘\n’  )
                        lines++ ;
                else if ( ch  ==  ‘\t’  )
                        tabs ++ ;

                else if ( ch  ==  ‘ ’  )
                        words ++ ;
                else
                        characters++;
                c = getc ( fp );
        }
fclose( fp );
printf(“Lines=%d Tabs=%d Words=%d Characters=%d”, lines, tabs, words, characters);
}
```

7. What is difference between Static Memory Allocation and Dynamic Memory Allocation

Static Memory Allocation	Dynamic Memory Allocation
• If memory is allocated to variables before execution of program starts then it is called static memory allocation.	• If memory is allocated at runtime (during execution of program) then it is called dynamic memory.
• It is fast and saves running time.	• It is bit slow.
• It allocates memory from stack.	• It allocates memory from heap
• It is preferred when size of an array is known in advance or variables are required during most of the time of execution of program.	• It is preferred when number of variables is not known in advance or very large in size.
• Allocated memory stays from start to end of program.	• Memory can be allocated at any time and can be released at any time.
• The storage space is given symbolic name known as variable and using this variable we can access value.	• The storage space allocated dynamically has no name and therefore its value can be accessed only through a pointer.

8. Explain Storage classes.

- Storage class decides the scope, lifetime and memory allocation of variable.
- Scope of a variable is the boundary within which a variable can be used.
- Four storage classes are available in C,
  1) Automatic (auto)
  2) Register (register)
  3) External (extern)
  4) Static (static)

1) automatic:

- Variables which are declared in function are of automatic storage class.
- Automatic variables are allocated storage in the main memory of the computer.
- Memory is allocated automatically upon entry to a function and freed automatically upon exit from the function.
- The scope of automatic variable is local to the function in which it is declared.
- It is not required to use the keyword **auto** because by default storage class within a block is auto.

Example:

```
int a;
auto int a;
```

2) **register:**

- Automatic variables are allocated storage in the main memory of the computer; However, for most computers, accessing data in memory is considerably slower than processing directly in the CPU.
- Register variables are stored in registers within the CPU where data can be stored and accessed quickly.
- Variables which are used repeatedly or whose access time should be fast may be declared to be of storage class **register**.
- Variables can be declared as a register: register int var;

3) **external:**

- Automatic and register variables have limited scope and limited lifetimes in which they are declared.
- Sometimes we need global variables which are accessible throughout the program.
- extern keyword defines a global variable that is visible to ALL functions.
- extern is also used when our program is stored in multiple files instead of single file.
- Memory for such variables is allocated when the program begins execution, and remains allocated until the program terminates. Memory allocated for an external variable is initialized to zero.
- Declaration for external variable is as follows: extern int var;

4) **static:**

- static keyword defines a global variable that is visible to ALL functions in same file.
- Memory allocated for static variable is initialized to zero.
- Static storage class can be specified for automatic as well as external variables such as:
  ```
  static extern int varx;     //static external
  static int var;             // static automatic
  ```
- Static automatic variables continue to exist even after the function terminates.
- The scope of static automatic variables is identical to that of automatic variables.

9. **Explain Input / Output functions.**

1) **scanf( )**

- It is used to read all types of data.
- It cannot read white space between strings.
- It can read multiple data at a time by multiple format specifier in one scanf( ).

  Example:
  ```
  scanf("%d%d", &a, &b);
  ```

2) **printf( )**

- It is used to display all types of data and messages.
- It can display multiple data at a time by multiple format specifier in one printf( ).

  Example:
  ```
  printf("a=%d b=%d", a, b);
  ```

Unformatted input output functions

1) **gets()**

- It is used to read a single string with white spaces.
- It is terminated by enter key or at end of line.

  Example:

  ```
  char str[10];
  gets(str);
  ```

2) getchar(), getche( ), getch( )

- It is used to read single character at a time.
- getchar( ) function requires enter key to terminate input while getche( ) and getch( ) does not require.
- getch( ) function does not display the input character while getchar( ) and getche( ) function display the input character on the screen.

  Example:

  ```
  char ch;
  ch = getchar();
  ch = getche();
  ch = getch();
  ```

3) puts()

- It is used to display a string at a time.

  Example:

  ```
  char str[]="Hello";
  puts(str );
  ```

4) putchar()

- It is used to display single character at a time.

  Example:

  ```
  putchar(ch);
  ```

Character checking functions.

Character checking functions are available in ctype.h header file.

1) isdigit(int); for checking number (0-9)
2) isalpha(int); for checking letter (A-Z or a-z)
3) isalnum(int); for checking letter (A-Z or a-z) or digit (0-9)
4) isspace(int); for checking empty space
5) islower(int); for checking letter (a-z)
6) isupper(int); for checking letter (A-Z)
7) ispunct(int); for checking punctuation symbols (like - : , ; , {, } , ?, . etc.)

10. Example: Write a program to check the entered character is digit or not

```
#include<stdio.h>
#include<conio.h>
#include<ctype.h>
void main()
{
        char c;
        clrscr();
        scanf("%c",&c);
```

```
        if(isdigit(c))
                printf("True");
        getch(); }
```

11. Explain command-line argument with example.

- Command-line arguments are given after the name of a program in command –line operating systems like DOS or Linux.
- Up to this point there is no argument in main() function.
- main() can accept two arguments :

  1)First argument is number of command-line arguments.

  2)Second argument is a full list of command-line argument.

  Syntax:

  int main(int argc, char *argv[])
- Here **argc** refers to the number of arguments passed and **argv[]** is a pointer array which point to each argument which passed to **main**.
- Following is a simple example which checks if there is any argument supplied from the command-line and take action accordingly:

  Example:

```
#include <stdio.h>
int main( int argc, char *argv[] )
{
  if( argc == 2 )
  {
    printf("The argument supplied is %s\n", argv[1]);
  }
  else if( argc > 2 )
  {
    printf("Too many arguments supplied.\n");
  }
  else
  {
    printf("One argument expected.\n");
  }
}
```

- Here **argv[0]** holds the name of the program itself and **argv[1]** is a pointer to the first command line argument supplied, and ***argv[n]** is the last argument.
- If no arguments are supplied, **argc** will be one and if you pass one argument then **argc** is set at 2.
- You pass all the command-line arguments separated by a space, but if argument itself has a space then you can pass such arguments by putting them inside double quotes "" or single quotes ‘’.

  Example:

```
$ ./a.out  testing1 testing2
Too many arguments supplied.
```