Back

FILE HANDLING, DYNAMIC MEMORY ALLOCATION

Back

- Array of Structure and pointers
- File Handling
- Opening & Closing file
- fread(), fwrite()
- Dynamic Memory Allocation

Back

ARRAY OF STRUCTURE AND POINTERS

Back

3.1.1 ARRAY OF STRUCTURES

- Definition:

  Structure is used to store the information of One particular object but if we need to store such 100 objects then **Array of Structure is used**.

- Example :

```
struct Bookinfo
        {
              char[20] bname;
              int pages;
              int price;
        }
        Book[100];
```

1.Here Book structure is used to Store the information of one Book.

2. In case if we need to store the Information of 100 books then Array of Structure is used.

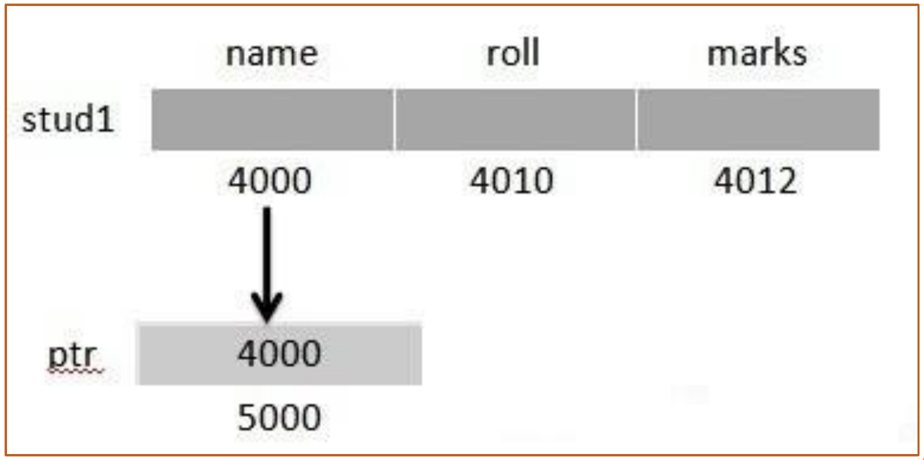
3. b1[0] stores the Information of 1st Book , b1[1] stores the information of 2nd Book and So on We can store the information of 100 books.

Back

3.1.2 POINTERS TO STRUCTURES

- Pointer to Structure in C Programming
  - Address of Pointer variable can be obtained using **'&'**operator.
  - **Address** of such Structure can be assigned to the **Pointer variable .**
  - **Pointer Variable** which stores the **address of Structure** must be declared as **Pointer to Structure .**
- Pointer to Structure Syntax :

```
struct student_database
        {
            char name[10];
            int roll;
            int marks;
        }stud1;
struct student_database *ptr;
 ptr = &stud1;
```



How to Access Structure Members?

Back

Simple program without Pointer variable:

```
#include <stdio.h>
 int main()
     {
        struct student database
               {
                 char name[10];
                 int roll; int marks;
                }stud1;
        stud1.roll = 10;
        stud1.marks = 90;
        printf("Roll Number : %d",stud1.roll);
        printf("Marks of Student : %d",stud1.marks);
        return 0;
      }
```

Output:

```
Roll Number : 10
Marks of Student : 90
```

Back

Simple program with Pointer

variable:

```
#include <stdio.h>
int main(int argc, char *argv[])
   {
      struct student_database
            {
               char name[10];
               int roll;
               int marks;
            }stud1 = {"Pritesh",90,90};
      struct student_database *ptr;
      ptr = &stud1;
      printf("Roll Number : %d",(*ptr).roll);
      printf("Marks of Student : %d",(*ptr).marks);
      return 0;
   }
```

Output:

```
Roll Number : 90
Marks of Student : 90
```

Back

3.1.3 ARRAY OF POINTERS

- Array of pointers

Back

FILE HANDLING

3.2.1 INTRODUCTION OF FILE HANDLING

- Drawbacks of Traditional I/O System.

1. Until now we are using **Console Oriented I/O functions**.
   "Console Application" means an application that has a text-based interface. (black screen window))

2. Most applications require a large amount of data , **if this data is entered through console then it will be quite time consuming task.**

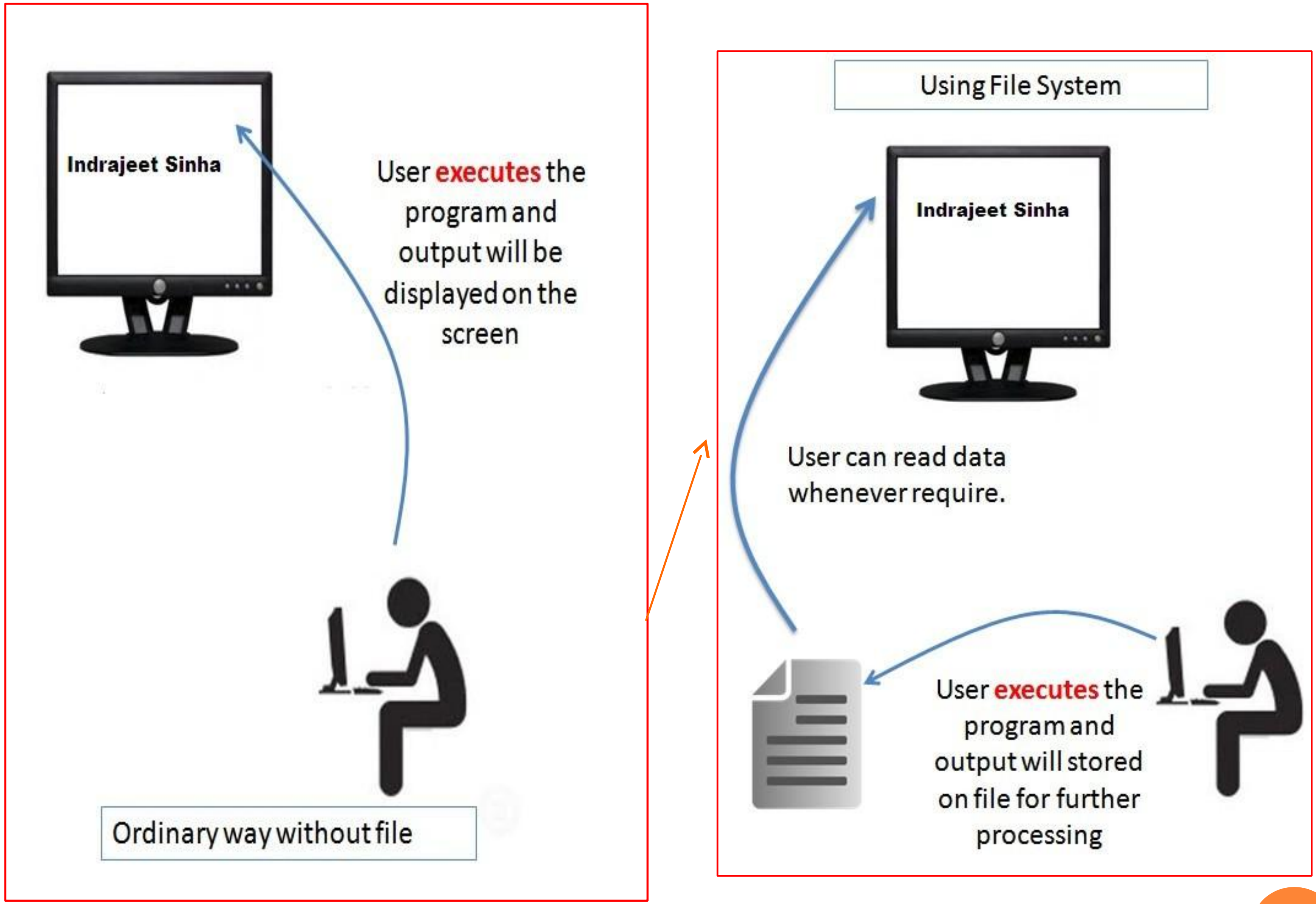
3. Main drawback of using Traditional I/O :- **data is temporary** (and will not be available during re-execution)

File handling in C :

1. New way of dealing with **data is file handling**.
2. Data is **stored onto the disk** and can be retrieve whenever require.
3. Output of the program may be stored onto the disk
4. In C we have many **functions that deals with file handling**
5. A file is a collection of bytes stored on a **secondary storagedevice** (generally a disk)

Collection of byte may be **interpreted as** –

- Single character
- Single Word
- Single Line
- Complete Structure.



User **executes** the program and output will be displayed on the screen

User can read data whenever require.

User **executes** the program and output will stored on file for further processing

3.2.2 OPENING & CLOSING FILE

- Opening Files

  We can use the **fopen( )** function to create a new file or to open an existing file. This call will initialize an object of the type **FILE**, which contains all the information necessary to control the stream. The prototype of this function call is as follows –

```
FILE *fopen( const char * filename, const char * mode );
```

Note:- Here, **filename** is a string literal, which will use to name of our file, and access **mode** can have any one.

Mode	Description
r	Opens an existing text file for reading purpose.
w	Opens a text file for writing. If it does not exist, then a new file is created. Here your program will start writing content from the beginning of the file.
a	Opens a text file for writing in appending mode. If it does not exist, then a new file is created. Here your program will start appending content in the existing file content.
r+	Opens a text file for both reading and writing.
w+	Opens a text file for both reading and writing. It first truncates the file to zero length if it exists, otherwise creates a file if it does not exist.
a+	Opens a text file for both reading and writing. It creates the file if it does not exist. The reading will start from the beginning but writing can only be appended.

If you are going to handle binary files, then we will use following access modes instead of the above mentioned

Mode	Description
rb	Opens an existing text file for reading purpose.
wb	Opens a text file for writing. If it does not exist, then a new file is created. Here your program will start writing content from the beginning of the file.
ab	Opens a text file for writing in appending mode. If it does not exist, then a new file is created. Here your program will start appending content in the existing file content.
r+b	Opens a text file for both reading and writing.
w+b	Opens a text file for both reading and writing. It first truncates the file to zero length if it exists, otherwise creates a file if it does not exist.
a+b	Opens a text file for both reading and writing. It creates the file if it does not exist. The reading will start from the beginning but writing can only be appended.

Closing file

To close a file, use the fclose( ) function. The prototype of this function is –

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

The **fclose(-)** function returns zero on success, or **EOF** if there is an error in closing the file. This function actually flushes any data still pending in the buffer to the file, closes the file, and releases any memory used for the file. The EOF is a constant defined in the header file **stdio.h**.

Back

Writing a File

Following is the simplest function to write individual characters to a stream –

```
int fputc( int c, FILE *fp );
```

The function **fputc()** writes the character value of the argument c to the output stream referenced by fp. It returns the written character written on success otherwise **EOF** if there is an error. You can use the following functions to write a null-terminated string to a stream –

```
int fputs( const char *s, FILE *fp );
```

The function **fputs()** writes the string **s** to the output stream referenced by fp. It returns a non-negative value on success, otherwise **EOF** is returned in case of any error.

Back

CONT..

```
#include <stdio.h>
main()
    {
      FILE *fp; fp = fopen("/tmp/test.txt", "w+");
      fprintf(fp, "This is testing for fprintf...\n");
      fputs("This is testing for fputs...\n", fp);
      fclose(fp);
     }
```

When the above code is compiled and executed, it creates a new file **test.txt** in /tmp directory and writes two lines using two different functions.

Back

CONT..

- Reading a File

Given below is the simplest function to read a single character from a file –

```
int fgetc( FILE * fp );
```

The **fgetc()** function reads a character from the input file referenced by fp. The return value is the character read, or in case of any error, it returns **EOF**. The following function allows to read a string from a stream –

```
char *fgets( char *buf, int n, FILE *fp );
```

The functions **fgets()** reads up to n-1 characters from the input stream referenced by fp. It copies the read string into the buffer **buf**, appending a **null** character to terminate the string.

Back

CONT..

- If this function encounters a newline character '\n' or the end of the file EOF before they have read the maximum number of characters, then it returns only the characters read up to that point including the new line character. You can also use **int fscanf(FILE *fp, const char *format, ...)** function to read strings from a file, but it stops reading after encountering the first space character.

Back

CONT..

```
main()
   {
        FILE *fp;
        char buff[255];
         fp = fopen("/tmp/test.txt", "r");
        fscanf(fp, "%s", buff); printf("1 : %s\n", buff );
 fgets(buff, 255, (FILE*)fp); printf("2: %s\n",buff );
 fgets(buff, 255, (FILE*)fp); printf("3: %s\n",buff );
 fclose(fp);
    }
```

Output:-

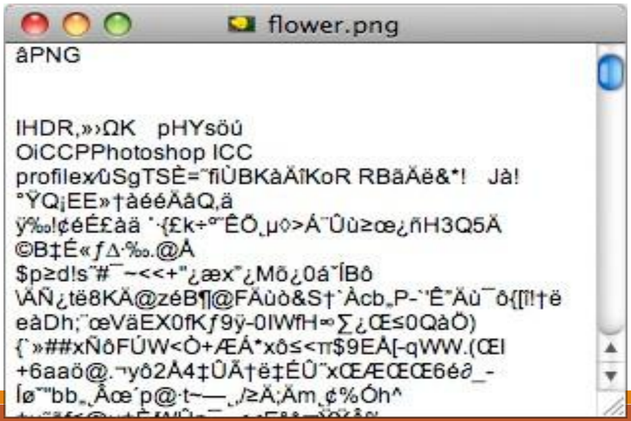
```
1 : This
2: is testing for fprintf...
3: This is testing for fputs...
```

Note: First, **fscanf()** read just **This** because after that, it encountered a space, second call is for **fgets()** which reads the remaining line till it encountered end of line. Finally, the last call **fgets()** reads the second line completely.

Back

3.2.3 BINARY & TEXT FILES

- Definition
- Text File:-A **text file** is a file that is properly understood as a sequence of character data, separated into lines. when a text file is displayed as a sequence of characters, it is easily human-readable.
- Binary File:- A **binary file** is anything else. A binary file will include some data that is not written using a character-encoding standard some number would be represented using binary within the file, instead of using the character representation of its various digits



```
âPNG


IHDR,»›ΩK   pHYsöú
OiCCPPhotoshop ICC
profilex/ûSgTSÈ=˜ﬁÙBKàÄîKoR RBäÄë&*!   Jà!
°ŸQ¡EE»†àééÄåQ,ä
ÿ‰!¢éÉ£àä ˙·{£k÷°˝ÊÕ¸µ◊>Á¨Ûù≥œ¿ñH3Q5Ä
©B‡É«ƒ∆·‰.@Å
$p≥d!s˜#¯~<<+"¿æx"¿Mõ¿0á˘ÎBô
\ÄÑ¿të8KÄ@zéB¶@FÄùò&S†`Àcb„P-`'Ê"Äù¯ô{[î!†ë
eàDh;¨œVäEX0fKƒ9ÿ-0IWfH∞∑¿Œ≤0QàÖ)
{`»##xÑôFÚW<Ò+ÆÁ*xô≤<π$9EÅ[-qWW.(ŒI
+6aaö@.¬yô2Å4‡ÛÄ†ë‡ÉÛ¨xŒÆŒŒ6é∂_-
Îø˜"bb„¸Åœ´p@·t~—¸„/≥Ä;Äm¸¢%Óh^
```

Back

FPRINTF(), FSCANF(), FEOF()

Lecture no.- 30,

UNIT- III

3.2.4 FPRINTF(), FSCANF(), FEOF()

Back

- C library function - fprintf()

Description

The C library function **int fprintf(FILE *stream, const char *format, ...)** sends formatted output to a stream.

Declaration

```
int fprintf(FILE *stream, const char *format, ...)
```

Parameters

- **stream** – This is the pointer to a FILE object that identifies the stream.
- **format** – This is the C string that contains the text to be written to the stream. It can optionally contain embedded format tags that are replaced by the values specified in subsequent additional arguments and formatted as requested. Format tags prototype is **%[flags][width][.precision][length]specifier ,** which is explained in next slide.

Back

specifier	Output
c	Character
d or i	Signed decimal integer
e	Scientific notation (mantissa/exponent) using e character
E	Scientific notation (mantissa/exponent) using E character
f	Decimal floating point
g	Uses the shorter of %e or %f
G	Uses the shorter of %E or %f

Back

CONT..

o	**Signed octal**
s	String of characters
u	Unsigned decimal integer
x	Unsigned hexadecimal integer
X	Unsigned hexadecimal integer (capital letters)
p	Pointer address
n	Nothing printed
%	Character

Back

Example:

```
#include <stdio.h>
#include <stdlib.h>
int main()
   {
         FILE * fp; fp = fopen ("file.txt", "w+");
         fprintf(fp, "%s %s %s %d", "We", "are", "in", 2012);
         fclose(fp);
         return(0);
   }
```

Let us compile and run the above program that will create a file **file.txt** with the following content −

```
We are in 2012
```

Back

C library function - fscanf(),

Description

The C library function **int fscanf(FILE *stream, const char *format, ...)** reads formatted input from a stream.

Declaration

Following is the declaration for fscanf() function.

```
int fscanf(FILE *stream, const char *format, ...)
```

Parameters

stream – This is the pointer to a FILE object that identifies the stream.

format – This is the C string that contains one or more of the following items – *Whitespace character*, *Non-whitespace character* and *Format specifiers*. A format specifier will be as **[=%[*][width][modifiers]type=]**, which is explained in next slide.

Back

Argument	Description
*	This is an optional starting asterisk indicates that the data is to be read from the stream but ignored, i.e. it is not stored in the corresponding argument.
width	This specifies the maximum number of characters to be read in the current reading operation.
modifiers	Specifies a size different from int (in the case of d, i and n), unsigned int (in the case of o, u and x) or float (in the case of e, f and g) for the data pointed by the corresponding additional argument: h : short int (for d, i and n), or unsigned short int (for o, u and x) l : long int (for d, i and n), or unsigned long int (for o, u and x), or double (for e, f and g) L : long double (for e, f and g)
type	A character specifying the type of data to be read and how it is expected to be read. See next table.

Example

```
#include <stdio.h>
#include <stdlib.h>
    int main()
        {
            char str1[10],
            str2[10], str3[10];
            int year;
            FILE * fp; fp = fopen ("file.txt", "w+");
            fputs("We are in 2012", fp);
            rewind(fp);
            fscanf(fp, "%s %s %s %d", str1, str2, str3, &year);
            printf("Read String1 |%s|\n", str1 );
            printf("Read String2 |%s|\n", str2 );
            printf("Read String3 |%s|\n", str3 );
            printf("Read Integer |%d|\n", year );
            fclose(fp);
            return(0);   }
```

Output:

```
Read String1 |We|
Read String2 |are|
Read String3 |in|
Read Integer |
2012|
```

Back

Back

C LIBRARY FUNCTION - FEOF()

Description

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

Declaration

Following is the declaration for feof() function.

```
int feof(FILE *stream)Parameter
```

stream – This is the pointer to a FILE object that identifies the stream.

Return Value

This function returns a non-zero value when End-of-File indicator associated with the stream is set, else zero is returned.

EXAMPLE:

Back

```
#include <stdio.h>
int main ()
   {
          FILE *fp; int c; fp = fopen("file.txt","r");
          if(fp == NULL)
                    {
                      perror("Error in opening file");
                      return(-1);
                    } while(1)
          { c = fgetc(fp);
                    if( feof(fp) )
          { break ;
           }
            printf("%c", c);
          } fclose(fp);
          return(0);
   }
```

NOTE: Assuming we have a text file **file.txt**, which has the following content. This file will be used as an input for our example program –
This is Imperial

Output:
This is Imperial

Back

FREAD(), FWRITE()

Lecture no.- 31,
UNIT- III

Back

3.2.6 FREAD(), FWRITE()

- C library function - fread()

Description

The C library function **size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream)** reads data from the given **stream** into the array pointed to, by **ptr**.

Declaration

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

Parameters

ptr – This is the pointer to a block of memory with a minimum size of size*nmemb bytes.

size – This is the size in bytes of each element to be read.

nmemb – This is the number of elements, each one with a size of **size** bytes.

stream – This is the pointer to a FILE object that specifies an input stream.

Back

Example

```
#include <stdio.h>
 #include <string.h>
int main()
{
    FILE *fp; char c[] = "this is Indrajeet";
    char buffer[100];  /* Open file for both reading and writing */
    fp = fopen("file.txt", "w+"); /* Write data to the file */
    fwrite(c, strlen(c) + 1, 1, fp); /* Seek to the beginning of the file */
    fseek(fp, SEEK_SET, 0); /* Read and  display data */
    fread(buffer, strlen(c)+1, 1, fp);
    printf("%s\n", buffer);
    fclose(fp);
    return(0);
}
```

Output:- this is Imperial

Back

C library function - fwrite()

Description

The C library function **size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream)** writes data from the array pointed to, by **ptr** to the given **stream**.

Declaration

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

Parameters

ptr – This is the pointer to the array of elements to be written.

size – This is the size in bytes of each element to be written.

nmemb – This is the number of elements, each one with a size of **size** bytes.

stream – This is the pointer to a FILE object that specifies an output stream.

Back

```
#include<stdio.h>
int main ()
{
   FILE *fp; char str[] = "This is Arya First year Department";
   fp = fopen( "file.txt" , "w" );
   fwrite(str , 1 , sizeof(str) , fp );
   fclose(fp);
   return(0);
 }
```

Note:- Let us compile and run the above program that will create a file **file.txt** which will have following content –

This is Arya First year Department

Back

Now let's see the content of the above file using the following program

```
#include <stdio.h>
int main ()
   {
         FILE *fp; int c; fp = fopen("file.txt","r");
         while(1)
             {
                   c = fgetc(fp);
                    if( feof(fp) )
                   {
                      break ;
                   }
         printf("%c", c);
             }
         fclose(fp);
         return(0);
```

Output:

```
This is Arya First year
Department
```

3.2.8 SEQUENTIAL VS. RANDOM ACCESS FILES

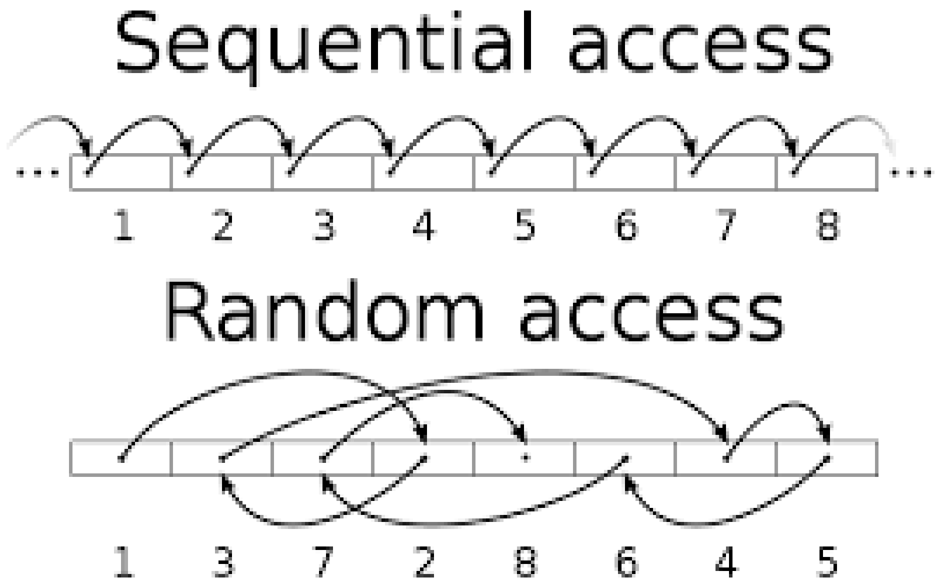
Back

- In computer programming, the two main types of file handling are:
  - Sequential;
  - Random access.
- **Definition:**

Sequential files are generally used in cases where the program processes the data in sequential fashion – i.e. counting words in a text file – although in some cases, random access can be feigned by moving backwards and forwards over a sequential file.

True random access file handling, however, only accesses the file at the point at which the data should be read or written, rather than having to process it sequentially. A hybrid approach is also possible whereby a part of the file is used for sequential access to locate something in the random access portion of the file, in much the same way that a File Allocation Table (FAT) works.

Back



Back

DYNAMIC MEMORY ALLOCATION

3.3.1 DYNAMIC MEMORY ALLOCATION

Back

- **C dynamic memory allocation** refers to performing manual memory management for dynamic memory allocation in the C programming language via a group of functions in the C standard library, namely **malloc, realloc, calloc and free**.

- 3.3.1.1 Malloc

Description

The C library function **void *malloc(size_t size)** allocates the requested memory and returns a pointer to it.

Declaration

```
void *malloc(size_t size)
```

Parameters

size -- This is the size of the memory block, in bytes.

Example

```
#include <stdio.h>
#include <stdlib.h>
  int main()
    {
        char *str; /* Initial memory allocation */
        str = (char *)
        malloc(15);
        strcpy(str, " Imperial College ");
        printf("String = %s, Address = %u\n", str, str); /* Reallocating memory */
        str = (char *) realloc(str, 25);
        strcat(str, ".com");
        printf("String = %s, Address  = %u\n", str, str);
        free(str);
        return(0);
    }
```

Output:

```
String = Imperial College,
Address = 355090448
String = Imperial College,
Address = 355090448
```

- Definition:

When **calloc** is used to allocate a block of memory, the allocated region is initialized to zeroes.

In contrast, malloc does not touch the contents of the allocated block of memory, which means it contains garbage values.

Description

The C library function **void *calloc(size_t nitems, size_t size)**allocates the requested memory and returns a pointer to it. The difference in **malloc** and **calloc** is that malloc does not set the memory to zero where as calloc sets allocated memory to zero.

Declaration

```
void *calloc(size_t nitems, size_t size)
```

Parameters

nitems -- This is the number of elements to be allocated.

size -- This is the size of elements.

Back

```
#include <stdio.h>
#include <stdlib.h>
int main()
    {
        int i, n; int *a;
        printf("Number of elements to be entered:");
         scanf("%d",&n);
         a = (int*)calloc(n, sizeof(int));
         printf("Enter %d numbers:\n",n);
         for( i=0 ; i < n ; i++ )
                {
                   scanf("%d",&a[i]);
                }
                 printf("The  numbers entered are: ");
                for( i=0 ; i < n ; i++ )
                            {
                                        printf("%d ",a[i]);
                            } free( a );
                 return(0);
     }
```

Output:

```
Number of elements to be
entered:3
Enter 3 numbers:
22
55
14
The numbers entered are: 22 55 14
```

3.3.1.3 FREE

Description

The C library function **void free(void *ptr)** deallocates the memory previously allocated by a call to calloc, malloc, or realloc.

Declaration

```
void free(void *ptr)
```

Parameters

ptr -- This is the pointer to a memory block previously allocated with malloc, calloc or realloc to be deallocated. If a null pointer is passed as argument, no action occurs.

Return Value

This function does not return any value.

Back

Example

```
#include <stdio.h>
 #include <stdlib.h>
#include <string.h>
 int main()
{
    char *str; /* Initial memory allocation */
    str = (char *) malloc(15);
    strcpy(str, “Indra jeet Sinha");
    printf("String = %s, Address = %u\n", str, str); /* Reallocating memory */
    str = (char *) realloc(str, 25);
    strcat(str, ".com");
    printf("String = %s, Address = %u\n", str, str); /* Deallocate allocated memory */
    free(str);
    return(0);
}
```

Output:

```
String = Indrajeet sinha, Address =
355090448 String = Indrajeet sinha,
Address = 355090448
```

3.3.1.4 REALLOC

Back

Description

The C library function **void *realloc(void *ptr, size_t size)**attempts to resize the memory block pointed to by **ptr** that was previously allocated with a call to **malloc** or **calloc**.

Declaration

```
void *realloc(void *ptr, size_t size)
```

Parameters

ptr -- This is the pointer to a memory block previously allocated with malloc, calloc or realloc to be reallocated. If this is NULL, a new block is allocated and a pointer to it is returned by the function.

size -- This is the new size for the memory block, in bytes. If it is 0 and ptr points to an existing block of memory, the memory block pointed by ptr is deallocated and a NULL pointer is returned.

Back

Return Value

This function returns a pointer to the newly allocated memory, or NULL if the request fails.

Example

```
#include <stdio.h>
#include <stdlib.h>
int main() {
char *str; /* Initial memory allocation */
str = (char *)
malloc(15);
strcpy(str, “Indrajeet Sinha");
printf("String = %s, Address = %u\n", str, str); /* Reallocating memory */
str = (char *)
realloc(str, 25);
strcat(str, ".com");
printf("String = %s, Address = %u\n", str, str);
free(str); return(0); }
```

Output:

```
String = Indrajeet sinha, Address =
355090448 String = Indrajeet sinha,
Address = 355090448
```